ANN-Based Autonomous Navigation for a Mobile Robot

Examinations in Effective ANN Layout Design and Training

Julian Beilhack

Master's Thesis

ANN-Based Autonomous Navigation for a Mobile Robot

Examinations in Effective ANN Layout Design and Training

Julian Beilhack

March 10, 2017

Institute for Data Processing
Technische Universität München

Julian Beilhack. *ANN-Based Autonomous Navigation for a Mobile Robot. Examinations in Effective ANN Layout Design and Training.* Master's Thesis, Technische Universität München, Munich, Germany, 2017.

Supervised by Prof. Dr.-Ing. K. Diepold and Prof. Dr. rer. nat. habil. Thomas Bräunl; submitted on March 10, 2017 to the Department of Electrical Engineering and Information Technology of the Technische Universität München.

Institute for Data Processing, Technische Universität München, 80290 München, Germany, `http://www.ldv.ei.tum.de`.

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

.... Munich,...................
Julian Beilhack

Abstract

In this thesis ANN layout and training methods for ANN based autonomous navigation is discussed. ANNs are trained to recognise a set of predefined situations. Movement commands are associated with every situation to achieve autonomous navigation. Experiments were conducted in the EyeSim simulator and with robots of the EyeBot family. Measurements from a laser distance sensor serve as input to the ANNs. The scenarios examined are autonomous maze and open-world way-point navigation. The process of finding a suitable ANN layout and training procedure is discussed for both scenarios. This includes determining the number of neurons for the input and hidden layer of the ANNs, a suitable stopping condition for the training and the number of training datasets. Further, performance enhancing manipulations on the data fed to the ANNs are discussed. Based on the developed ANNs, autonomous navigation for both scenarios is implemented on the EyeBot robots. The experiments show that ANNs with distance measurements as input can serve as a basis for autonomous navigation in indoor environments. Further it is shown, that economical choices in ANN neurons and training datasets can be made without limiting performance. This can be beneficial for systems with limited computational power and when the availability of training data is limited.

Acknowledgements

I would like to thank Prof. Thomas Bräunl at the University of Western Australia and Prof. Klaus Diepold at the Technische Universität München for their supervision of this thesis. Further I would like to thank Marcus Pham, Franco Hidalgo and Kai Lim for their support on the work with the EyeBot robots. Finally I want to thank my parents for making my studies possible through their support.

Contents

List of Figures

List of Tables

1. Introduction

Today, Artificial Neural Networks (ANNs) are a widely used machine learning method and part of many cutting edge technologies. Google's latest translation tool presented by Johnson et al. in [16] or its AI that beat the Go world champion described by Silver et al. in [25] are examples of the impressive AI feats achieved with the help of modern ANNs. Another innovative sector where ANNs are successfully employed is autonomous driving. Although ANN based autonomous driving has a long history, e.g. the road following algorithm presented by Pomerleau in [22] as early as 1989, only now autonomous vehicles are actually getting good enough to be ready for wide use on public streets. In 2016 Öfjäll et al. in [29] presented an algorithm, symbiotic online learning of associations and regression (SOLAR), which achieves autonomous navigation for cars even on snowy roads[1].

While ANNs used for autonomous navigation in public traffic are massive and process a multitude of inputs, this thesis examines the training, layout and potential use cases of small scale ANNs. The only input data provided are the measurements of a laser distance sensor. An obvious advantage of small ANNs as compared to more complex ones is the reduced computational power necessary to operate and to train them. Camera imagery, which is often used as input for ANNs in autonomous driving, has drawbacks in comparison with laser distance measurements. Processing it is computationally much more expansive and its quality is dependent on the lighting conditions. Therefore examining autonomous navigation driven by small scale ANNs fed by laser distance sensors is promising for applications on platforms with limited resources, like small autonomous robots.

Simple autonomous robots are developed for a wide variety of tasks. For example, in [14] Jaradat et al. present an autonomous mine detection robot. Yong-Kyun et al. describe a robot that uses ANNs for environment classification in [18]. In combination with classical reactive control they achieve autonomous behaviour. In [6] Correa et al. describe an autonomous surveillance robot. Their robot's indoor navigation is based on an ANN with input data from a Microsoft Kinect sensor [17]. Their ANN was trained to recognise the paths available to the robot. Combined with a topological map generated with the OpenCV library described in [21] they developed a robot that is capable of autonomously navigating in an indoor environment and surveilling it.

The first experiment conducted for this thesis treats a closely related problem, the autonomous navigation of a maze. Although ANNs are employed to tackle similar problems, like the ANN based wall-following robot presented by Sahu et al. in [7], the task can be solved by simpler means, like a feedback control as described by Turennout et al. in [27].

[1]Snowy roads are problematic due to missing road markers.

However, the problem is well fit to examine the process of determining an ANN layout and training process suited for a given navigation task and yielded various insights. Those insights were put to work in the second experiment where an open-world way-point navigation with obstacle avoidance was developed. The aspects discussed are the number of layers an ANN should have to solve the given task and the number of neurons per layer. Further, a training process is developed that prevents *overfitting*. The influence of the amount of data used for training is analysed. Finally, manipulations on the input data to improve ANN performance are discussed. The ANNs are trained to interpret their input distance measurements as one of a predefined set of states. By continuously evaluating the ANN and transitioning from state to state according to its outputs autonomous navigation was achieved. Each navigation scheme was first implemented in the EyeSim simulator described in section 3.3 and then ported onto the robot described in section 3.1. The ANNs used were implemented with the help of the FANN API described in section 3.4. Before the work on the autonomous navigation problems was begun, the EyeSim simulator API[2] was updated to match the newly implemented EyeBot API[3]. This not only served as a good way to get familiar with the topic, but also allowed to introduce a model of the laser sensor used for this thesis to the simulator.

Thesis Structure

Chapter 2 gives a brief introduction into the concepts of ANNs and PID control, which were employed in experiments. In Chapter 3 the hard- and software used for the experiments is presented. This includes the EyeBot robots, the Hokuyo laser sensor, the EyeSim simulator and the FANN API. Chapter 4 gives an overview of the processes that were updated in the EyeSim simulator. Further, the model of the Hokuyo laser sensor that was introduced to the simulator is described. Chapter 5 presents the process employed to gather the training and reference data for the ANNs. Chapter 6 and 7 described the maze navigation and open-world way-point navigation experiments. For both experiments, first the layout and training results are presented and discussed. Then a description of the implementation of the autonomous navigation on the EyeBot robots is given. In Chapter 8 the results are discussed and a conclusion is drawn.

[2]The simulator used for the experiments.
[3]The API used to control the robots used for the experiments.

2. Theoretical Principals

This chapter gives a brief introduction into the theoretical principals employed in this thesis. The first section treats artificial neural networks (ANNs). Basics of the layout and training of ANNs are presented. Further, the phenomenon of *overfitting* is explained. The section closes with a summary of the choices in ANN layouting and training discussed in the experimental part of this thesis. The second section introduces PID control, which was used to steer a robot through a maze in a straight fashion in the maze navigation experiment.

2.1. Artificial Neural Networks

Artificial neural networks (ANNs) are an attempt in machine learning to emulate biological neural networks like the human brain. An introduction to ANNs can be found in [19] by M. A. Nielsen and in [12] by Goodfellow et al. Figure 2.1 shows the structure of a simple ANN. Three vertical layers can be distinguished: the input, the hidden and the output layer. The most basic ANN would consist of only an input and an output layer. A network like that is, however, only capable of solving linear problems. To solve non-linear problems, at least one hidden layer is necessary. Many moderately complex problems, like handwriting recognition, can be solved by ANNs with one single hidden layer. On the other hand, ANNs for tasks like language translation, such as the one used by Google translate [16], require many hidden layers. For the experiments conducted for this thesis ANNs with one hidden layer were sufficient.

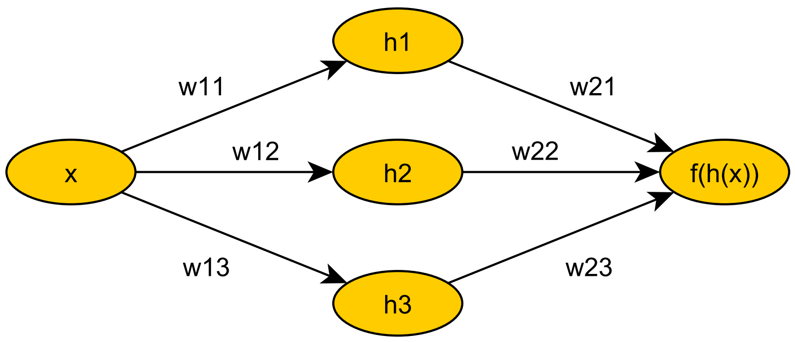


Figure 2.1.: Simple artificial neural network with one input neuron, 3 hidden neurons and one output neuron. *w11* - *w23*: weights of the inter-neuron connections.

2.1.1. Layout

Weighted edges connect the neurons. Like the action potential in biological neurons, artificial neurons have a so called *activation function* to threshold their activation. Activation functions can be a simple step function or smoother functions like the sigmoid, that allow values between "activated" and "not activated". The activation function employed in the ANNs presented in this paper is the symmetric sigmoid function which is defined by the following equation [24, p. 1266]:

$$\mathrm{sig}(z) = \frac{2}{1 + e^{-2 \cdot s \cdot z}} - 1 \tag{2.1}$$

Where s is a scaling factor and z the sum of the weighted inputs to the neuron. A plot of the sigmoid function for $s = 1$ can be seen in figure 2.2. Only neurons in the hidden and the output layer have activation functions.

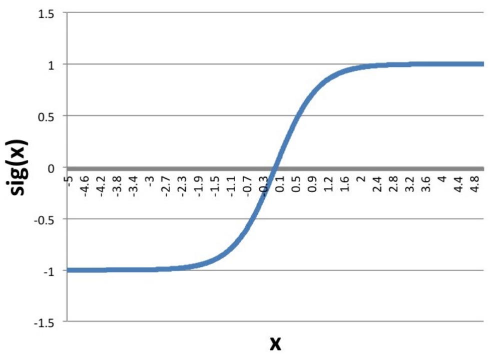


Figure 2.2.: Plot of the symmteric sigmoid function.

Input neurons represent one input parameter each and don't have an activation function. The output of neuron h_1 can therefore be expressed as a function of x as follows [19, chap. 1]:

$$h_{1,out} = \mathrm{sig}(w_{11} \cdot x) \tag{2.2}$$

The ANN's overall output is consequently desribed by the following equation [19, chap. 1]:

$$f(h(x)) = \mathrm{sig}\left(\sum_{n=1}^{3}(\mathrm{sig}(w_{1n} \cdot x) \cdot w_{2n})\right) \tag{2.3}$$

For the general case of N_{input} input neurons and N_{hidden} hidden neurons, the result of an output neuron f_l can be calculated as [19, chap. 1]:

$$h_k = \sum_{n=1}^{N_{input}}(x_n \cdot w_{in_{nk}}) \tag{2.4}$$

$$f_l = \text{sig}\left(\sum_{k=1}^{N_{hidden}} (\text{sig}(h_k) \cdot w_{out_{kl}})\right) \tag{2.5}$$

Where w_{in} are the weights connecting the input to the hidden layer and w_{out} the weights connecting the hidden to the output layer.

2.1.2. Training

An ANN learns by being trained. This is achieved by adapting the weights of the connection edges in a way, that certain inputs produce associated desired outputs. A widely used training method is the so called *backpropagation* algorithm. This iterative algorithm requires the neuron's activation function to be differentiable. Further, training data consisting of pairs of input values and the desired output values is needed. First, the weights of the connection edges are randomized. Each iteration of the algorithm then involves the following steps as described by Hecht-Nielsen in [9]:

1. Calculation of neuron outputs for the input data with current weights
2. Backpropagation to compute the difference between actual and desired outputs
3. Updating of the weights according to a chosen rule, e.g. $\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}}$ with learning rate η and error E

Those steps are repeated until the ANN meets specified performance criteria, for example the mean square error (MSE)[1] of the outputs is below a certain threshold. The standard training algorithm employed by the FANN API[2] is a specialised backpropagation, the iRPROP introduced by Igel et al. in [13], which is based on the RPROP described by Riedmiller et al. in [23]. The iRPROP is an adaptive gradient descent method. In this case adaptive means, that the learning rate is adapted automatically during the training process. The iRPROP consists of three steps [13]:

1. The learning rate is determined as:

$$\eta_{ij}^{(t)} = \begin{cases} \alpha^{+} \cdot \eta_{ij}^{(t-1)} & \text{for } \frac{\partial E}{\partial w_{ij}}^{(t)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t-1)} > 0 \\ \alpha^{-} \cdot \eta_{ij}^{(t-1)} & \text{for } \frac{\partial E}{\partial w_{ij}}^{(t)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t-1)} < 0 \\ \eta_{ij}^{(t-1)} & \text{else} \end{cases} \tag{2.6}$$

   Where (t) denotes the current iteration and (t-1) the iteration before that. Further, $0 < \alpha^{-} < 1 < \alpha^{+}$. The learning rate is bounded by $[\eta_{min}, \eta_{max}]$.

[1]The difference between desired and actual ANN outputs.

[2]The API used to implement the ANNs for this thesis

2. The weight updates are calculated as:

$$\Delta w_{ij}^{(t)} = \begin{cases} -\text{sign}(\frac{\partial E}{\partial w_{ij}}^{(t)}) \cdot \eta_{ij}^{(t)} & \text{for } \frac{\partial E}{\partial w_{ij}}^{(t)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t-1)} > 0 \\ -\Delta w_{ij}^{(t-1)} & \text{for } \frac{\partial E}{\partial w_{ij}}^{(t)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t-1)} < 0 \\ 0 & \text{else} \end{cases} \tag{2.7}$$

3. The weights are updated as: $w_{ij}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$

The main advantages of iRPROP compared to a simple backpropagation algorithm are the adaptive choice of the learning rate as well as taking into consideration the weight updates of former training iterations[3]. It was employed for every ANN trained for the experiments conducted for this thesis. Training with iRPROP converged after few training epochs[4] and never took longer than a couple of minutes with the ANN layouts used here.

2.1.3. Overfitting

Training algorithms for ANNs generally minimize the mean square error (MSE) of the output. The MSE is, however, not a very good quality feature for the purposes of the experiments conducted for this thesis. The actual goal is to create an ANN that performs well in general situations, not one that is trained in every peculiarity of the training data. Hence, Nielsen defines the *accuracy* of an ANN in [19, chap. 3] as follows:

$$accuracy = \frac{n_{correct}}{n_{total}} \tag{2.8}$$

Where n_{total} is the total number of reference datasets and $n_{correct}$ is the number of reference datasets for which the ANN yields the desired outputs. The reference data, like the training data, consists of sets of input and desired output values for the ANN. The only difference is, that it isn't used to train the ANN. Thus it can be utilised to measure the performance of the ANN in unknown situations. As described by Nielsen in [19, chapter 3] a decreasing mean square error doesn't necessarily correlate with an increasing accuracy. Often the accuracy plateaus after a certain number of training epochs while the error is still going down. This can be observed very clearly in figure 2.3 which shows the development of the accuracy and the MSE of an ANN over the number of training iterations. The data for this plot was gathered during the training process of an ANN trained on data from the Eye-Sim simulator. As can be seen, the accuracy reaches its maximum after about 15 training epochs while the MSE continues to go down. Since from the moment when the accuracy maximum is reached the ANN only learns the peculiarities of the training dataset without any improvement in generalization, ideally the training process should be stopped there to prevent the ANN from overfitting to the training data. In principal the accuracy might only

[3]Taking into consideration weight updates of former training iterations is called *backtracking*.
[4]Usually less than 50.

plateau for a certain amount of epochs before starting to improve again. This effect has, however, never[5] occurred in the execution of the experiments conducted for this paper. Another way to counteract overfitting according to Nielsen [19, chapter 3] is using larger sets of training data. However, acquiring additional data can be expensive or sometimes even impossible. Tracking the accuracy during training and stopping once it stops improving is therefore a good way to generate a well fit ANN without the need for larger datasets.

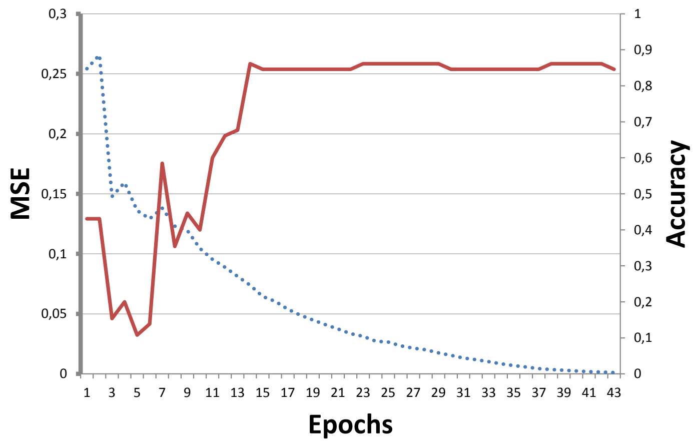


Figure 2.3.: Plot of the development of the MSE (blue) and the accuracy (red) of an ANN during training.

