Pforzheim University
School of Engineering

Project work

Simulation and evolutionary training of object collecting agents

by **Daniel Schembri** - matriculation number: 310026

Summer term 2015
31st August 2015

Examiner: Prof. Dr. Richard Alznauer
Supervisor: Dr. Christoph Ussfeller

Contents

Erklärung

Ich versichere, die beiliegende Projektarbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet zu haben.

Kraichtal, den 31. August 2015 Daniel Schembri

Abstract

This document was created for the project work at Pforzheim university in the course: Master of Science in Embedded Systems. This projectwork describes the development of a simulation environment for object collecting agents. The agents have a neural network that can be trained supervised or evolved by evolutionary algorithms.
The simulator was written in C++ using the 2D physics-engine Box2D by Erin Catto and using the OpenGl-frameworks GLUT, respectively Freeglut for accelerating the displaying of the simulation and GLUI for the graphical user interface.
The neural network framework is written by Jonathan Schwarz and detaily described in his projectwork ' A comparison of neural network types and learning techniques with an application in artificial life'.

Zusammenfassung

Dieser Projektbericht wurde im Rahmen der Projektarbeit des Studienganges Master of Science in Embedded Systems der Hochschule Pforzheim angefertigt.
Diese Projektarbeit beschreibt die Entwicklung eines Simulators für virtuelle Agenten, die Objekte in einer 2-dimensionalen Umgebung sammeln. Die Agenten besitzen ein neuronales Netz das wahlweise supervised oder evolutionär angepasst werden kann.
Der Simulator wurde in C++ entwickelt. Die Physics-engine Box2D von Erin Catto wird verwendet. Ebenso die OpenGL-Frameworks GLUT bzw. Freeglut zur beschleunigten Darstellung der Simulation und GLUI für die Graphische Benutzer Schnittstelle.
Das Framework der neuronalen Netze wurde von Jonathan Schwarz entwickelt und wird in seinem Projektbericht 'A comparison of neural network types and learning techniques with an application in artificial life' näher beschrieben.

List of acronyms

AABB Axis aligned bounding box

ANN Artificial neural network

GLUI OpenGL User Interface Library

GLUT OpenGL Utility Toolkit

MLP Multilayer perceptron

OpenGL Open Graphics Library

1 Introduction

This document was created at Pforzheim University in the course Master of Science in Embedded Systems. It was part of the project work.

1.1 Document structure

The first chapters explain the basic concepts of software agents, artificial neural networks and evolutionary algorithms. The following chapters introduce the used libraries and tools.
The main part describes the implementation of the simulator and the theoretical considerations. The last part concludes with the simulation results and a further view of the discussed topic.

1.2 Motivation

The classical method to solve a task mostly is to use a detailed prior knowledge. If this knowledge isn't sufficient it has to be acquired hard. The development of this solution therefore can be difficult, time-consuming and expensive. This circumstances cause an interest in solving tasks in an universal way. Creatures seems to solve problems easily by the abilities they received by evolution. This motivated researcher to apply this mechanisms in computer science. Thanks to the computing power of today evolution can be used in an abstract way to cultivate solutions and handle optimization problems.
The concept of software agents is one model of artificial intelligence. It's an abstract view of an intelligent system that perceives its environment and acts self-sufficiently.
Artificial neural networks are a concept to approximate target functions. These networks can be used for tasks with just partitial knowledge. They work well for finding patterns for example.
In this projectwork the main interest lies in agents, artificial neural networks and evolutionary training of these networks. The motivation is to create a suitable simulation environment and to compare different evolutionary algorithms.

1.3 Project task

The task of this project work is to develop a simulator to let agents collect objects in a 2-dimensional world. As follows a sketch of the main concept:

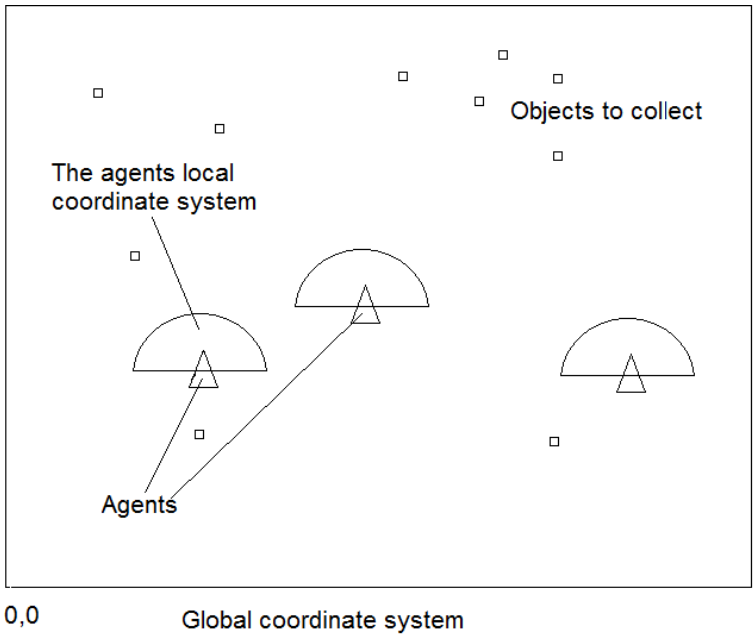


Figure 1.1: A sketch of the main concept

The agents and the objects they have to collect are spawned randomly. An agent has to collect as much objects as possible in a given time. Every agent has a sensor, represented by the semicircle in the sketch, to get the distance vector of an object. The object is just detected if it is in range of the sensor. For simplicity just the position of the nearest object is given, if more objects are in range of the sensor. Based on this input data the agent has two possibilities to act: it can rotate left or right and it can drive forward. Therefore the agents output has two parameters, defining the rotation-direction and speed and the velocity. To choose the parameters the agent has a logic, processing the input.
This logic is realized by a neural network that is trained supervised or evolved with special algorithms derived from evolution theory. The simulator has to simulate a realistic environment. Therefore a physics engine is used. The simulation has to be rated and the collected data displayed. The simulator uses the neural network framework developed by Jonathan Schwarz. Artificial neural networks are evolved in this project work to see if they can solve the task satisfactionally. The concept of software agents is used as a frame to get an abstract view of the used algorithms.

2 The concept of agents

An agent is a program that interacts self-sufficently depending on its actual state. In general, it has sensors, a processing logic and actuators. The input data will be processed to behave in the desired way. In the simplest case this could be a table containing the required output to every possible input. Most agents are found in the world wide web. For example webcrawlers, search-engines and bots are well known. In this project the concept of software agents is used as a modelling framework to differentiate the agent from the environment.
The following figure shows the general structure of an agent:

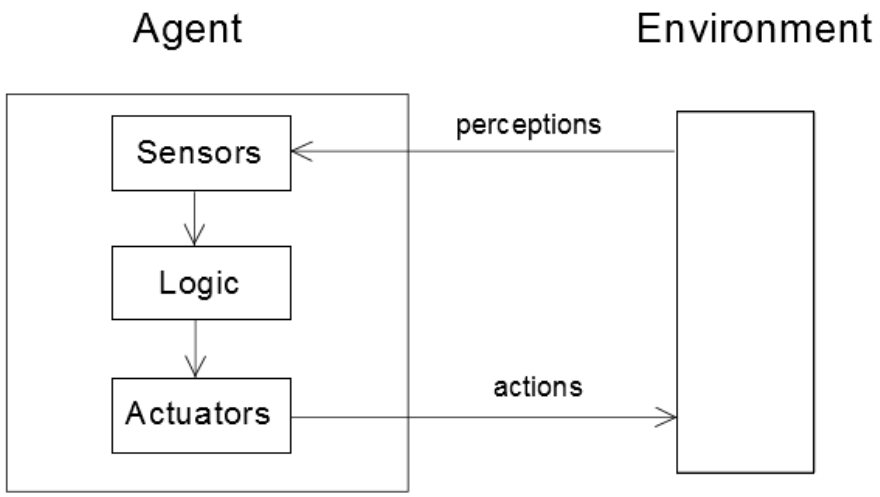


Figure 2.1: General structure of an agent (oriented on [RN10] Figure 2.9)

The logic part of an agent can be less or more complex. Just the actual perception can be processed or a specified sequence of perceptions. This could be realised by a simple script for example. In this project the agents logic is realised as a artificial neural network.
In the following sections the different realisation types of the logic is described in detail.

2.1 Rationality

An agent creates a sequence of actions. These actions cause a sequence of states in the corresponding environment. Every sequence of states can be rated. The agent is called rational if it can maximize its score, considering its sequence of perceptions and its previous knowledge of the problem.[RN10] It's important to distinguish rationality and perfection. An agent never will be perfectly solving a problem. It's unrealistic to expect the best possible solution for a problem in partitial observable

environments. It can solve the problem as it can be expected by the given perceptions. The agent can't predict random events.
In order to achieve rationality the following topics have to be considered:

- A sufficient previous knowledge of the problem
- A good rating of the resulting state sequence of the environment
- The perception sequence
- The possible actions the agent can execute

See also [RN10] chapter 2.2.1

2.2 Types of agents

By [RN10] agents can be classified in the following categories:

2.2.1 Simple reflex agent

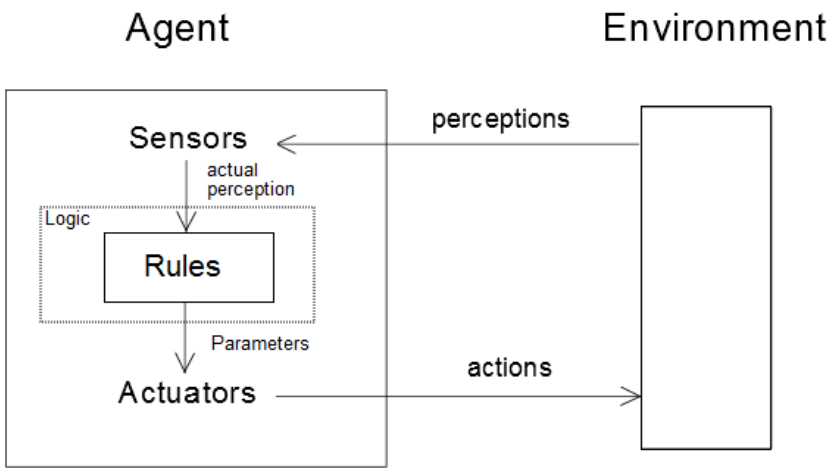


Figure 2.2: Structure of a simple reflex agent (oriented on [RN10] Figure 2.9)

The simplest type of an agent is the reflex based agent. This agent only processes the actual perception and act on the base of this actual state. In the simplest case the logic is realised as a lookup table or defined by simple rules. This type of agent can be used for simple tasks. The benefits are a fast implementation and a drastically reduced state space. The disadvantage lies within the simplicity. This agent has no memory and therefore doesn't include past states in his decisions. If the present state of the environment doesn't provide enough information or it is just partially observable, the agent could not solve the task. In partially observebale environments there can occur loops of wrong behaviour. This loops can be overcome if a random action can be chosen. For best functionality of a simple reflex agent the problem-environment should be fully observable.[1] For more complex tasks using the following described agent-types may be a better solution. ([RN10] chapter 2.4.2)

[1] See Chapter 2.3 for an explanaition of the environment

2.2.2 Model-based reflex agent

The model-based reflex agent expands the simple reflex agent by a memory. It internally creates a model of the world by past sensor data. This model is used to predict future states in the environment. This can be a moving object or changes in the environment made by the agent itself for example.
Internal states of the agent are used to approximate the state of the environment, if changes occur. The model is also used to estimate temporaly unobservable parts of the environment.
The model-based reflex agents ability to solve a task reaches its limit, if it has to act in a new environment or other agents acting in the world, too. The internally created model then may not fit anymore. ([RN10] chapter 2.4.3)

2.2.3 Goal-based agent

For some tasks it isn't enough to know the correct action on a specified state at the environment, like reflex-based agents do. It can be important for an agent to know the goal behind those actions. This is necessary for the generalization of a problem. For example: For a given route A to B there can be a description for a sequence of actions the agent has to execute in order to reach the target B. It is more difficult to describe specific rules for getting from one point to another in general, as other routes may differ a lot. It's therefore too complex to describe all rules in detail; hencefore a generalization of this task has to be modeled.
The benefit is an agent that can solve complex tasks. The drawback is a need of more computing time. For real-time tasks this can be an obstacle. ([RN10] chapter 2.4.4)

2.2.4 Utility-based agent

The utility-based agent expands the goal-based agent by the fact, that it can choose between several solutions. Therefore the agent has a rating function to choose the best or at least a good enough solution by several criteria. This rating function can be more complex then computing solutions of the task itself.
The development of such complex agent and corresponding rating algorithms can be difficult and is an own topic of computer science and includes research areas like perception, representation, reasoning and learning. ([RN10] chapter 2.4.5)

2.2.5 Learning agent

The complete development of an agent is difficult and time-consuming. Therefore Alan Turing searched in 1950 for a way to speed up this process.[Tur50] The concept of the learning agent was created. The learning agent has four components:

- **Performance element**
- **Critic**
- **Learning element**
- **Problem generator**

The **performance element** is the executive part of the agent. It processes sensor data and makes actions.

