Matthias Eder, BSc

Using particle filters and machine learning approaches for state estimation on robot localization scoring

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ass.-Prof. Dipl-ing. Dr.techn Gerald Steinbauer

Institute for Software Technology (IST)

Graz, October 2017

Abstract

Robot localization is crucial part of modern robot systems and is a well studied topic. There are lots of different techniques which can be used to estimate the position of a robot in relation to an environment. Typically because of the uncertainty in the robot domain, these techniques use some kind of filtering approach to determine the current robots pose, using various types of sensors like 2D lasers and the representation of the environment. To determine how well the robot knows its pose we introduced the term localization scoring. Localization scoring analyses how precise a robot is localized. It typically uses the robots position and additional sensor data to check how well the stated position fits into the environment. In this thesis an existing implementation based on a particle filter will be analyzed for finding features such as the distribution of the environment of different kind which can be used to score the accuracy of robot localization. It then is examined if those features are useful for robot localization scoring by using them in machine learning approaches. One way of finding a feature is by training neural networks on the particle cloud. This allows to find a complex feature that holds a lot of information about the localization state of the robot. To find the best fitting neural network different types and structures are evaluated. Results show that convolutional as well as recurrent networks have a good performance in detecting a localization feature. Also the desired network complexity is minimal since a particle filter generates information which simple network structures can already be classified. Using extracted localization features and combining them in boosting algorithms like AdaBoost shows that a reasonable output is trained which can be used to estimate the robots localization quality.

Kurz Zusammenfassung

Roboter Lokalisierung ist ein wichtiger Bestandteil moderner Roboter und ein gut untersuchtes Forschungsgebiet. Es gibt viele verschiedene Techniken die verwendet werden können um die Position eines Roboters in einer Umgebung zu bestimmen. Um die Genauigkeit eines Roboters in der Umgebung zu erhöhen werden meist Umgebungsdaten und Sensoren wie 2D Laser verwendet. Um festzustellen wie genau ein Roboter seine Position begann man damit die angegebene Position zu bewerten. Dabei geht es darum zu analysieren wie genau der Roboter lokalisiert ist und wie sicher er sich ist, and der richtigen Stelle zu sein. Typischerweise werden dafür die geschätzte Position des Roboters, die Umgebung und verschiedene Sensordaten benutzt um festzustellen wie gut die geschätzte Position in die Umgebung passt. Im Grunde geht es darum festzustellen ob ein Roboter lokalisiert oder delokalisiert ist. In dieser Arbeit wird eine bereits bestehende Implementation eines Partikelfilters analysiert um bestehende Merkmale, wie Verteilung der Partikel, zu finden. Diese Merkmale werden dann verwendet um die Genauigkeit einer angegebenen Position zu bewerten. Um dies zu erreichen werden die gefundenen Merkmale kombiniert und in verschiedenen maschinellen Lernansätzen verwendet. Eine Möglichkeit um ein Merkmal zu finden besteht darin, neuronale Netzwerke auf die Partikelwolke zu trainieren. Dies ermöglicht es ein komplexes Merkmal zu extrahieren welches Informationen über den Lokalisierungszustand des Roboters enthält. Um ein passendes neuronales Netzwerk zu finden werden verschiedene Arten und Strukturen ausgewertet. Die Ergebnisse zeigen, dass sowohl faltende neuronale Netze (CNNs) als auch rekurrente Netzwerke eine hohe Genauigkeit bei der Erkennung des Lokalisierungszustandes erzielen. Des Weiteren wurde herausgefunden, dass die dazu benötigte Netzwerkkomplexität minimal ist da ein Partikelfilter nur einfache Informationen erzeugt die schon mit kleineren Netzwerkstrukturen erkannt werden können. Um eine Gesamtgenauigkeit zu erhalten wurden alle gefundenen Merkmale kombiniert und in maschinelle Lernalgorithmen wie AdaBoost kombiniert. Dabei wurde gezeigt, dass bei Verwendung von vernünftigen Merkmalen der Lokalisierungszustand eines Roboters sehr gut geschätzt werden kann.

Contents

Chapter 1

Introduction

This Master's thesis deals with topics which are currently hot discussed. It discusses particle filter [1, 12], a common technique for robot localization, and evaluates the information which these techniques produce. This information is then used in training deep learning networks to find out if it can be used for estimating the accuracy of a robots position.
The peculiarity is that usually particle filters are used in combination with machine learning approaches for state estimation [160], pattern recognition [19] or to track objects [112]. This thesis does not combine those techniques for improving either of these combinations. Instead, machine learning approaches are used to recognize patterns in particle filters information for estimating the quality of the localization of the robot.

In this chapter we are going to motivate the problem which we like to handle and present the initial situation of the thesis.

1.1 Initial Situation

Robot localization is a well discussed topic nowadays. There are lots of different techniques which can be used to estimate the position of a robot in an environment. Typically, these techniques use some kind of state estimation technique to determine the current robots pose, using various types of sensors like 2D lasers. Localization algorithms are known to have some accuracy issues since they only work with probabilistic models and thus do not always report the correct position [10, 146]. To determine how well the robot knows its pose in relation to its environment we introduced the term localization scoring. Localization scoring analyses how well a robot is localized. It typically uses the robots position and additional sensor data to check how well the stated position fits. Determining how well a robot is localized is also known to be difficult. As it is not obvious to determine the position of a robot using only simple sensors, it is also a challenge to find a method which delivers a good estimation of the robots current position estimation quality.

A typical approach used for localization is the so-called Monte Carlo method which is based on the particle filter [147, 34, 146]. This algorithm uses particles to represent the uncertainty about the robots pose. Different types of sensor data are then used to reorganize the particles based on the likelihood of the processed data fitting into the environment [148, 80, 146]. In robotics, typically laser scanners are used in particle filters to evaluate the position. In an unambiguous environment particle start to form particle clusters after some time which represent the robots position and the uncertainty about it. Checking the accuracy of a stated position is not obvious using only laser sensor data. Many algorithms for localization scoring use simple methods like calculating the distance between scan and map points and then estimate the quality using defined thresholds which state how many scan points must match a certain distance to be localized [138, 82, 146]. Another method is to do some sort of line recognition within the map and scan. Those lines are then matched to evaluate the position of the robot [29]. Often many simple techniques with various parameterizations are combined to improve the quality of a scoring algorithm. In those combined method one also needs weighting parameters to define the importance of different scoring features. There also exist more complex approaches but many of them need to set additional parameters which are difficult to estimate since it is not an obvious task to determine and rank important features [76, 61, 163, 148]. At the end the performance is often unreliable and a lot of work has to be invested to find parameters for a certain environment which fit best.

1.2 Thesis Outline

First the problems addressed in this thesis are stated more formally and the aim of of this thesis is defined. After showing our goals and used methods some basics are explained which help to orientate through this work. Also related topics which were found in research are described and discussed. Then particle filters are analyzed and information is searched which can be applied for machine learning approaches. This information is then filtered and stored as training data for deep learning networks. The data is then used to train deep learning networks to find an acceptable solution. To make sure that the selected solution is optimal, different network structures with different network parameterizations will be presented and trained. The outcome is also investigated using different validation sets, not only created test sets but also real world data sets. The result of this solution is then also combined with classical techniques used nowadays to score the localization accuracy of a robot.

1.3 Problem definition

This section handles the definition of problems which are handled in this master thesis. Problems of the current situation are revealed and explained in further detail. Subsequently it is discussed which of these problems are addressed in this thesis and how we plan to solve them.

1.3.1 Basic Problems in Robot Localization

Since the very first beginning of self-localizing robots some fundamental problems are known which researcher tried to overcome with various solutions.

The basic problems are now shortly discussed.

The first problem which is addressed is the **global localization** problem [109, 105]. Given a robot and an arbitrary environment which the robot may know. The robot does not obtain information about its initial position within this environment. The task is now to determine the robots location within the environment using only data retrieved from simple sensors like 2D laser-range finders. Thus the pose has to be estimated from scratch. While humans have lots of information available to determine their current position it is not trivial to determine the initial location of a robot, using only limited information.

The next issue is the **kidnapped robot** problem [94]. It is similar to the global localization problem with the difference that the robot was already localized in the environment. While an autonomous robot is localized in an environment it is moved to an arbitrary location without giving any feedback. The robot then has to find its current position again using only the new information available. An essential problem which comes with the problems discussed so far is that the environment might have various areas which look similar. This results in multiple possible positions. Figure 1.1 illustrates this problem. The grey ellipse is the old position of the robot while the blue ellipse is the new position. Since the robot was carried away and did not receive any information about this movement he has to determine its new position. From both, the new and old robots position, the environment looks the same. Thus the robot might not even know he was carried to another position.

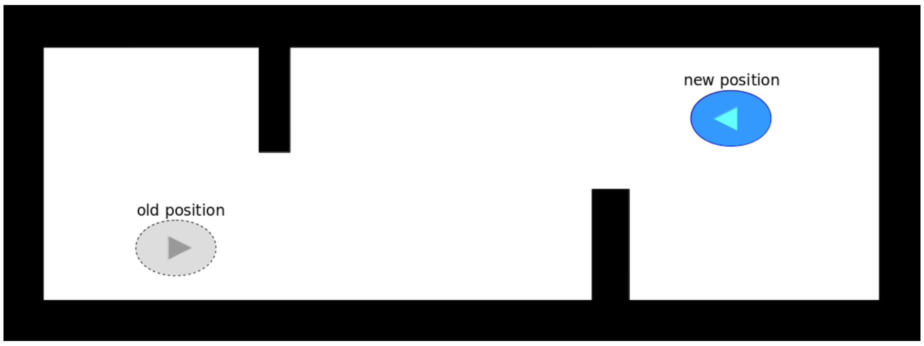


Figure 1.1: Illustration of the kidnapped robot problem

Another localization issue is the difference between **active and passive** localization [55, 135]. In passive localization the robot estimates its position using only incoming sensor data. It states that based on the localization information neither the orientation nor the motion of the robot can be adjusted [15]. In active localization the robot has the possibility to control the motion or orientation of the robot to improve the accuracy of the localization technique [15]. Thus it can choose where to look to gain necessary information for localizing the robot.

The last basic problem are **dynamic environments** [140]. When localizing a robot in an environment there is a difference between not changing the appearance of the setting over time and adapting its appearance based on new information. This new information may have various reasons like people or other robots moving around within this environment. This leads to the problem that there are artifacts within the environment which disturb a smooth localization. This results in the issue that the localization might become unreliable if there is a lot of noise within the environment.

In general, localization is hard because the pose can not be directly obtained. Some kind of sensors which measure the environment or the motion of the robot are needed. The problem of measuring the environment is, that it can be ambiguous and non trivial. Also the sensors for measuring the environment and the motion have systematic and non-systematic errors which lead to localization errors.

1.3.2 Current Localization Scoring approach

This section focuses on describing how the localization score is currently computed and which issues this computation has.

One important fact which has to be mentioned is that the robots localization and the scoring of its position accuracy are currently separated in two different modules which are independent from each other. A particle filter is used to localize the robot based on its movement. Therefore the odometry of the robot is taken and transformed to environmental coordinates. Since the odometry has a certain error this transformation is adapted with the help of the particle filter. This filter spreads out particles in the environment and calculates the position with the highest probability. Based on this found position the transformation between odometry and map is adapted such that the new position in the environment is the position with the highest probability. For a detailed explanation on how particle filters work see chapter 2.3.4.

To determine how accurate the localization of the robot works a separate module is used. This node uses four simple techniques to determine the position accuracy of the robot. The first technique uses a hough line detector [43] to receive lines from the environment and the 2D laser scan. Those lines are then compared and matched to each other. After evaluating those matching lines, the importance of this result is weighted together with the other three techniques. The second approach is to map single scan points on those previously calculated lines. Therefore every scan point is evaluated and it is checked how well it fits to the environmental lines.

The third technique uses a simple implementation of a ray-caster [132] to evaluate the position of the scan in the environment. Based on the robots position a ray-trace is sent out for each scan point and it is checked if this certain scan point lies in front of an obstacle, if it matches the obstacle or if it lies behind. This is then used to compute another scoring solution which is weighted with the rest. Figure 1.2 shows how the ray-casting technique is used to determine the robots position. The green points indicate a scan point, the black rectangle indicates a wall.

The last technique which is applied to score the accuracy of a robots position is the angle scoring. In this method the laser scan is taken and rotated in both directions for a certain degree. It is then evaluated if the rotated scan fits better to the environment than the original scan. For this evaluation the point-line matching or line-line matching as described above can be used. Figure 1.3 illustrates this technique. Therefore the scan is rotated a bit to see if it fits better

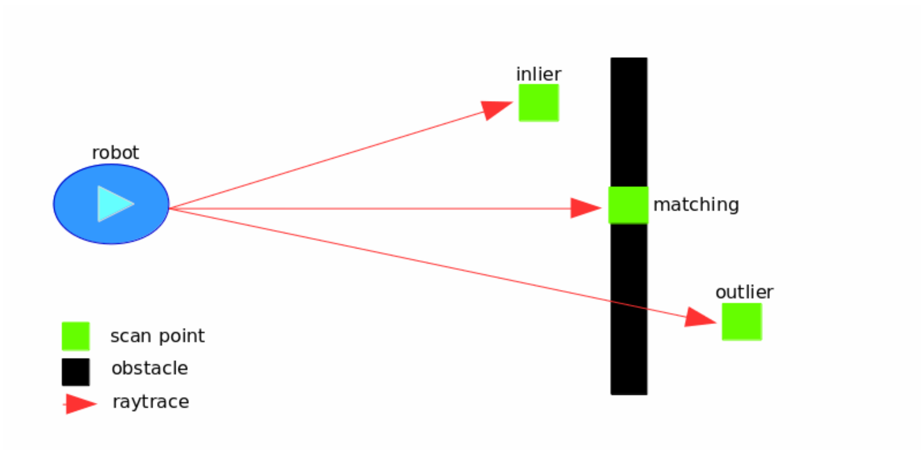


Figure 1.2: The ray-casting localization method. Determining the position of the scan point.

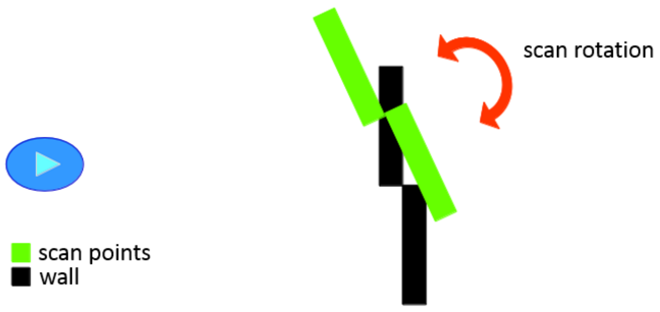


Figure 1.3: Localizing the robot by changing the scan angle

into the environment. This method takes care of small orientation errors.

When looking back at section 1.3.1 where active and passive localization were discussed one can now see that the particle filter is a passive localization method since it does not take over the control of the robot to improve the localization efficiency. It only takes the current observations and might adapt the robots position. The methods used for localization scoring are neither passive nor active localization. They are only used to determine how well a robot is localized without changing the actual position. But the score of this evaluation could be used to do some kind of error handling. When the score becomes critically low and the robots seems to be delocalized it could start to do some active localization.

The Problem of Parametrization

The four different techniques described above for scoring the accuracy of the robots position have one major issue. Every single technique uses multiple parameters which need to be set correctly such that the method can be assumed working. To receive optimal results one has to adjust those parameters for different environments and invest a lot of time to find the best fitting options.

For example, the first approach needs various parameters to be set, starting from the parameters for hough line detection [43], the matching distances and angles which state when a robot is delocalized, going to the weight of importance for every certain technique. All other methods also need predefined parameters like the rotation angle of the scan-angle method. This results in an huge number of parameters for which it is nearly impossible to find the best parameter combination.

The Inflation of Particle Filters

Particle filters based localization approaches are a good method to overcome some of the issues mentioned above [147, 146]. For example they are able to solve the global localization problem as well as the kidnapping problem. Another advantage of particle filters is that they are non-parametric [146] which means that no parameter has to be adjusted to achieve the optimal result. However, there also occur some minor problems. The first is that the accuracy and the performance depend on the number of particles spread over the environment. To keep this algorithm efficient the number of particles should be small while to keep it accurate the number of particles should be high[60, 50].

Another issue is the inflation of the particle filter. When having an environment which can not be clearly assigned to a scan, the particle cloud starts to inflate because the uncertainty about the current position increases. It is then hard to determine if the robot still is localized or if it is already delocalized. The best example for such an inflation are long straight corridors as shown in Figure 1.4. When driving through the corridor the position of the robot becomes imprecise by the particle filter. This is due to the issue that for the particle filter every position within this corridor looks similar. The current laser scan fits fine to the current robots position but also fits to various positions ahead and behind the robot. This leads to an inflated particle cloud which makes it hard to localize.

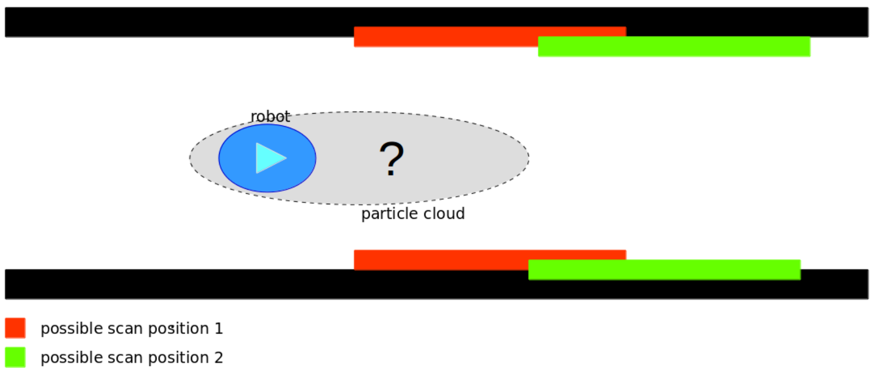


Figure 1.4: Inflating particle clouds due to a long corridor

1.4 Aim of the thesis

After having defined in Section 1.3 the problems which are known at the moment concerning robot localization and particle filters, it now will be discussed how these problems are addressed in this thesis. Therefore it is described what will be aimed with this thesis and how this is reached.