2.1.4. Choices in Layouting and Training

There is no simple rule of thumb to determine a suitable layout for an artificial neural network for any given problem. There are however guidelines as to which problems have to be adressed like presented by T. Rajkumar et al. in [26]. These include:

- Neuron's activation function
- Number of hidden layers
- Training algorithm
- Number input neurons
- Number of output neurons
- Number of hidden neurons for each hidden layer
- Number of training epochs

[5]Even for very high amounts of training epochs, i.e. 10000 as compared to the 61 in figure 2.3.

Different activation functions provided by the FANN API were tested. Those were the (non-symmetric) sigmoid, the step-wise sigmoid and the symmetric sigmoid. While no great differences could be detected, the symmetric sigmoid proved to be slightly better and was hence employed for all navigation experiments. Among the FANN training algorithms the iRPROP performed best in terms of numbers of training epochs until convergence and was therefore used for the experiments. As mentioned earlier, one hidden layer is necessary and sufficient for the problems presented here. The process involved to determine suitable configurations for the remaining parameters is described in Chapter 6 and 7 for each experiment.

2.2. PID Control

One of the most widely used control methods is the PID control. This section gives a brief introduction into its basics. Further information on this topic can be found in [28] by A. Visioli. PID stands for the three different terms of the PID control:

1. **P**roportional
2. **I**ntegral
3. **D**ifferential

PID controls are often implemented as closed feedback loop controllers as shown in figure 2.4. As can be seen in the figure, the controller input *e(t)* is the difference between the desired value *r(t)* and the actual value *y(t)* of the system variable to be controlled. The input to the plant, the *control variable*, *u(t)* is the output of the controller and is calculated to minimize *e(t)*. The controller equations follow as [28, chap. 1]:

$$e(t) = r(t) - y(t) \tag{2.9}$$

$$u(t) = K_p \cdot e(t) + K_i \int_0^t e(x)dx + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2.10}$$

With the proportional scaling parameter K_p, the integral scaling parameter K_i and the differential scaling parameter K_d. These have to be chosen so that *u(t)* minimizes *e(t)*.

For the use in a digital control, like in this thesis, theses equations have to be adapted. The pseudo code for a simple PID control algorithm is given in figure 2.5 [28, chap. 1]. The integral is translated into the sum of all previous errors, the derivative into the difference between the previous and the current error. Once *u* is calculated it is used as input to the plant. The plant could for example be a DC motor to be velocity controlled. The difference of the current velocity and the desired velocity would constitute *e*. The control variable could then be a PWM signal.

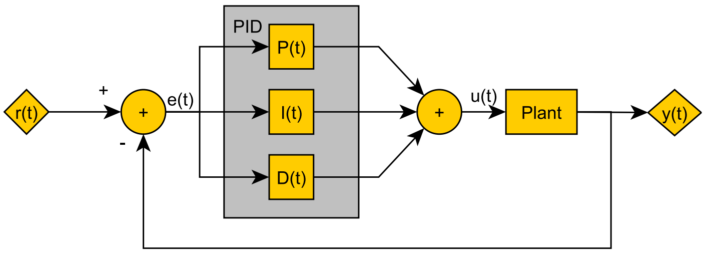


Figure 2.4.: Block diagram of a closed PID control loop.

```
e_old = 0;
i = 0;

while(control){
    y = get_y();
    e = r - y;
    i = i + e;
    d = e - e_old;
    e_old = e;
    u = kp * e + ki * i + kd * d;
    set_plant_input(u);
}
```

Figure 2.5.: Pseudo Code of a PID controlling algorithm.

The main challenge when implementing such a simple controller is to find suitable values for the *k* parameters. There are numerous methods to find such parameters. In most cases, however, these methods require a good model of the plant to be controlled or extensive measurements. In practice they are therefore often determined by simple experimental trial and error. This trial and error method was employed to find the parameters for the control presented in the maze navigation experiment of this thesis. Furthermore, sometimes the full PID term is not needed to adequately control a system, i.e. a P- or PI-control is sufficient. This was the case for the maze navigation control where a PI-control proved to be adequate.

3. Hardware and Software

In this chapter the hardware, that is the robots and sensors, and the software, that is the FANN API and the EyeSim simulator, used in the course of the work on this thesis are presented. The EyeBot robots, the EyeSim simulator and the Hokuyo laser distance sensor were provided by the robotics lab of the University of Western Australia. The API to interface the laser sensor and the ANN library were chosen independently.

3.1. EyeBot Robot

The robots on which the navigation schemes developed in the course of this thesis were implemented are part of the EyeBot family. They were developed by Prof. Thomas Bräunl at the University of Western Australia and are presented in [5]. The ones employed here are driven by two DC motors with connected encoders. They come equipped with a digital camera and position sensitive devices (PSDs)[1]. The control of the robots of the EyeBot family is divided into a high and a low level as shown in figure 3.1. The high level consists of a Raspberry Pi. Via calls to the EyeBot API installed on the Raspberry Pi the motors can be controlled, sensor measurements read and images from the camera retrieved. Further features of the EyeBot API include simple image processing capabilities and functions to control the LCD display connected to the Raspberry Pi. A ATxmega128A1U micro controller placed on a custom made IO-board represents the low level of the robot control. It is connected to the high level via USB. Movement commands received from the high level, e.g. "go straight for 10 cm", are translated into pulse width modulated (PWM) signals which drive the DC motors via a H-bridge. For the purpose of this thesis an additional laser distance sensor (see section 3.2) was fixed to the front of the robot and directly connected to the Raspberry Pi via USB. Energy is supplied either by directly connecting the robot to a power outlet or by a battery pack. Image 3.2 shows one of the robots used with a denotation of its components. The second EyeBot model used during the experiments can be seen in image 7.2.

The robot can be operated either with the help of the GUI for the Raspberry Pi LCD display provided by the EyeBot controller or by establishing a remote desktop connection via Wi-Fi. User programs are implemented in C and compiled with the custom *gccarm* script. By including the *eyebot.h* header file the functions described in [3] can be used to control the robot. An in-depth online documentation on the EyeBot controller can be found in [2]. The data sheet of the ATxmega128A1U micro controller can be found in [1].

[1]PSDs are distance sensors.

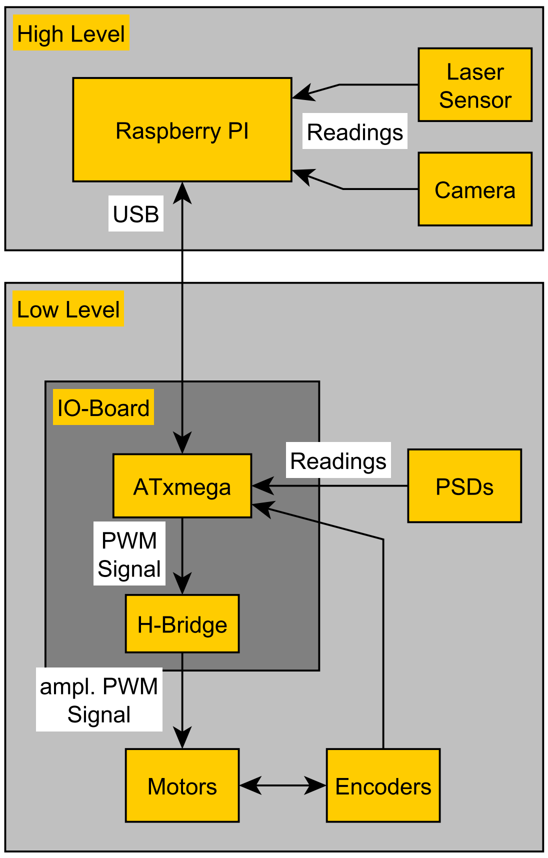


Figure 3.1.: Schematic of the high and low level components of an EyeBot robot. The high level consists of a Raspberry Pi. The low level is represented by a custom made IO-board with a ATxmega128A1U micro controller. The DC motors are driven by PWM signals via a H-bridge. PSD, digital camera and motor encoder data is provided to the high level functions via the micro controller. The laser distance sensor is directly connected to the Raspberry Pi via USB.

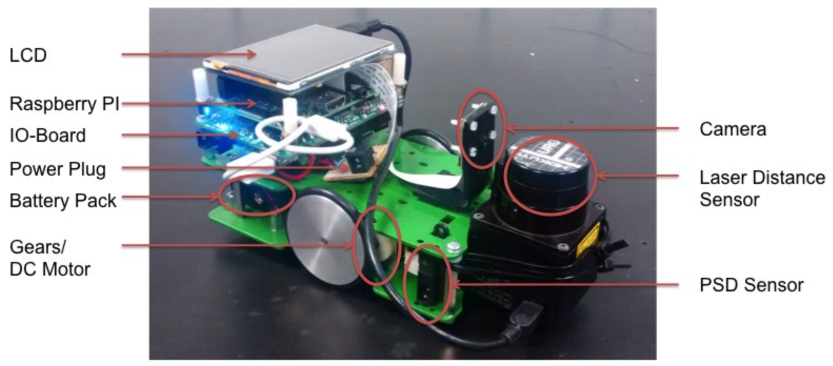


Figure 3.2.: Depiction of an EyeBot robot and denotation of its components. The robot is equipped with PSD sensors, a digital camera and a laser distance sensor. It is driven by two DC motors with encoders and powered by a battery pack or by directly plugging it into a power outlet.

3.2. Hokuyo Laser Sensor

To gather distance measurements of the robot's surroundings the Hokuyo Scanning Laser Range Finder URG-04LX-UG01 was employed. It covers an area of 240 degrees at an angular resolution of a approximately 0.36 degrees. The area radiated by the sensor is shown in image Figure 3.3 taken from its documentation [11]. This results in 682 measurement points with a maximum reach of 4 meters. The measurement accuracy is given as +/- 3% for distances between 20 mm to 4000 mm. The sensor can comfortably be connected to the robot's Raspberry Pi via USB, which also powers the sensor.

The C API provided by Hokuyo was used to interface the sensor with the EyeBot API. As of February 2017, big parts of the Hokuyo C API documentation were only available in Japanese. Since the functions behaviour is not always intuitive, using the sensor was made a little harder by that fact. Where it was available in English, the documentation was good and eventually all necessary communication with the sensor could be implemented in a stable fashion. Installation instructions and the API documentation can be found in [10]. An alternative to the Hokuyo sensor would be for example the Kinect sensor employed by Correa et al. in [6]. The sensor was modelled in the EyeSim simulator as described in Chapter 4.

3.3. EyeSim Simulator

The EyeSim simulator described in [4] offers an interface identical to the EyeBot API described above and allows to simulate multiple robots in parallel. The PSD sensors and digital camera used by the EyeBot robots are simulated as well as the motor control. For the purpose of this thesis the Hokuyo laser sensor described in the previous section was implemented for the simulator as described in chapter 5. In order to generate experiment

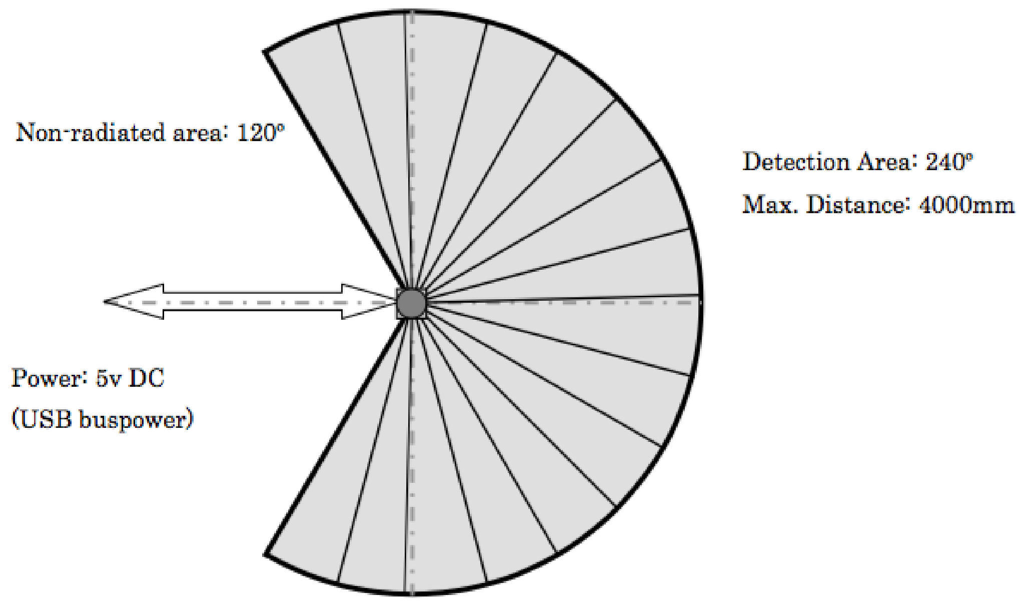


Figure 3.3.: Area radiated by the Hokuyo laser sensor taken from its documentation in [10].

environments, EyeSim offers the world and maze format to build maps. Both employ simple text files to define the maps and are easy to use. As an example image 3.4 shows a football field defined in the world format with six EyeBot robots. Each robot's camera output is displayed in a separate window. The *settings* button allows to change each robot's x- and y-position as well as their rotational angle. Simulation programs are implemented in C and compiled with the custom *gccsim* script. If compiled with the *gccarm* script instead, they can be employed on the physical robots without any changes necessary. The EyeSim simulator was used to develop and test the autonomous navigation strategies examined in this thesis before implementing them on the physical robots.

3.4. FANN API

The choice of the library employed in this paper to implement the neural networks fell on the Fast Artificial Neural Network Library (FANN) presented by Steffen Nissen in [20]. It offers various functions allowing the creation and training of neural networks as well as other useful utilities like the scaling and shuffling of input and training data. Its functionality covers the requirements to solve the problems presented in this paper and it is well documented. No problems concerning FANN's performance were encountered. The first reason for choosing FANN was its intuitive interface in C and its resulting compatibility with the EyeBot API. Compared with other, more complex, libraries like caffe described in [15],

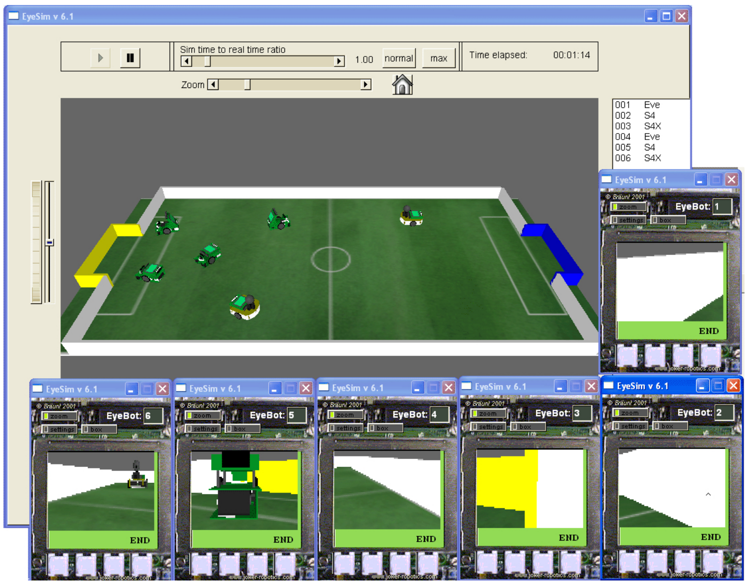


Figure 3.4.: EyeSim GUI with six EyeBot robots simulated as shown in [4]. Individual window for every robot's camera output. The map shown is a football pitch and was created in the EyeSim world format.

FANN was deemed to be better suited to address the rather simple pattern recognition problem presented in this paper without too much complexity overhead. A comparably complex neural network library that is as easily interfaced with C as FANN wasn't found and subsequently FANN was used.