The **learning element** is responsible for improvements of the agent. The learning element can contain a decision tree, a neural net, a support vector machine or other learning structures.

The **critic** rates the performance element and tells the learning element how to improve it.

A **problem generator** is used to improve learning by making new experiences. Without the problem generator the agent would work with its best actions, but would never try new actions, that may seem uneffective in the first way, but may prove to be more effective in a long term run.
It can be important for the agent to gain further knowledge by its perceptions. Therefore the agent has to make some actions to explore the world, if he acts in a unknown environment. An agent who doesn't learn can most likely not solve a problem if the environment changes. In nature animals are equipped with reflex actions to survive long enough to learn to adapt to their enviroment.

The most important part for improvements is the correlation between the **performance element** and the **learning element**.
([RN10] chapter 2.4.6)

2.3 The task environment

In the first case it's preferable to define the task completely. In [RN10] chapter 2.3 the task environment is defined by four categories:

• **Performance**	• **Actuators**
• **Environment**	• **Sensors**

The **performance** is defined by the rating the agent can achieve. The performance measure is done by metrics the developer has to choose before creating agents. Metrics can for example be how good an agent solves the problem, how fast it solves it, how efficient it uses ressources, etc. It is possible to rate the agent by multiple criteria.

The **environment** is the frame an agent solves a task within. If the sensors allow the agent to get the information of the whole environment, the environment is called **fully observeable**. The agent therefore can get all relevant information for its actions. Otherwise an environment is classified as **partially observeable**. For example this can be if the sensors are limited or influenced by interferences. If the agent has no sensors at all, the task environment is called **unobserveable**. Instinctively you would think this kind of agent would be totally useless, but there exist succesful algorithms to solve problems. For example a vacuuming robot can clean a room effectively. It therefore uses stochastic algorithms. In factories this is used effectively to turn objects on an assembly line. The benefit is less costs in sensors.

It can also be necessary to know if the environment is a **single-** or **multiagent**-based one. Multiple agents can **cooperate** or **compete**.

The environment is **deterministic** if all further states are predictable by the current state and there are no random events. Otherwise it is **stochastic**. In a stochastic environment there are given probabilities for each possible event. If not it is called **non-deterministic**. In reality there can exist deterministic environments with the disadvantage of too much possibilities. An agent can't compute every possible effect then. In this case the environment must be handled by the agent as if it were stochastic to get a chance of solving the task.

If the action of the agent does only depend on the actual environment state and does not influence future state the environment is called **episodic**. Otherwise it's called **sequential**. Chess for example is a sequential environment because every move can bring the game in a win- or lose-situation.

A **static** environment doesn't change. This makes it easier for the agent to solve a task. A **dynamic** environment on the other hand needs a continuously observation. It is possible that the environment is static, but the rating of the agent depends on the passed time. Then it's classified as **semidynamic**. An example for a static environment is chess. Playing chess with a clock is semidynamic.

The environment can have **discrete** or **continous** states. Chess has discrete states, as it is played move by move. It could get a continous character if the time to calculate a move plays a role. Driving a car is in general continously as accelerating and steering have continous units.

The environment can be **known** or **unknown**. That means that the agent has foreknowledge about the environment. This category has nothing to do with observability.

The actuators are the components an agent uses to interact with the environment. While developing an agent the types of actuators and, for physical agents, the degrees of freedom[2] may be considered.

The sensors are responsible for receiving data of the environment. With appropriate algorithms a better evaluation of sensor data is possible. The advantage can be a noticeable reduction of sensor costs.

[2]The degrees of freedom are the amount of independent parameters of a physical system that is necessary to describe the state of this system

3 Artificial neural networks

Artficial neural networks (ANN) are inspired by their biological equivalents. They are used in machine learning as an own class of classfiers, next to support vector machines and other mathematical methods. On the one hand there exist models offering biological plausability to research the operations in the human brain. On the other hand there exist models whos main focus lies in finding good mathematical solutions for technical tasks.
Artificial neural networks are often seen as a black box:

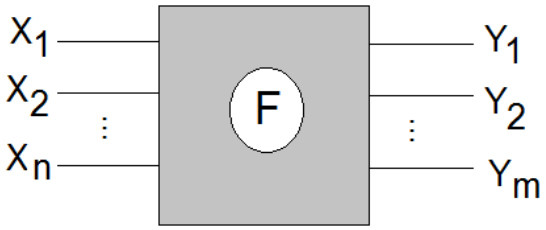


Figure 3.1: An ANN as black box [Roj96]

They approximate target functions. For a given input vector $\vec{X}$ an output vector $\vec{Y}$ can be computed. This happens on the base of a given data set or completely unsupervised. They map a function as follows[Roj96]:

$$F : R^n \rightarrow R^m \tag{3.1}$$

In general an ANN has an input layer, containing the inputvector $\vec{X}$ with n components, optional hidden layers and one output layer giving the resultvector $\vec{Y}$ with m components. All neurons, except the input neurons, have real weighted input connections, an activation function and output values. The input neurons do no computation. They just contain the input data and pass it to the next layer. In literature an artificial neural network is often showed as follows: [Roj96] [RW08]

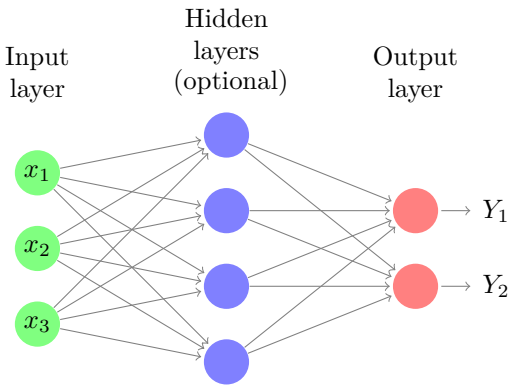


Figure 3.2: The general structure of an artificial neural network

This is a typical feed forward network. The computation results are passed from one layer to the next one. All values are passed at the same time. The values are weighted by a defined weight of the respective connection to the next neuron. It's possible to add a bias neuron to each layer. Bias neurons have always the value 1 and have also weighted connections to all neurons of the next layer. The neural network learns by adjusting the weights. In the most cases the weights are adjusted by a specified learning algorithm to fit the target functionality.
As follows the input of one neuron and the output result:

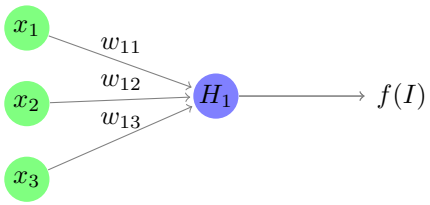
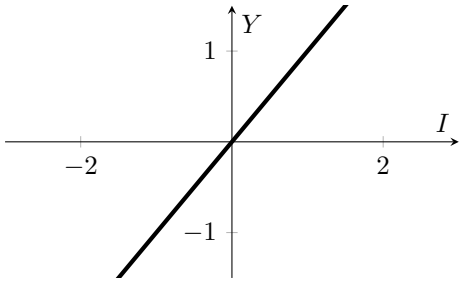


Figure 3.3: Computation of a neurons output

The total neuron input I is the sum of all input values x_i multiplied by their respective weight w_{1i}. The first index of the weight stands for the receiving neuron; the second index for the sending neuron. As follows the total input I of a receiving neuron:

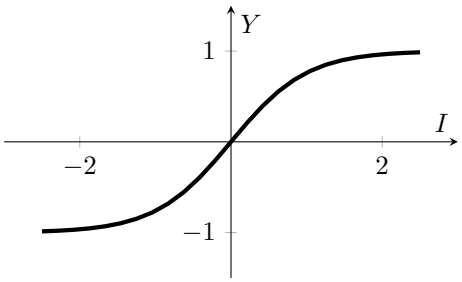
$$I = \sum_{i=1}^{n} (x_i * w_{1i}) \tag{3.2}$$

The output of the neuron is computed by a so called activation function $f(I)$, having as parameter the before determined total input I of the receiving neuron.
If the computing neuron is a hidden neuron, the output is also weighted and passed to the next units. If it is an output neuron then the output is a component of the Outputvector $\vec{Y}$. Usually all neurons of an entire layer have the same activation function.[RW08] As follows some commonly used activation functions:



$Y = I$

Linear function (here: identity function)

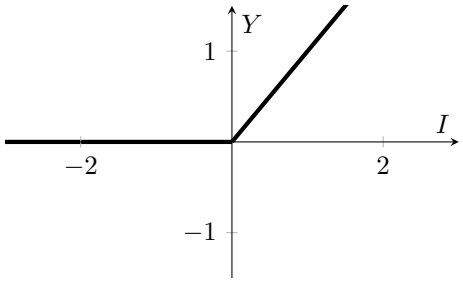


$Y = \tanh(I)$

Tangens hyperbolicus

Figure 3.6: Activation functions (oriented on [RW08])

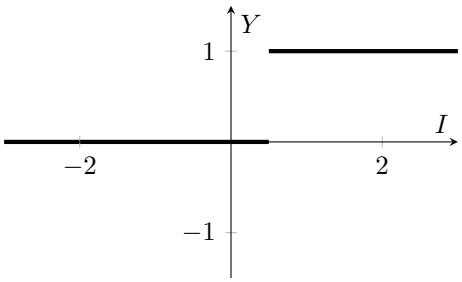
Activation functions can have a threshold S. The neurons output is zero if $I < S$:



$S = 0$

$$Y = \begin{cases} I & \text{if } I \geq S \\ 0 & \text{otherwise} \end{cases}$$

Linear function with threshold S



$S = 0.5$

$$Y = \begin{cases} 1 & \text{if } I \geq S \\ 0 & \text{otherwise} \end{cases}$$

Binary function with threshold S

Figure 3.9: Activation functions with threshold (oriented on [RW08])

Discrete activation functions can be used for classification of a dataset.
Continous functions can be used for regression. Regression analysis is the process of finding a target function to fit a sequence of datapoints. This is used for prediction of further datapoints.

As follows an example of the computation of a neural network. Assume there are 3 input neurons, 1 output neuron, no hidden neurons and no bias neurons. Assume there are the following input values and weights:

$$\begin{aligned} &x_1 = 3, \quad x_2 = -1, \quad x_3 = 0 \\ &w_{11} = 4, \quad w_{12} = 1, \quad w_{13} = 2 \end{aligned} \tag{3.3}$$

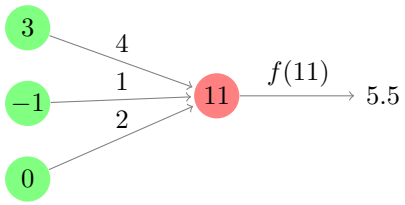


Figure 3.10: Example computation of a neuron

In this example the output neuron gets a total input of:

$$I = \sum_{i=1}^{3}(x_i * w_{1i}) = (3 * 4) + (-1 * 1) + (0 * 2) = 11 \tag{3.4}$$

This input may be processed in this example by a linear activation function with a threshold:

$$S = 0 \tag{3.5}$$

$$f(I) = \begin{cases} 0.5 * I & \text{if } I \geq S \\ 0 & \text{otherwise} \end{cases}$$

The final result of the output neuron's computation is:

$$f(I) = 0.5 * I = 0.5 * 11 = 5.5 \tag{3.6}$$

3.1 Network models

In the section before a feedforward network of the 2nd generation has been presented. In this network type the data is passed from one layer to the the next one. At the moment there are three generations of neural networks:

- **1. generation - Mcculloch-Pitts Neuron, Perceptron (binary)**
- **2. generation - real valued perceptron, multi layer perceptron(MLP) (real)**
- **3. generation - spiking neural networks (time encoding)**

The first generation works with binary in- and outputs, but real thresholds. The first model, the Mcculloch-Pitts Neuron was presented in 1943. The second generation was presented in the 90s. At that time scientists supposed that the firerate of a neuron, means the amount of binary signals in one second symbolise an encoding of information. Henceforth new models used real values representing these firerates instead of just binary values. Further research showed that biological neural networks can compute information faster than a second generation network. A third generation of artificial neural networks was modeled. Not just the firerate keeps information. Also the time delay between single signals encodes information. These signal pulses are called spikes. A third generation neural network is called a 'spiking neural network'. [Kla10]
There also exist other kind of topologies. For example there are recurrent network models in which a neuron can have a weighted connection to itself to process its own output in the next computation cycle of the network. Another topology is the kohonen map. In this topology all hidden neurons are interconnected. In this project work the focus lies on the 2nd generation neural networks and the feed forward topology.
As follows some different neuron- and networktypes are described by a chronological view.

3.1.1 The model of McCulloch and Pitts

The first mathematical model of a neuron was the McCulloch-Pitts-neuron, developed by Warren McCulloch and Walter Pitts in 1943.[MP43] They wanted to realise a simple, realistic neuron-model of the operations in the human brain and find out if the brain could compute turing computable functions. McCulloch-Pitts-neuron has binary inputs and one binary output. A real threshold has to be defined for each neuron, which reached, let the neuron fire[1] one or otherwise zero as output. In contrast to other models of further generations, it has no input weights. It's possible to add absolute suppressing inputs. If one of these has the value one, the neuron gives out zero, no matter of other inputs. Also it's possible to use relative suppressing inputs instead. For each relative suppressing input, the sum of all neuron inputs is decremented. [2]
Neural networks of this type of neuron can just learn by changing the threshold and the topology of the network. This causes an unnecessary workload due to its high effort in implementation.
At the following a figure of a single neuron is shown:

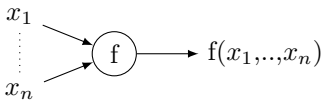


Figure 3.11: A McCulloch-Pitts-neuron

In general a neuron can be considered as a function. The neuron has n inputs and 1 output that can be connected to multiple neurons. It has an activation function that puts out the value one if the threshold S is reached.