The basic localization problems are well known and many different approaches exist which aim to solve some of these issues. This thesis should neither be a summary of those approaches nor should it develop new methods for robot localization. The aim of this thesis is to take a common laser-based localization approach using particle filters, use its information to support this approach and aim to improve its efficiency. This is done by analyzing the information provided by the particle filters and using it to train a deep neural network [134]. This method is used because it is not trivial to analyze the information of particle filters by hand. It is aimed to find a proper network which detects relevant information about the robot localization accuracy that can be used to score the accuracy of the robots position. Particle filters already handle a lot of basic issues which were described above. But since it does not reveal the accuracy of its solution that easily, it is hard to judge how well a robot is localized in a certain environment.

It also has to be mentioned that not the topic of robot localization itself is the topic of this thesis. It is about using an existing approach, respectively particle filters, and evaluate its accuracy by using the information revealed. Therefore it is about supporting and assessing the results of localization techniques.

To evaluate the accuracy of a particle filter based localization this thesis investigates the following three questions concerning robot localization scoring:

1. Can information provided by particle filters be used to score the accuracy of a position?
2. Does the temporal development of the particle filters information reveal information for localization scoring?
3. Can the approach of using particle filters for localization scoring be boosted using additional features concerning localization?

When analyzing a particle filter one is mainly able to observe the particle cloud. A simple approach of analyzing the information which is obtained by this cloud might be to apply statistical tools. Those tools can be used to determine basic statements about the robots localization quality. Since the particle filter models a complex system it might not be enough to simply apply statistical tools and make assumptions. The shape of a particle cloud can take various forms and thus a statistical statement like the standard deviation of the normal distribution might not lead to good results. To improve these results tough a collection of those statistical statements, called features, could be used for training boosting algorithms like AdaBoost. Additionally a neural network can be applied to detect complex features which cannot be obtained with the usual statistical tools.

To answer the questions above the particle filter will be analyzed for finding features of different kind which can be used to score the accuracy of robot localization. Then it is examined if those features are useful for robot localization scoring by using them in machine learning approaches. Therefore training data which is relevant for our selected machine learning approaches is generated. We focus on producing supervised data sets for training because we expect better results than using unsupervised data sets [40]. We also try to improve our approach by using large data sets. In general, it is difficult and time consuming to evaluate the ground truth of a robots position in real world environments since one needs to observe the correct robots pose by using external observation tools. To overcome this issue and generate a large supervised data set a robot simulation software is used for observing the information of particle filters [57]. The outcome of our machine learning approaches will then also be used on data which is generated in real world environments and verified manually.

To answer the first question a convolutional neural network (CNN) [83, 111, 86] is modeled and trained to see if there is a correlation between particle filter information (e.g. the pattern of the particles) and the localization score. Then a recurrent neural network (RNN) [91] is shaped to answer the second question and it is checked if the temporal development of particle filters reveal information about the localization accuracy of robots. After having evaluated and answered the first two questions, the best rated neural network of this thesis is used to answer the third question. Therefore, additional features which are already used for localization scoring are examined and combined with our objectives. It is then tried to boost the performance of our network by using an additional machine learning approach on those combined feature set. The outcome of these questions will then be examined, compared to each other and analyzed in detail.

Chapter 2

Prerequisites

This chapter describes all needed prerequisites that are used to comprehend this thesis. Therefore technical aspects as well as theoretical background which is needed to understand this thesis are discussed. First, basics of the common Robot Operating System (ROS) are explained and how it can be used to work with robots. Then various localization mechanisms are presented which are used nowadays to localize a robot or to score the accuracy of a robots localization, focusing on the particle filter and its implementation for robot localization.

Having explained how the localization can be done for a robot, the principles of neural networks are presented. Starting with some basics, going to deep learning architectures and how to train them. At the end two frameworks will be explained which are used in this thesis to apply machine learning approaches and for data generation.

2.1 Used materials and procedures

To successfully perform our work and satisfy all the defined aims we have to set our working materials and explain our used procedure.

The particle filter algorithm which is analyzed in this thesis is implemented into the Robot Operating System (ROS) [117]. This system provides a framework full of tools, drivers and libraries which allow you to develop robot software. It is widely used and easy to extend. ROS offers a code basis for different programming languages like C++, Java or Python. This thesis works with the programming language C++ and all further examples will be made in this language. Although the software is used to provide all components for a robot we do only work with one feature. Since we want to evaluate robot localization we focus only on the implementation of particle filters. This implementation can be found in the AMCL package that is offered by ROS. For receiving needed information about the robots localization and to generate training data, the robot operating systems communication channel is used.

When generating training data which can be classified into different states to train a deep neural classification network, a measurement has to be done to determine the robots stated position and the actual position. Therefore, a simulation software is used to simulate a robots hardware. By doing so the actual position of the robot can still be observed, even if the robot is delocalized. This simulation software is a self developed testing software of incubedIT which is based on the well known stage [57]. incubedIT is a company developing software for autonomous transportation robots. Stage is an open source robot simulator which is already supported by ROS.

To train various deep learning networks the Caffe framework is used [73]. It is and machine learning framework which was developed by the Berkeley university and by other contributors. Its aim is to offer a fast and modular platform which can be used to train neural networks.

There exist vast amounts of sensor types that can be used by a robot to interact with its environment [121, 2, 150]. Also for localizing a robot in its environment a lot of different sensor types can be used [80, 61]. To proceed with the current situation and to delimit the scope of this thesis, only 2D laser range finders and robot motion tracking are considered, concerning localization scoring.

The implementation part and thus all examples which are presented within this thesis are done using the operating system Ubuntu 14.04. It is not guaranteed that the implementation or other used materials work on other platforms in the same way. Using other operating systems might lead to errors in compilation or during runtime.

2.2 Robot Operating System

This section describes the Robot Operation System (ROS) which was introduced by the Stanford Artificial Intelligence Laboratory in 2007 [117]. As described before in Section 2.1, ROS is a framework which provides tools, drivers, a communication framework and libraries which can be used to write software for robots. This software was chosen because it is a well-known framework which is widely spread and currently used in research. Although it might not be the simplest solution for writing a single robot software it is one of the most used frameworks for robotic research. This might be due to a stable and maintained software core that is easily extendible. This thesis does not explain how to install ROS. A detailed description for that can be found in [110]. ROS offers a code basis for different programming languages like C++, Java or Python. This thesis works with the programming language C++ and all further examples will be made in this language. Some of the main advantages of ROS are highlighted in [110]:

1. **Distributed computation.** Not everything a robot needs can be computed on computer. Some processes might run on multiple machines. Therefore a communication framework is needed which allows the robot to communicate with other processes or computers. The best example would be the control of a robot where humans send commands from a separate computer to the robot. This interface is a distributed extension of the software.

2. **Software reuse.** Since robotics is a popular topic nowadays, many solutions for various tasks exist. This leads to a large collection of excellent

algorithms that can be used for main robotic tasks like navigation or localization. But those algorithms are only useful if there is a framework which offers those algorithms. ROS does so and thus one does not need to re-implement many algorithms for related tasks on robots.

3. **Rapid testing.** During the development of robotic software one also needs to test the implementation. This can be challenging when there is not always a robot nearby. To overcome this issue, ROS offers so-called *bagfiles*. This is a simple way to record robots sensor data and replay it as often as one likes.

To understand the implementation of a robots localization task, one needs to have knowledge about ROS. Therefore some basics are described in this chapter to understand the use of the robot operating system. First the structure is explained and how the system is started. Then some further terminologies like nodes, topics and parameters are explained. ROS is used as robot software to run the localization task. It is needed because information about the robots environment, especially sensor data and odometry, can be retrieved through this. All this data is required to make assumptions about the robots location.

2.2.1 Structure

To find the main directory of ROS, a standard path for the software location is defined when installing ROS. This path is defined on Ubuntu using an environment variable called *ROS_PACKAGE_PATH*. In there the code for the operating system can be found. ROS is organized in **packages**. Such packages hold different files that pursue a common purpose. Generally it includes executable files as well as code files. Each package contains a **manifest** file which defines the details of a package. The most important thing defined in there are dependencies. Those dependencies might be system dependencies or other ROS packages which contain needed capabilities.

2.2.2 The ROS Master

Packages, as explained above define the structure of ROS, but the aim is to execute some software which is defined in those packages. One of the goals of ROS is to keep everything simple and separated. Therefore many small programs, called **nodes** are started to run at the same time. Those nodes can operate independently from each other but sometimes need to share some information. To communicate with other nodes one needs the **ROS master**. This master is started at the beginning to offer a communication framework between small programs. This master keeps running the whole time while robotic software is used. To start the master one needs to call:

```
roscore
```

Once the master is running, one can start a ROS program from a package. Running a ROS program results in a running **node**. Those nodes execute the ROS dependent code which was written within the packages. One package can hold multiple executables that can be started as a node at the same time. To start a node one needs to call:

```
rosrun <package-name> <executable-name>
```

To see which nodes are currently running on the master the command

```
rosnode list
```

can be used. To retrieve special information about a node one can call:

```
rosnode info <node-name>
```

This outputs a list of topics which are published and subscribed by this node. Also a list of services that are offered by the node is shown. Topics and Services are explained later in this chapter.

2.2.3 Messages and Topics

So far only packages were described and how to start them but it was not discussed how the communication between nodes looks like. The main mechanism that is used for communication in ROS are **messages**. Those messages can hold information using standard types like integers, floats and strings. They can also contain another message type within that message, resulting in a structured message. ROS offers some standard messages that can be used to send specific data like odometry or sensor data to other nodes. But it also allows to define new message types which can have any structure using standard types and previously defined message types. Messages are defined in a *.msg* programming language which is then automatically converted into code which can be used in e.g. C++. Listing 2.1 shows an example for a msg file holding a string, an integer and another message type called *PoseWithCovariance* in the package *geometry_msgs*.

```
# Example ROS message position.msg in package sample_msgs
# This is used to generate a C++ class "Position" in
# package sample_msgs
string robot_name
uint32 robot_id
geometry_msgs/PoseWithCovariance pose
```

Listing 2.1: A small ROS message containing standard types and another message type

To share messages between nodes a channel is opened which allows to publish messages. This channel is also called **topic**. To identify a topic it has a name. The idea behind that is that a node can share information to other nodes by **publishing** messages to a certain topic. If a node wants to retrieve information it can **subscribe** to a certain topic in which it is interested in. To see which topics are currently active one can call:

```
rostopic list
```

To check which nodes publish a certain topic and which nodes subscribe it the following command can be used:

```
rostopic info <topic-name>
```

To observe the messages which are sent through a topic the command

```
rostopic echo <topic-name>
```

can be used. This command keeps listening to the topic and lists all the messages which are published in the meantime. Listing 2.2 shows an example for publishing a message within a node. It uses the previously defined message *position.msg* to advertise the topic *robot_location* and then publishes a message to the topic.

```
#include "ros/ros.h"
#include "sample_msgs/Position.h"

int main(int argc, char **argv)
{
  ...
  // open access point to communication with ROS system
  ros::NodeHandle n;
  //create publisher
  ros::Publisher location_pub =
    n.advertise<sample_msgs::Position>("robot_loc", 1);
  ...
  // create new message
  sample_msgs::Position msg;
  msg.robot_name = "robot_0"
  msg.robot_id = 0

  // publish message to topic
  location_pub.publish(msg)
  ...
}
```

Listing 2.2: An example for publishing a message within a node

To subscribe a message from a certain topic, the node needs to define a subscriber. For this subscriber a **callback** function is declared. This function is called when a new message arrives on the topic. Listing 2.3 shows an example for subscribing a message within a node. It also uses the message *position.msg* and subscribes to the topic *robot_location*. When a message arrives the callback function *onLocationCallback* is called and executed.

```
#include "ros/ros.h"
#include "sample_msgs/Position.h"

int main(int argc, char **argv)
{
  ...
  // open access point to communication with ROS system
  ros::NodeHandle n;

  //create publisher
  ros::Subscriber location_sub =
    n.subscribe("robot_loc", 1, onLocationCallback);
  ...
}

void onLocationCallback(
  const sample_msgs::Position::ConstPtr& msg)
{
  //output the robots name
  ROS_INFO("Name:%s", msg->robot_name.c_str());
}
```

Listing 2.3: An example for subscribing a message on a topic

2.2.4 Parameters

Another mechanism to get information into nodes are **parameters**. The idea is to have a parameter server running which holds parameters and their values. The main advantage of such a parameter server is that the values of parameters can change during runtime. When this happens, all nodes are informed and can update their given parameter value. This allows to adopt specific values within nodes without having to change and recompile the code. To get a list of the currently set parameters the command

```
rosparam list
```

can be used. To get the value for a parameter from the server one uses the *get* command:

```
rosparam get <parameter_name>
```

For setting a parameter the command *set* is used:

```
rosparam set <parameter_name> <parameter_value>
```

Listing 2.4 shows the use of parameters in a node. First a parameter *laser_range* is requested and printed to the terminal using a normal and a default version. The default version has the advantage that if the parameter is not found, it just sets the given default value while the normal version needs some error handling. For setting a value of a parameter there are two possibilities, either via a node handle or static.

```
#include "ros/ros.h"

int main(int argc, char **argv)
{
  ...
  // open access point to communication with ROS system
  ros::NodeHandle n;

  double range, range2;
  // normal get version
  if(n.getParam("laser_range", range))
  {
    ...
  }

  // default value get version
  n.param<double>("laser_range", range2, 30.0);

  // set with node handle
  n.setParam("laser_range", 35.0);

  // set static
  ros::param::set("laser_range", 35.0);
  ...
}
```

Listing 2.4: An example for parameter using within a node

Another possibility that is offered by parameters is to dynamically update them during the code execution by adding a callback function that is called whenever the parameter value is changed. Therefore one has to bind a parameter server to a defined callback function. When he parameter is then changed by other nodes or by the user this function is called and allows to react on the new value for a given parameter. This method is also called dynamic reconfiguration notification. This is often used if the parameters might change during runtime but for this thesis they play only a subordinate role.

2.2.5 Services

An alternative method for communicating between nodes are **service calls**. They have two main differences compared to messages:

1. Services are **bi-directional**. While messages are only sent through a topic to another node, no response is received. In service calls the channel is used in both directions. One node sends a service to another node and then waits for the other node to respond. This means that another node is needed which responds to the service call, otherwise it might never receive a response.

2. Service calls use a **one-to-one** communication. In contrast to messages which can be published and subscribed to a topic by multiple nodes, service calls are only done by one node. This node send a service call to an offered service and waits for a response. Since this connection is only

established between the service caller and service sender, no other node can listen to this service calls.

To get a list of all service which are active, the command

```
rosservice list
```

can be used. To get more information on a given service the command

```
rosservice info <service-name>
```

can be called. This command reveals information about the node which offers the service and about the service type which has to be used. To call a rosservice from the command line the following command is used:

```
rosservice call <service-name> <request-content>
```

Like messages, services are generated using a separate file. In this case it is a *.srv* file which describes the structure of the service. The service file holds two different definitions, separated by "− − −". The upper one is the definition of the request layout and the lower one is the structure of the response layout. Listing 2.5 gives a small example for a service file. It takes a string and an integer as request parameter and sends back a boolean value.

```
# Example ROS service robot_exists.srv in package
    sample_srvs
# This is used to generate a C++ class "RobotExists" in
# package sample_srvs
string robot_name
uint32 robot_id
---
bool exists
```

Listing 2.5: A small ROS service containing standard types

This message is then taken to generate a C++ code that can be used within a node to send and retrieve services. Listing 2.6 shows how a service is advertised by a node. It offers a service *robot_exists* and calls in a function *checkExistence* if the service is called.

```
#include "ros/ros.h"
#include "sample_srvs/RobotExists"

bool checkExistence(
  sample_msgs::RobotExists::Request& req,
  sample_msgs::RobotExists::Response& res)
{
  res.exists = true;
  return true;
}

int main(int argc, char **argv)
{
  ...
  // open access point to communication with ROS system
  ros::NodeHandle n;
  // offer service
  ros::ServiceServer srv =
    n.advertiseService("r_exists", checkExistence);
  ...
}
```

Listing 2.6: An example forthe usage of and service advertiser

To call a service, one needs to create a service client. Listing 2.7 shows an example of an client. It starts a client, calls it and prints the response.

```
#include "ros/ros.h"
#include "sample_srvs/RobotExists"

int main(int argc, char **argv)
{
  ...
  // open access point to communication with ROS system
  ros::NodeHandle n;
  //create service client
  ros::ServiceClient cl =
    n.serviceClient<sample_srvs::RobotExists>("r_exists");
  // create new service instance
  sample_srvs::RobotExists srv;
  srv.request.robot_name = "robot_0";

  //call service
  cl.call(srv);
  //print response
  ROS_INFO("exists:%d", srv.response.exists);
  ...
}
```

Listing 2.7: An example for a service call in a node

2.3 Localization of Robots

This section describes needed prerequisites for understanding the robots localization problem as described in Section 1.3. In there, the current localization scoring approach is already explained but for better understanding and for the sake of completeness a typical hough line transformation [43] and a basic ray-caster [132] will be presented. Those two techniques are currently used for scoring the accuracy of the robots localization and its information will later be used as features for boosting the deep learning results like discussed in Section 1.4.
Then additional features are presented which will be used for boosting the deep learning results. Those features are extracted from the localization approach from Ingemar J. Cox [29, 32, 31].
Another topic handled in this section are the principles of the basic particle filter [34, 146] which is used to train a deep neural network for state estimation on robot localization scoring. Since the particle filter is only a general algorithm, a more specific algorithm is needed that can be used for robot localization. Therefore also the Monte Carlo localization [147, 60, 146] will be discussed which is an extension of the particle filter for the use in robot localization.

2.3.1 Hough Line Transform

This section focuses on the hough transform. It is a method to detect lines, circles or other arbitrarily parametrizable geometric figures [96, 8]. The method discussed in this thesis is used for detecting straight lines, which is the simplest application of a hough transform [43].