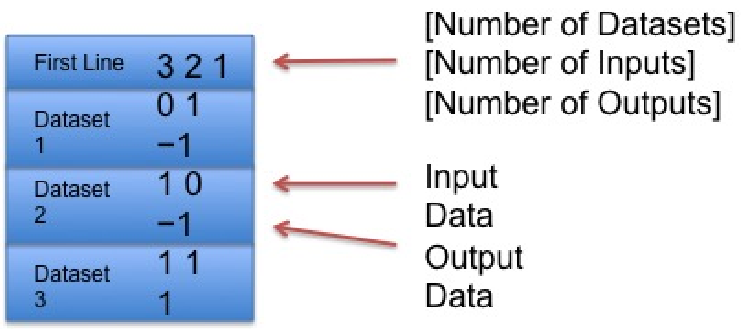


Figure 3.5.: FANN datafile format for the example of the and-function. First line: number of datasets, number of inputs and number of outputs separated by spaces. Then: lines of inputs separated by spaces followed by lines of outputs separated by spaces.

Work Flow

In the following the basic work flow to train and run an ANN with the FANN library will be explained. As a first step, data to train the ANN on is required. To make efficient training possible and save the data for future use, it is recommended to store the gathered data in (.data)-files of the FANN format shown in figure 3.5. The first line consists of the number of datasets, the number of inputs and the number of outputs separated by spaces. The following lines are a line of inputs separated by spaces followed by a line of outputs separated by spaces (if there is more than one).

Once a suitable datafile is generated, the function `FANN_create_standard` is called. Its parameters are the number of layers, the number of input neurons, the number of neurons in the hidden layers and the number of output neurons. It returns a `struct FANN *ann` representing the created neural network, which is then used in `FANN_train_on_file` to train the created network on the previously gathered training data. The remaining parameters are the name of the datafile to train on, a maximum number of training epochs, the number of training epochs after which a report is published to the terminal and a desired mean square error which when reached stops the training process. Finally the trained `struct FANN *ann` can be processed with the function `FANN_run`, where the input is an array of input data. Return value will be an array of [`num_output`] values in the interval [-1,1] representing the probability of each output. A simple program that trains a FANN network on the data shown in figure 3.5, executes it and prints the results to the terminal is shown in figure 3.6.

```
#include "fann.h"
#include <stdio.h>

int main(){
    int epochs_between_reports = 5;
    fann_type *out;
    /* MSE stopping value */
    float desired_error = 0.01;
    /* read data from file "and.data" */
    struct fann_train_data *data = fann_read_train_from_file(
        and.data);
    /* layers[numberOfLayers] = {inputNeurons,
        hiddenNeuronsLayer1...N, outputNeurons} */
    unsigned int layers[3] = {data->num_input, 4, data->
        num_output};
    /* create ANN with input, hidden and output layer
    number of neurons per layer defined in layers[] */
    struct fann *ann = fann_create_standard_array(3, layers);
    /* set activation functions */
    fann_set_activation_function_hidden(ann,
        FANN_SIGMOID_SYMMETRIC);
    fann_set_activation_function_output(ann,
        FANN_SIGMOID_SYMMETRIC);
    /* train ANN on data until MSE <= 0.01 */
    fann_train_on_data(ann, data, 1, epochs_between_reports,
        desired_error);

    int netInput[2] = [0,0];
    /* evaluate trained ANN for input [0,0] */
    out = fann_run(ann, netInput);
    /* print result, expectation out = 0 */
    printf("ANN␣Output:␣%lf", out[0]);

    /* save trained ANN to the file "and.net" */
    fann_save(ann, "and.net");
    fann_destroy(ann);
    fann_destroy_train(data);

}
```

Figure 3.6.: FANN program that trains and executes an ANN on the data shown above and prints the result to the terminal.

4. Update of the EyeSim API

As a prelude to the work on the experiments described in the next chapter, the API of the EyeSim simulator was updated. Since the EyeBot API used for the application programs of the physical robots was reimplemented from scratch, the EyeSim API had to be adapted to guarantee that simulator programs can be used on the robots without any changes necessary. The changes included:

- Function name changes
- Function Parameter changes
- Introduction of new functions
- Introduction of new camera resolutions
- Introduction of new LCD image sizes

Changing function names and adapting parameters was mostly straightforward yet time consuming. Most functions that had to be newly implemented, like `LCDCircle` to draw a circle on the LCD output, did not pose bigger problems either. The introduction of new LCD image sizes and camera resolutions, however, required a good understanding of the underlying processes. Since documentation on those processes, and the simulator in general, was thin, an overview of the changes made and a description of those processes is given in the first two sections of this chapter. This might help when those processes have to be further adapted in the future. During the work on the API updates the simulator was further expanded by a model of the Hokuyo laser distance sensor described in section 3.2. The model is described in the third section of this chapter.

4.1. Different LCD Image Sizes

The EyeSim GUI and image display is implemented with the Fast Light Toolkit (FLTK) API. A documentation of the FLTK API can be found in [8]. To display an image in EyeSim, first one of the functions `LCDImage`, `LCDImageGray` or `LCDImageBinary`, depending on the type of image to be displayed, is called. EyeBots simulated in EyeSim are modelled as objects of the struct `localRobi`. The API `LCDImage` functions call the respective `LCD_PutColorGraphics`, `LCD_PutGrayGraphics` or `LCD_PutColorGraphics` function of the respective localRobi's member of the class `eyeconsole`. The `LCD_PutGraphics` functions then call the `eyeconsole` member function `LCD_SetColorPixel` which set the graphics buffer of the `eyeconsole` member of the class `DualMode` according to the image to be displayed. Once this is done, the `LCD_PutGraphics` functions call the `redraw` function of the `DualMode` member. The `DualMode` function `redraw` in turn calls the `DualMode` function `draw` which calls the FLTK function `fl_draw_image`. This finally displays the image on the respective EyeBot's LCD. Figure 4.1 shows a flowchart of this process.

To allow for images of different sizes to be displayed, the EyeSim API function `IPSetSize` was implemented. It takes an integer associated with a resolution as parameter[1]. The camera resolution of the EyeBot for which the function is called is then set accordingly. Further, parameters for the image rows and columns were added to the `eyeconsole` `LCD_PutGraphics` functions. The `LCDImage` functions were then adapted to call the respective `LCD_PutGraphics` functions with the EyeBot's x and y camera resolution as image columns and rows parameters. Apart from adaptations in the `LCD_PutGraphics` functions allowing for images of varying size to be displayed based on the new parameters, the DualMode defines `PixX` and `PixY`, which limit the number of pixels allowed to be displayed per line and column, had to be increased. The reason why those defines existed in the first place is, that the LCD display window is of rigid size. Therefore, as a last step, the the size of the image display window was set to the maximum image size to be displayed. To achieve this, the constructor of the FLTK `console`[2] responsible for the GUI display was adapted accordingly.

4.2. Different Camera Resolutions

Camera images can be accessed in the simulator via the EyeSim function `CAMGet`. `CAMGet` calls the `render` function of the `Camera` member of the EyeBot for which it was called. The `render` function first calls the `draw` function of the `Camera's` `Main3DView` member. After that, the `data` variable of `Main3DViews's` `meRawImage` member is accessed via the standard C++ function `memcpy`. This raw image is then rectified according to the column and row offset of the image as well as the camera's x and y resolution. The process is depicted in the flowchart in figure 4.2.

To allow for varying camera resolutions, the class `Camera` was expanded by two variables for its x and y resolution respectively. The resolution can then be set by the `IPSetSize` EyeSim API function as described above. The necessary changes were integrated in the `render` function, which previously only allowed for rigid resolutions, and the dependent functions and classes were adapted accordingly. However, when accessing the camera after these changes, segmentation faults occur after a short period of time and the simulator crashes. Even after a extended search the cause for these faults could not be determined and this issue remains to be addressed.

[1]QQVGA: 0, QVGA: 1, VGA: 2, CAM1MP: 3, CAMHD: 4, CAM5MP: 5
[2]The constructor can be found in the file console.cpp.

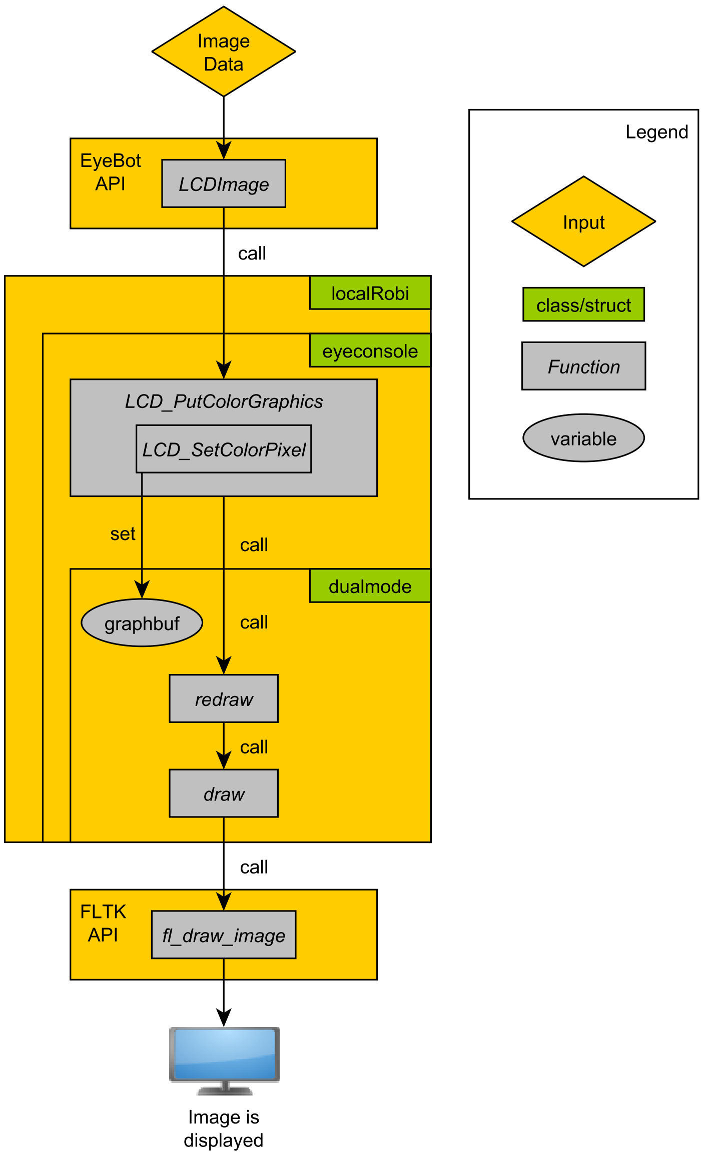


Figure 4.1.: Flowchart of the image display process of the EyeSim simulator. The EyeSim GUI and image display are implemented with the FLTK API.

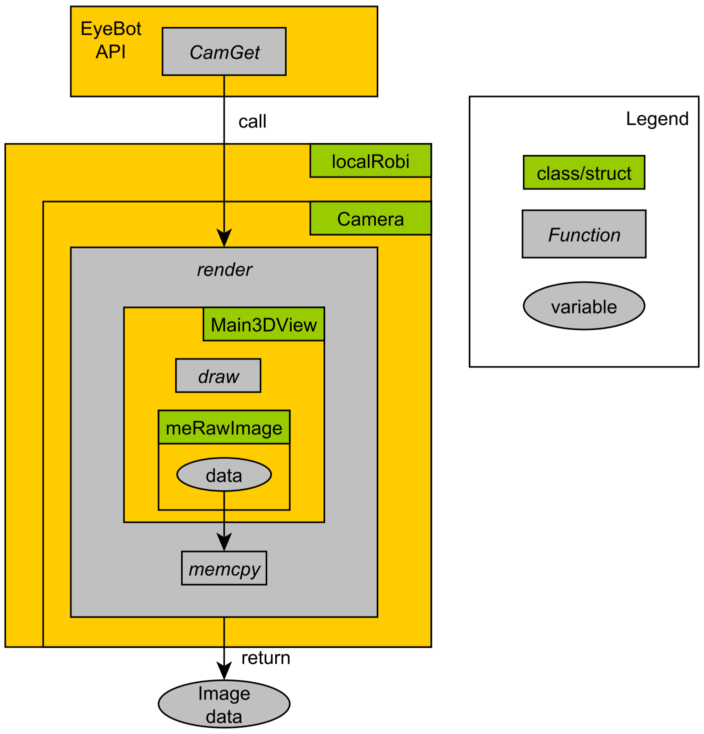


Figure 4.2.: Flowchart of the access to camera imagery process of the EyeSim simulator.

4.3. Laser Sensor Model

To allow the simulation of the ANN-based autonomous navigation applications developed in the course of this thesis in the EyeSim simulator, the Hokuyo laser distance sensor used as input for the ANNs had to be modelled in EyeSim. Since the simulations were only supposed to be proofs of concept the sensor model didn't have to be very close to the physical sensor in terms of noise and accuracy. Accordingly, the sensor was modelled as 682 PSD sensors concentrated on one single pixel. The PSD sensor model already included in the EyeSim simulator was used. The PSD sensors cover an area of 240 with a step width of about 0.36 degrees as shown in figure Figure 3.3. The EyeSim API was expanded by the functions `LaserGet` and `LaserPlot` which allow to retrieve the sensor readings and plot the sensor readings in polar coordinates respectively. The output of the plot function can be seen in the rightmost part of figure 5.1. The resulting model together with the implemented API functions allowed for a comfortable simulation of the experiments conducted for this thesis.

5. Methodology

The Experiments described in this thesis are implementations of ANN-based autonomous navigation in different scenarios. The ANNs employed are implemented with the help of the FANN (see section 3.4). Measurements gathered by the laser distance sensor (see section 3.2) serve as input for the ANNs. As a first step in every experiment, the navigation scenario was implemented in the EyeSim simulator (see section 3.3). This involved building maps in the EyeSim maze or world format, as well as implementing the necessary navigation commands.

Once the principals for solving the different problems had been worked out in the simulator, the solutions were ported to the physical robot. To make the robot ready for the experiments, first the FANN and the Hokuyo C API were installed on the robot's Raspberry Pi. The basic schematic in image 5.2 shows the signal flow of the navigation process. As can be seen, the Hokuyo laser sensor was connected to the robot's Raspberry Pi via USB. It was fixed to the front of the robot. Communication with the sensor was implemented with the help of the Hokuyo C API, based on which functions for plotting and reading of the distance measurements were realised.

The first section of this chapter describes the data gathering process for the training and reference data needed to train the ANNs. Section two presents the analysis functions implemented to find suitable ANN layouts.

5.1. Data Collection

Two types of training data had to be gathered: data from the EyeSim simulator and from the physical sensor. In order to collect the required data in the simulator, first the laser sensor employed by the physical robot was modelled in the simulator as described in the previous chapter. Functions to plot and collect the current sensor readings were implemented. As a platform for data collection in the simulator, an EyeSim Simulator program was implemented. Its GUI is shown in figure 5.1. To collect data, the robot is moved around with the help of the simulator settings. When the "Cap" button is pressed, the desired ANN output for the current position can be chosen. The current sensor readings together with their respective output are then automatically saved to a data file in the FANN format. The program also offers the possibility to train an ANN on the collected data and evaluate it for the robot's current position in order to test it. A similar program was implemented for the physical robot, only with the actual laser sensor as data source. The robot was moved

Function Name	**Description**
`LaserGet`	Gets current laser sensor values.
`LaserPlot`	Plots current laser sensor values in polar coordinates.
`appendData2file`	Generates data file in FANN format. Decision for desired output to current sensor readings. Both are saved to the generated file.
`scaleFannFile`	Scales number of input values in fileName by $\frac{1}{factor}$. Result is saved to newFileName.
`testNet`	Computes accuracy of a fann net.
`trainAnalysis`	Trains ANN and computes its accuracy for every epoch. Accuracy and MSE values are saved to resFile.
`accOverHiddenWrapper`	Trains ANNs with different N_{hidden}. Computes and saves their accuracy maxima.

Table 5.1.: List of functions for data gathering and ANN training analysis.

manually. Training data consisted of equal amounts of datasets[1] of comparable variance for every ANN output. Reference data was compiled following the same principals, the only difference being that it could not be the same measurements so as to have independent data to calculate the ANNs accuracy. In table 4.1 a list of the functions implemented for data collection and training analysis purposes can be found.