The overall input is computed by:

$$I = \sum_{i=1}^{n} x_i \quad x \in \{0, 1\} \tag{3.7}$$

x_n are the binary inputs.

The output Y is:

$$\begin{cases} 1 & \text{if } I \geq S \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

A function composition:

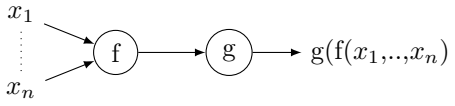


Figure 3.12: Function composition of two McCulloch-Pitts-neurons

The function composition is a concatenation of multiple neurons.

[1] In biology the word 'fire' means that the neuron puts out a signal
[2] In [Roj96] chapter 2.4.2 it's shown that Mcculloch-Pitts-networks with absolute suppressing inputs can be replaced by networks with relative suppressing inputs and vice versa. The topologies may differ

Applications The McCulloch-Pitts-neuron is biologically not plausible, because learning can just happen by changing the threshold and the network topology. This needs complex algorithms. But this neuron type can realise AND-, OR- and NOT-Gates. Therefore a network of neurons can realise every boolean logic and it is used in electrical engineering, because of realising logic gates efficently in contrast to classical gates. [Roj96] The usage of a McCulloch-Pitts network depends on the application type. Binary signals have the advantage of beeing less susceptible to interferences as multiple signal states. The drawback can be a complex topology.

3.1.2 Perceptron

The classical perceptron was published by Frank Rosenblatt in 1958[Ros58]. It is an artificial neural network with just an input- and an outputlayer with binary values. In contrast to the McCulloch-Pitts network it has additionally real weights. The advantage is that learning-algorithms just have to change the weights instead of the topology or the threshold. It's possible to define positive (stimulation), negative (supression) and zero (neutral) weights.
One main task of neural networks is to classify data. The network is therefore used as a seperator. The classical perceptron can seperate linear datasets.
As follows two examples of linear separable boolean functions:

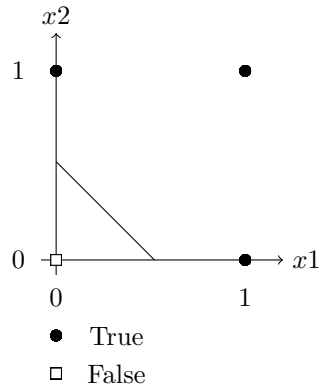
x1	x2	f(x1, x2)
0	0	0
0	1	1
1	0	1
1	1	1

(a) OR-Function: linear seperable

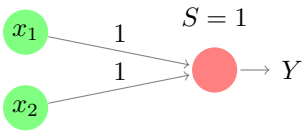
x1	x2	f(x1, x2)
0	0	0
0	1	0
1	0	0
1	1	1

(b) And-Function: linear separable

The perceptrons can be realised as follows:

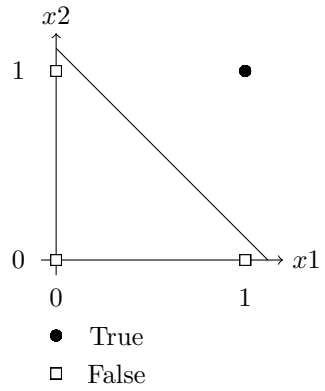


(c) A separating plane of the or-function

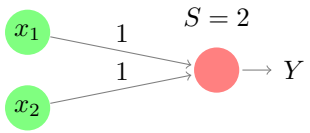


(d) Or-function perceptron

For the weights $w_1 = w_2 = 1$ the threshold has to be $0 < S \leq 1$.



(e) A separating plane of the and-function



(f) And-function perceptron

For the weights $w_1 = w_2 = 1$ the threshold has to be $1 < S \leq 2$

For two binary inputs there are just two functions that aren't linary separable. These are the **XOR**- and the **NXOR** function.

x1	x2	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

The two red marked functions aren't linearly seperable

Figure 3.13: Boolean functions for two parameters

In literature the impossibility of the linear separation of the XOR-function with a classical perceptron is called the **XOR-Problem**[3]:

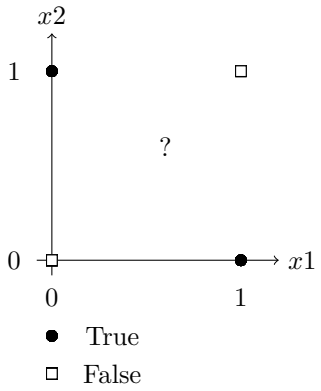


Figure 3.14: The XOR-separation problem

The solution of the problem is shown in the following section.

[3] see also [SM13] chapter 2.4 and [Roj96] chapter 3.2.3

Applications The perceptron can be used as pattern associator to assign an input pattern to a class. It can also be used as competitive network. Also linear seperable data can be categorised.

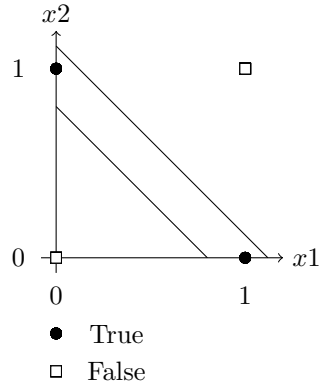
3.1.3 Multi layer perceptron

The multilayer perceptron (MLP) was intruduced by Minsky and Papert in 1969 [MP69]. It has an input-, an output- and one or multiple hiddenlayers (see figure 3.2). In contrast to the singlelayer perceptron it can solve the xor-problem and other non-linear problems. It can be trained super- or unsupervised.
A widely known example is the xor-problem:

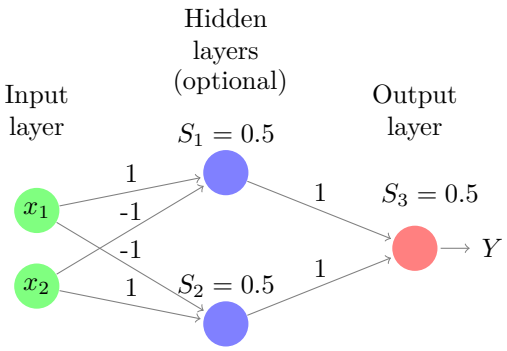
x1	x2	f(x1, x2)
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3.15: XOR-Function: Not linear seperable

Surely this problem can be solved by just two layers and with non-linear activation functions. But a linear separating plane is prefered, because it's easier to handle and in general faster at computation. To solve the XOR-Problem a hidden layer has to be added. With negative weights the activation of the respective hidden neuron can be surpressed:



(a) XOR: separation function



(b) The general structure of an artificial neural network

If both input neurons sends 0, no threshold is reached and the network gives 0 as output.
If one input neuron sends a 1 and the other input neuron sends 0 then the threshold of the respective hidden neuron is exceeded and it sends a 1 to the outputneuron. The output neuron's threshold is also exceeded and the network output is 1.
If both input neurons sends 1 the total input of both hidden neurons is 0, because the negative weights surpress the activation. The network output is 0.

The XOR-Problem shows that to separate a data set linear it's necessary to have at least one hidden layer. For an amount of 3 binary variables ($n = 3$) there exist 256 possible logical functions. 104 are linear separable. For $n = 4$ there are 65536 possible functions, only 1882 functions are linear separable. The ratio of linear separable to non linear separable functions tends to zero for $n \to \infty$.

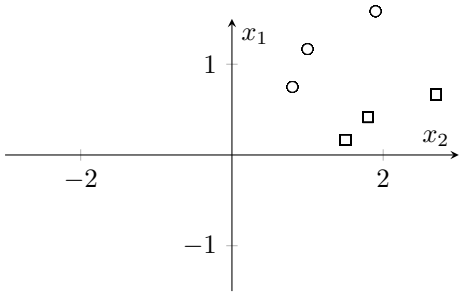
([Roj96] page 61) This shows that with increasing input parameters, hidden layers are necessary to compute this data.

Applications The MLP is used for classification and regression. It can approximate functions for complex problems.

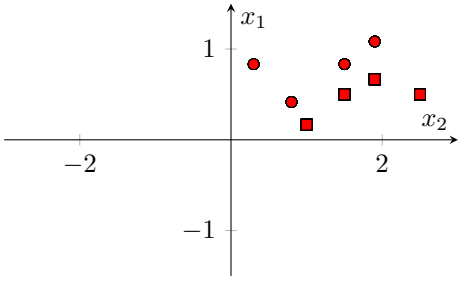
3.2 Learning

A neural network learns by adjusting the connection weights. One way of learning can happen supervised by a given training data set. The network is then trained by a learning algorithm. The network error, means the difference of the wanted output from the actual output, is minimized. After the training cycle the network is tested by a test data set, that must differ from the training data set. One big challenge is to avoid overfitting. Overfitting means the neural net learns the training set exactly, but may fail at the test data set. This behaviour can cause issues if the input data is not the same. A generalization of the net is wanted to cover a variation of the input data on the choosen task.[4] Therefore it's important to have an appropriate number of hidden neurons. If there are too few hidden neurons the network may not fit the target function properly. This is called underfitting. If there are too much neurons, overfitting may happen.

As follows an example of underfitting, overfitting and the wanted generalization of supervised learning at binary classification.
Assume there is a set of labeled data that is devided into a training set and a test set, to control the accuracy auf the network. There are two classes to separate:



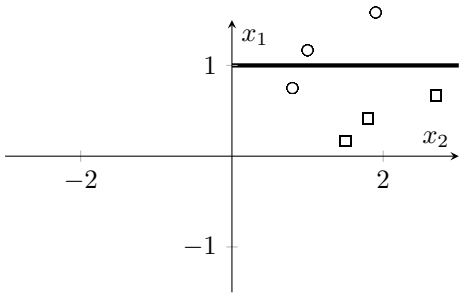
Training dataset



Test dataset

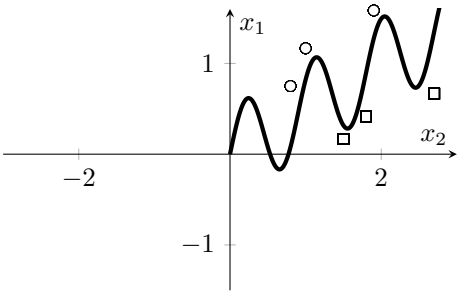
The neural network is trained by a learning algorithm[5]. In the following figures three different separation functions are shown, produced by three different neural networks distinguishing by a different amount of hidden neurons.

[4] A detailed describtion of supervised learning can be found at [Sch15]
[5] This may be the backpropagation algorithm. Described in [Sch15]



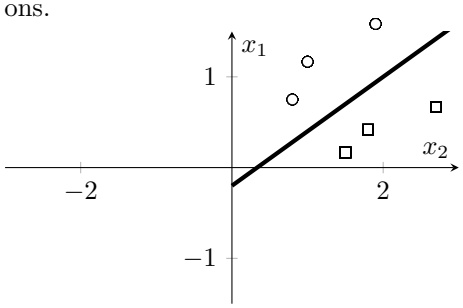
Underfitting

A too simple separating plane causes underfitting. This can happen if there are too few hidden neurons.



Overfitting

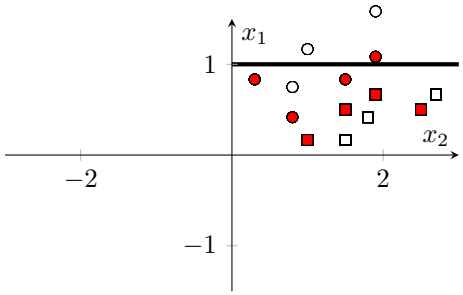
A too complex function causes overfitting. This can happen if there are too much hidden neurons.



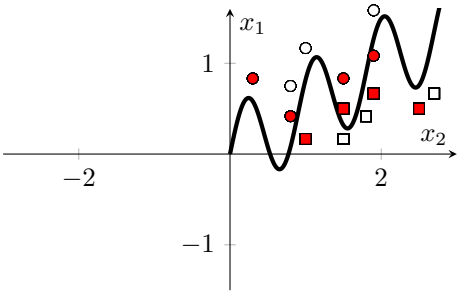
Perfect seperating plane

This plane separates the two classes perfectly because it has the maximal margin between the two classes.

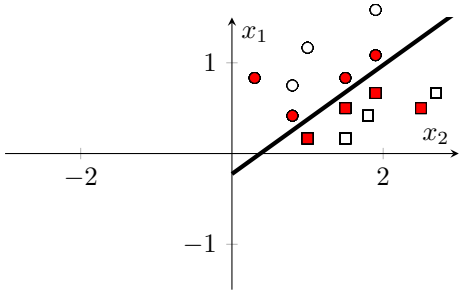
As follows the results of the three functions:



Underfitting



Overfitting



Perfect seperating plane

The figures show that the 1st and 2nd separation functions do not fit all data points. The 3rd

figure shows a good generalization, because the separating plane lies exactly in the middle of the two classes.
Other ways of learning are unsupervised and semi-supervised learning. At unsupervised learning the training data is completely unlabeled. The network has to find patterns in this data. This can happen by different algoritms described in literature. At semi-supervised learning the main part of the data set is unlabeled. But there exist a small amount of labeled data. This data can improve the learning accuracy.