Usually lines are represented using two parameters a and b. They are used in the equation $y = a * x + b$ to represent a line in two dimensional space. The problem with this equation comes with vertical lines. Those cannot be represented with this equation. To overcome this issue instead of a and b new parameters θ and r are introduced [43, 17, 54], where θ is the angle of the line and r is the distance from the origin respectively on the x-axis. Hough transform uses the formula represented in Equation 2.1 for defining a line. This formula can also be rewritten into Equation 2.2 to look similar to the basic line formula.

$$r = x * \cos\theta + y * \sin\theta \Leftrightarrow \tag{2.1}$$

$$y = -\frac{\cos\theta}{\sin\theta} * x + \frac{r}{\sin\theta} \tag{2.2}$$

All lines can now be represented by defining a $\theta \in [0, 360[$ and $r \geq 0$. A hough line is represented as a point in two dimensional parameter space with axis θ and r. Figure 2.1 illustrates the mapping of a line onto a point.

The hough transform uses this parameter space to map a pixel p_0 in an image to all lines that can pass through that point. This results in a sine-like curve in the hough space. When adding all pixels to the hough space, some lines may overlap and increase the intensity of this points. This results in local maxima which indicate the most likely lines within this image. Figure 2.2 shows the mapping of two points into the hough space. This results in an overlap which is a local maximum and thus is the most likely line for those two points. To

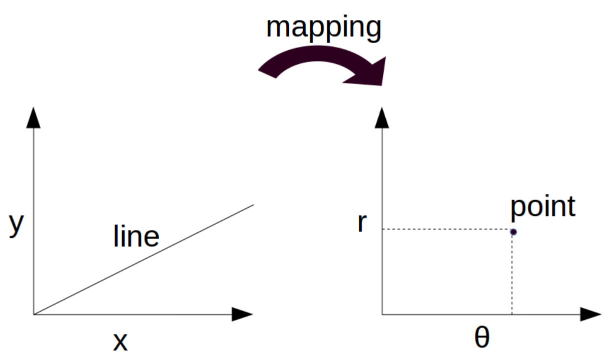


Figure 2.1: mapping a line to an hough point

detect lines in images often edge detection algorithms [153, 152] are used to highlight possible lines and improve the result of the hough transform. To work with the hough space in algorithms an so-called accumulator is needed. It covers the hough space and holds the value for each point. When a new pixel is added, the accumulator is increased for all bins where the lines pass through. To detect finite lines an approach called Progressive Probabilistic

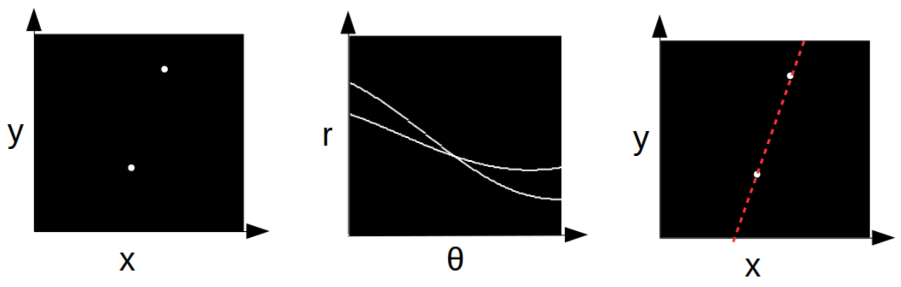


Figure 2.2: the process of detecting a hough line between two points.

Hough Transform [96, 54] is used. This approach searches along the infinite lines in the created edge image to detect finite lines. This procedure is not further discussed here. Having explained all necessary steps which are done in the hough transform for detecting lines one can now summarize it into an algorithm with four steps:

1. Apply an edge detection algorithm on image.
2. Map image points into hough space and store them in an accumulator.
3. Interpret the accumulator to detect lines of infinite length. This needs some thresholds and maybe other constraints.
4. Convert infinite lines to finite lines.

Hough transform is used for robot localization scoring to detect lines in an environment which is represented as a map. This map contains map points that are transformed into lines. The same is done using an image of the laser scan.

Those two images are then matched against each other to receive information about the robots localization accuracy. The matching is often done by searching the nearest map line to a scan line and by comparing its positions like the error in translation or rotation.

2.3.2 Ray Casting

To determine if the localization accuracy of a robot the ray casting method is used to check if a scan point lies in front of an obstacle, on an obstacle or behind. This section describes how this ray casting mechanism works. Ray casting is a method to detect intersections within an environment using rays which are sent out from a position with a certain angle [133, 47]. It is mostly used in computer graphics for 3D environments and is a solution for various problems [58, 65]. This thesis uses ray casting to determine the distance of the first obstacle that intersects with a ray. Since the robot uses a 2D representation of the environment only the algorithm for a 2D ray casting is presented.

To perform a ray casting one needs a 2D grid representation of the environment and a viewpoint which is the ray origin. This viewpoint consists of a position vector $\mathbf{p}$ and an orientation d. The environment grid contains cells which either are 0 or contain a positive value which indicate the probability of an obstacle. To start ray casting one needs to define the opening angle θ which determines the field of view of the robot. Another definition is the number of rays N which should be sent out through the grid and the maximum distance *max* which should be checked through the grid. To determine how far two rays are away from each other, one can now calculate the angle step α

$$\alpha = \frac{\theta}{N} \tag{2.3}$$

To orientation of the starting point for the first ray is the robots orientation rotated by $-\theta/2$. Using this starting orientation a ray is sent out to the grid map using a simple Bresenham algorithm [14, 4]. The Bresenham algorithm will not be presented within this thesis but for better understanding the principle is now shortly explained. Bresenham [14] developed a method to draw a line within a grid given only a starting point and an end point. It is a popular algorithm for line drawing or line following in computer graphics because it is fast and can only be done using simple integer calculations. For the purpose in this thesis the Bresenham algorithm gets a starting point and the orientation as input. It then internally calculates the end point using the maximum distance *max*. It then iterates over the cells in the grid that lie on this line and checks if a grid cell is blocked. The result of this algorithm is the distance to an obstacle or *max* if no obstacle was found. Listing 2.8 shows the ray casting method in pseudo code as it is used in the current implementation. The result of each ray is stored in a list which is then returned for further processing.

```
function raycasting(grid, $p$, $d$, $\theta$, $N$, $max$)
  result = empty list
  $\alpha$ = $\theta$ / $N$
  // calculate starting orientation
  $s$ = $d$ rotated by $-\theta$ / 2
  for (i = 0; i < N; i++)
    dist = bresenham(grid, $p$, $s$)
    result.append(dist)
    $s$ += $\alpha$
  return result
```

Listing 2.8: The raycasting algorithm as it is used in pseudo code

Figure 2.3 illustrates the algorithm with the defined parameters for better understanding. The red lines mark the field of view which the robot can see from its current viewpoint. The green lines indicate an ray which is sent out to the grid using the Bresenham algorithm. The algorithm starts on the left border orientation and then increases the orientation angle by α for each ray. To purpose of the ray casting method in robot localization is to cross check the

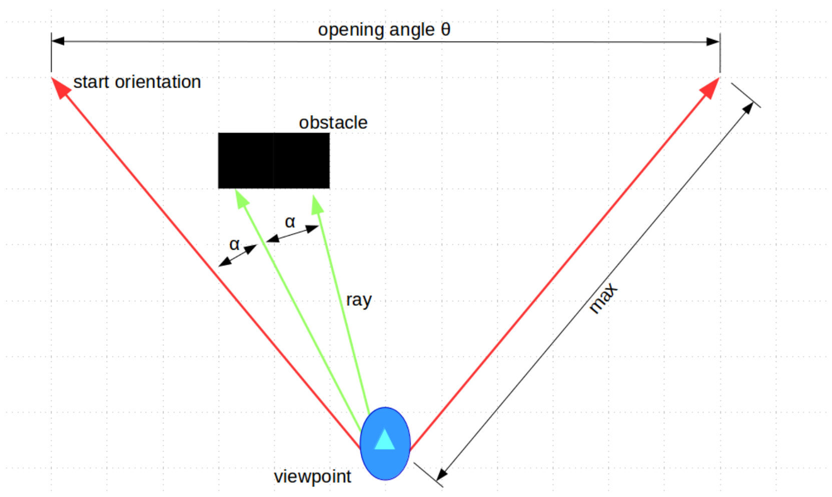


Figure 2.3: The ray casting illustrated for better understanding

retrieved laser points with its environment. Typically the parameters for the ray casting algorithm are defined by the laser configuration to match each scan point to one ray. Therefore, the opening angle, the number of rays and the maximum distance are defined according to the parameter of the laser scanner which is used.

2.3.3 Cox Approach for Position Estimation

This section describes the approach of Ingemar Cox who proposed a method for position estimation on mobile robots [32]. His approach was tested with a mobile robot called Blanche [107], a mobile robot which was designed to be low cost and uses only odometry and an optical range finder. He also proposed many new methods concerning the simultaneous localization and mapping problem (SLAM) [31, 89] but in this thesis only the approach for position estimation is discussed.

The general idea which Cox presented was to use only one optical range finder and odometry sensors to estimate the position of the robot. The odometry is defined as (x, y, θ) which is the position on a grid including the orientation of the robot with respect to a global or local coordinate frame. To perform this task Cox defined four components which are required [32]:

1. A map of the environment containing line segments as obstacles
2. A sensing mechanism to sense the environment using odometry and optical range sensors
3. An algorithm which matches the sensor data onto a map
4. An algorithm that estimates the precision between the matched dataset and which allows to combine the correction with the current position to optimize the robots position.

Since the environment of the robot is only stored within a grid map containing pixels as obstacles, one has to detect line segments which can be used for this approach. An example on how this could be achieved is the Hough transform as described in Section 2.3.1. The sensor data is represented as a list of $< r, \theta >$ tuples where r is the range of a scan point and θ is the scans angle w.r.t. to the robots orientation. Figure 2.4 shows a small part of a map including a scan. In this example the green scan points have a small rotation and a small translation such that it does not fit perfectly to the black map lines. The goal is to match the scan to the line segments which is basically a general problem that is discussed in computer vision. In computer vision the task is often to map an image with an arbitrary rotation and translation to a model. This is usually done by searching for features and determining the correspondence between the image and the model with the help of these features. In [30, 31] Cox proposed a matching

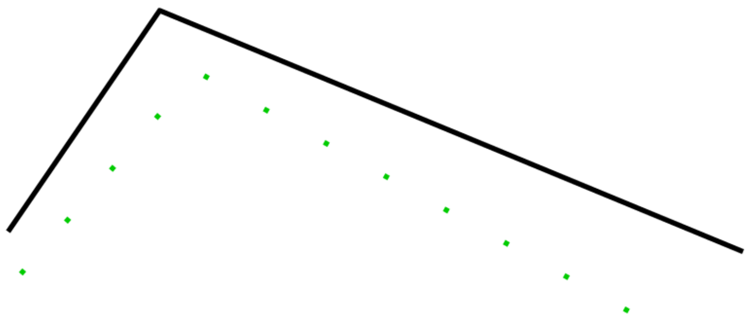


Figure 2.4: A simple example of a map with a displaced scan. The black lines indicate an obstacle within the map while the dots are the displaced scan points

algorithm for images which can be used if the displacement between model and image is small. Since the scan image is often near the correct position and only small adoptions have to be made, an algorithm can be used which maps a scan point to the closest line. A complete algorithm then consists out of four steps [32] and is leaned on the iterative closes point (ICP) algorithm:

1. For each scan calculate the *target*. The target is the nearest line segment to the point.
2. Search for a congruence where the squared distance is minimized for all scan points concerning their targets.
3. Move the scan image using the found congruence in step 2.
4. Repeat steps 1-3 until a certain threshold is reached.

Any congruence can be described using a translation t and a rotation θ and is denoted by $< t, \theta >$. The rotation θ is not estimated by taking an arbitrary origin but by using the center of gravity c of the scan image for rotation. The center of gravity c is a point within a model w.r.t. the center of gravity in the scan image such that is moves with the position of the image. This leads to the fact that a congruence $< t, \theta >$ can map any point x of an image to

$$x \to R(\theta)(x - c) + (c + t). \tag{2.4}$$

In this equation $R(\theta)$ i a rotation by an angle of θ in the clockwise direction. Thus it is denoted by

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.5}$$

To get a congruence for step 2 a value for (t, θ) is searched that minimizes

$$S = \sum_i ([R(\theta)(v_i - c) + (c + t)]^T u_i - r_i)^2 \tag{2.6}$$

where u_i is a unit vector, orthogonal to the infinite target line which corresponds to the scan point v_i. T indicates the transpose. r_i is the dot product of any point with u_i. To simplify the equation an approximation for $R(\theta)$ is done [32]

$$R(\theta) = \begin{bmatrix} 1 & -\theta \\ \theta & 1 \end{bmatrix}. \tag{2.7}$$

To get the congruence out of S, the derivate w.r.t. t and w.r.t. θ are set to 0 such that, after simplification, the following equation is found

$$\begin{bmatrix} M_2 & m_1 \\ m_1^T & m_0 \end{bmatrix} * \begin{bmatrix} t \\ \theta \end{bmatrix} = \begin{bmatrix} d_1 \\ d_0 \end{bmatrix} \tag{2.8}$$

where

$$\begin{aligned}
M_2 &= \sum_i u_i u_i^T \text{ (matrix)}, \\
m_1 &= \sum_i u_i(u_i^T v_i) \text{ (vector)}, \\
m_0 &= \sum_i (u_i^T v_i)^2 \text{ (scalar)}, \\
d_1 &= \sum_i u_i(r_i - u_i^T v_i) \text{ (vector)}, \\
d_0 &= \sum_i (r_i - u_i^T v_i)(u_i^T v_i) \text{ (scalar)}.
\end{aligned} \tag{2.9}$$

The localization approach from Cox can be used to estimate the position of a robot when the displacement of the scan image is small, i.e. the translation and rotation between model and scan is small. This method is currently not used to localize the robot which was a reason that it will not be considered in the deep learning approach but the resulting congruence $< t, \theta >$ might hold some information which can be used to boost the deep network which is trained in this thesis.

2.3.4 Particle Filters

This section presents the principles of particle filters [1, 12, 146]. It is a popular method for representing arbitrary probability distributions and solving state estimation problems. The technique behind particle filters is the Monte Carlo method [125, 122, 98] which already exists for over five decades. The main problem with particle filters is that they are computationally expensive which is a reason why other methods like the Kalman filter [156, 154] are good alternatives. Particle filter allow the analysis of complex systems which are non-linear and non-Gaussian. The goal is to deal with arbitrary probability distributions and model them correctly [42, 149]. Since the computational power has increased in the last years and particle filters are a non-parametric approach for solving complex models, they were applied in many different fields like neuroscience [129], biochemical networks [37], signal processing [6], economics [79] and robotics [34].

To understand the functionality behind particle filters, the theory is briefly introduced in this section. First, basics like the Hidden Markov Model, Bayesian Inferences and the Sequential Monte Carlo are explained. Then the particle filter algorithm is introduced before adopting it for robot localization problems [145].

Markov Model

To understand how particle filters work one has to know the basics of state-space models. Those start with a Markov Model which is a common approach for modelling sequences [49, 119, 114]. Let Q_n be a state variable where n indicates the time. Q_n can be a discrete or stochastic random variable which can take any state within its state space. The aim of the Markov Model is to evaluate a sequence of states $Q_1, ..., Q_N$, where N is the length of the sequence, and to assign a probability $P(Q_1, ..., Q_N)$ to this sequence. By remodeling this probability one receives the following factorization:

$$P(Q_{1:N}) = P(Q_1, ..., Q_N) = P(Q_1)P(Q_2|Q_1)P(Q_3|Q_2, Q_1) \cdot ... \cdot P(Q_N|Q_{N-1}, ..., Q_1) \tag{2.10}$$

This means that the number of probabilities increases exponentially with the length of the sequence N, since every sequence state can be combined with all its predecessor states [114]. To solve this problem the context is restricted. This also has the advantage that the model can be used for sequences of different size. The basic idea of the restriction is to not include all previous states for estimating the probability of the current state. Instead, only a few previous states are taken into account when estimating the probability of a state. The most common model is the bigram model which restricts the influence of previous states to one such that

$$P(Q_{1:N}) = P(Q_1, ..., Q_N) = P(Q_1) \sum_{i=2}^{N} P(Q_i|Q_{i-1}). \tag{2.11}$$

This model is also called Markov Model of 1st order. The parameter for this model are the prior probabilities $P(Q_1)$ for the first state at the beginning and the transition probability $P(Q_i|Q_{i-1})$. These probabilities can be trained using a maximum-likelihood estimation [27].

To summarize the terminology above, a Markov Model consists of the following parameters $\Theta = \{\pi, A\}$:

1. It holds a quantity of possible states $S = \{s_1, ..., s_{N_S}\}$ where N_S is the number of states
2. $\pi_i = P(Q_1 = i)$ is the probability for the state $i \in S$ at time 1. This is also called the prior probability.
3. $a_{ij} = P(Q_n = j|Q_{n-1} = i)$ is the transition probability from state $i \in S$ to $j \in S$. Those transition probabilities are summarized in matrix $A = [a_{ij}]_{N_S \times N_S}$.

Hidden Markov Model (HMM)

In contrast to Markov Models, Hidden Markov Models cannot observe the state Q_n. Here the states are "hidden" and only observable through observations X_n. Those X_n can be discrete or continuous. To conclude from an observation to a state, observation probabilities $P(X_n|Q_n)$ exist. This gives the probability to see X_n if we assume Q_n. It is assumed that observations only depend on the current state and not on other states or observations. Figure 2.5 shows how a HMM is structured. It illustrates how transition and observation probability work.