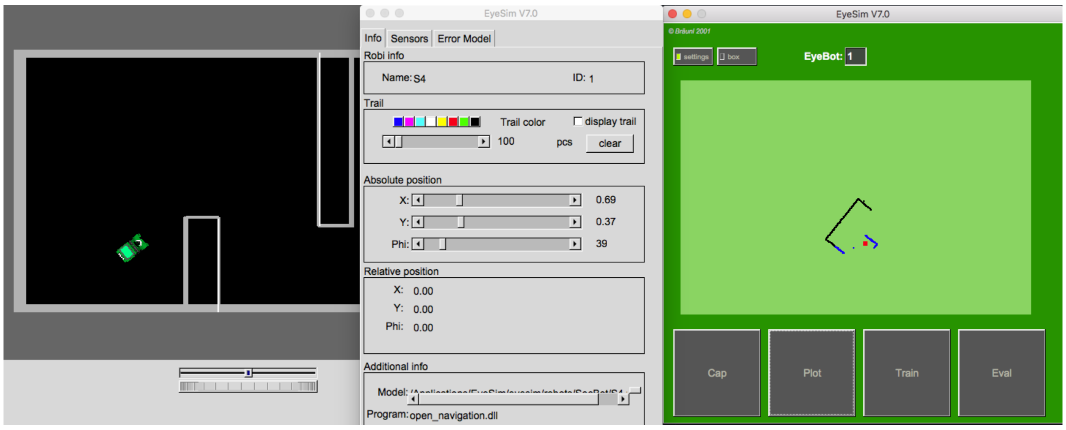
5.2. ANN Analysis Functions

This section presents the functions implemented in the course of this thesis for the analysis of ANN layout and training. Those functions were employed during the experiments described in the following sections. Functions to analyse the following aspects were developed:

- MSE and accuracy development during training
- Accuracy development over number of hidden neurons
- Accuracy development over number of training datasets

The functions are described in this order. Additionally their source code is given so as to allow the reader to completely understand where the data presented in the following sections came from.

[1]A dataset consists of input data and corresponding desired ANN output values.


EyeSim V7.0

Robi info

Name: S4 ID: 1

Trail

Trail color ☐ display trail

100 pcs

Absolute position

X: 0.69

Y: 0.37

Phi: 39

Relative position

X: 0.00

Y: 0.00

Phi: 0.00

Additional info

Model: /Applications/EyeSim/eyesim/robots/SocBot/S4

Program: open_navigation.dll

Figure 5.1.: EyeSim Simulator Program for data collection, ANN training and testing. *Cap*: Capture current sensor readings, choose desired output. Both are appended to the generated datafile in FANN-compatible structure. *Plot*: Plot current sensor readings as seen in the image. *Train*: Train a FANN net on the collected data. *Eval*: Evaluate the trained net for current sensor readings.

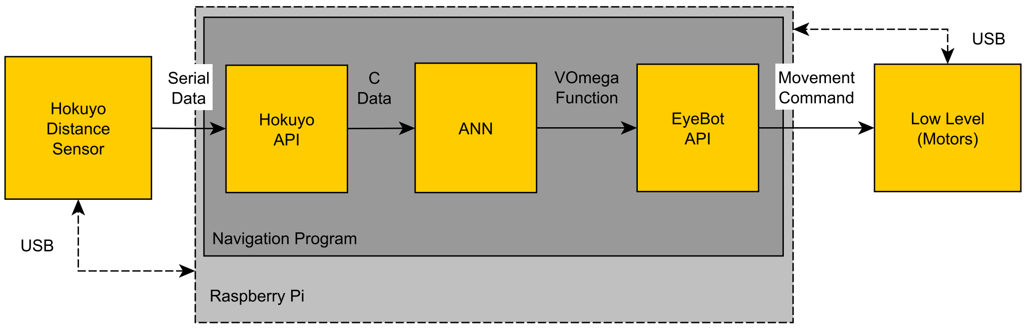


Figure 5.2.: Schematic of ANN-based navigation on an EyeBot robot. The Raspberry Pi communicates with the Hokuyo laser sensor via USB. The sensor is interfaced in C with the help of the API provided by Hokuyo. The movement commands associated with the ANN outputs drive the robot's DC motors via the EyeBot API's *VW* functions.

```
float testNet(char *reference_file, struct fann *ann,  ){
  fann_type *calc_out;
  int l,m;
  float total,nError=0;
  struct fann_train_data *data = fann_read_train_from_file(
     reference_file);
  //shuffle and scale reference data
  fann_scale_input_train_data(data, 0,1);
  fann_shuffle_train_data(data);

  for(int i=0; i<data->num_data; i++){
    calc_out = fann_test(ann, data->input[i], data->output[i]);
    l = largestMember(calc_out);
    m = largestMember(data->output[i]);
    if(l!=m){
      nError = nError+1;
    }
  }
  total = data->num_data;
  fann_destroy_train(data);
  return accuracy = (1-(nError/total));
}
```

Figure 5.3.: Source code of the analysis function `testNet` that computes the accuracy of a given ANN on a given reference datafile.

5.2.1. MSE and Accuracy Tracking during Training

To track the development of the MSE and the accuracy of an ANN during training, first the function `testNet` was implemented. This function gets a FANN network and a reference datafile as parameters and calculates the ANN's accuracy on the reference datasets. As can be seen in the source code of the function given in figure 5.3, the ANN is evaluated for every input set in the reference file. The result of this evaluation is then compared with the desired output given for those inputs in the reference datafile. If they don't match, the evaluation is counted as an error. The accuracy is then computed according to equation 2.8.

The function `trainAnalysis` trains a FANN network with a given number of hidden neurons on a given training datafile until it achieves a given accuracy on a given reference datafile. The accuracy of the ANN is calculated after each training epoch by calling the function `testNet` described above. Together with the MSE, the accuracy for every training epoch is saved to a text file for later analysis. The source code of the function is shown in figure 5.4.

5.2.2. Accuracy Tracking over Number of Hidden Neurons

To track the development of the accuracy of an ANN over the number of its hidden neurons, the function `accOverHidden` was implemented. It computes the accuracy for a number of

$$N_{ANNs} = \frac{nHiddenMax - nHiddenMin}{step} \tag{5.1}$$

ANNs in the interval of [*nHiddenMin*, *nHiddenMax*] hidden neurons with a step width of *step*. Interval and step width are parameters of the function. To do so, iteratively ANNs with [*nHiddenMin*, ..., *nHiddenMax*] hidden neurons are trained for 100 epochs. The accuracy is calculated after every training epoch by calling the function `testNet` described in the previous subsection. The accuracy maximum together with the training epoch it occurred in are saved to a given results file for every number of hidden neurons. The source code of this function is given in figure 5.5. Since the execution of this function can take quite long depending on the interval and step width chosen, the function `accOverHiddenWrapper` was implemented. It allows to queue calls of `accOverHidden` for different layouts, i.e. numbers of input neurons. This allowed to get all desired analysis done by letting `accOverHiddenWrapper` run over night.

5.2.3. Accuracy Tracking over Number of Training Datasets

The function `datasetAnalysis` was implemented to track the development of the accuracy of an ANN of a given layout over the number of training datasets it is trained on. It takes the number of files with different numbers of datasets as parameter. The files have to be named "ds-[number].data". They should consist of numbers of training datasets for every ANN output out of an interval that is to be analysed. E.g. if one wants to analyse the development over [1,...,15] datasets, the files ["ds-1.data",...,"ds-15.data"] should consist of [1,...,15] training data pairs in the FANN format. 50 ANNs are then trained for every training file and the mean accuracy for every training file is saved to a results file for later analysis. Figure 5.6 shows the source code of the function.

```
void trainAnalysis(char *reference_file, char *train_file, char *
    net_name, char *result_file, int n_hidden, float
    accuracy_desired){
  float acc, mse;
  char tmp[25];
  const float desired_error = (const float) 0.00001;
  const unsigned int max_epochs = 50;
  const unsigned int epochs_between_reports = 1;

  struct fann_train_data *data = fann_read_train_from_file(
      train_file);

  unsigned int layers[3] = {data->num_input, n_hidden, data->
      num_output};
  struct fann *ann = fann_create_standard_array(3, layers);
  FILE *file = fopen(result_file, "w");
  //scale training data to values to [0,1] and shuffle it
  fann_scale_input_train_data(data, 0,1);
  fann_shuffle_train_data(data);
  //set activation functions
  fann_set_activation_function_hidden(ann, FANN_SIGMOID_SYMMETRIC
      );
  fann_set_activation_function_output(ann, FANN_SIGMOID_SYMMETRIC
      );

  //train ANN
  for(int i=0; i<max_epochs; i++){
    fann_train_on_data(ann, data, 1, epochs_between_reports,
        desired_error);
    mse = fann_get_MSE(ann);
    testNet(reference_file, ann, &acc);
    sprintf(tmp, "%lf\t%lf\n", mse, acc);
    fputs(tmp, file);
    if (accuracy_desired < acc){
      break;
    }
  }
  //save
  fann_save(ann, netName);
  fann_destroy(ann);
  fann_save_train(data,"scaled_data.data");
  fann_destroy_train(data);
  fclose(file);
}
```

Figure 5.4.: Source code of the analysis function `trainAnalysis` that tracks the development of the MSE and the accuracy of a given ANN over the number of training epochs and saves the results to a given file.

```
void accOverHidden(char *reference_file, char *train_file, int
    nHiddenMin, int nHiddenMax, int step, FILE *results){
  float acc;
  const float desired_error = (const float) 0.00001;
  const unsigned int max_epochs = 100;
  const unsigned int epochs_between_reports = 0; //0: no reports
      printed by fann API
  struct fann_train_data *data = fann_read_train_from_file(
      train_file);
  *max = 0;
  for(int nHidden=nHiddenMin; nHidden<=nHiddenMax; nHidden=
      nHidden+step){
    unsigned int layers[3] = {data->num_input, nHidden, data->
        num_output}; //layers[numberOfLayers] = {inputNeurons,
        hiddenNeuronsLayer1...N, outputNeurons}
    struct fann *ann = fann_create_standard_array(3, layers);
    //scale training data to values bt. [0,1] and shuffle it
    fann_scale_input_train_data(data, 0,1);
    fann_shuffle_train_data(data);
    //set activation functions
    fann_set_activation_function_hidden(ann,
        FANN_SIGMOID_SYMMETRIC);
    fann_set_activation_function_output(ann,
        FANN_SIGMOID_SYMMETRIC);

    //train ANN
    for(int i=0; i<max_epochs; i++){
      fann_train_on_data(ann, data, 1, epochs_between_reports,
          desired_error);
      testNet(reference_file, ann, &acc);
      if(acc>*max){
        max = acc;
        epoch = i;
      }
    }

    fprintf(results,"%lf␣%lf␣%d\n", max, epoch);
    fann_destroy(ann);
  }
  fann_destroy_train(data);
}
```

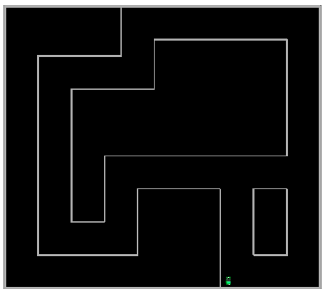
Figure 5.5.: Source code of the analysis function `accOverHidden` that tracks the development of the accuracy of a given ANN over the number of hidden neurons it consists of and saves the results to a given file.

```
void datasetAnalysis(int n, char* reference_data, char* results){
  FILE *results = fopen(results, "w");
  float accuracy, mean;
  struct fann *ann;
  char file_name[30];

  for(int k=0; k<=n; k++){
    sprintf(file_name,"ds-%d.data", k);
    mean = 0;
    for(int i = 0; i<50; i++){
      trainAnalysis( reference_data, file_name, "fann.net", "
         accuracy31.txt", 30);
      ann = fann_create_from_file("fann.net");
      testNet( reference_data,  ann,  &accuracy );

      mean = mean+accuracy;
      accuracy = 0;
      fann_destroy(ann);
    }

    mean = mean/50;
    fprintf(results,"%lf\n", mean);

  }
  fclose(results);
 }
```

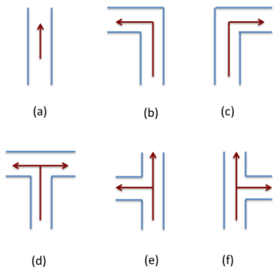
Figure 5.6.: Source code of the analysis function `datasetAnalysis` that tracks the development of the accuracy of a given ANN over the number training datasets it is trained on and saves the results to a given file.

6. Maze Navigation

In the course of the first experiment, autonomous navigation in a maze like the one shown in image 6.1a was implemented. The outputs of the ANN for this experiment were defined as the six situations shown in figure 6.1b. The navigation strategy is straightforward: each situation is interpreted as a state defined by the paths available to the robot and a movement command is associated with every state. The output of the ANN, which is constantly evaluated, determines the state. The first section of this chapter describes the layout and the training process determined to be suitable for the ANN. The second section presents and discusses the implementation of the navigation on the EyeBot robots.



(a) Test maze



(b) Situations of possible movement directions recognised by the ANN.

Figure 6.1.: Test maze for the robot and situations recognised by the ANN. The situations in **(b)** are: **(a)** straight, **(b)** left corner, **(c)** right corner, **(d)** left or right corner, **(e)** left or straight, **(f)** right or straight.

6.1. ANN Layout Design and Training

In this section the layout and training of the ANN employed for the autonomous maze navigation scheme is presented. The following choices and issues will be discussed:

- Number of input neurons[1]

[1]Four different numbers of input neurons have been examined: 31, 62, 341 and 682. 682 is the total number

- Number of hidden neurons
- Training
- Number of training datasets
- Input data manipulation

All data presented in this subsection was gathered in the EyeSim simulator.

6.1.1. Number of Input Neurons

One of the first choices when designing an ANN layout is the number of input neurons. The input data for our ANN are the distance measurement points provided by the laser sensor. Therefore, at first, all 682 measurement points were used. However, as can be seen in the graphs shown in figure 6.2, a higher number of inputs doesn't necessarily mean that a higher accuracy in recognising reference data is achieved. The accuracy in the shown graphs is even slightly better for 62 input values as compared to 341 or 682. Results do, however, fluctuate with every training so the differences in accuracy in figure 6.2 for the different numbers of input neurons are negligible. The data for the graph was produced by the function `trainAnalysis`.

The main conclusion that can be drawn from those graphs is, that the accuracy is just as good for lower numbers of inputs as it is for the full 682. Since training and evaluating ANNs gets harder and takes longer with growing numbers of neurons, economic choices should always be preferred. Therefore 62 was chosen as the number of input values. Having 70, 50 or 62 input neurons would make very little difference. The arbitrary number of 62 was chosen out of the convenience of it being an integer factor of 682. The reduced number of data points was achieved by computing the mean value of eleven measurement points[2].

6.1.2. Number of Hidden Neurons

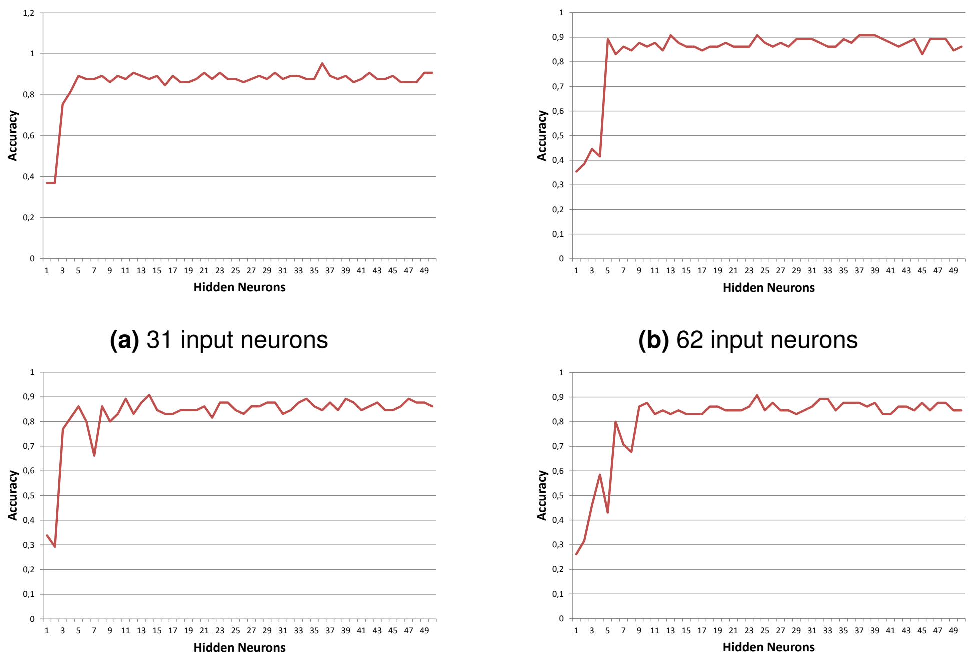
The numbers of hidden layers and of neurons per hidden layer affect the time it takes to train and run a neural network as well as the complexity of problems it can solve. Generally it is more difficult to train bigger, and therefore more complex, networks than it is to train small ones. Choosing those numbers is therefore an important design decision and an economic choice should be preferred here as well. The choice of the number of hidden layers is rather trivial in our case. Networks consisting only of an input and an output layer can only solve linear problems. Thus, since the given navigation problem is clearly non-linear, at least one hidden layer is necessary. On the other hand, most non-linear problems,

of measurement points provided by the employed laser distance sensor. The integer factors of 682 have been chosen as suitable additional values.