An alternative to these learning methods is the evolutionary optimization of the network. This may belong to the unsupervised learning methods. Populations of nets are cultivated and rated by a fitness function. For these there exist many optimization algorithms.[6]

In this project work the topology and type of the networks are kept and only the weights are changed by the evolution process.

[6]Some of them are described in chapter 4

4 Optimization with evolutionary algorithms

Optimization is widely used, for example in information technology, engineering and economics. Applications are the routing of circuits, the optimal usage of machinery, the traveling-salesman-problem[1] and others. It is often difficult to create a mathematical model for such optimization problems. Over the last decades methods were developed who use principles of the evolution. The idea is to reverse engineer nature's evolution process, because animals seem to solve complex problems easily.[GKK04]
The simplest method is the selection method. In this method a random initialized parameter set will be generated. A parameter set is called a solution[2],because it could possibly solve the task sufficiently. The solution is rated to determine the fitness, which represents the quality of the solution. This solution is randomly changed[3] and rated. This process is repeated for a specified amount of cycles or until the optimization condition is reached. The solution with the best fitness, determined in this process, will be kept. [Kin94]
A set of solutions is called a population. Another method of optimization is the recombination of solutions. Mostly the best solutions are recombined with randomly chosen ones. This and other methods are called evolutionary algorithms. To use an evolutionary algorithm, the problem doesn't have to be in a defined form; therefore it hasn't to be linear or differentiable.
Evolutionary algorithms differ from classical optimization algorithms by their probabilistic and metaheuristic nature. The general idea is not to compute a solution than rather 'cultivate' it. Classical algorithms are specified for some kind of problem. Evolutionary algorithms acting probabilistic (random changes). Metaheuristic means that these algorithms are used in a more general way, means the task has not to be a specific kind of problem. But it's not guaranteed that they find a good solution.

In this project evolutionary algortihms are used to find the best (unsupervised) neural network to control one agent. The idea is to use the amount of objects an agent collects over a defined time as fitness value. The evolutionary algorithm changes the agent's weights of the feed-forward multilayer perceptron.

4.1 The general form of the optimization process

In general the optimzation process is defined by the following components:

- **Search space**
- **Rating function (fitness- or error function)**
- **Optimization algorithm**

[1] The traveling salesman problem describes to find the shortest route between cities
[2] In literature also called chromosome or individual
[3] In literature a random change of a chromosome is also called mutation

The **search space** D is the domain of the optimization function. It defines the general form of a solution. Figure 4.2 and 4.3 are examples of search spaces. Every point in the search space is a possible solution of the optimization task. A solution x is a vector of n components. Each solution gets a real value to determine its quality. F is a function that maps D to the set of real numbers. A solution x with the best quality is wanted, means $F(x)$ is optimal. Depending on the task a minimal or maximal value of $F(x)$ is wanted. As follows the general formula of an optimzation problem:

$$\begin{aligned} F : D &\mapsto R \\ D &\subset R^n \\ x &\in D \end{aligned} \tag{4.1}$$

The amount of parameters n a solution has, has to be defined before an optimization algortihm can be used efficiently. If there are too much possibilities, for example by having too much parameters, the algorithm may never find a satisfying solution. Therefore it's recommended to reduce the search space and define the general form of a parameter set early. Also a transformation of the search space can speed up the optimization process.
Evolutionary algorithms have the advantage that there is no need for deep knowledge of a problem. But at least a basic prior knowledge of the task is necessary.

The **rating function** $F(x)$ gives the quality value for a solution. At maximization problems it's wanted to maximize a fitness value. At minimization problems it's wanted to minimize an error value. A maximization problem can be transformed into a minimization problem and vice versa by inverting the rating function. In some cases constraints have to be considered. Some parameters may have limits they must not exceed. In industry often multiple rating functions are defined, because not only the quality of a solution is important, but also efficiency, cost reduction, etc. These functions $F_i(x)$ can be weighted and a general rating function can be computed by:

$$F(x) = \sum_{i=1}^{n} w_i * F_i(x) \tag{4.2}$$

For maximization tasks $w_i > 0$ and for minimization tasks $w_i < 0$. [GKK04]
It's possible that the rating of a solution includes not only a rating function but also a complex simulation or real testing. An example is the rating of the aerodynamic properties of a car in a wind tunnel.

The **optimization algorithm** starts with a random initialised solution. It changes the solution each step until the optimization condition is reached or the process is aborted. Different algorithms are explained in more detail from section 4.2.
The No-Free-Lunch- theoreme, introduced by David Wolpert and William G. Macready [WM95] states that for the set of all optimization problems there exist no universal algorithm that solves all problems well. An algorithm may solve a subset of these problems well, but may fail at other ones. Therefore it's important to classify the task first and select a proven algorithm.

As follows an optimization process in general:

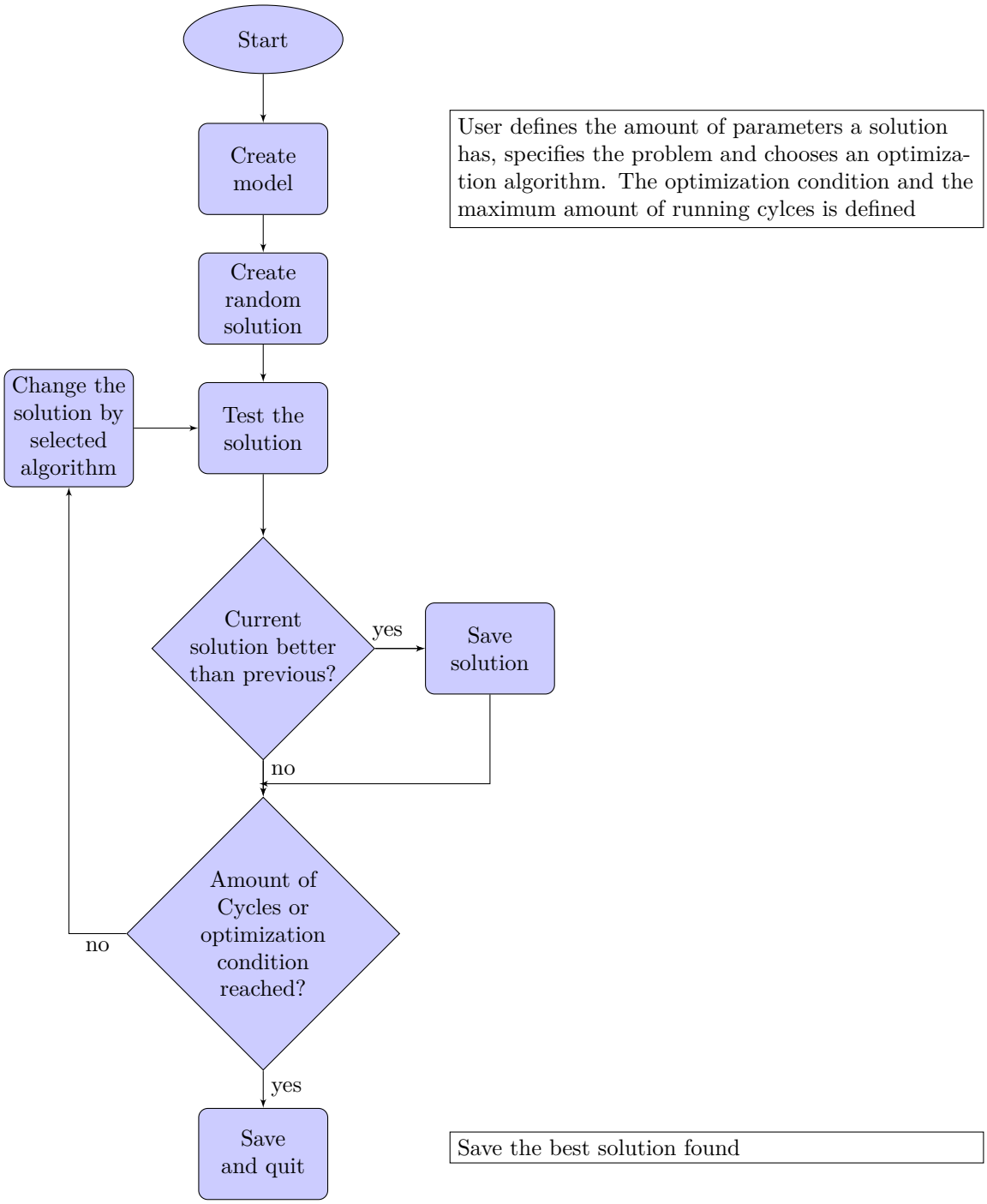


Figure 4.1: The general process of optimization

4.2 The hill climber method

The hillclimber method is a basic method of an evolutionary algorithm, developed by Ingo Rechenberg in 1973.[Rec73] Parameters of a system are randomly changed untill a minimum or maximum of a rating function is reached. The value of the parameter-changes per iteration step are limited.

The main algorithm is:

1. Generate a randomly initialized solution.
2. Change the solution's parameter by limited random delta-values.
3. Test if the solution has a better fitness than the old one and replace it. Otherwise refuse the new solution.
4. If the optimization condition is reached abort, otherwise go to step 2.

The algorihm can be imagined as a blind hillclimber. The hillclimber makes a random step. If he gets higher, he makes the next step. Otherwise he takes back his last step and makes another random step. The main problem is, that this algorithm mostly converges in a local optimum. [Kin94]

Reasons to use this kind of algorithm:

- For many nonlinear problems there are no alternative solution approaches.
- Implementing this method is easy.

As follows two examples of optimization problems:

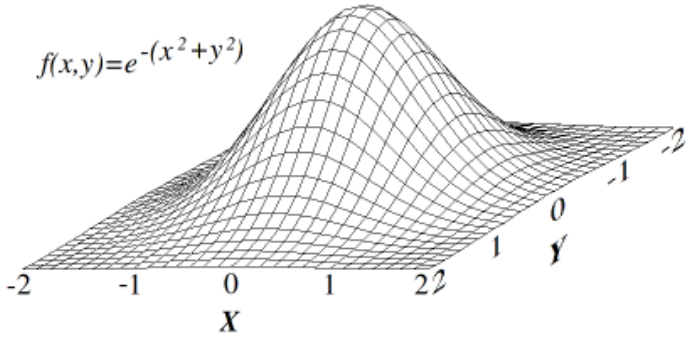


Figure 4.2: A convex function. Ideal for the hillclimbing method [Wik15a].

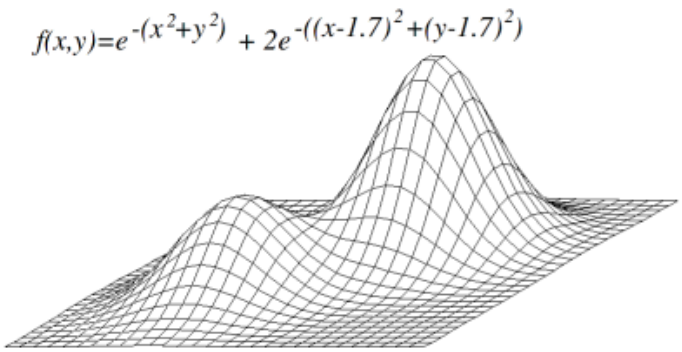


Figure 4.3: A function with two optima. Hillclimbing could end in the worse optimum if it starts at a bad coordinate. [Wik15a].

4.3 Variants of the hill climber method

In the main-method of the hill climber algorithm just a better fitness value is accepted after an iteration step and a worse one is rejected. The problem is that this algorithm will mostly get stuck on a local optimum, instead of finding the global one. To solve this problem temporal low fitness values should be allowed. It should be possible for the algorithm to leave a local optimum to find a better one. To realise this, there have been developed some extended versions of this method:

- A decrease of the fitness is allowed by a specified probability (Simulated annealing)
- A decrease of the fitness is allowed untill a maximal deterioration. (Threshold accepting, deluge method)

4.4 Simulated annealing

Simulated annealing is a probabilistic optimization method. It is inspired by the annealing of fluid materia to a solid aggregate states in metallurgy. While cooling down a material the thermodynamic free energy has to get as minimal as possible to get a clear crystalline structure. Therefore it's suggested to cool down to material slowly for a better probability of getting a clear solid body.
Analog in optimization the algorithm starts with a high temperature T, means reaching a wide area in the search space. That way it's possible to reach more maxima. At the beginning each solution has the same probability to get selected. With each iteration the temperature is reduced and the reached area is getting smaller. Selecting better solutions is now more probalistic. Towards the end the algorithm commutes in an optimum and behaves like the standard hillclimbing algorithm.
Slowly reducing the temperature increases the chance to find the global optimum. Cooling to fast leads to commuting early into a local optimum instead.