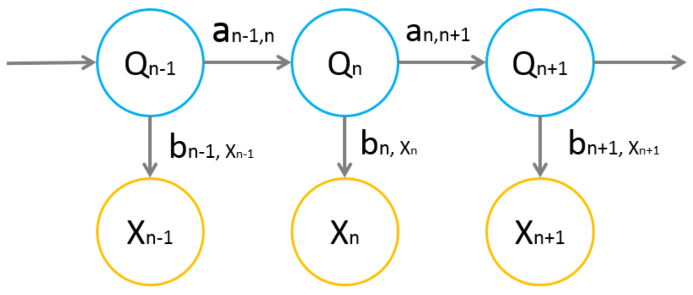


Figure 2.5: Structure of a Hidden Markov Model

The parameter of a HMM $\Theta = \{\pi, A, B\}$ are the same like for the Markov Model and additionally the following extensions are valid:

1. HMMs have additional observation probabilities which can be discrete or continuous. Those probabilities are summarized in the symbol B.
   (a) discrete observations: $x_n \in \{\theta_1, ..., \theta_k\} : b_{i,x_n} = P(X_n = x_n|Q_n = i)$

(b) continuous observations: $x_n \in \mathbb{R}^d : b_{i,x_n} = P(x_n|Q_n = i)$. The continuous observations can be modelled using different distributions.

2. For HMMs only Markov Models of 1st order are used. So the current state depends only on one previous state.

3. Observations X_n depend only on Q_n.

The goal of Hidden Markov Models is to estimate the state Q_N, given all the observations up to that time $P(Q_N|X_1 : N)$. Another approach would be to find the posterior distribution $P(Q_{1:N}|X_{1:N})$ sequence of states. This distribution can be computed using different methods. This thesis handles the **Bayesian Inference** and the **Sequential Monte Carlo method**.

Bayesian Inference

Bayesian inference is a method which uses Bayes' theorem to update a probability distribution as more observations are made [99, 11]. It uses the Bayes theorem

$$P(Q_{1:N}|X_{1:N}) = \frac{P(Q_{1:N}, X_{1:N})}{P(X_{1:N})} \tag{2.12}$$

to rewrite the joint probability

$$P(Q_{1:N}, X_{1:N}) = P(Q_{1:N-1}, X_{1:N-1})P(Q_N|Q_{N-1})P(X_N|Q_N). \tag{2.13}$$

This can again be used to form the equation

$$P(Q_{1:N}|X_{1:N}) = P(Q_{1:N-1}, X_{1:N-1})P(Q_N|Q_{N-1})P(X_N|Q_N) \tag{2.14}$$

At the end two steps are retrieved. The **update step**

$$P(Q_N|X_{1:N}) = P(X_N|Q_N)P(Q_N|X_{1:N-1}) \tag{2.15}$$

and the **prediction step**

$$P(Q_N|X_{1:N-1}) = P(Q_N|Q_{N-1})P(Q_{N-1}|X_{1:N-1}) \tag{2.16}$$

where $P(Q_{N-1}|X_{1:N-1})$ is known through recursion and $P(Q_N|Q_{N-1})$ is the transition probability. Bayesian Inference has on major problem: It is often hard to deal with these distributions in closed-form. This happens especially in non-Gaussian and non-linear models.

Sequential Monte Carlo Method (SMC)

A solution for the intractable distributions are Sequential Monte Carlo methods. Those methods are the predecessor of particle filters [5, 33]. They are used to approximate the distribution by representing probability distributions using a large number of particles K. A property which also holds for particle filters is that as $K \to \infty$, the distribution converges to the correct distribution [41]. For SMC one can use **importance sampling** [104] to simulate K independent and identically distributed (i.i.d.) particles $Q_{1:N}^{(i)}|_{i=1}^{K}$ according to an arbitrary importance sampling distribution $\hat{\pi}(Q_{1:N}|X_{1:N})$. This is done to overcome the problem of computationally expensive sampling and to sample complex distributions. The empirical estimates are then

$$P_K(Q_{1:N}|X_{1:N}) = \frac{1}{K}\sum_{i=1}^{K} \delta_{Q^{(i)}_{1:N}}(Q_{1:N}) W^{(i)}_N \tag{2.17}$$

where $\delta_{Q^{(i)}_{1:N}}$ denotes the delta function which is located in $Q^{(i)}_{1:N}$. $W^{(i)}_N$ holds the importance weights with

$$W^{(i)}_N = \frac{w(Q^{(i)}_{1:N})}{\sum_j w(Q^{(j)}_{1:N})} \tag{2.18}$$

and

$$w(Q_{1:N}) = \frac{P(Q_{1:N}|X_{1:N})}{\hat{\pi}(Q_{1:N}|X_{1:N})}. \tag{2.19}$$

An issue which occurs with importance sampling is that it can be hard to find a good importance distribution and it is not usable for recursive estimation. This means when a new observation X_{N+1} is made, the previously predicted samples and weights cannot be reused. A solution for this issue is **sequential importance sampling** [93]. Lets assume that one can factor the importance distribution as

$$\hat{\pi}(Q_{1:N}|X_{1:N}) = \hat{\pi}(Q_{1:N-1}|X_{1:N-1})\hat{\pi}(Q_N|Q_{1:N-1}, X_{1:N}) \tag{2.20}$$

where the first multiplicand is the importance distribution at time $N-1$ and the second multiplicand is the extension to time N. Equation 2.20 can then be reformed into

$$\hat{\pi}(Q_{1:N}|X_{1:N}) = \hat{\pi}(Q_1|X_1)\prod_{n=2}^{N}\hat{\pi}(Q_n|Q_{1:n-1}, X_{1:n}) \tag{2.21}$$

and out of this the importance weight can be recursively evaluated

$$W^{(i)}_N \approx W^{(i)}_{N-1}\frac{P(X_N|Q^{(i)}_N)P(Q^{(i)}_N|Q^{(i)}_{N-1})}{\hat{\pi}(Q^{(i)}_N|Q^{(i)}_{1:N-1}, X_{1:N})}. \tag{2.22}$$

Having defined this, one can now simulate

$$Q^{(i)}_N \sim \hat{\pi}(Q_n|Q^{(i)}_{1:N-1}, X_{1:N}) \tag{2.23}$$

and update the weight $W^{(i)}_N$ for $Q^{(i)}_{1:N}$ based on the previously calculated weights $W^{(i)}_{N-1}$. To reduce the variance one can take the newly generated distribution for sampling. This is called resampling and improves the results of Sequential Monte Carlo. Providing the current weighted particles, one can resample as follows:

$$P(Q_N|X_{1:N}) \approx \sum_{i=1}^{K} W^{(i)}_N \delta_{Q^{(i)}_N}. \tag{2.24}$$

Since this equation replaces the particles with new ones one has to keep in mind that a particle with a small weight is unlikely to be drawn and a particle with a large weight might be drawn multiple times.

Particle Filter Algorithm

The sections above describes the background which is needed to understand particle filters. This section now deals with the algorithm that combines theoretical and practical knowledge. The key idea of particle filters is to spread particles in space which represent the posterior distribution [146]. Instead of using a parametric form for representing the distribution, particle filters generate samples based on its own distribution. An advantage is that this is non-parametric and that it can represent complex distributions.

A problem which occurs with the sampling step in Equation 2.24 is that a discrete distribution is used to approximate a continuous one. This leads to the issue that in discrete space the probability for two particles which are sampled by its distribution to be identical is greater than 0. In contrast, the continuous distribution never draws two particles identically. In the particle filter one solves this issue by approximating the distribution from Equation 2.24 using a kernel density estimate which works with the particles instead of using them directly. Thus we can rewrite the resampling equation as

$$P(Q_N|X_{1:N}) \approx \sum_{i=1}^{K} W_N^{(i)} KF(Q_N - Q_N^{(i)}). \tag{2.25}$$

where $KF(Q_N - Q_N^{(i)})$ denotes a kernel function which is located at $Q_N^{(i)}$. One can now define a complete particle filter using the state and update equations from **sequential importance sampling** and combine it with the resampling step from Equation 2.25.

To use the formal equations which were described in this section, one needs an algorithm which combines all the necessary steps [146]. For the algorithm we define a set of particles

$$X_t = x_t^{[1]}, x_t^{[2]}, ..., x_t^{[M]} \tag{2.26}$$

where each particle $x_t^{[m]}|_1^M$ represents the state space at time t. More concrete, each particle is a representation of the state at time t. M is the number of particles. The complexity of the algorithm increases with the number of particles chosen.

```
function ParticleFilter($X_{t-1}, u_t, z_t$)
  $\bar{X}_t = X_t = \emptyset$
  for $m = 1$ to $M$ do
    sample $x_t^{[m]} = P(x_t|u_t, x_{t-1}^{[m]})$
    $w_t^{[m]} = P(z_t|x_t^{[m]})$
    $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
  endfor
  for $m = 1$ to $M$ do
    draw $i$ with probability $\approx w_t^{[i]}$
    add $x_t^{[i]}$ to $X_t$
  endfor
  return $X_t$
```

Listing 2.9: The particle filter algorithm [146]

The aim of a particle filter is to approximate the probability of x_t by a set of particles X_t and to update the probability based on the observations. Listing 2.9 describes the algorithm for a particle filter. u_t denotes some action which is taken at the current time step. This is chosen using the transition probability defined in Section 2.3.4. z_t denotes the observation which was made after executing action u_t. Sampling can e.g. be done with one of the previously discussed methods, the importance sampling or the sequential importance sampling. The particle filter can be split up in three main parts

1. **Transition** is the fist phase. It applies the given action u_t to each particle, which lets them move.
2. **Evidence**: In this phase the particles which were moved in the transition phase are weighted again based on the observation z_t.
3. After having weight the samples **resampling** takes place. In this phase M new samples are created, based on the new weights from the evidence phase.

Monte Carlo Localization

Having defined the basic particle filter algorithm one can now use it to localize a robot within its environment. This section adapts the particle filter which was described in Section 2.3.4 and presents an example for a needed motion model and a measurement model. Monte Carlo Localization is an approach which is based on particle filtering and can be applied to the local as well as global localization problem [146, 147, 51]. To use the particle filter one has to define a motion model which samples a particle based on the actual motion. In robot localization this is usually done by using a motion model which relies on some measurements of the odometry. In scope of this work an motion model based on odometry sensors is used. The sampled particles are then used to determine their importance by calculating their weights, using a measurement model. Using the same notations as in Section 2.3.4, one can adapt the particle filter to receive the Monte Carlo localization algorithm which is presented in Listing 2.10.

```
function MCL($X_{t-1}, u_t, z_t, m$)
  $\bar{X}_t = X_t = \emptyset$
  for $m = 1$ to $M$ do
    $x_t^{[m]}$ = sample_motion_model($u_t, x_{t-1}^{[m]}$)
    $w_t^{[m]}$ = measurement_model($z_t, x_t^{[m]}, m$)
    $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
  endfor
  for $m = 1$ to $M$ do
   draw $i$ with probability $\approx w_t^{[i]}$
   add $x_t^{[i]}$ to $X_t$
  endfor
return $X_t$
```

Listing 2.10: The Monte Carlo localization algorithm [146]

For the motion model one can basically choose between *velocity motion model* and *odometry based motion model*. Velocity models use dead reckoning to determine the robots position while odometry based models use wheel encoders to determine the motion [146, 45]. In this thesis an implementation of the **odometry motion model** is used to sample particles. It samples a particle x_{t-1} based on an given action u_t which is the transition from one state into another. x_{t-1} and u_t hold position information which is denoted as

$$x_{t-1} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}, u_t = \begin{pmatrix} \bar{x}_{t-1} \\ \bar{x}_t \end{pmatrix} \tag{2.27}$$

with

$$\bar{x}_{t-1} = \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{\theta} \end{pmatrix}, \bar{x}_t = \begin{pmatrix} \bar{x}' \\ \bar{y}' \\ \bar{\theta}' \end{pmatrix}. \tag{2.28}$$

the algorithm uses four parameters α_1 to α_4 which represent the error in odometry sensors. Listing 2.11 shows the pseudo code of the currently used motion model. The function *sample*(σ^2) uses a zero-centered distribution to generate a random sample [146]. Using a normal distribution this could be e.g.

$$\frac{1}{2} \sum_{i=1}^{12} \text{rand}(-\sigma, \sigma) \tag{2.29}$$

where *rand* is a pseudo random number generator which uses a uniform distribution based on the parameters.

```
function sample_motion_model($u_t$,$x_{t-1}$)
    $\delta_{rot1} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
    $\delta_{trans} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
    $\delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1}$

    $\hat{\delta}_{rot1} = \delta_{rot1} - \text{sample}(\alpha_1 \delta_{rot1}^2 + \alpha_2 \delta_{trans}^2)$
    $\hat{\delta}_{trans} = \delta_{trans} - \text{sample}(\alpha_3 \delta_{trans}^2 + \alpha_4 \delta_{rot1}^2 + \alpha_4 \delta_{rot2}^2)$
    $\hat{\delta}_{rot2} = \delta_{rot2} - \text{sample}(\alpha_1 \delta_{rot2}^2 + \alpha_2 \delta_{trans}^2)$

    $x' = x + \hat{\delta}_{trans} \cdot \cos(\theta + \hat{\delta}_{rot1})$
    $y' = y + \hat{\delta}_{trans} \cdot \sin(\theta + \hat{\delta}_{rot1})$
    $\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$

    return $x_t = (x', y', \theta')^T$
```

Listing 2.11: The odmometry motion model [146]

Figure 2.6 shows the three parameters δ_{trans}, δ_{rot1} and δ_{rot2} of the motion model. The movement of a robot in one time interval $t-1$ to t is approximated by a translation δ_{trans} and two rotations. One at the beginning and at the end (δ_{rot1} and δ_{rot2}).

To measure the environment one needs a **measurement model**. It uses data which is retrieved from the real world environment by sensors. Nowadays there are various sensor types like range finders, cameras or tactile sensors which retrieve information of the environment and use different measurement models [146]. In this thesis a currently used model for a 2D laser range finder is described. Lasers are similar to sonar sensors, they send out signals and record the received echo. The difference between those two sensors are the signal

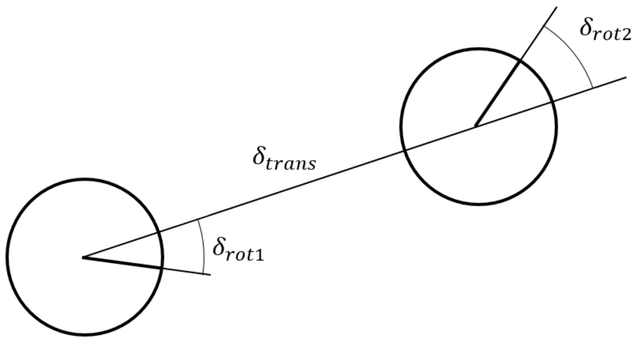


Figure 2.6: Visualization of the odometry model

types which are sent out. In case of sonar an ultrasonic beam is emitted while laser sensors use light beams. Since those two methods work in the same way they both can be used for the measurement model described below.

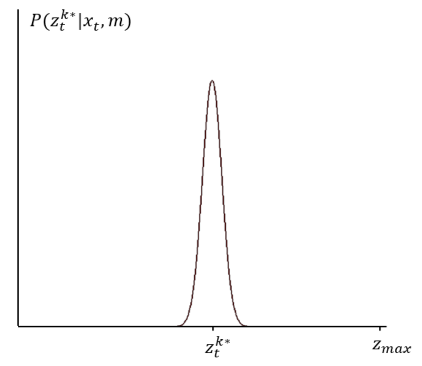
```
function measurement_model($z_t, x_t, m$)
  $q = 1$
  for $k = 1$ to $K$ do
    compute $z_t^{k*}$ for the measurement $z_t^k$ using ray casting
    $p = z_{hit} \cdot P_{hit}(z_t^k | x_t, m) + z_{short} \cdot P_{short}(z_t^k | x_t, m)$
        $+ z_{max} \cdot P_{max}(z_t^k | x_t, m) + z_{rand} \cdot P_{rand}(z_t^k | x_t, m)$
    $q = q \cdot p$
  return $q$
```

Listing 2.12: The measurement model [146]

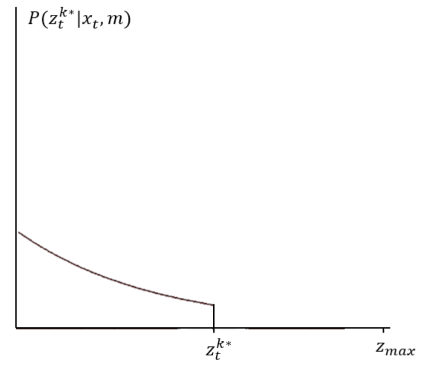
Listing 2.12 describes the currently used measurement model for beam range finders. It uses four parameters $z_{hit}, z_{short}, z_{max}, z_{rand}$ which weight the outcome of four different distributions with $z_{hit} + z_{short} + z_{max} + z_{rand} = 1$. The algorithm uses a loop to iterate over all K scan points. Since the robot can be displaced and the given scan range z_t^k might not be correct, z_t^{k*} is computed. It is the actual range of the current scan point. To receive this value, typically ray casting is done.

The beam measurement model uses four different sections within the range to compute the probabilities. Those sections depend on the actual measurement retrieved from z_t^{k*}. P_{hit} is a Gaussian distribution at the location where it is most likely to hit an object. P_{short} defines the probability for a scan point to be in the short range using an exponential distribution. P_{max} defines the probability for the maximum range and P_{rand} adds a random factor to the probabilities, both using uniform distribution. Figure 2.7 illustrates the sections for the measurement distribution.