[2]Or in the case of 341 input neurons of every second, for 31 input neurons every 22.

including handwriting recognition, can be solved by ANNs with only a single hidden layer. Since the problem of recognising handwriting is very similar to our situation recognition problem, a layout with one hidden layer was chosen and proved to be sufficient.

As to the number of hidden neurons in the hidden layer, a series of experiments was conducted. The accuracy of ANNs with different numbers of input neurons was evaluated after 100 training epochs for hidden neuron numbers between one and 49. The function `accOverHidden` was employed to carry out this analysis. It was found that a rather low number of hidden neurons is enough to produce ANNs with good accuracy. As can be seen in figure 6.2, the accuracy of ANNs reaches its maximum rather quickly. For the number of input neurons decided upon in the previous section, 62, the accuracy maximum is reached at about nine hidden neurons. To have some reserve, a number of hidden neurons of 30 was selected. Training the resulting ANN of 62 input and 30 hidden neurons gave good results and the surplus hidden neurons should leave plenty of room if one would for example want to add further situations to be recognised by the ANN.



(a) 31 input neurons

(b) 62 input neurons

(c) 341 input neurons

(d) 682 input neurons

Figure 6.2.: Plots of the development of the accuracy of the trained ANNs after 100 training epochs over the number of hidden neurons. ANNs with 31, 62, 341 and 682 input neurons were examined.

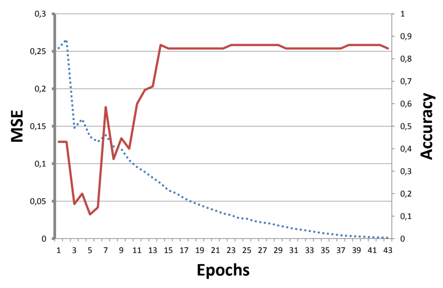
6.1.3. Training

When training an ANN the goal is to achieve the maximum accuracy possible for the respective layout while preventing the ANN from overfitting. To examine the phenomenon of overfitting, the development of the mean square error (MSE) and the accuracy of ANNs with different numbers of input neurons was compared. To do so, accuracy and MSE were evaluated after each epoch during training with the help of the function `trainAnalysis`. The resulting graphs are shown in figure 6.3. During the first few training epochs strong oscillations can be observed in the plots. This is due to the untrained state of the ANNs which causes almost random outputs. Subsequently, MSE and accuracy are randomly oscillating as well. As can be seen in all four cases, accuracy stagnates after a certain amount of training epochs while the mean square error still decreases. Every epoch after the accuracy reached its maximum is futile and only leads to an unnecessarily good fitting to the data the ANN is trained on and a loss of its generalisation power. This loss of generalisation power could be observed when training ANNs with very low MSE stopping values. The ANN would then perfectly recognise the exact training data but moving the robot by only a few centimetres would result in different (and wrong) outputs.

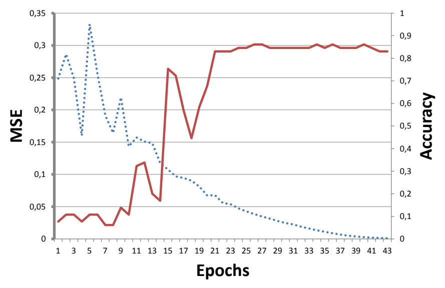
Out of the box, the FANN library only provides MSE and maximum training epoch number as stopping values for training. To effectively prevent the ANNs from overfitting, a function to track the development of the accuracy of the ANN in training was implemented. After each training epoch, the training was interrupted to evaluate the ANN for a set of reference data different from the training data. The amount of correctly recognised reference data pairs was counted to compute the accuracy. The training was stopped when a given accuracy was reached. The accuracy stopping value was chosen according to the results of the previous examinations. Usually a value of 85% was chosen. This proved to be an effective method in combating overfitting while still getting ANNs with accuracy close to the possible maximum.

6.1.4. Number of Training Datasets

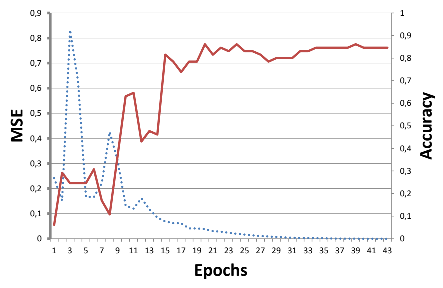
One of the biggest workloads when implementing simple ANNs like the ones described here is gathering training data. It is therefore of economic interest to know how many datasets are necessary to achieve a certain accuracy. To examine this the function `datasetAnalysis` described earlier was implemented. To execute this analysis a number of datasets is of course necessary. Economic benefit can, however, be achieved when for example adding new output neurons to an existing ANN or implementing a new ANN of comparable complexity. The results of this analysis can then be used as benchmarks as to how many datasets are required.



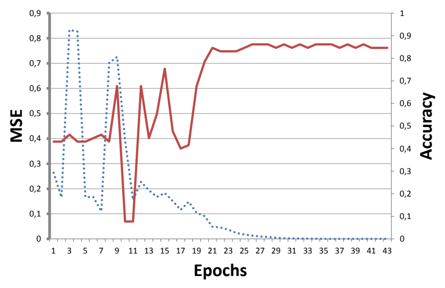
(a) 31 input neurons



(b) 62 input neurons



(c) 341 input neurons



(d) 682 input neurons

Figure 6.3.: Plots of the development of mean square error (blue) and accuracy (red) of the trained ANNs over the number of training epochs. ANNs with 31, 62, 341 and 682 input neurons were examined.

Figure 6.4 shows the results yielded by `datasetAnalysis` for the ANN with 62 input neurons, 30 hidden neurons and 6 output neurons that was determined to be suitable for the maze navigation problem. As can be observed, the accuracy of the ANN rises steeply until a number of seven datasets is used for its training. The accuracy gain for more training datasets beyond that is rather small. It can therefore be concluded that a number of about 10 training datasets should be sufficient to train ANNs of similar complexity for a output.

6.1.5. Input Data Manipulation

Scaling the input and training data can improve ANN performance as well as its behaviour during training. Scaling the input data to values between [0,1] for our case has decreased the number of training epochs required to reach a desired mean square error and therefore decreased the training time. The FANN function `FANN_scale_input` was employed to scale the inputs. When scaling the input data for training, it has to be scaled to the same interval when actually running the trained ANN, in order to produce usable results.

Furthermore, all distance values greater than one meter were set to zero. For one, this makes gathering data much easier, since only the close proximity is evaluated. For example what comes after a corner is irrelevant. Only snapshots of a left corner are required, without the necessity of snapshots of all possible situations beyond that corner.

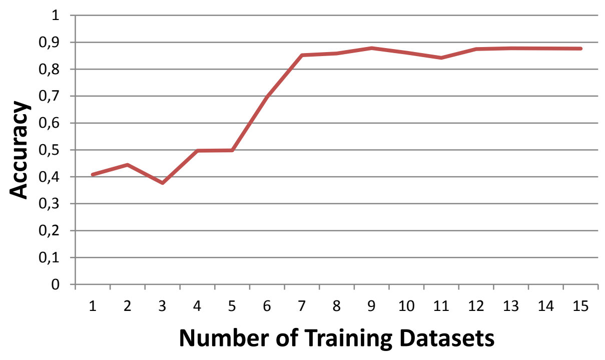


Figure 6.4.: Development of an ANN with 62 input, 30 hidden and 6 output neurons over the number of datasets per output neuron used as training data.

On the other hand it also improved the accuracy of the ANN since noise produced by further away path conditions, irrelevant to the current situation, is reduced. Of course this only makes sense in scenarios like a maze where further away situations are mostly unimportant.

6.2. Application on the EyeBot Robot

In this section, the implementation of the maze navigation experiment on the EyeBot robots is described. Based on the results of the previous subsections, an ANN with 62 input neurons and 30 hidden neurons was chosen for the implementation of the autonomous maze navigation on the EyeBot robot. The ANN was trained on data gathered by the Hokuyo laser sensor and which was scaled and shuffled as described above. The navigation process was implemented as a state machine, where the states were the paths available to the robot at a given time. Based on the ANN output for the current sensor readings one of the six states shown in figure 6.1b was chosen. Each state was associated with a movement command as follows:

- Straight $\rightarrow$ go straight
- Left $\rightarrow$ go around left corner
- Right $\rightarrow$ go around right corner
- Left or right $\rightarrow$ if way point is set, go in that direction; else user decides

- Left or straight $\rightarrow$ if way point is set, go in that direction; else user decides
- Right or straight $\rightarrow$ if way point is set, go in that direction; else user decides

After the movement command was executed, the ANN was evaluated again and a new iteration began. Figure 6.5 shows the flowchart of the navigation process which was implemented in the function `mazeNav`. A list of the functions implemented for data gathering and navigation on the physical robot for this experiment can be found in table 4.2[3].

6.2.1. Movement Commands

Going straight was implemented as the function `piStraight` by employing a PI controller to keep the robot in the middle of the maze path. The controller is described by the following equations:

$$width = distance_{left} + distance_{right} \tag{6.1}$$

$$e = (width/2) - distance_{left} \tag{6.2}$$

$$\Delta_{\phi} = \phi - \phi_{desired} \tag{6.3}$$

$$u = \Delta_{\phi} + (p \cdot e + i \cdot sum) \tag{6.4}$$

$$sum = sum + e \tag{6.5}$$

Where $distance_{left/right}$ is the distance to the wall 90 degrees to either side gathered by the laser sensor. ϕ is the rotational angle of the vehicle relative to its starting position and was provided by the vehicle's encoder via the EyeBot function *VWGetPosition*. $\phi_{desired}$ is the rotational angle the vehicle should have in order to go straight parallel to the pathway walls. It was assumed that the robot starts parallel to the walls. Since $\phi = 0$ at the start, $\phi_{desired,1} = 0$. Then $\phi_{desired}$ was incremented by $+/-90$deg[4] after a left/right corner. Using the ϕ measurements for the control helped prevent the robot from oscillating around the middle of the pathway and therefore made the control smoother. As can be seen from the equation 6.2, the error is zero when the distance from the left-hand side wall is exactly half the total width of the pathway. The robot is therefore controlled to stay in the middle. The current error times the *p* parameter and the sum of all previous errors times the *i* parameter plus the difference between the robot's ϕ and $\phi_{desired}$ yields the control variable *u*. The control parameters *p* and *i* were determined experimentally. The control loop was implemented as a function that first reads the current sensor values and computes *u*. Then the EyeBot function `VWCurve(int dist, int angle, int speed)` is called with *u* as the angle and fixed values for speed and linear distance. Further, the integration sum can be reset by setting the `piStraight's reset` parameter to one. This functionality is necessary because after a corner is taken the position of the vehicle relative to the wall

[3]For training analysis, the functions from table 4.1 were reused.

[4]The angle provided by `VWGetPosition` actually runs from [0,180] and then [-179,0]. This had to be taken into consideration when incrementing $\phi_{desired}$.

changes, which makes the previous integration sum obsolete. With the control parameters $p = 0.25$ and $i = 0.015$ the function reliably kept the robot in the middle.

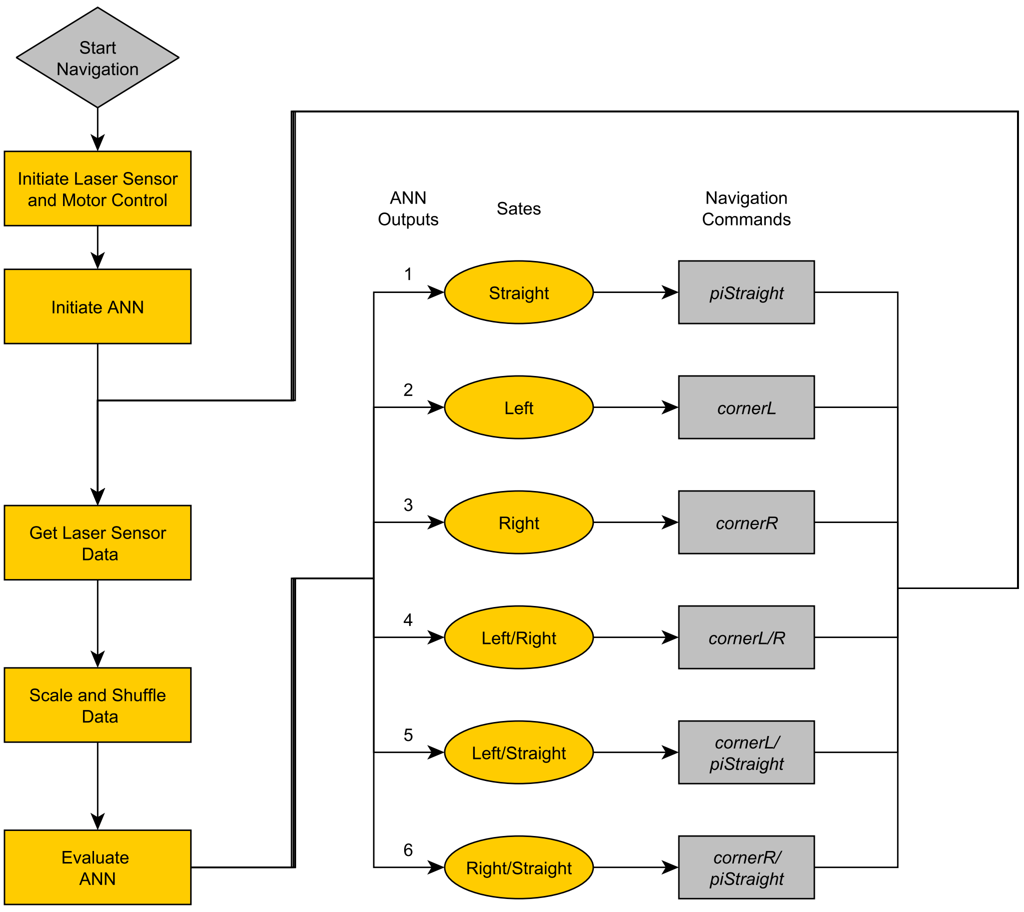


Figure 6.5.: Flowchart of the navigation process `mazeNav` of experiment 1 (maze navigation). First communication with the laser sensor, the ANN and the motor control are initiated. Then, the ANN is evaluated continuously with the current laser distance sensor readings as input. Based on its output, one of six states is chosen. Every state represents a set of paths available to the robot. Associated with every state is a movement command. After the execution of the associated movement command of the chosen state, the next iteration begins and the ANN is evaluated again.

Corners were taken by going first straight, i.e. calling *piStraight*, until a

$$\Delta = distance_{left/right} - distance_{old} > thr \tag{6.6}$$

is detected. Where $distance_{left/right}$ is the distance from the wall 90 degrees to the left (or right if going around a right corner) and $distance_{old}$ the value of the last iteration. The

Function Name	**Description**
`open_urg_sensor`	Establishes connection with sensor at USB port.
`LaserGet`	Returns current laser sensor values.
`LaserPlot`	Plots current laser sensor values in polar coordinates.
`appendData2file`	Generates data file in FANN format. Decision for desired output to current sensor readings. Both are saved to the generated file.
`mazeNav`	Evaluates ANN in a loop and executes movement commands associated with ANN outputs.
`piStraight`	Controls vehicle distance from left-hand side wall to keep it close to the middle of maze pathway. *reset* = 1 resets integration sum.
`cornerR/cornerL`	Navigates robot around right or left maze corner.
`runNet`	Evaluates ANN for current sensor readings and returns index of largest output value.
`largestMember`	Returns index of largest member of a <fann_type> array.

Table 6.1.: List of functions for data gathering and navigation on physical robot in experiment 1.

threshold value *thr* was set to 5 cm. Then *VWCurve* is called with 90 degrees as the angle. This was implemented in the functions *CornerR* and *CornerL*. When one of the crossroads was detected, first either a decision screen was shown to the user, or if way points were set, the direction to the next way point was calculated. Based on that either the respective corner function or *piStraight* was called.

6.2.2. Performance

Figure 6.6 shows plots of the development of the accuracy of ANNs[5] trained on data gathered in the simulator and by the physical laser sensor. The plot data was produced by the function `trainAnalysis`. The reference data used for accuracy calculation was gathered by the physical sensor and consisted of 60 data pairs, ten pairs per output neuron. Both sets of training data consisted of 90 data pairs, 15 per output neuron. The ANN trained on data gathered by the sensor shows decent accuracy values of about 80%. The one trained on simulation data on the other hand does not come close to any reasonable performance. The dominating reason is very likely the elevated noise in the sensor which is not being simulated. Additionally, the metal bars connecting the wall pieces used for the

[5]Both ANNs had 62 input neurons and 30 hidden neurons. The training and reference data was processed as described in the previous chapter.

experimental set-up cause very peculiar sensor readings. Since the simulation of the laser sensor is based on PSD sensors, it is not too surprising that the resulting values are not comparable with values actually produced by the sensor.