First the difference of the fitness between the previous solution x and the new solution x' is computed by:

$$r = F(x') - F(x) \tag{4.3}$$

Then the probability of selecting a solution with lower fitness is computed:

$$p(r) = \frac{1}{1 + exp(-r/T)} \tag{4.4}$$

At the beginning a big T tends to equalize the probability of all solutions. For $T \to \infty$ all solutions have the same chance to get selected. Lowering T gives better solutions priority.
Simulated annealing is suitable for functions with many local optima. [Kin94]

4.5 Crossover

The crossover algorithm represents the idea of cultivating a population of individual solutions. After rating the population, the best individual is chosen and crossed with a random one of the actual population. There are two children created, who can optionally be mutated. This is done until a new generation of individuals is created. The process is repeated until a satisfying solution is found or the process is aborted.[4]

[4] For a detailed description see [Sch15]

5 The Box2D-Physicsengine

Box2D is a 2-dimensional physics engine, created by Erin Catto[Cat11]. It is licensed under the zlib-license, hence it is non commercial and open source. The Box2D library is written platform-independently in C++. It is used mainly for game development. Many portations into other programming languages, like Java and C# exist.
Box2D is used by the project to simulate a realistic physical environment, proving the agents behaviour. The project is based on the testbed GUI of Erin Catto. This program is a testing application for general phyiscs functionality. It uses GLUT respectively freeglut to accelerate graphics. Testbed is also licensed by the zlib-license and a modification is allowed, naming the origin of this software. In this project work it is modified and enhanced to create a simulation environment for virtual agents. The following sections are a summary of [Cat11] and [ifo14].

5.1 Core concepts

5.1.1 Bodies

Bodies are the main objects, affected by physics simulation.
A body contains the following values:

- **2D position vector**
- **Angle**
- **Velocity**
- **Angular velocity**
- **Rotational inertia**
- **Mass**
- **One or more fixtures**

A body contains one or more fixtures, containing the shape and more properties of the body.
If a body is moved, the fixtures are also moved.

Body definitions

To create a body, a body definition has to be created first. It's holding, among other things, the body type, linear- and angular damping values. The body definition can also be used to set the start- position and angle of the body. This can increase start-up performance.

Body types

There are three types of bodies:

- **Static body**
- **Kinematic body**
- **Dynamic body**

A **static body** is excluded from the simulation. It also doesn't move. Static bodies are used as world-borders for example. Static bodies don't collide with other static- or kinematic bodies, but

dynamic bodies can collide with them.

A **kinematic body** can move by setting its velocity, but isn't influenced by forces. Kinematic bodies don't collide with other static- or kinematic bodies, but dynamic bodies can collide with them.

The **dynamic body** is the commonly used bodytype in Box2D-simulations. It is fully simulated and can be moved by the user or by forces, as its usual way. Also it can collide with any other object.

Damping, friction and gravity scale

Damping reduces the velocity (linear and angular) of the body. **Friction** reduces the velocity of a body at collision with other bodies.
The damping parameter can be zero for no damping and infinite for full damping. In general a value between 0 and 0.1 is used.
The **gravity scale** is a factor that determines how much the worlds gravity influences the body. 0 determines no influence. The standard value is 1.

Activation and Sleeping

A body can be set asleep. This is used to save cpu-time as for sleeping bodies no computation of physics is necessary. The body will be woken up, if another body collides with it. A body can be completely excluded from simulation by setting it inactive instead. In contrast to the sleeping mode it will not be woken up.

5.1.2 Fixtures

Fixtures give bodies a shape and additional properties. A body can have multiple fixtures, with their positions relativly to the body's origin.
The following properties are part of a fixture:

- **A shape**
- **Broad-phase proxies**
- **Density**
- **Friction**
- **Restitution**
- **Collision filtering flags**
- **Pointer to the affiliated body**
- **User data**
- **Sensor flag**

A single **shape** is contained by the fixture. The shape can be a rectangular, a polygon or a circle. Shapes aren't linked to the body directly, because they may be used independly of the simulation. Therefore the fixture is interposed between body and shape. In Box2D the maximum amount of vertices is defaultly set to eight. A vertex defines the position-vector of a body's edge. A polygon therefore can be created with a minimum of three position-vectors and maximally with eight.

Broad-phase proxies are used internally by Box2D to accelerate collision detection. Therefore a dynamic tree containing the collision-pairs in form of AABB's[1] are used. In the most cases it isn't necessary for the user to use proxies.

The **densitiy** is used for computation of the body's mass.

[1] AABB: Axis aligned bounding box. AABBs are used for fast computing of collision detection

The **friction** of two fixtures is multiplied at collision.

Restitution makes a fixture elastic. Usually this value is set between 0 und 1.

Collision filtering flags can be used to just allow specified groups of fixtures to collide.

A **pointer to the affiliated body** is stored in the fixture.

User data is used as a 'hook' to identify a fixture by an id or add it some self-defined properties. User data is a 'void'-pointer. Therefore it can contain any datatype.

The **sensor flag** defines that a fixture doesn't interact physically with other fixtures, except by detecting collisions.

A fixture is created by initializing a fixture definition and passing its address to the body:

```
1 b2FixtureDef fixtureDef;
  fixtureDef.shape = &myShape;
  fixtureDef.density = 1.0f;
  b2Fixture* myFixture = myBody->CreateFixture(&fixtureDef);
```

Listing 5.1: Creation of a fixture [Cat11]

5.1.3 Joints

Joints are used to connect two or more bodies. For more detailed information read the Box2D manual Chapter 8.[Cat11]

5.1.4 World

A world is the simulation environment. It contains all bodies and joints and manages all aspects for simulation. It is the main entity.
The world represents a factory pattern, because it can create and destroy bodies and joints.

Simulation parameters

The physics simulation is controlled by three parameters:

- **timeStep** (float32) (Recommended: 1.0f /60.0f respectively 60Hz)
- **velocityIterations** (int32) (Recommended: 8)
- **positionIterations** (int32) (Recommended: 3)

These parameters are used by the step-method of the world in which one single simulation step is processed.

The **timeStep** parameter defines the time-resolution on which the simulation will update the environment. A lower timestep improves the quality of the simulation, but on costs of performance. A higher timestep increases the performance, but the simulation can be inaccurate by too low values. The default value is 60Hz. The author of Box2D suggests not to use values lower than 30Hz. As set, the timestep should not be changed as it could cause unwanted behaviour. For a faster simulation

the step-method should called multiple times instead.

Velocityiterations is the resolution of computing impulses to move bodies precisly. The default value is 8.

Positioniterations defines the resolution of overlapping bodies. This is important for a correct collision detection of bodies. The default value is 3.

[Cat11] [ifo14]

5.2 Collision detection

In Box2D collisions are computed between two fixtures. A contact object, containing information about collision coordinates, normals and impulses among ohter things, is created if a collision happens. A contact object can be used to manage a collision.
An AABB is an 'axis aligned bounding box'. It completly contains a shape. It is used for fast performance of computing geometrical operations.[Tou83]

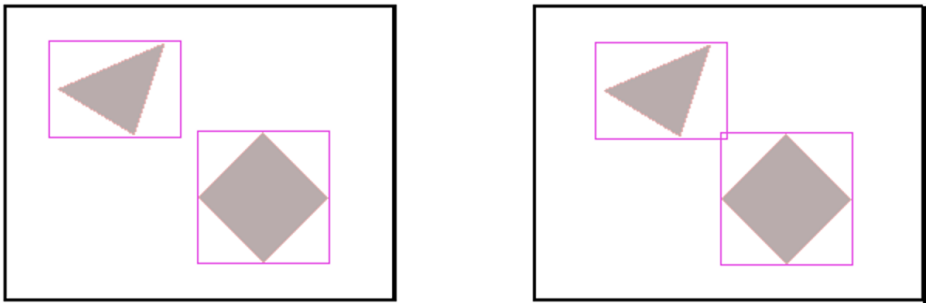


Figure 5.1: Two AABBs are overlapping [ifo14]

If two AABBs are overlapping, a contact between the two fixtures is created, but the isTouching()-method returns 'False', because the shapes aren't overlapping.
Sometimes, fast moving bodies miss collisions and incorectly pass through each other. This effect is called tunneling. Therefore it is possible to define the body as a 'bullet' for high resolution at collisions. The physics engine then uses a method called continous collision detection (CCD)[Ber04] for this body. Usually this method is just used for high speed and static bodies, because it costs cpu performance. A body can be defined as a bullet by its definition instance:

```
1 bodyDef.bullet = true;
```

Listing 5.2: Define fixture as bullet before creation

Or after the creation of a body by the following method:

```
1 body->SetBullet(true);
```

Listing 5.3: Define fixture as bullet after creation

The following figure shows the processing of a collision:

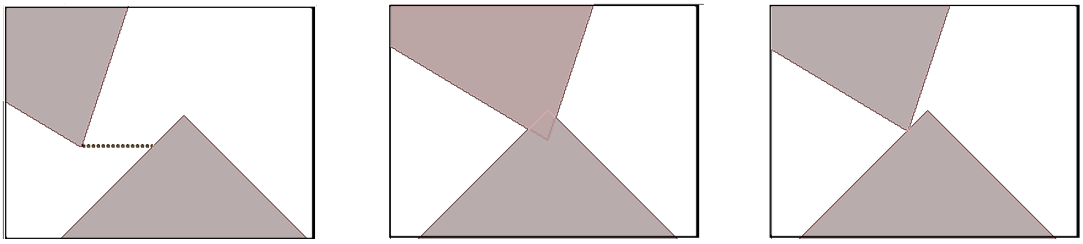


Figure 5.2: Two fixtures are overlapping [ifo14]

The first picture shows the beginning of a contact of two bodies. A contact object is created, because the AABBs of the two bodies are overlapping.
The second picture shows non-bullet bodies tunneling each other. Incorrectly the collision is missed. This can happen if the bodies move too fast.
The third picture shows the correct result of a collision as it should be. For fast moving bodies this can be acquired by defining a body as bullet.

5.2.1 Contact listener

The contact listener executes callback-methods if the events 'Begin contact' or 'End contact' are triggered. On a collision a contact object is created containing information about the fixtures collision. Contacts are used to manage collisions between two fixtures.
To access the contact and the two colliding fixture a contact listener should be used.
The collision-computation processes four phases:

- **Begin contact**
- **Presolve**
- **Postsolve**
- **End contact**

```
/b2ContactListener
// Called when two fixtures begin to touch
virtual void BeginContact(b2Contact* contact);
// Called when two fixtures cease to touch
virtual void EndContact(b2Contact* contact);
// Called after collision is detected
virtual void PreSolve(b2Contact* contact, const b2Manifold* oldManifold);
// Called after computing the results of the collision
virtual void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse);
```

Listing 5.4: Contact listener methods

Begin contact

This callback is executed when two fixtures begin touching.

End contact

The end contact callback is called when the two colliding fixtures cease touching.

Presolve

The presolve callback is executed after detection of a collision, but before computing the resulting impulses. It contains a pointer to the contact.

Postsolve

The postsolve callback is executed after computation and applying the contactimpulses of the collision. It contains a pointer to the contact and also the computed impulse data.

5.2.2 Collision filtering

The default behaviour is that all fixtures can collide with each other. If a user wants to specify which groups of fixtures can collide and which pass through each other, he can use category- and mask bits to manage 16 groups of fixtures. If more precise behaviour is necessary group indizes can be used or own rules can be defined by overriding the 'ShouldCollide'-method.

Category- and maskbits

These bits are 16-Bit wide and therefore 16 groups are supported. Category bits declare on which groups a fixture belongs. A fixture has to belong to minimal one and maximal 16 groups. The mask bit defines the group of fixtures this fixture can collide with. The default values are 0x0001 for category bits and 0xFFFF for mask bits, meaning every fixture can collide with each other.

Group indizes

Group indizes can be used to get a more specified behaviour as only with category- and maskbits. The standard value of the group index for a fixture is zero, means group index isn't used, instead the category- and maskbits are used. The value otherwise can be positive or negative.
For a value nonzero the following rules are applied at collision:

- if both group indizes are different, use the category/mask rules as described above
- if both group indizes are equal and positive, collide
- if both group indizes are equal and negative, don't collide

Contact filter

For an userdefined specification of collision behaviour the following virtual method can be overwritten:

```
1 bool b2ContactFilter::ShouldCollide(b2Fixture* fixtureA, b2Fixture* fixtureB);
```

Listing 5.5: The contact filter method

5.3 DebugDraw

The debugdraw class is used as a debugging feature to display the physics simulation of Box2D. It uses OpenGL. Debugdraw can draw the shapes, contact points, contact normals, the AABBs, etc. As the debugdraw offers all wanted basic shapes it is used by the simulator.

To use the debugdraw first the debugdraw object has to be instanciated and registered to the world by:

```
m_world->setDebugDraw(&m_debugDraw);
```

Listing 5.6: Registering a debugdraw instance to the world instance

Setting the flags to determine what has to be drawn:

```
uint32 flags = 0;

flags += b2Draw::e_shapeBit; //Enable drawing shapes
flags += b2Draw::e_jointBit; //Enable drawing joints
flags += b2Draw::e_aabbBit; //Enable drawing AABBs
flags += b2Draw::e_centerOfMassBit; //Enable drawing center of masses

m_debugDraw.SetFlags(flags);
```

Listing 5.7: Initializing DebugDraw

After each calling of the step-method the following method has to be called:

```
m_world->DrawDebugData();
```

Listing 5.8: Using debugdraw after each step

This draws every shape in the body list of the world.