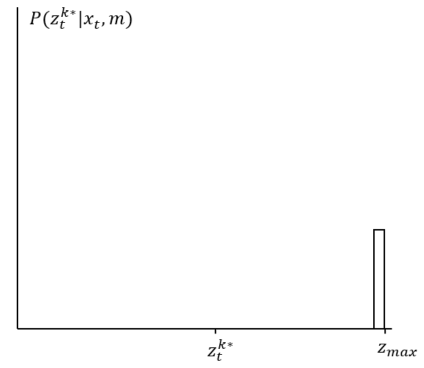
Putting all four sections together results in a pseudo-density of the mixture distribution [146]. This mixture distribution is calculated in listing 2.12 and is



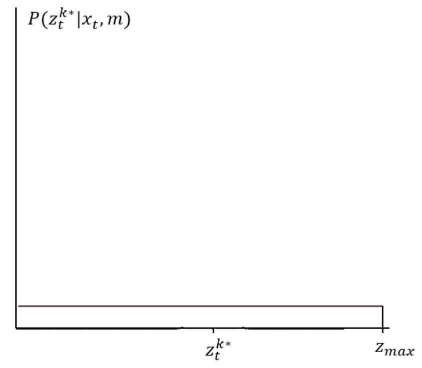
(a) Gaussian distribution P_{hit}



(b) Exponential distribution P_{short}



(c) Uniform distribution P_{max}



(d) Uniform distribution P_{rand}

Figure 2.7: Sections of measurement model as described in [146]

mathematically defined as

$$w_t = \prod_{k=1}^{K} P(z_t^k|x_t, m) = \prod_{k=1}^{K} \begin{pmatrix} z_{hit} \\ z_{short} \\ z_{max} \\ z_{rand} \end{pmatrix}^T \begin{pmatrix} P_{hit}(z_t^k|x_t, m) \\ P_{short}(z_t^k|x_t, m) \\ P_{max}(z_t^k|x_t, m) \\ P_{rand}(z_t^k|x_t, m) \end{pmatrix}. \tag{2.30}$$

where w_t is the resulting weight for particle x_t and scan observation z_t. Figure 2.8 shows the complete pseudo-density of the measurement model.

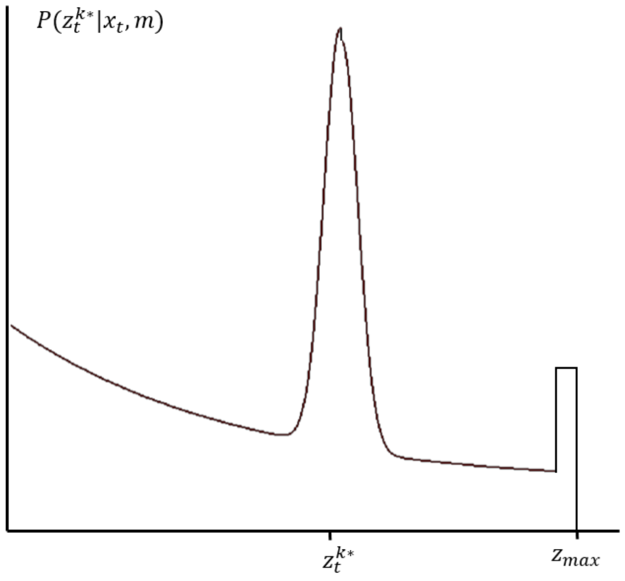


Figure 2.8: Pseudo-density of the beam range mixture model

2.4 Neural Networks

After having explained the major prerequisites to understand the robots localization, the next important topic is discussed. Neural networks are a key topic in this thesis. They are needed to train and validate datasets which are generated to estimate the position accuracy of a robot.

This section focuses on the most important things which are needed for understanding neural networks. First the basics are explained to get a rough overview and to refresh the knowledge on neural networks. Then it is explained which data can be used to train a neural network and how it has to be used. It is then shown how prepared data can be used to train a neural network. Therefore two main learning algorithms are discussed. Having discussed the basics of neural networks, more complex network architectures which are needed in this thesis are presented. Therefore all necessary information like the principle idea and the learning mechanism are discussed.

2.4.1 Basics

An artificial neural network is a crucial machine learning concept which is trained to make decisions based on the input data [62, 68]. The idea of this machine learning approach comes from nature or, to be more precise, from brains [62, 25]. It is leaned to simulate a functioning brain which consists of neurons that are connected with each other. The principle of neural networks is similar but the output is decided by the way neurons are connected with each other, how the network is trained and which data is used.

Neurons

In neural networks a simple component is called neuron. A simple type of such neurons is a **perceptron** which was introduced by Frank Rosenblatt [123]. The idea of a perceptron is to take several binary inputs $x_1, ..., x_n$ and use

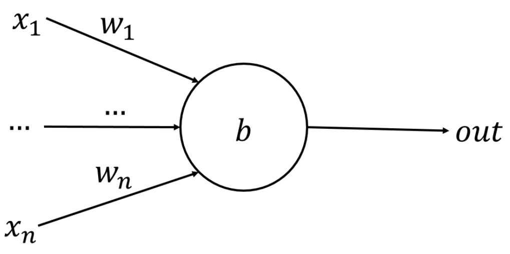


Figure 2.9: A component of neural networks: the perceptron

them to produce one binary output *out*. To compute a meaningful output Rosenblatt introduced weights $w_1, ..., w_n$ which are real numbers and indicate the importance of a single input x_i. The output is then calculated using the sum of the inputs and its weights $\sum_j w_j x_j$ and a given threshold b. Since the output is binary it can either be 0 or 1. This is done by checking the sum of the weighted inputs using the threshold

$$out = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq b \\ 1 & \text{if } \sum_j w_j x_j > b \end{cases}. \tag{2.31}$$

Usually, neurons are illustrated like in Figure 2.9. To simplify Equation 2.31 one can write the weighted sum $\sum_j w_j x_j$ as dot product, resulting in $\mathbf{w}^T \cdot \mathbf{x} = \sum_j w_j x_j$ where **w** and **x** indicate vectors. The next step is to make the threshold from Equation 2.31 to the other side. To further simplify the equation, the threshold is from now on called *bias* and set negative $b = -b$. The outcome of those modifications is then

$$out = \begin{cases} 0 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b > 0 \end{cases}. \tag{2.32}$$

By changing the weights **w** of the inputs and the bias b one can change the outcome of the model [62]. This is the basic idea behind training neural networks. The action of selecting an output for the perceptron is called **activation function**. It defines how well a neuron is satisfied to the input. In the case of perceptrons it is a so-called *step function* which either 0 or 1, depending on the input. To find the best value for weights such that a wished output is retrieved some kind of artificial training is needed. The approach of optimizing the weights is to adopt the weights with a small error to see how the output changes. However, when using perceptrons and its step-activation function like introduced above, a small change in a weight can cause the output to flip e.g. from 0 to 1. This might change the behaviour of a network completely and might lead to an incorrect network. To overcome this issue **sigmoid neurons** were introduced [92]. A sigmoid neuron uses a **sigmoid activation function** and holds the same variables like a perceptron: weights **w**, inputs **x** and a bias

b. Another difference is that inputs $\mathbf{x}$ are not restricted to be binary but to be a value between $0 \leq x_i \leq 1$. Also the output is not 0 or 1 but computed as

$$out = \sigma(\mathbf{w}^T * \mathbf{x} + b) \tag{2.33}$$

where σ is the sigmoid function which is defined as

$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \tag{2.34}$$

such that the output can be calculated by

$$out = \frac{1}{1 + \exp^{-\sum_j w_j x_j - b}}. \tag{2.35}$$

Basically, sigmoid neurons obtain their advantage from the smoothness of the sigmoid function. Due to this smoothness a small change $\triangle w_i$ in weight w_i or $\triangle b$ in bias b will only produce a small change $\triangle out$ in the output. In fact, the change in the output $\triangle out$ can be approximated by

$$\triangle out \approx \sum_j \frac{\partial out}{\partial w_j} \triangle w_j + \frac{\partial out}{\partial b} \triangle b \tag{2.36}$$

where $\partial out / \partial w_j$ denotes the partial derivative of the output w.r.t. w_j and $\partial out / \partial b$ is the partial derivative w.r.t. b.

Constructing Neural Networks

Understanding neurons and their functionality, one can now start to construct a neural network. Basically, a neural network is the combination of various neurons which are connected with each other. To get an output for a certain input set one then has to propagate the input values through the network using the defined weights and biases. The creator of a network can construct any kind of network structure by just adding neurons to the network and connecting them with other neurons [25]. That is the theory. In practice the construction of networks follows some conventions. Often a network is build up using **layers**. Those layers are a part of neural networks and contain a certain number of neurons that are connected with each other in a specific way.
First of all, one needs to define the number of input neurons. This is done by choosing the number of needed inputs. Every input enters the network through its own input neuron which can then be distributed and combined to other neurons. Putting those input neurons together one receives an **input layer**. The same holds for the **output layer**. By defining a desired number of outputs one creates the output layer. If one likes to only sent inputs to an output layer, one can already construct a network for his needs. In practice, more layers are added between input and output layer. Those are called **hidden layers** and can contain any number of neurons. Also the connections within the layers can be chosen arbitrarily. There are different types of hidden layers, some of them are presented and discussed in Section 2.4.4. Figure 2.10 shows an example of a neural network. It uses three input neurons and two output neurons. In-between a hidden layer with four neurons which are fully connected to the other layers is added.

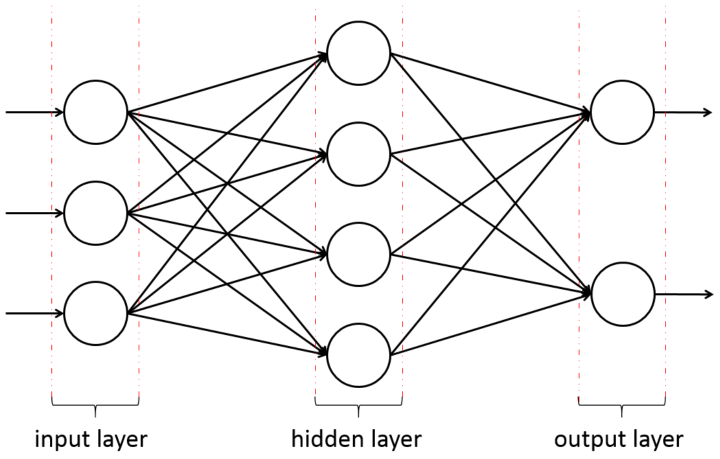


Figure 2.10: An example construction of a neural network

2.4.2 Data Generation for Learning

To optimize the performance of a network one has to adapt the weights and biases of all neurons. This is done by applying training algorithms as described in Section 2.4.3. However, to train a network and to optimize the neurons one needs a vast amount of data [40, 81, 66, 21]. This data is a collection of input sets which are used to train, test and validate a network. Therefore, the data is split up in three sets: the training set, the test set and the validation set. The training set is used to adapt the weights and biased of the neurons within the network while the test and validation set verify the network structure. There are different possibilities for training which use different types of datasets. Based on the data structure one can separate the learning methods into unsupervised, supervised and semi-supervised learning. This thesis deals with supervised data but for the sake of completeness the other two possibilities are also briefly explained. To see how data is generated which is used for this thesis look at Section 4.3.1.

Underfitting and Overfitting

Since the network is trained only on training data, the danger of overfitting and underfitting occurs [36, 151]. Depending on the network structure and the training set, the training algorithm might tend to overfit. The definition of overfitting is a low training error and a high test error. This results in a network that works perfectly fine with the training set but does not find the correct output for new input data. The main reason for overfitting is a training set that contains only specific information and does not represent the general situation. The network is optimized based on the specialized data which then leads to a low training error but when testing it, the error raises. When the training data is not specific enough or the information is not extracted correctly underfitting might occur. Then the training error and the test error is high. A reason for underfitting is wrong or imprecise input data. If data is used that does not hold relevant information to train, the network cannot identify necessary features to optimize the weights and biases. This results in a high training and testing error. To reduce the risk of over- and underfitting, a test set is used to check how well a network performs on datasets which are not trained. This check is done during the training to possibly adapt the training steps. After training the validation set is used to finally validate the received network. This is used to see how data which is completely independent from the training performs.

Supervised Learning

Supervised learning is the task to train the network given pre-labelled data [40, 81]. This means the data which is used to train the network knows what the result should be. Typically it is represented as pair of an input vector and the desired value for the output. Learning algorithms which use this kind of training sets are more likely to score high accuracy in training since the network structure can be adapted according to the desired output. Formally, given N labelled training samples of the form $\langle x_i, y_i \rangle$ where x_i is the input vector and y_i is the desired output label. A learning algorithm then produces a network where the input optimally describes the label such that $f : X \rightarrow Y$, where X is the space of the input which is mapped onto the output space Y. The most common machine learning approaches which use supervised data are support vector machines (SVM) [141], linear regression [161], logistic regression [26] aw well as neural networks.

Unsupervised Learning

Unsupervised learning approaches do not have a desired output label for their training data [66]. Although neural networks are often more efficient when using supervised training samples, some research has proven that also good results with unsupervised data can be achieved [88]. The problem which unsupervised learning has to face is that it cannot evaluate the accuracy of the current network structure while learning. To overcome this issue many different approaches are used like the k-means algorithm [77] for clustering or hebbian learning for neural networks [101]. Preprocessing of the given data is a common approach to improve the accuracy of algorithms which use unlabelled data. This can be done with methods like the principal component analysis (PCA) [75].

Semi-Supervised Learning

Semi-supervised learning is a modification of supervised learning [21]. It is suited for methods which can combine both, labelled and unlabelled data. Typically there exists only a small amount of labelled data and a vast amount of unlabelled data. It is shown that the accuracy of a machine learning algorithm which uses unlabelled data combined with a small amount of labelled data is considerable improved [21]. This technique is often used when it is hard to generate supervised data e.g. in robotics where one often needs a human agent who labels a small amount of data.

2.4.3 Training Neural Networks

Having constructed the structure of a neural network and prepared the desired training data, one can now start to search for the optimal weight/bias parametrization to solve a given problem. This can be done by using different training algorithms. Some of them are described in this section. When training networks a lot of training data is used. Based on this data the weights are adapted and optimized. However, be aware that neural networks do not guarantee an optimal solution and might get stuck in local minima [113, 62]. This section introduces two main methods which are used to train neural networks. While backpropagation is an algorithm which is widely used to train different types of networks [85], backpropagation through time focuses on networks which want to learn sequences like it is used in recurrent networks [143].

Basically there are two different network structures. **Feedworward** networks allow only a propagation of the input data to the next layer. This means that the data stream flows from the input to the output without any loops in the network. In recurrent network structures, single loops are allowed. Single loops are created by using the output of a neuron as input of the same neuron again. For a more detailled description see Section 2.4.4. Thus it is possible to propagate data back to a predecessor layer. Backpropagation is an algorithm which can be used to train feedworward networks. It is not possible to train a recurrent structure with that algorithm. Therefore an adoption was made and backpropagation through time was invented.

Backpropagation

To train a neural network it is necessary to train an error function. By selecting an error function and reducing the error, one can improve the performance of the network [126]. A common approach therefore is backpropagation which uses gradient descent [146]. Given the error function one can measure the difference between the desired output and the actual output. Then this gap is used to adapt the weights and biases of the network to reduce the error.

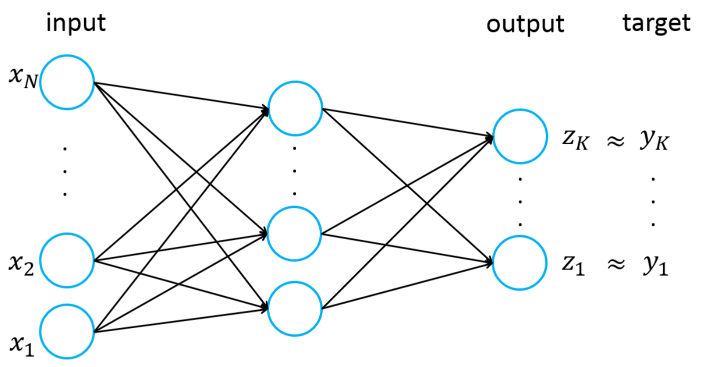


Figure 2.11: The network structure for backpropagation

In backpropagation, the error of a single neuron can be calculated by

$$e_k = z_k - y_k \tag{2.37}$$

where z_k is the actual output of neuron k and y_k is the desired output (label). Given K output neurons, one can define the classical measurement function of the error which is the sum of the squared errors

$$E^{(i)} = \frac{1}{2}\sum_{k=0}^{K} e_k^2 = \frac{1}{2}\sum_{k=0}^{K}(z_k - y_k)^2. \tag{2.38}$$

Figure 2.11 shows the notation of the used variables above. To reduce the error of the function one has to choose new weights that minimize the measurement of the error. Since no analytic solution is possible, gradient descent is used [87]. To improve the network and update its weights, the gradient of the error function is needed. To calculate the gradient two message passes are needed. The **forward pass** and the **backward pass**. Using these passes, the error gradient can be calculated as

$$\frac{\partial E^{(i)}}{\partial w_{kj}} = \delta_k z_j \tag{2.39}$$

where w_{kj} is the weight between neuron j and k, δ_k is the error of neuron k and z_j is the output of neuron j. Forward pass calculates the activation and outputs of all neurons $\mathbf{z}$. Therefore the input $\mathbf{x}$ is used and propagated through the network. To start, every input neuron receives the input from the vector, then the output for every neuron j which comes after the input layer is calculated as

$$z_j^{(a)} = f_j^{(a)}\left(\sum_{l=1}^{A} w_{jl}^{(a)} z_l\right) \tag{2.40}$$

where a is the a-th layer within the network with $1 < a \leq b$. This is defined because the first layer only receives the input values without weights which are propagated forward. b is the number of layers. A is the number of inputs for neuron j and z_l is the output of the predecessor neuron. f_j is the activation function which is used. As described in Section 2.4.1, a sigmoid function $\sigma(x)$ is often used. Of course every neuron can have its own activation function which might lead to various different activation functions within a network.
After all outputs z_j of the output layer are calculated, backward transmission takes place. Here, the error δ_k is calculated backwards. Starting from the output neurons with

$$\delta_k = \frac{\partial f_k(a_k)}{\partial a_k} e_k \tag{2.41}$$

where f_k is the activation function for neuron k and a_k is the activation value. As described in Section 2.4.1, the activation value is the sum of the weighted inputs and its bias $a_k = \mathbf{w}^T\mathbf{x} + b$. For every hidden neuron, the error is calculated as

$$\delta_k = \frac{\partial f_k(a_k)}{\partial a_k} \sum_{r \in post(k)} \delta_r w_{rk} \tag{2.42}$$

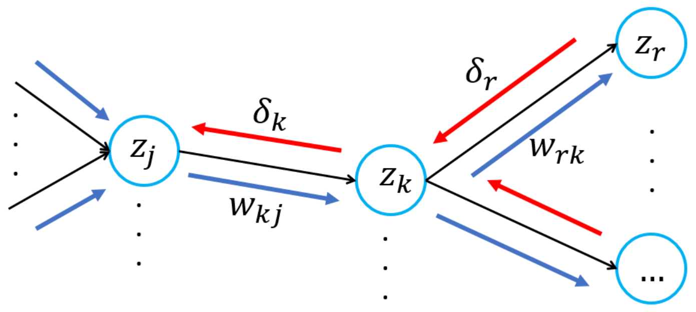


Figure 2.12: Forward transmission and backward transmission

where $post(k)$ are the successor neurons of k. Figure 2.12 illustrates the forward and backward transmission. First, forward transmission takes place which is shown with the blue arrows. Then the error is propagated back by calculating δ_k and sending it back with the red arrows.