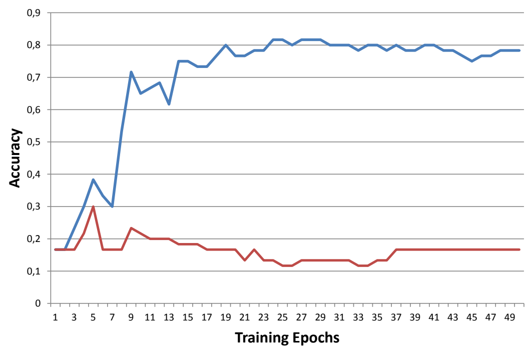


Figure 6.6.: Graphs of the development of the accuracy over 50 training epochs for training data gathered by the physical sensor (blue) and in the simulation (red). Reference data for accuracy calculation in this case was gathered with the physical laser sensor. The ANN trained on simulation data shows very bad results. The one trained on physical data shows reasonable accuracy values of about 80%.

6.3. Results

The most suitable layout found for the described navigation problem, was a network with 62 input neurons. Training was stopped after 30 training epochs. The training and input data were scaled to values in the interval of [0,1] to ensure effective training. In addition, distance values over one meter were set to zero to simplify training and reference data gathering. The resulting ANNs performed well in the EyeSim simulator when trained on data gathered in the simulator and on the physical robot when trained on data from the Hokuyo sensor. The 80% accuracy achieved by the latter sometimes resulted in right corners falsely detected as "right corner or straight", analogously for left corners. While this is not a safety issue, because the user could simply choose the correct movement command in this particular situation, it could still be improved upon. For one, a larger set of training data would certainly result in a higher accuracy. In addition, the results of the ANN could be filtered by a decision tree. For example when "right corner or straight" was falsely detected in a right corner, the ANN output value for "right corner" was still quite high, usually greater than 50%. So by not simply choosing the highest output value but taking into consideration the other output values as well, performance might be further enhanced. As to the training of the ANN, it was important not to train the ANN for too many epochs to prevent overfitting. This could be achieved by stopping when the network accuracy ceases to improve. In the case of the ANN described above, this would be after about 30 training epochs. The network resulting from this layout and training strategy performed well on different test mazes.

In conclusion, the maze navigation problem was adequately solved with the help of an ANN. By conducting the examinations described in this section a layout was found, that with

$$62 + 30 + 6 = 98$$

neurons had considerably fewer neurons than any layout with the full 682 distance measurement points as input. This results in a simpler[6] and quicker training process. Autonomous maze navigation can be solved by simpler means, e.g. a wall-following algorithm. It was, however, an ideal problem to gather experience in the training of laser sensor based ANNs. This experience was put to use in the open-world way-point navigation described in the following section.

[6]In terms of required training data.

7. Way-point Navigation

The problem to be solved in experiment 2 was the autonomous navigation to a way-point in an open-world setting. The navigation developed was based on two ANNs. One to recognise obstacles in the way and one the determine available paths around them. Figure 7.1 shows the objects the training data was gathered on. In image 7.2 a typical test environment for the EyeBot way-point navigation can be seen. The first section of this chapter describes the layout and the training process determined to be suitable for the ANNs. The second section presents and discusses the implementation of the navigation on the EyeBot robots.

7.1. ANN Layout Design and Training

In this section the layout and training of the ANNs employed for the autonomous maze navigation scheme is presented. The following choices and issues will be discussed:

- Number of input[1] and hidden neurons
- Training
- Number of training datasets

All data presented in this section was gathered in the EyeSim simulator. The same input data manipulation procedures as described in the previous chapter were used for this experiment.

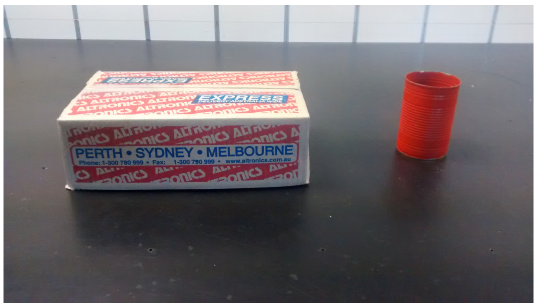


Figure 7.1.: Objects used for ANN training in experiment 2. Left: large object. Right: small object.

[1] Four different numbers of input neurons have been examined: 31, 62, 341 and 682. 682 is the total number of measurement points provided by the employed laser distance sensor. The integer factors of 682 have been chosen as suitable additional values.

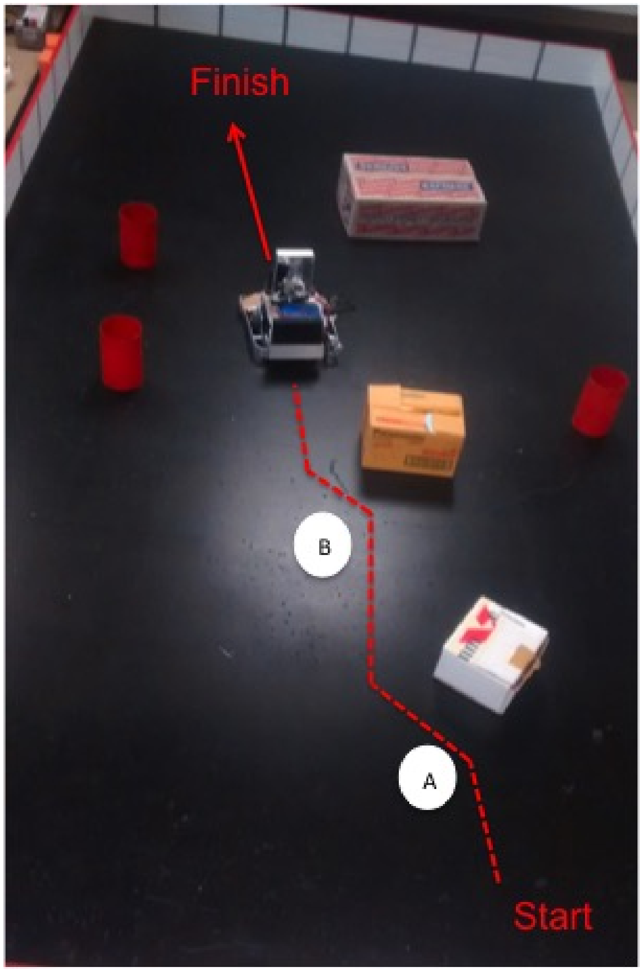


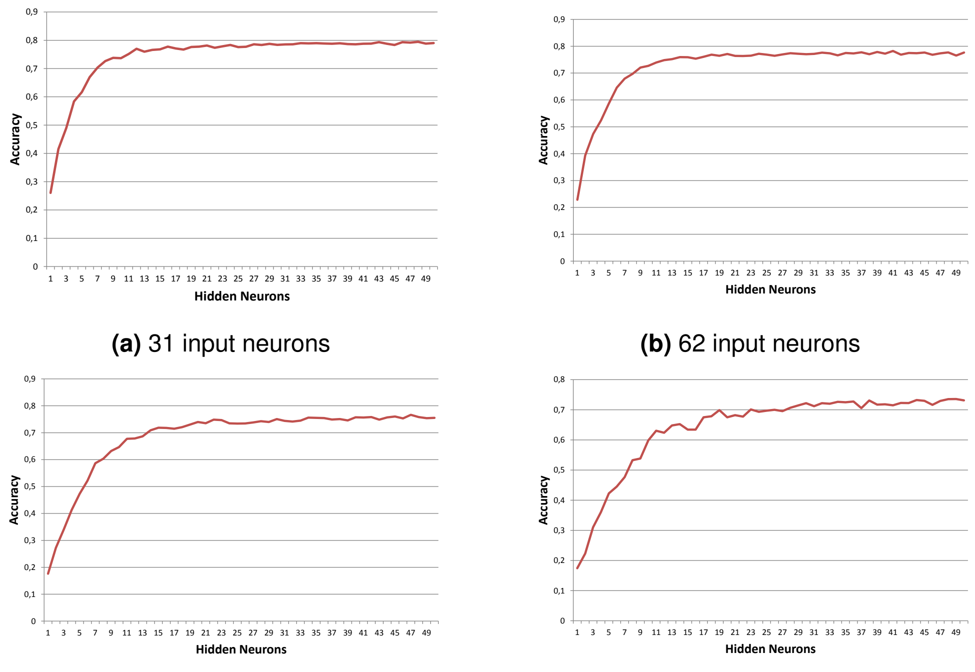
Figure 7.2.: Way-point navigation test environment. At point A a small object was detected and dodged, at point B a large object was detected and dodged.

7.1.1. Number of Hidden and Input Neurons

Since variance between training results for identical layouts was even higher than in experiment one, the maximum accuracy for every number of hidden neurons was averaged over 50 separate training processes with the help of the function `accOverHidden`. This means that

$$50[[1{,}50]\ \text{hidden neurons}] \cdot 50[\text{average}] \cdot 4[\text{input neuron cases}] = 10000 \tag{7.1}$$

ANNs were trained to calculate the results shown in figure 7.3. For more complex ANNs this would not be feasible because one training process alone can take several hours or more. In the case of the ANNs examined here, however, the whole process only took about two hours and the data was significantly improved by this procedure. The same overall trend of approaching a maximum value as in figure 6.2 can be observed. Maximum values here lie between an accuracy of about 70% for 682 input neurons and 80% for 31 input neurons. While the layouts with 31 and 62 input neurons show little improvement in accuracy for more than 15 neurons in the hidden layer, the layout with 341 input neurons shows an increase in maximum accuracy of about 5% for 30 as opposed to only 15 hidden neurons. The ANNs with 682 not only showed the weakest accuracy maxima but also required the highest amount of hidden neurons to reach them. For 35 hidden neurons an accuracy maximum of about 72% was achieved.



(a) 31 input neurons

(b) 62 input neurons

(c) 341 input neurons

(d) 682 input neurons

Figure 7.3.: Plots of the development of the accuracy after 100 training epochs over the number of hidden neurons. 31, 62, 341 and 682 input neurons examined.

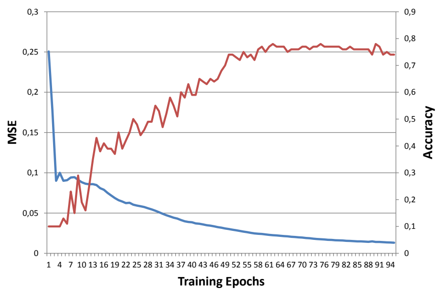
As in experiment one, it can therefore be concluded that a lower number of input neurons does not cause a loss in accuracy. In this case the layouts with 31 and 62 input neurons even show a significantly better performance as the ones with 341 and 682. Since they reach their accuracy maxima for lower numbers of hidden neurons as well, the logical choice for the layout was either one with 31 or 62 input neurons. To have some room for potential future expansions in complexity, the layout with 62 input neurons and 30 hidden neurons was determined to be used for the implementation on the EyeBot. As experiment one showed, the Hokuyo laser sensor generates considerable noise which this overhead in neurons should be able to cope with, as it did in experiment one.

Another finding that could be made due to the averaging over 50 ANNs per number of hidden neuron was, that the ANNs reached their maximum accuracy after about 70 training epochs in average. The number of training epochs it took to reach the maximum was tracked in experiment one as well. However, the number fluctuated too strongly to be of any use. This value was used in further training processes as an orientation for when to stop the training.

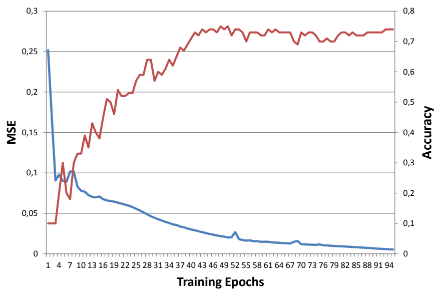
Since ANN2 achieved exceptional accuracy with the simplest convenient layout of 31 input neurons and five hidden neurons as described in the next subsection, its layout shall not be further discussed here.

7.1.2. Training

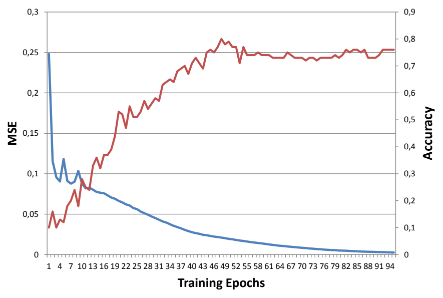
Image 7.4 shows plots of the development of the MSE and the accuracy over the number of training epochs produced by the function `trainAnalysis` for ANN1. Like in experiment one, layouts with 31, 62, 341 and 682 input neurons were compared. Again, like in figure 6.3, the MSE continued to decrease after the respective accuracy maxima were reached.



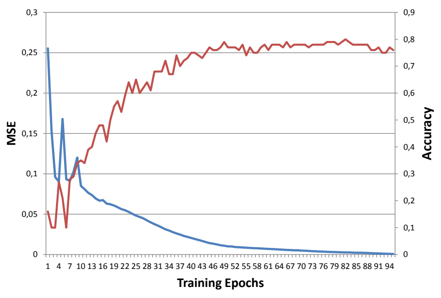
(a) 31 input neurons



(b) 62 input neurons



(c) 341 input neurons



(d) 682 input neurons

Figure 7.4.: Experiment 2: Plots of the development of mean square error (blue) and accuracy (red) over the number of training epochs for ANN1. 31, 62, 341 and 682 input neurons examined.

Above, the number of 70 training epochs was found to be when ANNs of those layouts in average reach their maximum accuracy. Given, that the plots in image 7.4 only show the development of one single ANN per number of input neurons, this seems like a reasonable result.

The training process developed in experiment one was again used for the ANNs for the implementation of experiment two on the EyeBots. I.e. training until a given accuracy is reached or until accuracy ceases to improve. In addition a maximum number of 100 epochs was introduced. If the desired accuracy wasn't reached after this amount of training epochs, a new training process was started.

The plot produced by `trainAnalysis` for ANN2 with 31 input neurons and five hidden neurons shown in 7.5 is a little curious. That is, it actually achieved an accuracy of 100% on a reference dataset of 15 data pairs for each output. Since this result seemed suspicious, the analysis was repeated several times. However, the results were always similar. Testing the ANN for data other than the reference data confirmed its exceptional accuracy. As the plot shows, this accuracy was reach after only 6 training epochs. ANN2 for the implementation on the EyeBot was therefore trained with an accuracy of 100% as stopping value.

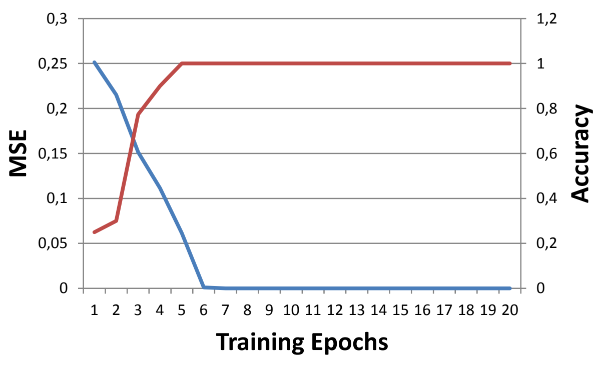


Figure 7.5.: Experiment 2: Plots of the development of mean square error (blue) and accuracy (red) over the number of training epochs for ANN2.

7.1.3. Number of Training Datasets

In figure 7.6 the plot of the development of the accuracy of ANN1 over the number of training datasets used can be seen. The layout with 62 input neurons and 30 hidden neurons determined in the previous subsections was used as input for the function `datasetAnalysis` to gain this data. As can be seen, the accuracy increases rapidly until ten datasets used for training. From then on it keeps increasing, but at a slow pace. 25 as compared to ten datasets meant an increase in accuracy of about 3%. While in experiment one a number of around 12 datasets would have been deemed sufficient for a plot like this, demands to ANN1's accuracy were a little higher, as described in the following subsection. This is why here a number of 25 datasets was determined to be gathered for the training on physical sensor data.

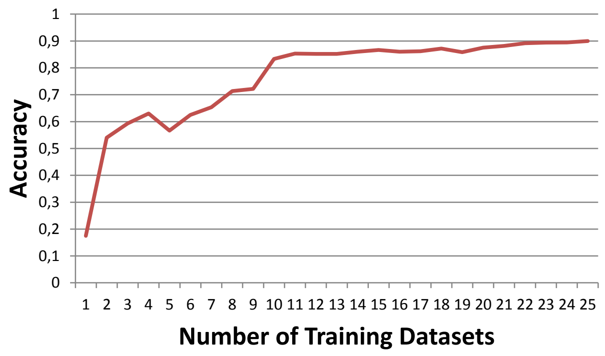


Figure 7.6.: Experiment 2: Development of an ANN with 62 input, 30 hidden and 4 output neurons over the number of datasets per output neuron used as training data.