5.4 The program flow of a Box2D project

As follows, the general program flow of a Box2D project:

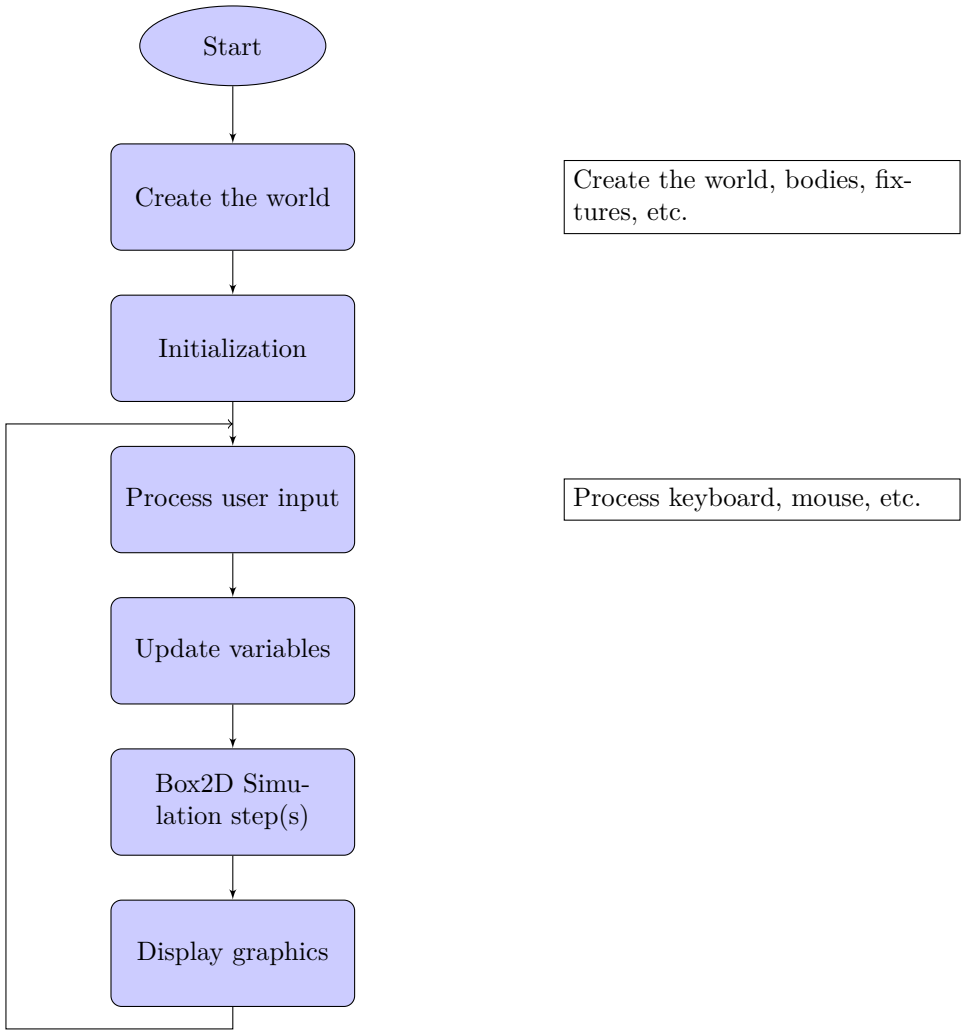


Figure 5.3: Flowchart of a Box2D project

At first the Box2d world and affiliated objects have to be created and initialized. It follows a cycle of processing the user inputs, updating the variables and simulating the physics. The objects can optionally be displayed.
The physics simulation can be speeded up by calling the simulation step multiple times before continuing the program flow. Also avoiding the display of graphics speeds up the program.

6 OpenGL

OpenGL stands for 'Open Graphics Library'[SA09]. It is widely used for developing graphics applications and was published in 1992 by Silicon Graphics. Since 2006 it is maintained by the Khronos Group. It is licensed under an open source license. The logo and the trademark are licensed under a trademark-license. OpenGL is platform-independent.
It is used by the project to accelerrate the simulator graphics.
The Box2D-Testbed uses GLUT respectively Freeglut and GLUI. Thererefore theses libraries are also used in this project.

6.1 The OpenGL Utility Toolkit (GLUT)

The OpenGL Utility Toolkit (GLUT) was developed by Mark Kilgard[Kil96]. It is used to extend OpenGL-Applications by window-management. GLUT is used for small- and medium sized OpenGL-applications.

6.1.1 Freeglut

Freeglut is developed by Pawel W. Olszta[Ols15] and is an alternative to GLUT. It is also a substitute for GLUT to use at Microsoft Windows. Freeglut is still maintained. Freeglut is used, because the originally GLUT-library is deprecated and not maintained anymore.

6.1.2 Program structure

In this section the basic structure of a freeglut program is explained. The following code shows the initialization and creation of a window:

```
1  #include "glut.h"

   int mainWindow;  //Window handle

5  int main(int argc, char* argv[]){
     // Glut initialitation
     glutInit(&argc, argv);
     glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
     glutInitWindowPosition(0, 0);
10   glutInitWindowSize(1024, 768);
     mainWindow = glutCreateWindow("Title");
     //Register function callbacks
     glutDisplayFunc(Display);
     glutKeyboardFunc(Keyhandler);
15   glutReshapeFunc(Resizehandler);
     //Here optionally a GLUI-window can be included, see Chapter 6.3
     //Enter glut main loop, usually no return from this function
     glutMainLoop();
     return 0;}
```

Listing 6.1: Initialization of a GLUT program

In the first lines glut is initialised, a window with specified position, size and a title is created. The variable 'mainWindow' is of type int. It stores the adress of the window-structure.

In the second part of the codesnippet the handler functions are set. The handler functions have to be static functions. At the end the glut main loop is entered. Usually this function never returns. In Freeglut there exist functions to leave and re-enter the main loop.
The following Codesnippet shows the drawing of a simple triangle:

```
static void Display()
{
  //Set the color to white
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
  //Clear the Scene with selected color
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  //Switch to Modelview Matrix
  glMatrixMode(GL_MODELVIEW);
  //Replace the current matrix by the identity matrix
  glLoadIdentity();
  //Draw a blue triangle into the buffer
  glColor3f(0.0f, 0.0f, 1.0f);
  glBegin(GL_TRIANGLES);
  glVertex3f(-0.5f, 0.5f, -5.0f);
  glVertex3f(-1.0f, 1.5f, -5.0f);
  glVertex3f(-1.5f, 0.5f, -5.0f);
  glEnd();
  //Finally draw the scene on screen
  glutSwapBuffers();
}
```

Listing 6.2: Drawing some graphics

The first line sets the draw-color to white and with no transparency. The color is formatted in RGBA. The first three parameters represent the colors red, green and blue. The last parameter is alpha, the opacity of the figure. 0.0f means no quantity of the according parameter and 1.0f as the full quantity. This is used by the second line to clear the scene. Then the viewmode is set, the position of the camera is loaded. With 'glBegin' the drawing of an object is started. In this example a triangle is drawn. Therefore the three vertices of the triangle has to OpenGl. This is done by. Each object has to finished with the 'glEnd'-command. At the end the buffer is swapped and the drawn scene is displayed at the screen.

```
static void Resize(int w, int h)
{
//Set the viewport of the camera
glViewport(0, 0, w, h);
//Switch to the camera perspective matrix
glMatrixMode(GL_PROJECTION);
//Reset the camera matrix
glLoadIdentity();
//Set camera angle, width-to-height ratio, near z and far z clipping coordinate
gluPerspective(45.0f, (double)w / (double)h, 1.0f, 200.0f);
}
```

Listing 6.3: Resize camera perspective on window resize

The first line sets the viewports position and size. The following operations should than applied on the projection matrix. Therefore the matrix is resetted to the identity matrix and the clipping of the world the camera should record is set. The near z-coordinate sets the 2D-viewing plane of the camera. The 3D-world will be projected on this 2D-plane. This is the picture that the user will finally see. The far-z perspective defines how deep the sight of the camera into the world is. Objects out of range aren't considered in this case.

6.2 The OpenGL User Interface Library(GLUI)

The OpenGL User Inteface Library (GLUI) is a GLUT-based user interface library developed by Paul Rademacher[Rad06]. It is written in C++ and usable for multiple platforms.
GLUI is used to implement GUI controls, like buttons, labels, listboxes, etc.
The following figure shows some components of GLUI:

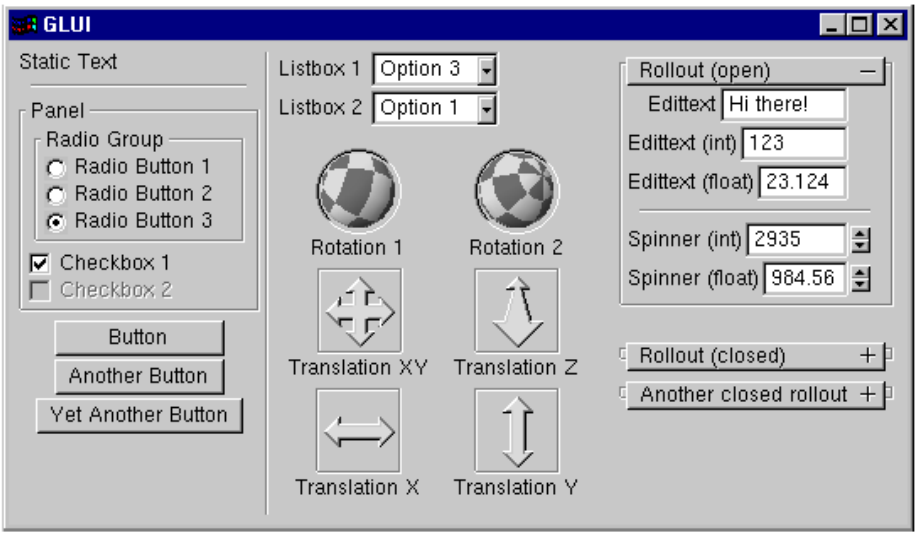


Figure 6.1: Some GLUI Components [Rad06]

6.3 A GLUI program

In this section is described how a GLUI-window is set up. The following code is included after the creation of a GLUT-window and before entering the glut main loop (see listing 6.1) :

```
#include "glui.h"
//GLUI window instance
GLUI *glui;

int main(int argc, char* argv[])
{
 ...
 //after creation of glut-window
 glui = GLUI_Master.create_glui("GLUI", 0);
 //Add some components
 glui->add_statictext( "Simple GLUI Example" );
 glui->add_separator();
 glui->add_checkbox( "Wireframe", &wireframe, 1, control_cb );
 GLUI_Spinner *segment_spinner =
 glui->add_spinner( "Segments:",GLUI_SPINNER_INT, &segments );
 segment_spinner->set_int_limits( 3, 60, GLUI_LIMIT_WRAP );
 GLUI_EditText *edittext =
 glui->add_edittext( "Text:", GLUI_EDITTEXT_TEXT, text );
 //Set the main gfx window
 glui->set_main_gfx_window(main_window);
 //Register the Idle callback with GLUI (instead of with GLUT)
 GLUI_Master.set_glutIdleFunc(GlutIdle);
}
```

Listing 6.4: Creating a GLUI-window

First the glui-instance is initialized. In the following lines gui-components are added. Some components need a variable to point on. This is necessary to store input values from the user. The components can be dimensionized, e.g. the spinners minimum and maximum value is set. At the end the corrosponding glut window has to be registered in the glui-window. Callbacks are now defined. This has to be static functions, like in GLUT.
'GLUI_Master' is a global GLUI-object used to register callbacks, as glui-windows use for example the idle-callback extensively to control components status.

7 Implementation

7.1 Concept

The main concept is to cultivate a population of agents by different evolutionary algorithms to find a satisfying solution. Each agent has a sensor, representing an own coordinate system. The agents have to collect as much objects as possible within a given time. Therefore it has a feed forward multilayer perceptron. The objects and the agents are spawned randomly every generation. After collecting an object a new one is spawned instantly on a random position.

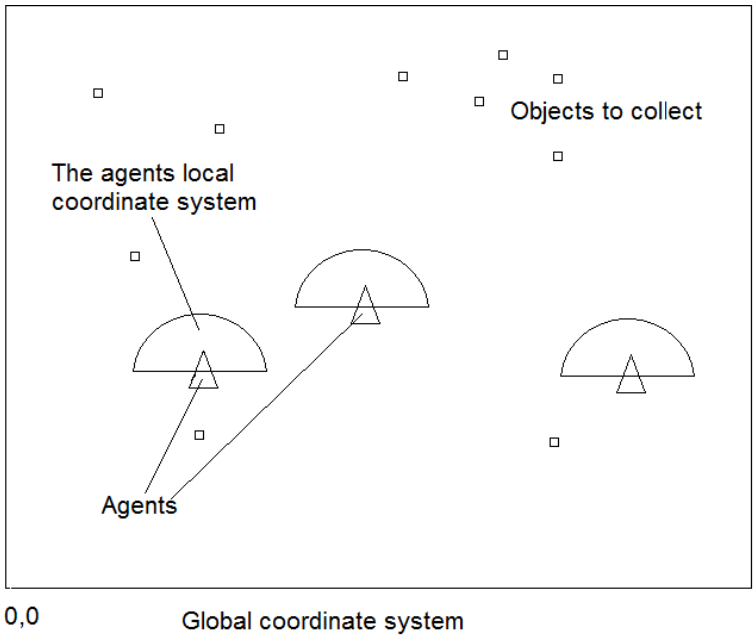


Figure 7.1: A sketch of the main concept

7.2 The task environment

By the definitions of a task environment, explained in chapter 2.3, the simulation environment can be categorised as:

• **partially observeable**	• **episodic**
• **single agent based or mulitagend based and competive**	• **semidynamic**
	• **continous**
• **deterministic**	• **first unknown then known**

The simulation environment is **partially observeable** because the agents perception is limited by the sensors range.