This algorithm is usually executed for each sample set until a stopping criteria is reached. Since this algorithm does not guarantee an optimal solution, it might converge to a local minimum. It is also not guaranteed that the algorithm converges. To train a network using backpropagation two different methods can be used which determine when the error gradient is calculated. In **batch learning** the error gradient is calculated from a collection of training samples and then accumulated. So the update of the weights is only done after a collection of samples is seen:

$$w_{kj} = w_{kj} - \eta \nabla E \tag{2.43}$$

where η is parameter which defines the learning rate and

$$E = \sum_{i}^{m} E^{(i)}. \tag{2.44}$$

In **online learning**, the weights are updated after each sample i is propagated forward

$$w_{kj} = w_{kj} - \eta \nabla E^{(i)}. \tag{2.45}$$

This method is often much faster and can be used when the data is coming in at real-time. It also has the possibility to escape from a local minimum since every single sample may enforce a significant weight change.

Backpropagation Through Time (BPTT)

To train a network structure which allows loops within layers, the backpropagation through time was invented [157]. It is a method which allows to train recurrent network structures. Recurrent networks are often used to train data sequences, since it can learn to remember information from previous inputs [9, 97]. In contrast to feedforward networks, a recurrent structure is able to encode longer past information, thus it is suitable for sequence modelling. Instead of explaining the complete functionality, this thesis only describes the difference of the structure, the difference in calculating the output and give the error cost function for using BPTT.

The advantage of BPTT is that it tends to be faster for training sequence models than other basic optimization techniques like evolutionary optimization [162]. An disadvantage is that it has some issues with local optima. Local optima are a bigger problem in BPTT than it in feedforward networks.
In a recurrent network errors can be reused for propagating forward. This can be done to a certain number of layers such that a wished sequence can be used. This process is usually called **unfolding** [7]. Due to unfolding the loops of the network structure can be dissolved. Typically a parameter k is defined which determines how deep a loop should be unfolded. When unfolding takes place, one adds a sequence of neurons which take an input x_i and the previously calculated output z_i. Since the output z_i does not exist for the first neuron, a initial value for z_1 has to be specified. This is usually a vector of zeros. Additionally also a time instance t has to be given to the network. Figure 2.13 shows an example of and unfolded network with $k = 3$. it takes neuron f and unfolds it three times. z_t is an initial input which has to be defined. x_i are the sequential inputs at time i. After unfolding the neuron is concatenated with the next neuron to continue processing.

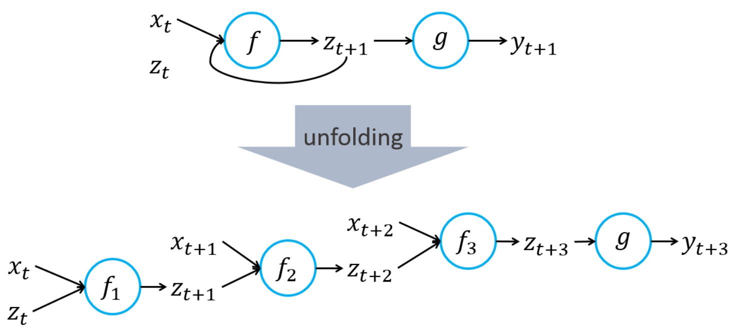


Figure 2.13: An unfolded recurrent neural network with $k = 3$

For recurrent networks a new cost function is defined which measures the error. It is called cross-entropy [124] and shown to perform well in recurrent neural networks. The error function is then defined as

$$E^{(i)} = -\sum_{k=0}^{K} (z_k \ln y_k + (1 - z_k) \ln(1 - y_k)) \tag{2.46}$$

and

$$E = \sum_{i}^{m} E^{(i)}, \tag{2.47}$$

following the same definition as before for the simple backpropagation algorithm. The recurrent neurons also calculate their outputs differently, since they have more inputs.

$$z_j^{(a)}(t) = f_j^{(a)}\left(\sum_{l=1}^{A} w_{jl}^{a} z_l(t) + \sum_{h=1}^{B} w_{jh}^{(a)} z_h(t-1)\right) \tag{2.48}$$

The main difference is the second term. This term adds the weighted output of the neuron form the previous time step, where B is the number of outputs from the previous time step which flow into the current calculation. Also a time interval t has to be introduced. To change the weight one has then to calculate the error δ_k for every output neuron and propagate it back through the unfolded network.

2.4.4 Deep Learning Architectures

This section deals with different network structures that are commonly used for various fields of application. Since a network with a single layer can not solve complex problems, more hidden layers are added to an network. This is basically the definition of a deep neural network [35]. A network becomes deep if many hidden layers are added. Those deep layers can have different forms and structures. There are some common structures which have proved their value over time for certain applications. This section introduces some network structures which are used for state estimation on robot localization scoring. First the convolutional neural networks are introduced. It is a common network for feature detection and image classification [83]. Then neural networks for sequence classification are presented, i.e. recurrent neural network and the long-short term memory. Those networks allow to identify sequences and are used for e.g. speech recognition [67, 91]. After that a combination of both, recurrent network and convolutional network is presented and discussed.

Convolutional Neural Networks (CNN)

Convolutonal neural networks are commonly used to detect features in images which help to classify an image. Therefore, the image is used as input of a neural network structure. In convolutional neural networks one applies a small filter several times on various positions of the image. This method reduces the number of parameters which have to be learned and thus also overfitting is prevented [111]. Typically several types of feature detectors are used where each of them have their own network layer. Those feature detectors are trained to extract valuable information out of the image. Those detectors are then applied on different positions of the image to search for a certain feature. This allows a part of the image to be represented in the same way. Convolutional neural networks use three basic ideas: local receptive fields, shared weights and pooling.

Local Receptive Fields
To understand a convolutional neural network it helps to think of an image as the input. Thus the inputs are organized as a grid. As in artificial neural networks, the input neurons are connected to a hidden layer. The first difference is that not every input pixel is connected to every hidden neuron but the connections are made in small, localized regions from the input image. This

region is called **local receptive field** for the hidden layer. It has a fixed size of L rows and M columns. This local receptive field is then slided across the complete image resulting in a different hidden neuron connection for every field. Figure 2.14 shows an example of the first hidden layer, using a $5x5$ local receptive field. Depending on the size of the local receptive field, the number of hidden neurons in the feature layer decreases. Since the field alsways has to map at least one input neuron to a hidden neuron, the hidden layer is never bigger than the input layer.

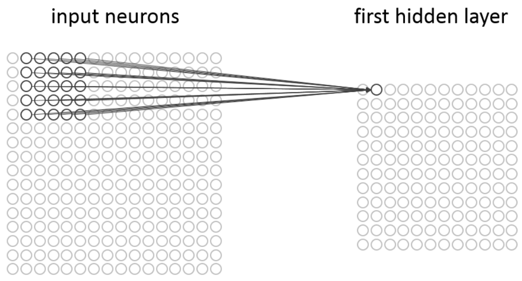


Figure 2.14: A 5x5 local receptive field over the image and creating first hidden layer

Shared Weights and Biases
From Section 2.4.1 one already knows that each neuron holds a bias and weights for its inputs. For each hidden neuron the same weights and bias from the local receptive field is used. So for the (j, k)-th hidden neuron the output is

$$\sigma\left(b + \sum_{l=0}^{L}\sum_{m=0}^{M} w_{l,m}a_{j+l,k+m}\right). \tag{2.49}$$

Here, σ is a sigmoid activation function, b is the bias, $w_{l,m}$ is the array of the input weights and $a_{x,y}$ denotes the input activation at position (x, y).

This leads to the fact that every neuron in the first hidden layer searches for the same feature but on different positions within the map grid. This fact also leads to the advantage of a convolutional network. They do not care about the location of a feature in an image and therefore one does not have to care about the translation of images. Since those hidden layers detect features in the image, it is also called a **feature map**. There is also a possibility to detect various features from the input image by adding more feature maps. Figure 2.15 shows an example input layer with three feature maps. the collection of all feature maps which come from the same input layer is called **convolutional layer** and is the key idea of CNNs which helps to reduce the amount of parameters.

Pooling Layers
Another common layer which a convolutional network uses are so-called pooling layers. Pooling layers usually are inserted after the use of convolutional layers. They are used to simplify the received feature information which was generated in the feature maps. More exact, a pooling layer uses the output of

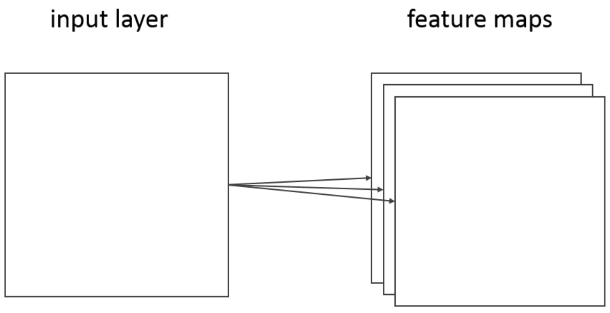


Figure 2.15: Using several feature maps to detect different kind of features

a feature map and prepares a summarized layer. This pooling layer takes a defined number of input neurons, organized in a *NxN* grid, and reduces this number to one. An example is the **max-pooling** method which searches for the maximum activation from the inputs and sends this value to the output. Figure 2.16 shows an example of a 2*x*2 max-pooling layer. It takes four input neurons and only outputs the highest value of these four inputs. Since a

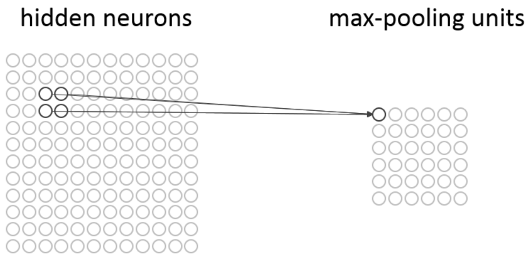


Figure 2.16: Using the max-pooling layer of size (2x2) on the output of the feature map

convolutional layer holds many feature maps, also many pooling layer exist. Every pooling layer is used for a single feature map where the pooling is applied separately. All those layer types can then be used to build a complete convolutional neural network. One just has to put all layers together to obtain a fully functional feature detector. A complex neural network can also contain many convolutional layers which are concatenated. To obtain a correct classification one also needs to add a desired number of output neurons where each neuron indicates a possible solution. Figure 2.17 shows the general structure of a complete CNN containing an input layer, a convolutional layer with three feature maps, corresponding pooling layers and output neurons for solving the problem.

Recurrent Neural Networks (RNN)

The next structure which was already shortly discussed in Section 2.4.3 are recurrent neural networks. RNNs are allowed to form directed cycles within

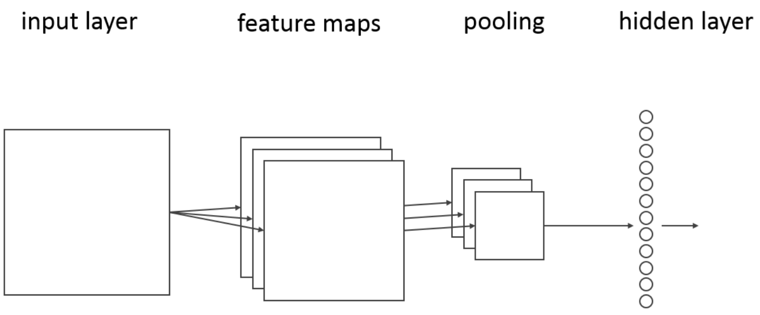


Figure 2.17: The main structure of a convolutional layer which forwards its output to a hidden layer

the network and thus can propagate through time. Due to those loops a kind of internal memory is generates which allows to store information from previous inputs [91, 136]. This internal memory allows to remember previous input information and can change its behaviour based on this input. Thus it is also a common network to train sequences [64] since it performs good on recognizing sequences. The best example for the use of a recurrent network is speech recognition [120]. Speech recognition uses the directed loops within the network structure to store the previous words. Thus the probability of a word occurring depends on the words which came before. This makes sense since a sentence follows grammatical rules and thus not every word combination is equally likely. There are several different implementations of recurrent networks which solve different problems in all kinds of fields. Some of the most common network structures are fully connected RNNs [158], bi-directional RNNs [136] and Long-Short Term memory networks [67]. This thesis only deals with Long-Short Term Memory structures since it will also be used in the implementation part.

A recurrent network is typically organized with RNN units. Such a unit maps the input to a hidden state which then again maps the hidden state to the output. It uses different weight parameters W, b which are trained to receive the output. Figure 2.18 illustrates a simple RNN unit. It receives its input x_t and uses it to calculate the hidden layer h_t and the output value z_t.

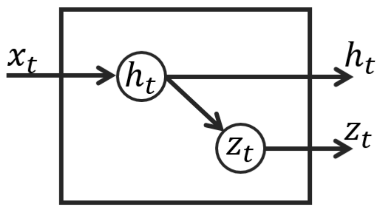


Figure 2.18: Illustration of a RNN unit

The hidden layer vector h_t holds the input and is trained to remember sequences

$$h_t = \sigma_g(W_{h1}x_t + W_{h2}h_t + b_h) \tag{2.50}$$

where x_t is the input vector, σ_g is the sigmoid activation function, h_t is the hidden vector and W_{h1}, W_{h2}, b_h are parameter matrices and vector. The output vector z_t defines what information is passed on

$$z_t = \sigma_g(W_z h_t + b_z) \tag{2.51}$$

where W_z, b_z are parameters and h_t is the output of the hidden layer.

Long-Short Term Memory (LSTM)

LSTM is a recurrent neural network architecture which has an universal field of application. It is shown that given enough network units it is able to compute the same things like a traditional computer can do [67]. A Long-Short Term Memory network is good at classifying, predicting and processing time series. Its special power is that it is also good in learning and evaluating time series which have a time lag of unknown size between two events. It is well suited for training data that really has to remember longer time series and also focuses on the current input. This is a reason why in many fields LSTM is better than other recurrent architectures.

A Long-Short Term Memory network uses so-called LSTM units which are essential for the design of such a network. Those units can solely be used or in addition to other network units. A LSTM unit is a collection of recurrent neurons which is good at remembering information for either a long or short period of time. The period of how long information should be remembered depends on the trained LSTM unit. The reason why LSTM units perform well at their tasks is that they do not use an activation function in its recurrent components. This leads to the advantage that stored information does not get iteratively squashed over time. Also the gradient does not tend to fade away when Backpropagation Through Time is applied for training. LSTM units are usually implemented in **blocks** which contain several units. Such an block typically contains three or four **gates** which control the information flow into or out of the memory. These gates are implemented with the help of logistic functions that compute a probability between 0 and 1. The most common gates are the input gate, the output gate and the forget gate. The **input gate** controls the extent which defines when a new value flows into the memory. A **forget gate** defines how much the unit is allowed to forget and which values remain in memory. An **output gate** defines which values in memory are used to compute the output activation in the block. The only weights W and U in an LSTM block are used to direct the operation of the gates as shown in Figure 2.19.

Elements of a Traditional LSTM

To understand the typical structure of a LSTM network, a LSTM unit is explained. A traditional LSTM uses three gates and a Hadamard product $\circ$. The Hadamard product is defined over two matrices $A, B \in \mathbb{R}^{m \times n}$ such that it is

$$A \circ B = (a_{ij} \cdot b_{ij}) = \begin{pmatrix} a_{11} \cdot b_{11} & \dots & a_{1n} \cdot b_{1n} \\ \dots & \dots & \dots \\ a_{m1} \cdot b_{m1} & \dots & a_{mn} \cdot b_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n} \tag{2.52}$$

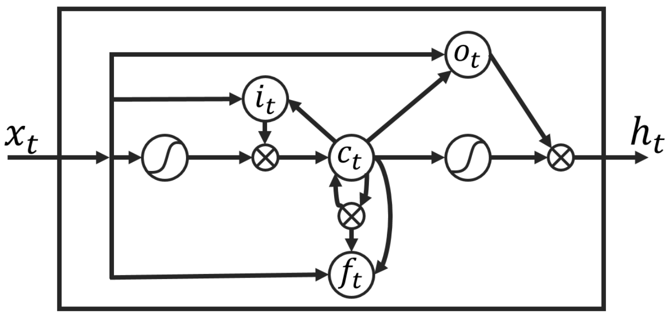


Figure 2.19: A simple LSTM block with only input, output and forget gates. LSTM blocks may have more gates

The forget gate vector f_t holds the weight of how much information should be remembered.

$$f_t = \sigma_g(W_f x_t + U_f h_t - 1 + b_f) \tag{2.53}$$

where x_t is the input vector, σ_g is the sigmoid activation function, h_t is the output vector and W, U and b are parameter matrices and vector. The output gate vector o_t defines what information is used to generate the output and is denoted as

$$o_t = \sigma_g(W_o x_t + U_o h_t - 1 + b_o). \tag{2.54}$$

The input gate vector i_t is defined similarly and which holds the weight for acquiring new information

$$i_t = \sigma_g(W_i x_t + U_i h_t - 1 + b_i), \tag{2.55}$$

The cell state vector c_t is defined as

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_g(W_c x_t + U_c h_{t-1} + b_c) \tag{2.56}$$

where σ_g is the hyperbolic tangent activation function and $c_0 = 0$. The resulting vector h_t also uses the hyperbolic tangent activation function for its steps, except the first one because its first value is defined as $h_0 = 0$

$$h_t = o_t \circ \sigma_c(c_t). \tag{2.57}$$

Long-Short Recurrent Convolutional Networks (LRCN)

Many tasks which are used in computer vision need sequential processing of inputs as well as feature recognition. Until now convolutional neural networks can only evaluate an image for one time step. When a sequence of input images should be trained only few approaches are known. To overcome this issue a new approach for was introduced which is applicable for visual recognition and description [39]. It combines convolutional layers with the recurrent LSTM architecture. In this method, first features are extracted using convolutional layers. Then the result of these layers is used as input to LSTM units. They learn the sequential part and generate the output. Figure 2.20 shows the general structure of a LRCN network. It takes t sequences as input, uses a convolutional layer to detect features which are then trained for sequence learning. Then after the LSTM unit a output is predicted. This is repeated through the complete sequence to receive the output of the sequence. If one wants to predict the data while the sequence is running, one can also get this output while the sequence is running through the network.