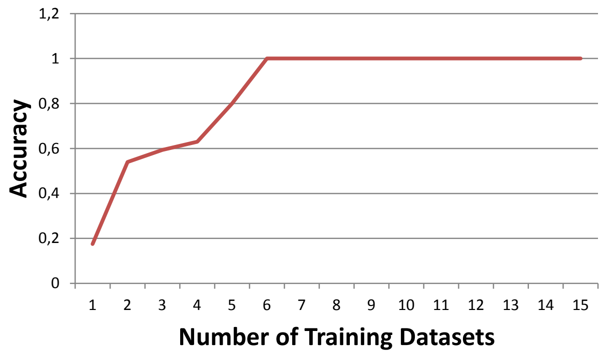


Figure 7.7.: Experiment 2: Development of an ANN with 31 input, 5 hidden and 2 output neurons over the number of datasets per output neuron used as training data.

Figure 7.7 shows the results of the analysis for ANN2. The accuracy maximum of 100% was reached for seven sets of training data. It was concluded that a number of ten training datasets should make sure, that this accuracy is reached reliably.

7.2. Application on the EyeBot Robot

The autonomous open-world way-point navigation was implemented on the EyeBot robots based on two ANNs. The first one was responsible for the recognition of four types of obstacles: walls, corners, small and large objects. When a small or large object was recognised to block the way, the second ANN was used to determine which way around it was available. In the case of a detected corner, the robot was backed up and turned around by the function `corner`. When a wall was found to be in the way, the function `wall` was called to find a way around it. As long as no obstacle was detected, the robot was directly navigated towards the way-point. A suitable layout for the first ANN was found to be 62 input neurons and 30 hidden neurons, based on the results of the previous sections. ANN2 was implemented as an ANN with 31 input, five hidden and two output neurons. Both were trained on data gathered by the Hokuyo laser sensor in the way described in Chapter 5. Figure 7.8 shows the flowchart of the navigation process implemented in the function `wpNav`.

7.2.1. Movement Commands

The function `navigateWoObstacle` was called when no obstacles were detected. It navigates the robot towards the way-point. To do so, it gets the robot's current rotation angle and the way-point coordinates as parameters. The rotational angle was accessed via the EyeBot function `VWGetPosition`. First, the remaining distance is calculated with the help of the Pythagoras Theorem:

$$c = \sqrt{x^2 + y^2} \tag{7.2}$$

With x and y being the way-point coordinates. Then the angle α and the angle the robot had to turn, Δ, are calculated as:

$$\alpha = \arcsin(\frac{b}{c}) \tag{7.3}$$

$$\Delta = \alpha - \phi \tag{7.4}$$

To avoid turning angles greater than 180 degrees, the following adaptation is made:

- if ($\Delta > 180°$): $\Delta = -(360° - \Delta)$
- if ($\Delta <$ -180°): $\Delta = (360° + \Delta)$

The EyeBot function `VWTurn` was then called with Δ as the turning angle. Finally the robot is set into motion by calling `VWSetSpeed`.

When an object was detected in the way of the robot, `avoidObstacle` was called. This function first evaluates ANN2 to determine an available path around the object blocking the way. Based on the output of ANN2 `VWCurve` is called to go around the object either around the right or around the left.

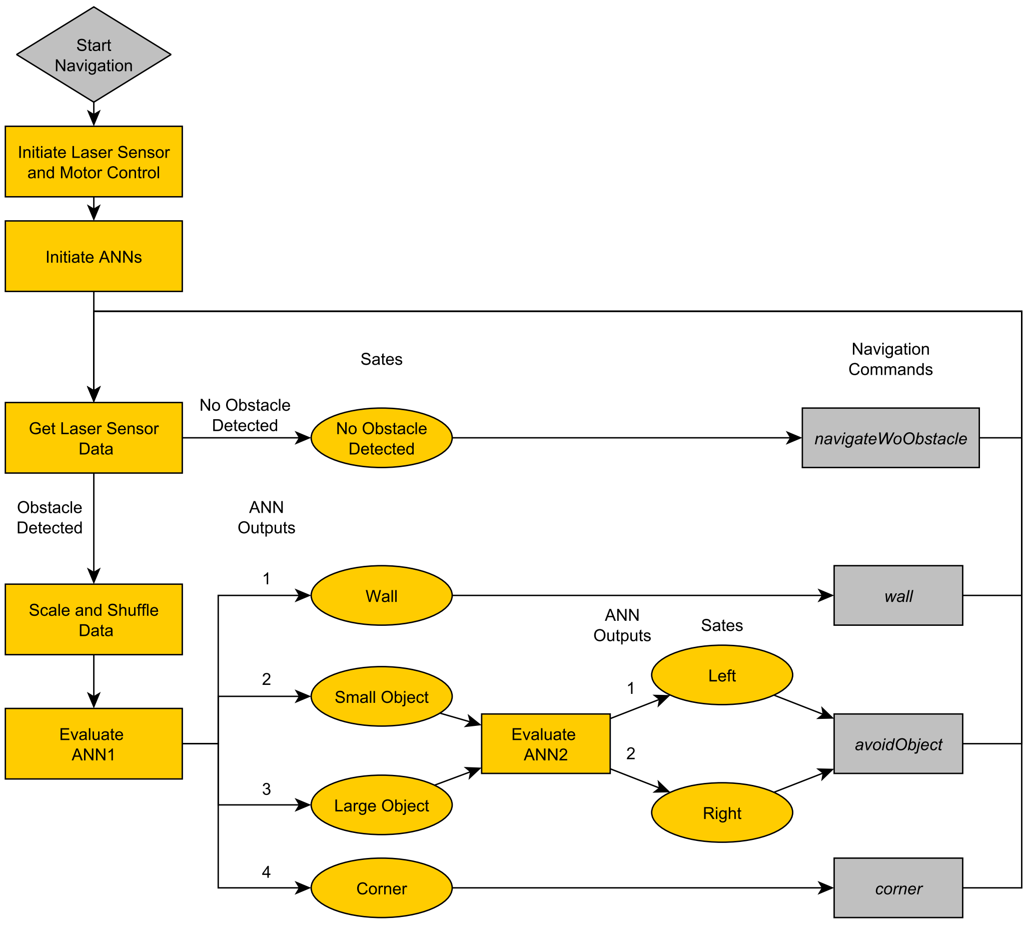
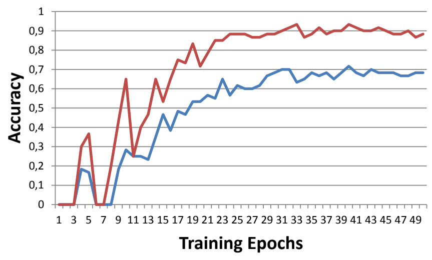


Figure 7.8.: Flowchart of the navigation process of experiment 2 (way-point navigation). First communication with the laser sensor, the ANNs and the motor control are initiated. Then, navigation to the way-point is started and laser sensor readings are continuously checked. If an obstacle is detected (readings < thr) ANN1 is evaluated to determine the kind of obstacle. Based on its output, one of four states is chosen. If the robot faces a wall or is stuck in a corner the respective movement commands are executed. If the robot faces an obstacle, a ANN2 is evaluated to find a way around the object. Then the object is dodged accordingly.

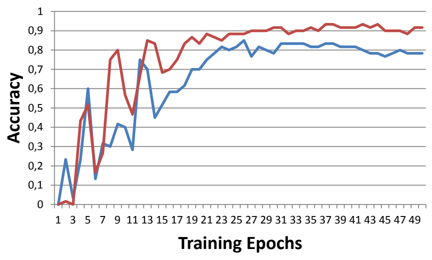
If both a path on the right and on the left is available, the more direct one to the way-point is chosen. The function `wall` was called when a wall was found to be in the robot's way. First, the robot is aligned to the wall in a 90 degree angle, then it searches for a way around the wall. This is done by first going into one direction along the wall. If an opening is found, the robot is steered through it with the help of the functions `cornerR/L` developed in experiment one. If another wall is met, the robot is turned around by 180 degrees with the help of `VWTurn` and an opening is searched for in this direction. If the robot ends up in a corner, the function `corner` is called, which backs him up and turns him around by 180 degrees.

7.2.2. Performance

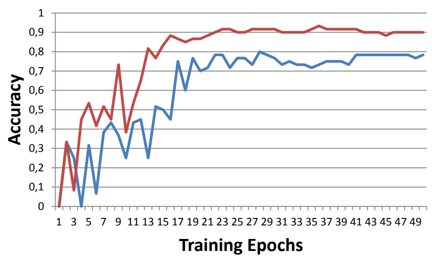
The first set of ANNs trained for ANN1 (see figure 7.8) on data gathered by the laser sensor for this experiment reached accuracies of 70% to 80%. 25 datasets per output were used as determined above. While an accuracy of 80% proved to be sufficient for the maze navigation scheme developed in experiment 1, way-point navigation did not go smooth with these ANNs. To analyse the ANNs performance in more detail, the function `trainingAnalysis` was expanded to count the number of times each individual data pair was falsely recognized during accuracy computation[2]. Like this it was found that the detection of small objects was very inefficient - for all layouts examined, at least 10 out of the 15 reference data pairs for small objects were falsely recognised.



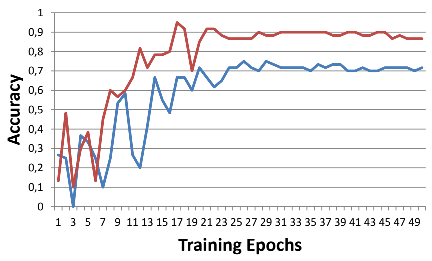
(a) 31 input neurons



(b) 62 input neurons



(c) 341 input neurons



(d) 682 input neurons

Figure 7.9.: Comparison of accuracy development during training with different sets of training data.

To address this issue, the new sets of training data were gathered for the small object. This lead to improvements in accuracy of 10%-20% as can be observed in 7.9. The reason

[2]As opposed to only count the overall number of falsely recognised data pairs.

for the poor results with the first set of training data was very likely sloppy work when collecting the data. This data was collected last out of the training data for all outputs. This resulted in a slightly tired data gatherer. While for all other outputs close attention was paid the get datasets of the object from as many different angles and distances as possible, this was neglected for the small object. The incredible improvement in accuracy of 10%-20% gained by gathering new, better data goes to show how important the quality, not only the quantity, of training data is. However, it is hard to define rules as to what makes good training data. This is something that could be examined more closely in future work.

No performance issues were met with ANN2 (see figure 7.8). The accuracy of 100% achieved proved sufficient to reliably determine an available path around the detected obstacles.

7.3. Results

The most suitable ANN layout found for ANN1 (see figure 7.8) was a network with 62 input neurons and 30 hidden neurons. Training was stopped when a desired accuracy of 90% was reached. This occured in average after about 70 training epochs. The training data was scaled and shuffled as described in the description of experiment one. By conducting the analysis described in the previous section, a layout for ANN1 was found that with

$$62 + 30 + 4 = 96$$

neurons in total is considerably slimmer than the straightforward choice of the full 682 inputs plus hidden and output neurons. On top of that the determined layout even performed better than the larger ANNs. The fact, that an ANN with 62 input and 30 hidden neurons was found to be suitable to solve the maze navigation problem in experiment one is curious. It was interpreted as a hint, that both problems are of very similar computational complexity. After the performance enhancement achieved by expanding the set of training data described in the previous subsection, ANN1 reliably detected the different obstacles. The total amount of neurons in ANN2 of

$$31 + 5 + 2 = 38$$

is also much lower than it would have been with the full set of 682 inputs.

In conclusion, the autonomous navigation implemented around the two ANNs solved the open-world way-point navigation problem satisfyingly. The analysis of the different layouts conducted throughout experiment one and two allowed to find economic and effective ANN layouts to base the autonomous navigation strategies on.

8. Conclusion

In the course of this thesis a process for the development of ANN based autonomous navigation was presented. Special attention was payed to the determination of economical ANN layouts as well as an efficient training process. To achieve this the following methods were proposed:

- Comparison of ANNs with different numbers of input neurons in terms of accuracy
- Comparison of ANNs with different numbers of hidden neurons in terms of accuracy
- Tracking accuracy during training as stopping condition
- Comparison of ANNs trained on different numbers of training datasets

The examination of layouts of different sizes in neurons offers the obvious advantage of finding economic layouts. By implementing the presented analyis functions these methods can be executed efficiently. The generally small size of ANNs necessary to solve similar problems, allows for a thorough analysis. This is due to the comparably short period of time required to train the networks. This allows to train a big number of them and compare them amongst each other. Since, as stated earlier, no general rule for finding suitable ANN layouts is available, this seems a reasonable approach to finding such layouts. The ANNs layouted and trained with the help of these methods were effective in solving the autonomous maze navigation and open-world way-point navigation problems posed. Economical ANN layouts in terms of numbers of neurons were found. This resulted in ANNs that could be trained quickly and required little computational power. Further, overhead in the amount of training data was avoided. The ANN layouts were first tested in the EyeSim simulator before being deployed on the robots. The reason for that is the gathering of data which was much simpler and much less time consuming in the simulator than with the EyeBot robots. Together with the analysis of the required amount of training data, this saved much time and allowed for the testing of many different ANN configurations. Because of the time saved by gathering data in the simulator the total workload was therefore actually decreased by the double implementation in the simulator and on the physical robots.

Use cases for autonomous navigation strategies based on ANNs of similar complexity as those examined in the course of this thesis would naturally be such with a limited available computational power. This would be for example small autonomous robots. A limiting factor is the environment in which the data provided by a laser sensor as the one used here provides sufficient data for autonomous navigation. For one, the sensor only provides data of a plain area about 7 cm above the ground. This means obstacles smaller than 7 cm cannot be detected. Also if one would want to use the sensor to guide along a wall, the wall would necessarily have to be at least 7 cm high. Environments suitable for the deployment of autonomous navigation strategies similar to those described in this thesis are structured spaces. Such spaces can be found for example in indoor environments. Apart from surveillance robots, autonomous cleaning and maintenance robots could be navigated based on ANN/sensor calibrations developed in this thesis. Further, the results of this thesis are planned to serve as the basis for an ANN based road following alogrithm for a autonomous race car developed at the University of Western Australia.

Based on the layout and training methods presented in this thesis ANNs for many different scenarios can be designed in a methodological and efficient manner. Another aspect that should be fruitful to examine is the quality and composition of the training data necessary to train ANNs with certain characteristics. Since gathering training data is one of the biggest workloads when training ANNs for new use cases, knowing exactly how much and what kind of data one needs to train the desired network can be of great economical benefit.

Bibliography

1. Atmel. Atxmega128a1u data-sheet. http://www.atmel.com/images/Atmel-8385-8-and-16-bit-AVR-Microcontroller-ATxmega64A1U-ATxmega128A1U_datasheet.pdf. [Online; accessed 8-February-2017].

2. T. Bräunl. EyeBot API. http://robotics.ee.uwa.edu.au/eyebot7/doxygen/html/index.html, . [Online; accessed 11-November-2016].

3. T. Bräunl. EyeBot Functions. http://robotics.ee.uwa.edu.au/eyebot7/Robios7.html, . [Online; accessed 30-January-2017].

4. T. Bräunl. EyeSim Simulator. http://robotics.ee.uwa.edu.au/eyebot/doc/sim/sim.html, . [Online; accessed 11-November-2016].

5. T. Bräunl. *Embedded Robotics*. Springer-Verlag Berlin Heidelberg, 2003.

6. D.S.O. Correa, D.F. Sciotti, M.G. Prado, D.O. Sales, D.F. Wolf, and F.S. Osorio. Mobile robots navigation in indoor environments using kinect sensor. In *Critical Embedded Systems (CBSEC), 2012 Second Brazilian Conference on*, pp. 36–41. IEEE, 2012.

7. T. Dash, S.R. Sahu, T. Nayak, and G. Mishra. Neural network approach to control wall-following robot navigation. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, pp. 1072–1076. IEEE, 2014.

8. fltk.org. Fltk documentation. http://www.fltk.org/index.php. [Online; accessed 11-February-2017].

9. R. Hecht-Nielsen et al.. Theory of the backpropagation neural network. In *Neural Networks*, 1(Supplement-1), pp. 445–448, 1988.

10. L. Hokuyo Automatic Co. URG C Library document. http://urgwidget.sourceforge.net/html/. [Online; accessed 14-January-2017].