It can be both, **single agent based** or **mulitagend based and competive**, depending on the population size. If there is just one agent to train it's single agent based. With more agents the simulation is multiagent based and competive, because the best agent will 'survive' the evolution process.

The environment is **deterministic**. The start positions are chosen randomly, but the process of collecting the objects is from this point on deterministic as each step to get objects can be desribed step by step.

It's **episodic** as the actual state does not change the general method of collecting further objects. The task environment is **semidynamic**, even if there are multiple agents the environment is considered by a static behaviour for simplicity. Also there is a time limit for collecting objects.

Continuity is given, because an agent handles with real values, like the object's position, the agent's velocity and rotation. The time for one generation is also continous.

The **knowledge** about the environment is in the first case **unknown** as the neural net is not trained, respectively evolved. On a successful evolution process the agent's knowledge turns from unknown to **known**, because the agent gets a model of the environment, represented by the correct weights of the neural net. The knowledge of the environment has nothing to do with the observeability. The knowledge just categorieses the agent's ability to understand the specific rules of the environment. The environment task doens't change. As soon as the agent understands the principles of the environment, it remains knowing.

7.3 The agent's logic

In this project work a simple reflex agent is realized. The agents processing logic is realized by a multilayer perceptron. That means the agent has no memory and just processes the actual perception. This categorises the agent to a simple reflex agent.

7.3.1 The neural network

The neural network outputs are real valued and continous, because factors of velocity and rotation speed are wanted. The evolution process of the network is just changing the weights and not the topology. The two possible outputs are divided into two independent ANN, each having 2 input-, 8 hidden- and 1 output neuron. Additionally the input- and the hidden layer each get 1 bias neuron, because if no object is in range the agent should still drive around the world until it detects one. The activation function of each neuron is a tangens hyperbolicus with no threshold. This scales the output as described in this section.
The input range of both ANNs is:

$$x \in [-20.0\ ;\ 20.0] \tag{7.1}$$
$$y \in [0.0\ ;\ 20.0]$$

In the case that no object is in range, the input vector is $\binom{0}{0}$.

The output is scaled. The output range of the velocity ANN is:

$$v \in [0.0\ ;\ 1.0] \tag{7.2}$$

The output is multiplied with the agent's maximum velocity.

And for the rotation ANN:

$$a \in [-1.0\ ;\ 1.0] \tag{7.3}$$

This output is multiplied with the agent's maximum rotation speed.

7.3.2 Evolutionary optimization

Each neuron of a layer, including bias neurons, is connected with all neurons of the next layer. It follows that the evolutionary algorithm gets the following amount of parameters it has to optimize:

$$3 * 8 + 9 * 1 = 33\ weights \tag{7.4}$$

This causes two 33-dimensional search spaces, one for finding the correct velocity and one to find the correct rotation speed. The rating function is a **fitness function**, because a maximum amount of collected objects is wanted.

7.4 The agent's actuator

The agent can rotate left or right. The agent can drive just forwards. It can determine the factor of velocity and rotation until the maximum, defined in the Box2D environment.
The physiscs engine damps the agent's movement and its rotation. So it will stop after a while if it use its actuators.

7.5 The agent's sensor

The agents sensor is a semicircle detecting just collectable objects in a given radius.[1] The sensor returns a vector containing the X- and Y-coordinate relative to the agents actual position. For simplicity just the vector of the nearest object is recognized. The X- and Y values have to be transformed. This is described as follows:

[1] The semicircle sensor shape is based on [ifo14]

Coordinate system transformation

Before the computation of the outputvector of the neural network can be done the local coordinate system of the agent has to be determined. To get the relative position of an object the global coordinate system has to be transformed into the agents local coordinate system, considering the agents actual angle.
Therefore at first a translation is made, then the rotation of the agent is applied.
As follows a figure of the translation and rotation of the coordinate system:

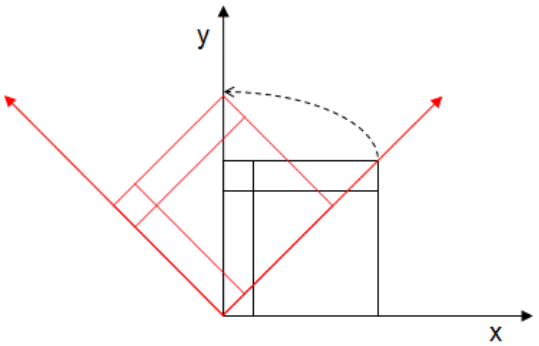


Figure 7.2: Rotated coordinate system [Wik15b]

Vector $\vec{P_a}$ holds the position of the agent and $\vec{P_o}$ the position of the object in the global coordinate system.
At first the coordinate of the object relative to the agent has to be determined by:

$$\vec{P} = \vec{P_o} - \vec{P_a} \tag{7.5}$$

Then the actual angle α of the agent has to be considered. Therefore the final position vector $\vec{P'}$ and the corresponding components x' and y' has to be computed by rotating the vector $\vec{P}$:
This is described by the following transformation matrix: [Wid09]

$$\vec{P'} = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} * \vec{P} \tag{7.6}$$

So the components of vector $\vec{P'}$ are computed as follows:

$$\begin{aligned} x' &= x * \cos\alpha + y * \sin\alpha \\ y' &= -x * \sin\alpha + y * \cos\alpha \end{aligned} \tag{7.7}$$

7.6 Software concept

7.6.1 Use-Cases

As follows the use cases of the simulation software:

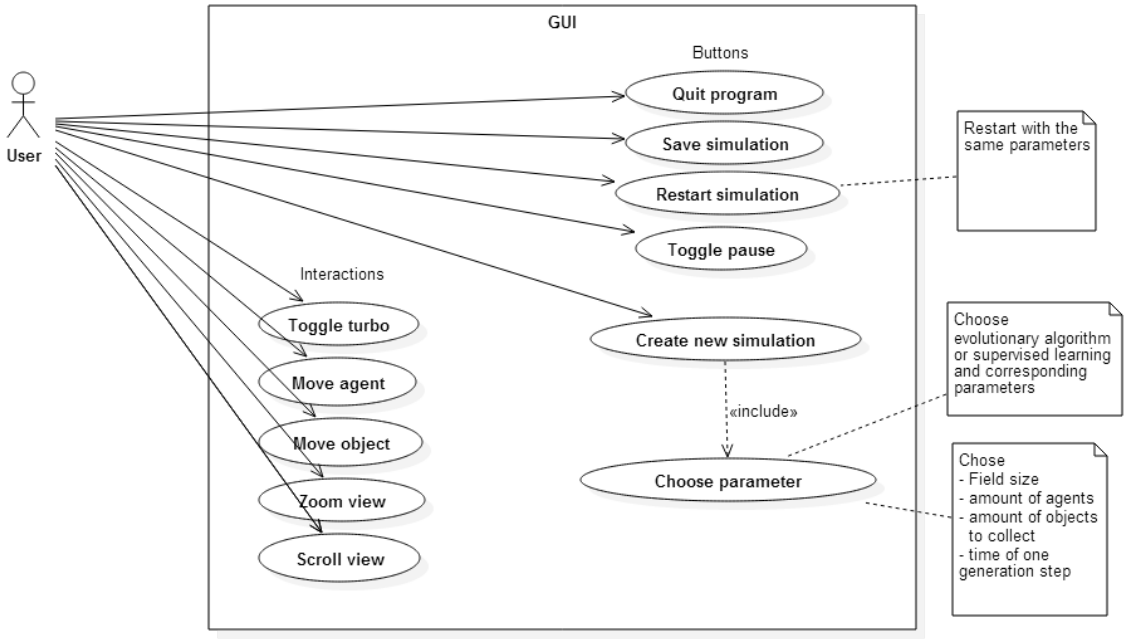


Figure 7.3: GUI: Use Cases

The user is able to create and start, pause, resume, restart the simulation. He is able to save the actual simulation state and quit the simulator.
While simulating the user can interact with the simulation environment. The simulation can be speeded up by toggling a turbo mode. The computation power then is used for simulation only. The view can be scrolled and zoomed.
The user can also move and hold an agent or an object by mouse. This can be helpful to make some experiments and to see how a single agent reacts.

7.6.2 Classdiagram

The main structure of the simulator contains a graphical user interface (GUI), a simulation environment, a class of the agent, a class for the evolutionary optimization logic and a neural network framework[2]:

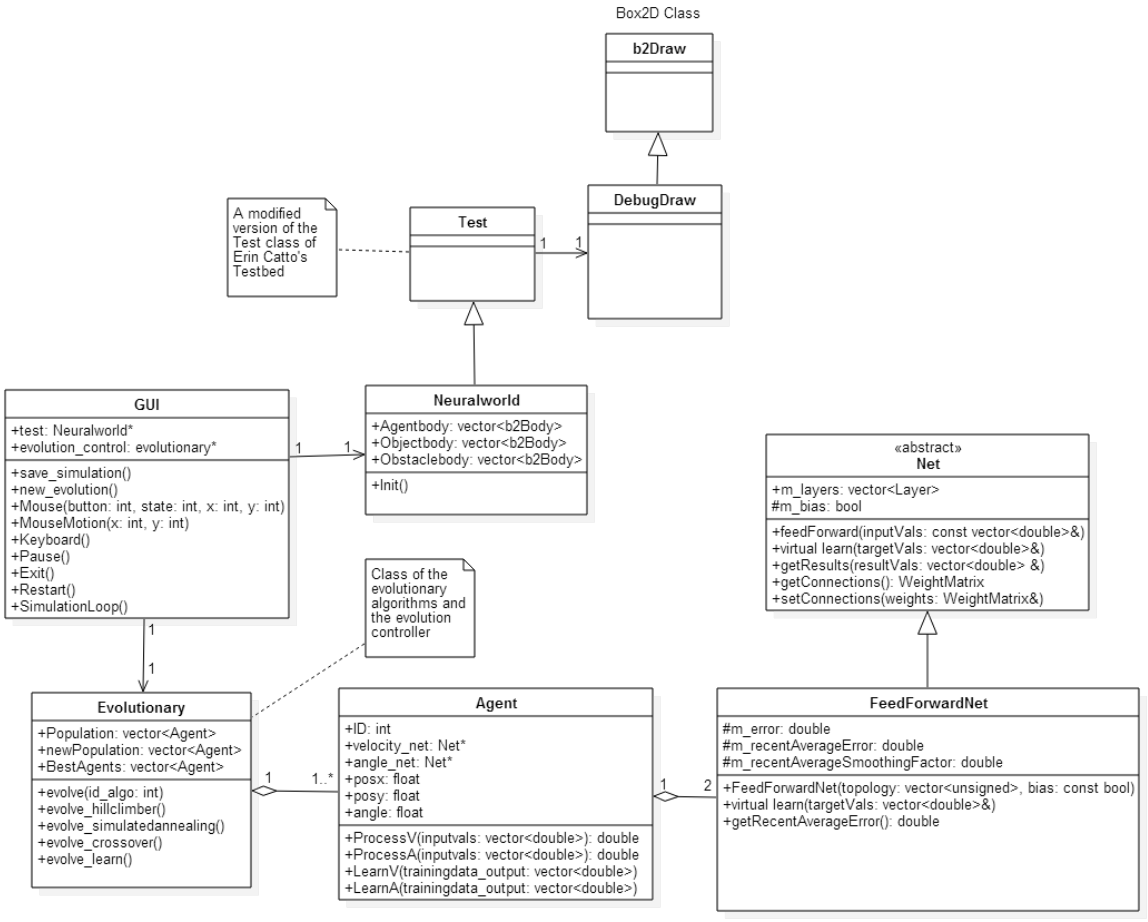


Figure 7.4: Class diagram of the simulator

GUI
The Class GUI is the main class in the simulation software. It contains one evolution controller (Evolutionary) and one simulation environment (Neuralworld). The GUI manages the user's keyboard- and mouse input.

Neuralworld
Neuralworld is the Box2D simulation environment. It contains a world and bodies for each agent, object and the world borders. The world borders are realized as static bodies. Neuralworld inherits functionalites, like interaction by mouse and drawing shapes and status messages from 'Test'.