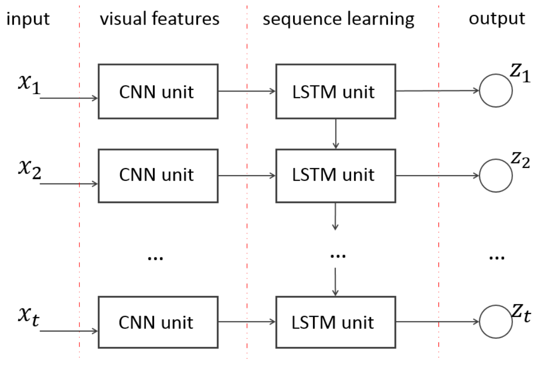


Figure 2.20: A general architecture of a LRCN network

LRCN is deep in terms of time and space. It can be applied on various vision tasks and can also handle sequential inputs as well as sequential outputs. In theory it should be easy to extend existing recognition tools to support them with LRCN mechanisms.

2.5 Caffe

This section discusses some theoretical knowledge which is needed for training deep neural networks with the framework Caffe [73]. Therefore it is presented how Caffe trains its networks and how to use it. A focus is set especially on how data has to be prepared that it is usable for the framework, how networks are modelled and how the framework is used to actually train a network.

2.5.1 Stochastic Gradient Descent (SGD)

As already defined in Section 2.4 it is necessary to define an error function to train a neural network. This error function is then minimized while learning takes place. This method does not guarantee an optimal solution and can converge to a local minimum, leading to a non-optimal solution. There are many different error functions $J(\theta)$ which were proposed to train a neural network. The most common method which is used is the so-called gradient descent. This method was already introduced in Section 2.4.3 but now it will be mentioned a bit more in detail. A simple cost function which can be used for gradient descent is linear regression which is defined as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)}) \right)^2 \tag{2.58}$$

where θ is a given parameter vector that should be optimized, m is the number of training samples used, h_θ is the hypothesis on which the samples should be relied on, $x^{(i)}$ is the i-th training sample and $y^{(i)}$ is the desired output. This cost function tries to find a linear function that is optimal to the given parameter space. It tries find a hypothesis such that $h_\theta(\mathbf{x}) = \mathbf{x}^T\theta \approx y$.

Gradient descent takes some arbitrary cost function $J(\theta)$ and tries to reduce the error. It calculates an error by using the cost function and slightly adapts the parameter space until is converges. The basic gradient descent algorithm is defined as

$$\theta_j = \theta_j - \eta \cdot \frac{\partial}{\partial \theta_j} J(\theta) \tag{2.59}$$

where θ_j indicates the parameter vector at iteration j, η is a defined learning rate with $0 < \eta < 1$ and $\partial/\partial\theta_j$ is the partial derivative of the cost function $J(\theta)$ with respect to θ_j.

Using the example cost function from Equation 2.58 and combining it with gradient descent one receives

$$\theta_j = \theta_j - 2\eta \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)}) \right) \cdot x_j^{(i)} \tag{2.60}$$

where 2η is the learning rate, $h_\theta(x^{(i)} - y^{(i)}$ is the error which is received by the current hypothesis and $x_j^{(i)}$ is the input.
It is computationally intensive to train a neural network with the complete cost function when the number of training samples is high because all training samples have to be computed before applying one training step. Another approach for improving this is **stochastic gradient descent** [164]. This method uses the same definitions as before but instead of training the complete cost function, a parameter update is only performed by a few training samples. The number of training examples is defined as **batch size**. It ca be defined as a number which is less then the training size. If it is one then only one training sample is used for updating the parameter space. The equation for stochastic gradient descent then looks like

$$\theta_j = \theta_j - \eta \cdot \frac{\partial}{\partial \theta_j} J(\theta, \mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \tag{2.61}$$

where $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$ are a batch of training samples. The batch is usually selected by taking randomly selected data samples but for training sequences the order of the sequence has to stay intact. As SGD recomputes the gradient based only on a batch of training samples, it can perform faster parameter updates and often also converges faster. This method can also be used for online learning. Since only a batch of training samples is used to update the parameters, a certain fluctuation occurs over the training steps. This happens because a certain batch might not perform so good for the complete training set and therefore increases the overall training error. This makes it difficult to find an exact minimum which can be used. But on the other hand it is also possible to escape local minima with this method.

The framework of Caffe uses stochastic gradient descent for training a neural network structure. The batch size can be defined by the user and thus also the amount of oscillation can be chosen individually.

2.5.2 Preparing Data for Caffe

To train a neural network using a given framework one usually has to prepare the retrieved data such that it is applicable for a particular learning algorithm. The same holds for the Caffe framework and thus one has to prepare its data according to the given specifications. This thesis only focuses on supervised learning and thus also needs to prepare the desired labels for training. When starting to prepare the training data one needs to prepare sets of input data and output labels. The best and easiest way for representing input data in Caffe are images. This allows to store information within one data object that can be read by the Caffe framework. For this thesis it also has the advantage to observe the images since convolutional training is commonly applied for pattern recognition on images. The generated data also has to be split up into a training set and a test set. One can individually choose how many test samples one would like to use but in general 20 − 40% are used for testing. Having a collection of images and knowing the corresponding labels, one can use two basic principles of preparing the data for Caffe.

The first method is to create a so-called **LMDB** file. Those files allow high-performance processing of its inputs. To create such a file one needs a list of files and the corresponding labels. A small section of a training file could look like

```
...
path/to/file/image37.png  0
path/to/file/image45.png  0
path/to/file/image1.png   1
path/to/file/image13.png  1
path/to/file/image785.png 0
...
```

where each image has a label of 0 or 1. Caffe then offers an executable called *convert_imageset* which can be executed to generate a LMDB file.

```
./convert_imageset path/to/train.txt path/to/store/lmdb
```

The main advantage of this method is that it is easy to generate a training set which can be used by the framework. However, disadvantages are that it does not allow multiple labels for an image which can be annoying if one would like to train data with multiple labels and it does not allow floating labels such that the label must be a positive integer representation.

To solve these problems one can use data which is represented in **HDF5** format. This format allows to store multiple labels for an input and also allows them to be a float. The disadvantage which comes with this data format is that it does not offer a simple conversion of images into HDF5 format. Instead one has to download a library which can then be used to create data sets. For this thesis a python library named *h5py* is used to generate correct data samples but therefore one also needs basic knowledge in python. To simplify the HDF5 data generation one could also use an image list containing the labels as above.

Caffe loads the data row by row into the network. To use random ordering one can shuffle the created image list or, for LMDB, could add the option *-shuffle* when creating the set.

2.5.3 Defining a Network Structure

Having prepared the data which can be used for training, one can define the desired network structure. Caffe offers a complete network generation solution in python where one can define the desired structure in python and then the desired structure is generated. Typically the structure of a network is defined in a *.prototxt* file. This file contains all necessary information which is needed for understanding the structure. It contains a set of layers which hold specific parameters and define what the input should be and where the output is sent to.

To make structuring easier many standard layer types which can be used are supported. For the input also different layers are defined such that the framework can distinguish between e.g. LMDB and HDF5 input data. Some of the most important layers which are used in this thesis are

1. Convolution Layers
2. Pooling Layers
3. Long-Short Term Memory Layers
4. Inner Product Layers (fully connected layer)
5. Dropout Layers
6. Softmax Layers

Listing 2.13 shows a part of the *prototxt* file. It defines a Pooling layer with the name *pool1*. The parameter *bottom* in line 4 defines where the layer should be added within the network. In this example the layer is added at the bottom of the layer *conv1* which is probably a convolution layer. The parameter *top* in line 5 defines under which name the output should be stored. *pooling_param* in line 6-9 are special parameters which are set for defining the structure of the pooling layer. In there it is defined that max-pooling should be used with a kernel size of 2.

```
 1  layer {
 2    name: "pool1"
 3    type: "Pooling"
 4    bottom: "conv1"
 5    top: "pool1"
 6    pooling_param {
 7      pool: MAX
 8      kernel_size: 2
 9      stride: 2
10    }
11  }
```

Listing 2.13: An example layer which is contained in the .prototxt file

Figure 2.21 illustrates the flow graphically. Top and bottom therefore can be seen as separate positions where the data is stored or retrieved.
Figure 2.22 shows a small network as an example. It uses an data layer to retrieve the input which is stored at *data* and *label*. The input data is then used by a fully connected layer and sent on to the softmax layer. The softmax layer

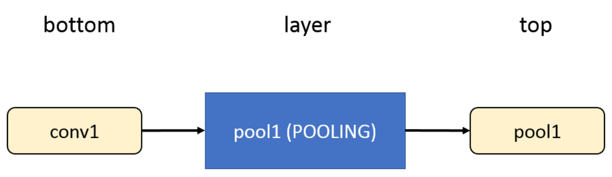


Figure 2.21: The flow of a Caffe layer illustrated

then also receives the label and can compare them for adapting the learning weights.

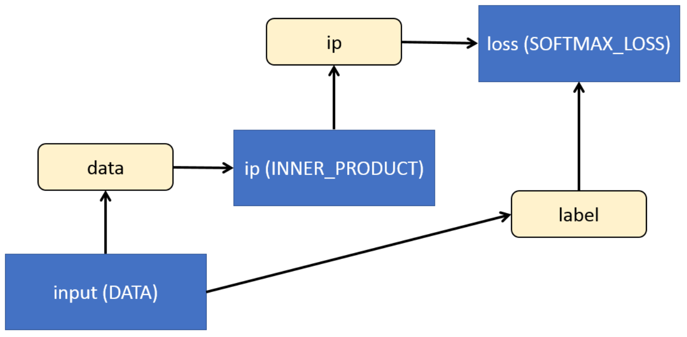


Figure 2.22: Example Caffe network

Basically any desired network structure can be modelled by adding various layers into the structure file and linking them together. Since it does not make sense to arbitrarily combine different layers one can lean on common network structures when designing a new network. One of the most popular network structure which has been useful in various fields is the so-called *LeNet* structure [3]. This structure is a convolutional neural network which was introduced to recognize handwritten digits.

2.5.4 Training a Network with Caffe

When the training data is prepared and a valid network structure is defined, one can start to train the network. To train a network using Caffe one has to define a so-called *solver file* which also ends with *.prototxt*. In the solver file all necessary parameters are set which are then applied for the network. Listing 2.14 shows an example file of such a solver file.

```
 1  net: "path/to/network.prototxt"
 2  test_iter: 80
 3  test_interval: 500
 4
 5  base_lr: 0.01
 6  momentum: 0.9
 7  weight_decay: 0.0005
 8  lr_policy: "inv"
 9  gamma: 0.0001
10  power: 0.75
11
12  display: 100
13  max_iter: 10000
14  snapshot: 5000
15  snapshot_prefix: "path/to/result/name"
16  solver_mode: GPU
```

Listing 2.14: Basic content of a solver file for Caffe

Line 1 defines the path to the network file which has to be loaded. This is necessary because the framework only asks for the solver file and not for the structure file. Line 2-3 define at which interval the network should be tested and how many batches should be used to test. This file says that every 500 iterations a test should be done with 80 batches. Line 5-10 define some principle network parameters like the learning rate for gradient descent and other parameters which are used within the network. Line 12 defines how often the current progress should be printed to the command line and line 13 states how many iterations should be done. Line 14-15 define where the resulting file which contains all trained network parameters should be stored and at which iteration a temporary snapshot should be made. Line 16 describes the processor that should be used for training. This can either be CPU or GPU.

Having adapted the solver file to the specific needs, one can start the training. This can be done by using the provided executable and tell it to train the solver file

```
./caffe train --solver=path/to/solver.prototxt [options]
```

After training the framework creates two files. The first is a *.caffemodel* file which contains all necessary values like weights and biases which were trained for the network. The second file is a *.solverstate* file that stores the last iteration step which was done with SGD. This solverstate file could then be used to continue with the training if e.g. a training was cancelled or more training steps then defined in the solver file are needed.

2.6 OpenCV

Open Source Computer Vision is a library that offers efficient algorithms which are designed for real-time applications [71, 70]. The main fields of application are computer vision and machine learning. It offers libraries which can be used in C++, C and Python. It also contains various machine learning algorithms that were already trained and are ready for use. The best example are implementations for face recognition and object detection. Since the complete

OpenCV library is very powerful, only a few parts are mentioned which are needed within this thesis. First various methods are presented which are used for generating and handling images. Then it is shown how a trained Caffe network can be loaded into C++ with the help of OpenCV. This is shown since the trained neural network has to be implemented into a ROS node. At the end of this section a Boosting method is described which is used to train given features for robot localization scoring.

2.6.1 Image Handling

Since OpenCV is a library for computer vision, it seems quite obvious that it can also do simple image handling. To understand how training samples can be created with the help of OpenCV, a short introduction is given in handling images [13]. Images in C++ are stored using a class called *cv::Mat*. It is used to represent each pixel in a $N \times M$ matrix. The representation of pixel x_{nm} depends on the image type. If a pixel contains multiple channels, like RGB, the pixel can be represented as a vector. In this thesis only grey images are handled and thus a unsigned character with a range of $0 \leq x_{nm} \leq 255$ is used. An image in C++ can be created as

```
cv::Mat img = cv::Mat(r, c, CV_8UC1, cv::Scalar(0));
```

where *img* is created with *r* rows and *c* columns. The type *CV_8UC1* states that each pixel in the image should be represented by eight unsigned bits and one channel. *cv::Scalar(0)* states the initial colour of the image. In this case only a black-white Scalar is used where 0 indicates a white background. Scalars can also represent other color schemes like RGB where three color values are inserted. After creating an instance of an image one can draw different objects into the image e.g. lines, circles or rectangles. This thesis needs only to draw circles so a code for that is shown but all other figures are drawn similarly.

```
cv::Point pixel(x, y);
cv::circle(img, pixel, 1, cv::Scalar(254), CV_FILLED);
```

In this command a new point is created which indicates a pixel at position (x, y). Then the circle function is called which draws a circle in image *img*, at center *pixel*, colour 254 and it states that the circle should be filled. Another method which is used in this thesis is to rotate an image. This can be done by defining the center of rotation, creating an rotation matrix which knows the center and the angle to be turned and rotating the image. The result of the rotation is stored in a new image instance.

```
cv::Point center(x, y);
cv::Mat rot = cv::getRotationMatrix2D(center, angle, 1);
cv::Mat result;
cv::warpAffine(img, result, rot, img.size());
```

The last thing which is used in this thesis for handling images is to cut them.

```
cv::Rect roi(start_x, start_y, width, height);
cv::Mat cut(img, roi);
```

In the example above, the instance *roi* indicates a region of interest which is represented as a rectangle. This rectangle starts at column *start_x* and at row *start_y* in the image. The position indicates the lower left corner from where it should be cut. Then the width and height of the new image are given and a new image *cut* is created.

2.6.2 Working with trained Neural Networks

Having trained a neural network it is used to identify labels on new data. In the case of this thesis the neural network should be used on live data for robot localization scoring. Therefore it needs to be embedded in a ROS node which receives all necessary information, builds an input set with this data and then uses the input set to estimate the localization quality. Neural networks are often hard to learn but once they are learned it is easy to classify new input data [134]. Thus it can be used on live-data and get a real-time localization status. To use a trained network in a ROS node the OpenCV library is needed. OpenCV offers an interface for loading a trained Caffe model and using it for classification. Listing 2.15 shows an example code of how a Caffe model is loaded using OpenCV and how a new image is classified.

```
 1 #include <opencv2/dnn.hpp>
 2 #include <opencv2/imgproc.hpp>
 3 #include <opencv2/highgui.hpp>
 4 using namespace cv;
 5 using namespace cv::dnn;
 6
 7 int main() {
 8   Net net = readNetFromCaffe(network_txt, model_bin);
 9   int x = 36, y = 36;
10   Mat img = generateNewImage();
11   Mat blob = blobFromImage(img, 1, Size(x, y),Scalar(0));
12
13   net.setInput(blob, ”data”);
14   Mat res = net.forward(”loss”);
15
16   Mat prop_mat = res.reshape(1, 1);
17   Point class_pt;
18   double* prob;
19   minMaxLoc(prop_mat, NULL, prob, NULL, &class_pt);
20   int id = class_pt.x;
21 }
```

Listing 2.15: Loading the trained Caffe network into ROS using OpenCV

In line 8 the given Caffe network is loaded using the method *readNetFromCaffe*. This is a method which is in the OpenCV library and takes two arguments as parameter. *network_txt* is the path to the network structure which was used to train the network and *model_bin* is the path to the *.caffemodel* file which was produced after the network was trained. Line 10 generates a new image file that should be checked using live data retrieved from ROS topics. In line 11 a blob is created. A blob is a 4D matrix which is created using the given image. It also optionally scales the image and if needed resizes it to a size of x and y. The given Scalar is an optional input that may contain mean values which are subtracted from the image channels. Line 13 sets the new blob as input for the loaded network and states where should be stored at. Here the input blob is stored at position *data* which is used to load the first layer. In line 14 the network is propagated forward until the layer *loss* is reached. The result is then reshaped to a smaller matrix in line 16 which can then be used to analyse the calculated label. Line 19 analyses the given result and stores the found label with its probability in two local variables. In the end one has the variable *id* which contains the found label and the variable *prob* which is a pointer to the probability for the found label.