11. L. Hokuyo Automatic Co. Scanning Laser Range Finder URG-04LX-UG01 Datasheet. https://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_UG01_spec_en.pdf, 2009. [Online; accessed 11-November-2016].

12. A.C. Ian Goodfellow, Yoshua Bengio. *Deep Learning*. The MIT Press, 2016.

13. C. Igel and M. Hüsken. Improving the rprop learning algorithm. In *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*, volume 2000, pp. 115–121. Citeseer, 2000.

14. M.A. Jaradat, M.N. BaniSalim, and F.H. Awad. Autonomous navigation robot for landmine detection applications. In *2012 8th International Symposium on Mechatronics and its Applications*, pp. 1–5. April 2012. doi:10.1109/ISMA.2012.6215189.

15. Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *arXiv preprint arXiv:1408.5093*, 2014.

16. M. Johnson, M. Schuster, Q.V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado et al.. Google's multilingual neural machine translation system: Enabling zero-shot translation. 2016.

17. Microsoft. Kinect sensor specifications. `https://msdn.microsoft.com/en-us/library/jj131033.aspx`. [Online; accessed 5-March-2017].

18. Y.K. Na and S.Y. Oh. Hybrid control for autonomous mobile robot navigation using neural network based behavior modules and environment classification. In *Autonomous Robots*, 15(2), pp. 193–206, 2003.

19. M.A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

20. S. Nissen. Fast Artificial Neural Network Library. `http://fann.sourceforge.net/fann.pdf`, 2004. [Online; accessed 11-November-2016].

21. OpenCV.org. Opencv documentation. `http://opencv.org/`. [Online; accessed 8-February-2017].

22. D.A. Pomerleau. *ALVINN, an autonomous land vehicle in a neural network*. Technical Report, Carnegie Mellon University, Computer Science Department, 1989.

23. M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference On*, pp. 586–591. IEEE, 1993.

24. P. Sibi, S.A. Jones, and P. Siddarth. Analysis of different activation functions using back propagation neural networks. In *Journal of Theoretical and Applied Information Technology*, 47(3), pp. 1264–1268, 2013.

25. D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot et al.. Mastering the game of go with deep neural networks and tree search. In *Nature*, 529(7587), pp. 484–489, 2016.

26. R. Thirumalainambi and J. Bardina. Training data requirement for a neural network to predict aerodynamic coefficients. In *AeroSense 2003*, pp. 92–103. International Society for Optics and Photonics, 2003.

27. P. Van Turennout, G. Honderd, and L. Van Schelven. Wall-following control of a mobile robot. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pp. 280–285. IEEE, 1992.

28. A. Visioli. *Practical PID Control*. Springer-Verlag London, 2006.

29. K. Öfjäll, M. Felsberg, and A. Robinson. Visual autonomous road following by symbiotic online learning. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pp. 136–143. June 2016. doi:10.1109/IVS.2016.7535377.

A. Appendix

A.1. Abbreveations

ANN	Artificial Neural Network
API	Application Programming Interface
FANN	Fast Artificial Neural Network Library
FLTK	Fast Light Toolkit
FSM	Finite State Machine
GUI	Graphical User Interface
IO	Input/Output
MSE	Mean Square Error
PSD	Position Sensitive Device
USB	Universal Serial Bus

A.2. Installation URG and Fann on Raspberry

Fann:

- Download from `http://leenissen.dk/FANN/wp/download/`
- Unzip and navigate to folder in terminal
- $ cmake .
- $ sudo make install
- adapt gccarm file in /home/pi/eyebot/bin

URG:

- Download from `http://urgwidget.sourceforge.net/html/`
- Unzip and navigate to folder in terminal
- $ make
- $ sudo make install
- adapt gccarm file in /home/pi/eyebot/bin

A.3. Header Files

A.3.1. Simulation

The header file split into the figures A.1, A.2, A.3, A.4, A.5 and A.6 contain the function headers of the functions implemented for the EyeSim simulator.

```
#include "eyebot.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "fann.h"

#define REFNET "ref.net"      //functioning net for reference
   data collection
#define indexR  426 //index of laser sensor value 90 deg to robis
    right
#define indexL  256 //index of laser sensor value 90 deg to robis
    left
#define NMAN 6 //number of neural net output manouvers
#define PI 3.14159265359
#define LROBI 180 //length of the robot
#define THR 1000
#define VLIN 100


/**
Run fann net with current sensor readings as input

The reference data file is used to scale the input data

@param  ann fann net to be run
@param  dataFile reference data file
@return
*/
void runNet(struct fann *ann, char *dataFile);
```

Figure A.1.: Function headers of the functions implemented for the EyeSim simulator (1).

```
/**
Train fann net

Trains the net until desired error is reached

@param  dataFile fann data file to train the net on
@param  netName file name the resulting net is to be saved to (.
   net)
@param  error desired mean square error
@return
*/
void trainNet(char *dataFile, char *netName, double error);
/**
Run net for test data collection

Collects sensor data and saves it to a file

@param  ann fann net to be run
@param  filename file to save collected data to
@return number of collected datasets
*/
int runRefNet(struct fann *ann, char *filename);
/**
Get index of largest member in a <fann_type> array

@param  array array to be searched
@return index of largest member
*/
int largestMember(fann_type* arrray);
/**
Get index of smallest member in a <int> array

@param  array array to be searched
@return index of smallest member
*/
int smallestIntMember(int* array);
/**
Get index of second largest member in a <fann_type> array

@param  array array to be searched
@return index of second largest member
*/
int secondMember(fann_type* array);
/**
Initiate fann data file

@param  filename name of file to be initialised
@return
*/
void initFile(char *filename);
```

Figure A.2.: Function headers of the functions implemented for the EyeSim simulator (2).

```
/**
Add header line with number of datasets to fann data file

Has to be done last and seperately because the number of datasets
    needs
to be known

@param  filename name of file to be finished
@param  n number of datasets in the file
@return
*/
void finishFile(char *filename,int n);
void writeData( FILE *file);
/**
Go around corner and save data

Collects sensor data, writes it to file and updates nData

@param  situation 1 - right corner, 2 - left corner, 3 -
   crossroads T, 4 - crossroads right, 5 crossroads left
@param  dir 1 - right, 2 - left
@param  file file collected sensor data is written to
@param  nData number of datasets
@return
*/
void refCorner(int situation, int dir, FILE *file, int *nData);
/**
navigate directly to next waypoint

assumes no obstacles in the way

@param  i current waypoint
@param  waypoints array containing all waypoints
@return 0 if successful
*/
int navigateWoObstacle(int i, int waypoints[NWP][2]);
/**
Navigate using combination of waypoints and ann

@param  ann fann net to be employed
@param  waypoints array containing all waypoints
@return 1 when all wps have been reached; 0 else
*/
int driveProcess(struct fann* ann, int waypoints[NWP][2]);
```

Figure A.3.: Function headers of the functions implemented for the EyeSim simulator (3).

```
/**
Navigate around a left corner

@param  waypoints array containing all waypoints
@param  wp current waypoint
@return
*/
void goAroundCornerL(int waypoints[NWP][2], int wp);
/**
Navigate around a right corner

@param  waypoints array containing all waypoints
@param  wp current waypoint
@return
*/
void goAroundCornerR(int waypoints[NWP][2], int wp);
/**
Navigates to the middle of the road and align parallel to it

@param
@return
*/
void alignInMiddle();
/**
Align parallel to a wall to the left

@param
@return
*/
void alignLeft();
/**
Align parallel to a wall to the right

@param
@return
*/
void alignRight();
```

Figure A.4.: Function headers of the functions implemented for the EyeSim simulator (4).

```
/**
scales the number of intputs of a fann file by 1/factor

the values in the resulting file are the mean values

@param  fileName fann file to be scaled
@param  newFileName file the results are saved to
@param  factor scaling factor
@return
*/
void scaleFannFile(char *fileName, char *newFileName, int factor)
    ;
/**
trains net and computes its accuracy for every epoch

@param  refFile fann data file with reference data for accuracy
    computation
@param  trainFile fann data file to train the net on
@param  netName name of the (.net)-file the resulting net is
    saved to
@param  resFile accuracy and MSE for every epoch are saved to
    this file (for analysis purposes)
@param  nHidden number of hidden neurons the net is to have
@return
*/
void trainAnalysis(char *refFile, char *trainFile, char *netName,
     char *resFile, int nHidden);
/**
Analyses influence of amount of training data

Trains 50 ANNs for [1...15] datasets per output and saves
    accuracy mean for every
number of datasets to a .txt file

@param
@return
*/
void datasetAnalysis();
```

Figure A.5.: Function headers of the functions implemented for the EyeSim simulator (5).

```
/**
computes accuracy of a fann net

@param  testFile: fann data file with reference data for accuracy
     computation
@param  ann: fann net struct to be tested
@param  accuracy: var to return the accuracy
@param  accuracyThr: var to return the accuracy with thresholding
@return
*/
void testNet(char *testFile, struct fann *ann, float *accuracy,
    float *accuracyThr);
/**
Appends current Laser Sensor data & desired NN output to data
    file

fopen(filenName, "a") is used to open the file

@param  filename name of file to be finished
@param  outputs array with names for the outputs of the ANN
@return
*/
void appendData2File(char *fileName, char outputs[NMAN][30]);
/**
navigate directly to next waypoint

assumes no obstacles in the way

@param  waypoint coordinates to be navigated to
@param  phi current rotation of the robot
@return 0 if successful
*/
int navigateWoObstacle(int waypoint[2], int phi);
/**
Navigate using ANN

@param  ann fann net to be employed
@param  waypoints array containing all waypoints
@return 1 when all wps have been reached; 0 else
*/
int driveProcessExp1(struct fann* ann);
/**
Navigate using combination of waypoints and ANN

@param  ann fann net to be employed
@param  waypoints array containing all waypoints
@return 1 when all wps have been reached; 0 else
*/
int driveProcessExp2(struct fann* ann, int waypoints[NWP][2]);
```

Figure A.6.: Function headers of the functions implemented for the EyeSim simulator (6).

A.3.2. Experiment

The header file split into the figures A.7, A.8, A.9, A.10 and A.11 contain the function headers of the functions implemented for the EyeBot robots.

```
#include "urg_sensor.h"
#include "urg_utils.h"
#include "fann.h"
#include "eyebot.h"

#define PI 3.14159
#define LROBI 100
#define NLASER 682
#define VLIN 100
#define NMAN 2  //number of nn output manouvers
#define NWP  4 //number of waypoints
#define DCORNER 20 //dist the robi moves in search of edge in
    goAroundCornerR/L [mm]
/** Vehicle Movement Commands **/
/**
PI control vehicle distance from wall

Controls vehicle distance from left-hand side wall to keep it
    close to the middle of maze pathway.

@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  indexL index of urg distance value 90 deg to the left
@param  indexR index of urg distance value 90 deg to the right
@param  reset 1 if control sum should be reset
@return 0 if successful
*/
int piStraight(urg_t *urg,long *time_stamp, long *data, int
    indexL, int indexR, int phiDes, int reset);
```

Figure A.7.: Function headers of the functions implemented for the EyeBot robots (1).

```
/**
Go around left corner

@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  indexL index of urg distance value 90 deg to the left
@param  indexR index of urg distance value 90 deg to the right
@param  phiDes phi of the vehicle at start
@return 0 if successful
*/
void cornerL(urg_t *urg, long *time_stamp, long *data, int indexL
    , int indexR, int phiDes);

/** Sensor Functions **/
/**
Plot current sensor readings in polar coordinates

@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  data_old array of size NLASER consisting of the last plot
    's sensor readings
@return 0 if successful
*/
void LaserPlot(urg_t *urg, long *time_stamp, long data[], long
    data_old[]); //Plot Sensor measurements data[] on LCD
/**
Get current sensor readings

@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@return number of readings if successful, -1 if failed
*/
int LaserGet(urg_t *urg, long *data, long *time_stamp);
/**
Initiate communication with the laser sensor

@param  urg sensor handle
@return 0 if successful, -1 if failed
*/
int open_urg_sensor(urg_t *urg);
```

Figure A.8.: Function headers of the functions implemented for the EyeBot robots (2).

```
/** ANN Functions **/
/**
Navigation Routine for experiment 1

@param  ann      fann ANN object to be used for navigation
@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  indexL index of urg distance value 90 deg to the left
@param  indexR index of urg distance value 90 deg to the right
@param  dataFile name of reference data file for ANN
@return 0 if successful
*/
int navRoutineExp1(struct fann* ann, urg_t *urg, long *time_stamp
    , long *data,int indexL, int indexR,char *dataFile);
/**
Navigation Routine for experiment 2

@param  ann      fann ANN object to be used for navigation
@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  indexL index of urg distance value 90 deg to the left
@param  indexR index of urg distance value 90 deg to the right
@param  dataFile name of reference data file for ANN
@return 0 if successful
*/
int navRoutineExp2(struct fann* ann, urg_t *urg, long *time_stamp
    , long *data,int indexL, int indexR,char *dataFile);
```

Figure A.9.: Function headers of the functions implemented for the EyeBot robots (3).

```
/**
Evaluate ANN

Return array of possibilty values for each outputneuron

@param  ann     fann ANN object to be used for navigation
@param  dataFile name of reference data file for ANN
@param  urg sensor handle
@param  timestamp timestamp handle
@return Array of possibilty values for each outputneuron
*/
fann_type* evalNet(struct fann *ann, char *dataFile, urg_t *urg,
   long *time_stamp);
/**
Evaluate ANN

Return array of possibilty values for each outputneuron

@param  ann     fann ANN object to be used for navigation
@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  indexL index of urg distance value 90 deg to the left
@param  indexR index of urg distance value 90 deg to the right
@param  dataFile name of reference data file for ANN
@return Index of most likely outputneuron
*/
int runNet(struct fann *ann, char *dataFile, urg_t *urg,long *
   time_stamp, long *data, int indexL, int indexR);
/** Utility Fuctions **/
/**
Append current sensor readings to data file

Creates file if it does not exist; overwrites existing file

@param  urg sensor handle
@param  timestamp timestamp handle
@param  data array of size NLASER to tmpsave sensor readings
@param  dataFile name of the datafile to generate/overwrite
@return 0 if successful
*/
int appendData2File(urg_t *urg, long *time_stamp, char *dataFile)
   ;
```

Figure A.10.: Function headers of the functions implemented for the EyeBot robots (4).

```
int init(urg_t *urg, int *indexL, int *indexR, int green); //
    initiate sensor and v-omega drive; return -1 if failed, 1 if
    successful
void terminate(urg_t *urg);
/**
Get index of largest member in a <fann_type> array

@param  array array to be searched
@return index of largest member
*/
int largestMember(fann_type* array); //get position of largest
    member in a <fann_type> array
/**
Get index of second largest member in a <fann_type> array

@param  array array to be searched
@return index of second largest member
*/
int secondMember(fann_type* array);
/**
Print current position (x,y,phi)
@return
*/
void printPos();
/**
Navigate directly to next waypoint, assuming no obstacles in the
    way

@param  waypoint way-point in cartesian coordinates (x,y)
@return 0 if successful
*/
int navigateWoObstacle(int waypoint[2]);
```

Figure A.11.: Function headers of the functions implemented for the EyeBot robots (5).

A.4. Example Program

The program in figure A.12 allows to plot the current laser sensor measurements in polar coordinates, run the maze navigation described in chapter 6 and gather laser sensor data for ANN training.

```
#include "eyebot.h"
#include "netLaser.h"
int main(){
        int indexL, indexR; //index of 90 and 270 deg measurement
        int mean, k=0;
        long *data = NULL, time_stamp, data_old[NLASER];
        urg_t urg;
        struct fann *ann = fann_create_from_file("physical.net");
        VWSetPosition(0,0,0);
        if(init(&urg, &indexL, &indexR, 0)<0){return 1;}
        data = (long *)malloc(urg_max_data_size(&urg) * sizeof(
           data[0]));
        while(1){
            LCDMenu("Plot","Navigate","Capture","Speed␣0");
            k=KEYGet();
            switch(k){
                case KEY1: LaserGet(&urg,data,&time_stamp);
                           LaserPlot(&urg,data,data_old);
                           break;
                case KEY2: navRoutine(ann,&urg,&time_stamp,data,
                    indexL,indexR, "ref.data");
                           break;
                case KEY3: appendData2File(&urg, &time_stamp, "
                    sensor.data");
                           break;
                case KEY4: VWSetSpeed(0,0);
                           break;
                           }}}
```

Figure A.12.: EyeBot program that allows to plot the current laser sensor measurements in polar coordinates, run the maze navigation described in chapter 6 and gather laser sensor data for ANN training.