[2]The ANN framework is described in [Sch15]

Evolutionary
Evolutionary is the evolution controller class. It contains methods to apply the evolutionary algorithms. Also it contains a population of agents, realized as a vector. At least one agent has to be in the population. In the vector 'best_agents' the best agents found during the optimization process are saved. In the actual simulator version just the best agent is saved. To use the crossover algorithm a second vector of agents was added.

Agent
The agent's class contains an id, two feed forward multilayer perceptrons for velocity and rotation and member variables of the position and angle. The methods 'ProcessV' and 'ProcessA' return the computation results of the neural net. 'V' stands for velocity and 'A' for angle. 'LearnV' and 'LearnA' are methods to train the corresponding network by the given dataset, if supervised learning was chosen.

Net
'Net' is the main class of the neural network by framework[Sch15]. It is abstract and has a vector of layers, holding input-, hidden- and the output layer. The method 'FeedForward' let the network compute its output. The results is stored in a vector and can be accessed by calling the 'getResults'-method.

FeedForwardNet
'FeedForwardNet' is also part of the neural network framework. It is the network type used in this project work. It inherits from the abstract class 'Net'. An agent posses two feed forward networks.

7.7 The graphical user interface

The graphical user interface (GUI) was created with GLUI[3] As follows a figure of the GUI:

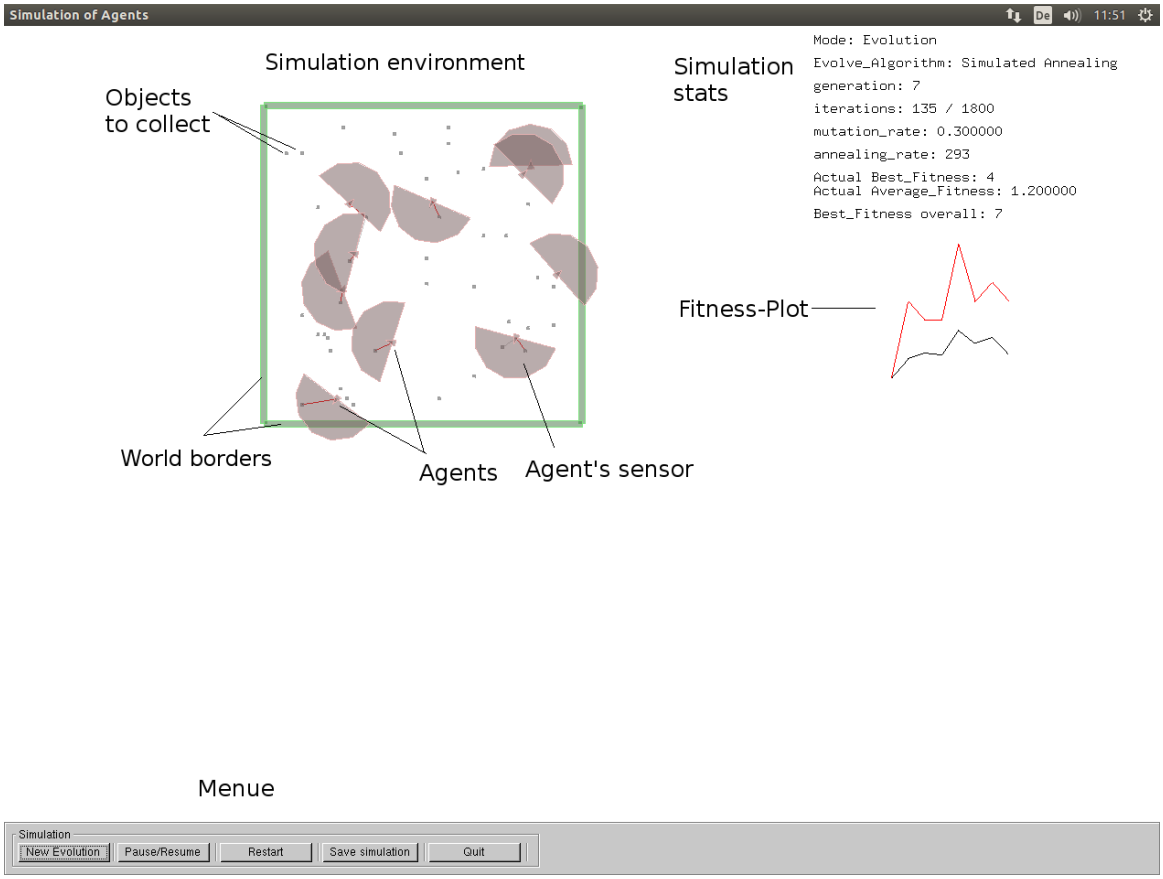


Figure 7.5: The simulation GUI

The GUI is divided into three parts. At the bottom is a menue to create, save and pause the simulation.
At the middle of the window the simulation is visualized. The red line shows the distance to the nearest object. The grey lines are the distances to other objects in the sensors range.
And at the right side there are stats and two plots, representing the best- and the average fitness values of each generation.
The time to measure the fitness of one generation can be chosen.
As follows a figure of the 'new evolution' menue:

[3]see chapter 6.2

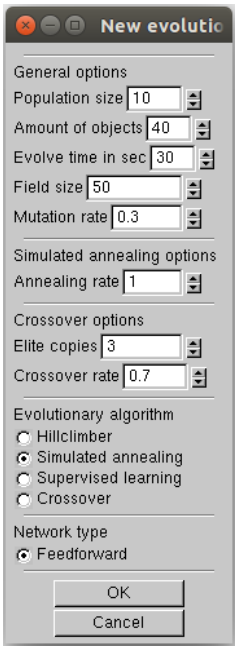


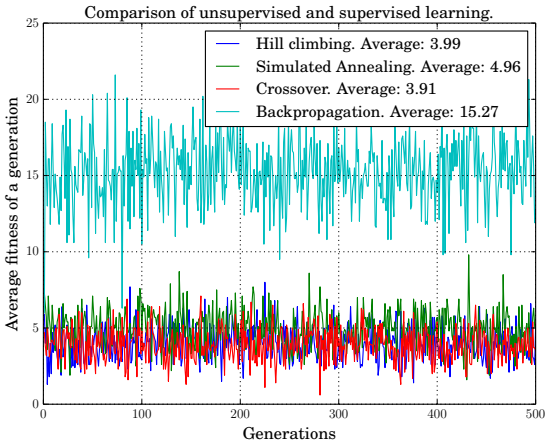
Figure 7.6: Create a new evolution process

The user can choose an evolutionary algortihm or the supervised learning method.[4] He can also change parameters like the field size, the amount of agents and objects and algorithm parameters. There are given standard values and limits for each parameter.

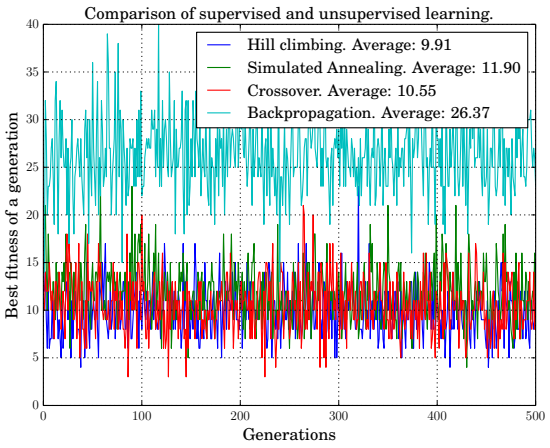
[4]The crossover- and the supervised training algorithm were developed by [Sch15]

8 Results of the simulation

The result shows that evolutionary algorithms doesn't solve the task sufficiently. Some agents just spinning around and the collection of objects is just a lucky shot. But while optimizing, the process evolved some unexpected agents. Instead of directly driving to an object, these agents drive straightly through the whole environment. They rotate just sometimes and then continue driving straight ahead. They seem to exploit the fact that objects are spawned randomly in an uniform distribution. Nonetheless a supervised training, means to teach the agent to directly drive from object to object shows much better results.[Sch15]
As follows the average- and the best fitnesses of an evolution process:



(a) Average fitnesses of the evolution [Sch15]



(b) Best fitnesses of the evolution [Sch15]

9 Conclusion

The evolutionary algorithms showed that it's possible to find solutions someone would not expect in the first case. As example an agent who is driving 'blindly' across the environment and collecting a huge amount of objects. The evolutionary algorithms didn't find a satisfying solution. The main cause could be a too large amount of hidden neurons that causes overfitting. Further tests with less hidden neurons have to be made. The project work shows that too much parameters can make an evolutionary algorithm insufficient. Therefore a good prior knowledge can be helpful in modeling the optimization process.
Neural networks are a probabilistic mehtod to solve problems. To use these networks effectively some rules of thumb have to be considered. Too much hidden neurons can cause overfitting and an unnecessary effort. Also the usage of multiple hidden layers doesn't improve the accuracy in the most cases.
A separation of the task into multiple subtasks and a distribution to multiple independent neural networks can improve the efficiency and give a clearer view of the task. To simplify the optimization process the transformation and preprocessing of the input data may be helpful to reduce the search space and let the network adjust it weights just for the essential task.
There are many methods to use neural networks. A successful method can be a deep learning neural network.[YL15] In contrast to a classical feed forward network this means splitting a task into multiple sub tasks and having multiple hidden layers. Each layer solves a subtask and passes the result to the next layer, instead of letting all hidden layers solve the main task. This abstraction can improve the performance significantly.

List of Figures

Listings

Bibliography

[Ber04] Bergen, Gino van d.: Ray Casting against General Convex Objects with Application to Continuous Collision Detection. (2004), June

[Cat11] Catto, Erin: Box2D v2.2.0 User Manual. (2011). http://www.box2d.org/manual.html

[GKK04] Gerdes, Ingrid ; Klawonn, Frank ; Kruse, Rudolf: *Evolutionäre Algorithmen.* 1. Wiesbaden : Vieweg Verlag, 2004

[ifo14] http://www.iforce2d.net/b2dtut/

[Kil96] Kilgard, Mark J., Silicon Graphics, Inc., Specification, 1996. https://www.opengl.org/documentation/specs/glut/spec3/spec3.html

[Kin94] Kinnebrock, Werner: *Optimierung mit genetischen und selektiven Algorithmen.* München : Oldenbourg Verlag GmbH, 1994

[Kla10] Klass, Jan: *Neuronale Netze der 3. Generation und Anwendungsgebiete*, Hochschule Offenburg, Paper, 2010. http://kcode.de/wordpress/wp-content/uploads/2010/10/Paper_NN3Gen.pdf

[MP43] McCulloch, W. ; Pitts, W.: *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, 1943. – 115–533 S.

[MP69] Minsky, Marvin ; Papert, Seymour: *Perceptrons: An Introduction to Computional Geometry.* Cambridge, Massachusetts : MIT Press, 1969

[Ols15] Olszta, Pawel W., Online, 2015. http://freeglut.sourceforge.net/

[Rad06] Rademacher, Paul, Online, 2006. http://glui.sourceforge.net/

[Rec73] Rechenberg, Ingo: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Stuttgart : Frommann Holzboog Verlag, 1973

[RN10] Russel, Stuart ; Norvig, Peter: *Artificial Intelligence - A modern approach.* Pearson Education Inc., 2010

[Roj96] Rojas, Raúl: *Theorie der neuronalen Netze - Eine systematische Einführung.* Springer Verlag, 1996

[Ros58] Rosenblatt, Frank: The perceptron: a probalistic model for information storage and organization in the brain. In: *Psychological Review* 65 (1958)

[RW08] Rey, Daniel G. ; Wender, Karl F.: *Neuronale Netze.* Bern : Verlag Hans Hübner, 2008

[SA09] Segal, Mark ; Akeley, Kurt: The OpenGL Graphics System: A Specification (Version 3.1). (2009), May. https://www.opengl.org/registry/doc/glspec31.20090528.pdf

[Sch15] Schwarz, Jonathan: *A comparison of neural network types and learning techniques with an application in artificial life*, Hochschule Pforzheim, project work, 2015

[SM13] Schembri, Daniel ; Marschall, Paul: *Vergleich der Lernfähigkeit von drei neuronalen Netzwerkmodellen*, DHBW Karlsruhe, Studienarbeit, 2013

[Tou83] Toussaint, Godfried: Solving Geometric Problems with the Rotating Calipers. (1983)

[Tur50] Turing, Alan: Computing machinery and intelligence. In: *Mind* 459 (1950), S. 433 –460

[Wid09] Widnall, S.: *Lecture L3 - Vectors, Matrices and Coordinate Transformations*, Massachusetts Institute of Technology, Article, 2009. http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec03.pdf

[Wik15a] Wikipedia, Online, 2015. http://en.wikipedia.org/wiki/Hill_climbing

[Wik15b] Wikipedia, Online, 2015. https://de.wikipedia.org/wiki/Koordinatentransformation

[WM95] Wolpert, David H. ; Macready, William G.: No free lunch theorems for search. 1995. – Forschungsbericht

[YL15] Yann LeCun, Geoffrey H. Yoshua Bengio B. Yoshua Bengio: Deep learning. In: *Nature* 52 (2015). http://www.docdroid.net/11p1b/hinton.pdf.html

Appendix

The source code and the associated documents of this project are on the enclosed CD / Archive:

- The project report in pdf-format
- The source code of the simulator
- A short tutorial of the simulator
- Associated sources in pdf-format