2.6.3 AdaBoost

Boosting is a machine learning approach which uses supervised data for classification. The idea is to combine multiple weak classifier which does not hold enough information about a certain class itself and combine it to one strong classifier that can be used for identifying classes [130, 131]. OpenCV offers an adaptive boost algorithm which takes N training samples $(x_i, y_i), 1 \leq i \leq N$ with $x_i \in \mathbb{R}^K$ and $y_i \in -1, +1$. x_i is the input vector which contains K different components that are used for training. y_i is the desired label which is either -1 or $+1$. There exist several variants of boosting algorithm which all have a similar structure [52]. In this thesis the standard discrete AdaBoost algorithm which uses two classes is presented. It uses its N-sized input set and initializes the weights for each input sample with $w_i = 1/N$. Then a weak classifier $f_m(x)$, the weighted training error ϵ_m and the scaling factor c_m is computed. Then the weights are increased for input samples that have been wrongly classified. After this step the weights are normalized and the steps for finding a new weak classifier are repeated form M times. At the end a final classifier $F(x)$ is found which uses the sign of the weighted sum of the input set. Thus, the final discrete AdaBoost algorithm contains the following steps [108, 90]:

1. Collect N supervised samples $(x_i, y_i), 1 \leq i \leq N$ with $x_i \in \mathbb{R}^K$ and $y_i \in -1, +1$.
2. Assign initial weights to all samples as $w_i = 1/N, 1 \leq i \leq N$.
3. For m in $1 \leq m \leq M$ do
   (a) Find the weak classifier $f_m(x) \in -1, +1$ which is based on the weights w_i.
   (b) calculate the error
   $$\epsilon_m = \exp\left(-\sum_{i=1}^{N} w_i y_i f_m(x_i)\right) \tag{2.62}$$
   (c) calculate the scaling factor
   $$c_m = \frac{1}{2}\log\left(\frac{1-\epsilon_m}{\epsilon_m}\right) \tag{2.63}$$
   (d) update the weights as
   $$w_i = \frac{1}{Z} \cdot w_i \cdot \exp(-y_i c_m f_m(x_i)) \tag{2.64}$$
   where Z is factor for normalization such that $\sum_i w_i = 1$.

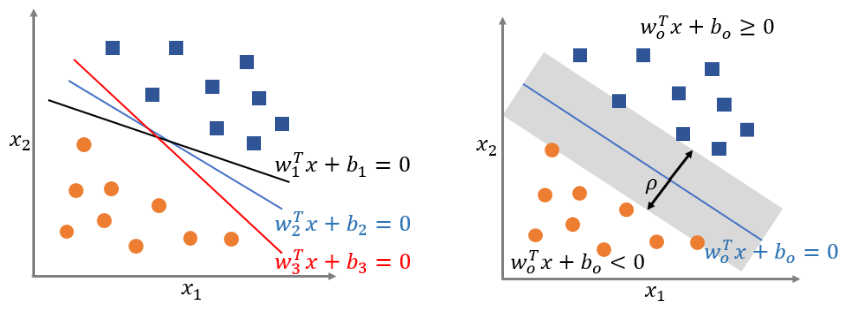
4. generate new function $F(x)$ for classification with

$$F(x) = sgn\left(\sum_{m=1}^{M} c_m f_m(x)\right) \tag{2.65}$$

This algorithm produces a function $F(x)$ which is either -1 or $+1$. $F(x)$ is based on the sum of the calculated weights. It is currently implemented in OpenCV and is used in this thesis to boost the performance of weak classifiers.

2.6.4 Support Vector Machines

A Support Vector Machine (SVM) is a popular tool for classification [141, 69]. It searches for an optimal hyperplane which can be used to separate two classes. It takes training samples as input which are classified and outputs an optimal hyperplane which can be used to categorize new samples. To find the optimal plane which separates the two classes best is not a trivial task. Consider m training samples $(\mathbf{x}^{(i)}, y^{(i)}), 1 \leq i \leq m$ where $\mathbf{x}^{(i)}$ is a sample which is labelled with $y^{(i)} \in \{-1, 1\}$. If those samples are linearly separable, multiple possible solutions exist which could be applied. Figure 2.23a illustrates the problem on a two dimensional problem which is linearly separable. The task is now to find



(a) An example of multiple possible hyperplane solutions

(b) An optimal hyperplane found with SVM

Figure 2.23: Determining the optimal hyperplane out of multiple solutions using SVM

the best hyperplane, determined with $\mathbf{w}_o$ and b_o, that maximizes the separation space between the two classes

$$\mathbf{w}_o^T \mathbf{x} + b_o = 0 \tag{2.66}$$

To apply SVM one has to define support vectors. Those are the samples $\mathbf{x}^{(s)}$ which are closest to the separation hyperplane. They are used to define an optimal separation hyperplane. The evaluation function is then

$$h(\mathbf{x}) = \mathbf{w}_o^T \mathbf{x}^{(i)} + b_o \tag{2.67}$$

which is used to classify new samples $\mathbf{x}$. This is done based on the sign of $h(\mathbf{x})$. The distance of a sample $\mathbf{x}^{(i)}$ is given as

$$r = \frac{h(\mathbf{x}^{(i)})}{||\mathbf{w}_o||} \tag{2.68}$$

where $||\mathbf{w}_o||$ is the euclidean norm

$$||\mathbf{w}|| = \sqrt{\sum_i w_i^2}. \tag{2.69}$$

Since $||\mathbf{w}_o||$ and b_o can be scaled without changing the separation hyperplane, one can choose the support vectors $x^{(s)}$ such that

$$h(\mathbf{x}^{(s)}) = \mathbf{w}_o^T \mathbf{x}^{(s)} + b_o = \pm 1, \text{ for } y^{(s)} = \pm 1. \tag{2.70}$$

The distance of a support vector is then defined as

$$r = \frac{h(\mathbf{x}^{(s)})}{||\mathbf{w}_o||} = \frac{\pm 1}{||\mathbf{w}_o||} \tag{2.71}$$

and the maximized margin is then given as

$$\rho = 2|r| = \frac{2}{||\mathbf{w}_o||}. \tag{2.72}$$

To find the optimal hyperplane one has to maximize the margin. This can be done by minimizing $1/2||\mathbf{w}||^2$. The optimization problem is then given as

$$\mathbf{w_o} = \arg\min_{w} \frac{1}{2}||\mathbf{w}||^2 \tag{2.73}$$

and can be solved with the help of Lagrange multipliers

$$J(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i [y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1] \tag{2.74}$$

where $\alpha_i \geq 0$ are Lagrange multipliers. The optimal solution can then be found in the saddle of

$$\min_{\mathbf{w}} \max_{\alpha} J(\mathbf{w}, b, \alpha). \tag{2.75}$$

The Lagrange multipliers α_i can then be found by using the dual form and solving it with the help of quadratic optimization

$$Q(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{T(i)} \mathbf{x}^{(i)} \tag{2.76}$$

where

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \text{ and } \alpha_i \geq 0. \tag{2.77}$$

The Lagrange multipliers can then be used to find the solution as a linear combination of training vectors as

$$\begin{aligned} \mathbf{w}_o &= \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)}, \\ b_o &= y^{(i)} - \mathbf{w}_o^T \mathbf{x}^{(i)}. \end{aligned} \tag{2.78}$$

If data samples are not linearly separable, so-called **kernel methods** can be applied. Those methods take the training samples which are not linearly separable and map them into a higher dimensional space with the use of some non-linear transformation $\varphi(.)$ [28]. Such a kernel function returns the inner product of data points within some space

$$K(\mathbf{x}_1, \mathbf{x}_2) = \varphi(\mathbf{x}_1)^T \varphi(\mathbf{x}_2). \tag{2.79}$$

Kernels allow to operate in high-dimensional feature spaces without having to compute the coordinates within this space. This is also called the **kernel trick** and has better performance than explicit computation methods.

Chapter 3

Related Research

This section discusses literature and research topics that are related to this thesis. There exist various different research areas that are currently quite popular and many publications concerning those areas are presented in this thesis. This section tries to cover all related research areas and presents the latest progress which was achieved within these areas. Therefore new and valuable literature that is connected with this thesis was selected. Another interesting search phrase is the exact topic itself, hence also research on estimating the robots location accuracy with neural networks and particle filters was done. Surprisingly, no publications were found that exactly match this topic. Most of the publications on robot localization focus on the process of localizing a robot and not on measuring its accuracy. Also research showed no valuable results on using neural networks for analysing particle filters. Instead it was often used to improve the accuracy of neural networks by combining both methods.

Before searching related research one has to determine research areas that are important within the thesis. Those areas are then examined to find valuable publications. The found publications are then analysed on their relevance and collected within this section. By evaluating and summarizing those research areas one can then design an overall concept that can be used to solve the problems which were defined in section 1.3. The most related research areas that are addressed in this thesis are

- Latest research on robot localization and its accuracy
- Convolutional neural networks and its progress
- Application of long-short term memory networks
- Pattern recognition with neural networks
- Development of particle filters
- The use of particle filters in combination with neural networks

Those research areas are investigated carefully to gain enough information that is necessary to address the problem formulations of this thesis and to set a knowledge base that is up to date such that the results are valuable and not outdated.

At first the current topics in robot localization are analysed. Therefore it is searched for promising methods that might be useful in the near future. Then, the latest progress on CNNs and LSTMs is discussed and current applications are analysed. After that, it is searched on how pattern recognition can be done by using neural networks and how particle filters might be applied to different topics. Also the combination of particle filters and neural networks is presented.

3.1 Robot Localization and its Accuracy

The first topic which is discussed in here is the current research in robot localization and in analysing the accuracy of some localization methods. Therefore, two methods for feature detection are shown that use only a grid map and a laser scanner. Both feature detectors aim to detect significant corners for robot localization. Then a publication on the localization accuracy of a robot that uses particle filters in combination with scan matching is discussed.

3.1.1 Using the Laser Scan for Feature Detection

A novel approach is proposed by Kallasi et al. who introduced a new method for detecting features to localize and navigate robots [76]. They propose two new feature detectors called *Fast Adaptive Laser Keypoint Orientation-Invariant* (FALKO) and *Orthogonal Corner* (OC). Those two detectors are an improvement of the *Fast Laser Interest Region Transform* (FLIRT) [148] approach that can be used to detect high curvature points in laser scan images. While the FLIRT method searches for general features which depend on the viewpoint of the robot, FALKO and OC are designed to detect stable features like corner walls. The difference between the two proposed methods is that FALKO detects features by selecting meaningful neighbours and scoring the cornerness of a feature and OC uses orthogonal alignments to detect important features.

A good feature detector is important for robot localization since those features can be used to map a laser scan into a given environment. If a detector can find a lot of meaningful features in the laser scan that look nearly the same in the grid map, one can match the scan into the environment with high accuracy. This thesis now presents the two detectors that are proposed by Kallasi et al. [76].

Orthogonal Corner Detector

The orthogonal corner detector uses the fact that indoor environments often have many straight walls that are aligned in an orthogonal direction. It uses the Hough Transform $HT(\theta, r)$ and its Hessian representation of scan points S to map each point into the hough parameter space as described in Section 2.3.1. For transforming scan points into the Hough Space only a subset of the parameter space of size $n_\theta \times n_r$ is used where the cells are centered in $[\theta_t, r_s]$, with $0 \leq t < n_r$ and $0 \leq s < n_\theta$. Here θ_t is defined as $\theta_t = t\triangle\theta$ with $\triangle\theta = \pi/n_\theta$ and $r_s = \triangle r(s - n_r/2)$. This is then used to calculate the Hough Spectrum $HS(\theta_t)$ [20]

which is defined as the squared sum of all distances in the Hough space

$$HS(\theta_t) = \sum_{r_s \in r} HT(\theta_t, r_s)^2. \tag{3.1}$$

The OC detector then determines the dominant direction $\bar{\theta}$ which is the absolute maximum of the Orthogonal Hough Spectrum

$$\bar{\theta} = \max(OHS(\theta_t)) = \max(HS(\theta_t) + HS(\theta_{t+n_\theta/2})). \tag{3.2}$$

Here $\theta_{t+n_\theta/2}$ is used since the cells are centered. The dominant direction is then used to rotate every scan point p_i by $-\bar{\theta}$, resulting in a new set $\bar{S}$. Every point p_i defines its own neighbourhood radius r_i which depends on the distance to the viewpoint $||p_i||$. The radius is then calculated as

$$r_i = a \exp(b||p_i||) \tag{3.3}$$

where a, b are parameters which can be chosen individually. This can then be used to define three neighbour sets as

$$C(p_i) = \{p_j \in \bar{S} : ||p_j - p_i|| < r_i\} \tag{3.4}$$

$$C_x(p_i) = \{p_j \in C(p_i) : |p_{j,x} - p_{i,x}| < w \wedge |p_{j,y} - p_{i,y}| > w\} \tag{3.5}$$

$$C_y(p_i) = \{p_j \in C(p_i) : |p_{j,y} - p_{i,y}| < w \wedge |p_{j,x} - p_{i,x}| > w\} \tag{3.6}$$

where w is a threshold on point alignment. One can then score the neighbourhood of a point by checking the alignment into both directions. The larger a C_x and C_y set is, the higher is its score. The values for determining the score are defined as $n_x = |C_x(p_i)|$ and $n_y = |C_y(p_i)|$. Those values are then used to score the neighbourhood of a point and check if it is a valuable corner as

$$score(p_i) = \frac{n_x + n_y}{\epsilon + |n_x - n_y|} \tag{3.7}$$

where ϵ is the allowed error. This score can then be used to score corners and thus find out how well the given scan matches int its environment.

FALKO Detector

The Fast Adaptive Laser Keypoint Orientation-invariant detector uses 2D range data to detect edge intersections. It computes its neighbours like in the OC detector, Equation 3.4. This set is then split up into two subsets

$$C_L(p_i) = \{p_j \in C(p_i) : j < i\}, \tag{3.8}$$

$$C_R(p_i) = \{p_j \in C(p_i) : j > i\}. \tag{3.9}$$

Then the cardinality of a point is checked and if $|C_L| < 2$ or $|C_R| < 2$ the point is thrown away from the possible corner set. Then the two endpoints x_L and x_R for a corner candidate are calculated. They are defined as

$$x_L = p_{j_{\min}}, j_{\min} = \arg\min_j \{p_j \in C_L(p_i)\} \tag{3.10}$$

$$x_R = p_{j_{\max}}, j_{\max} = \arg\max_j \{p_j \in C_R(p_i)\}. \tag{3.11}$$

These variables are then used to form a geometrical triangle $\triangle p_i, x_L, x_R$ which is then evaluated. The connection between x_L, x_R is the base of the triangle $\overline{x_L, x_R}$. If the length of this base line or the height of the triangle is lower than some defined threshold r_i/β, the point is thrown away. Here, β is a parameter which has to be chosen. Then a polar grid is used on a point p_i that separates the neighbourhood into circular sectors. Then the neighbour points are used to compute the orientation based on the candidate point, such that

$$\phi_{j,L} = \left\lfloor \frac{s_n}{2\pi} \tan^{-1}\left(\frac{p_{j,y} - p_{i,y}}{p_{j,x} - p_{i,x}}\right)\right\rfloor, \forall p_j \in C_L(p_i) \tag{3.12}$$

$$\phi_{j,R} = \left\lfloor \frac{s_n}{2\pi} \tan^{-1}\left(\frac{p_{j,y} - p_{i,y}}{p_{j,x} - p_{i,x}}\right)\right\rfloor, \forall p_j \in C_R(p_i) \tag{3.13}$$

where s_n is the number of sectors that are used in the polar grid. One can now define a distance function for quantized orientations in the sector units as

$$d_\theta(\phi_1, \phi_2) = \left(\left((\phi_1 + \phi_2) + \frac{s_n}{2}\right) \mod s_n\right) - \frac{s_n}{2}. \tag{3.14}$$

The distance function can then be used to score the corners of a point candidate and its neighbours. Therefore the left and the right side are calculated as

$$score_L(p_i) = \sum_{h=i-1}^{j_{\min}} \sum_{k=h-1}^{j_{\min}} |d_\theta(\phi_{h,L}, \phi_{k,L})| \tag{3.15}$$

$$score_R(p_i) = \sum_{h=i+1}^{j_{\max}} \sum_{k=h+1}^{j_{\max}} |d_\theta(\phi_{h,R}, \phi_{k,R})| \tag{3.16}$$

which can then be summarized into one score function which measures the alignment of two point sets.

$$score(p_i) = score_L(p_i) + score_R(p_i). \tag{3.17}$$

FALKO and OC are both approaches that are a good to detect features and to score them. By detecting features in the environment and in the scan, one can try to match them and thus localize the robot. Those features can also be used as additional boosting parameter for AdaBoost of Support Vector Machines when it comes to improving the localization scoring accuracy.

3.1.2 Combining Particle Filters with Laser Scan Matcher

A publication which was presented in 2012 by Röwekämper et al. evaluates the position accuracy of a mobile robot localization method that is based on particle filtering and laser scan matching [127]. For evaluation they used a motion capture system that tracks the position of a robot within its environment with high accuracy [103]. The localization system which was used in their evaluation was a combination of basic state-of-the-art approaches. Therefore Monte Carlo Localization (see Chapter 2.3.4), a scan matching procedure [155] and the Kullback-Leibler distance sampling (KLD-sampling) [50] were used. KLD-sampling is used to improve the efficiency of the particle filter by dynamically adapting the number of samples. Thus, if the robot is focused only a small number of particles is used whereas a large number of particles is needed if the uncertainty of a location is high. Scan matching algorithms try to improve the localization of a robot by matching a given laser scan to a grid map. This is achieved by slightly rotating and translating the scan image and checking whether the adoption improves the matched scan points.

Röwekämper et al. defined three requirements for the localization approach [127]. It should be efficient, robust and accurate. To fulfill these requirements some adoptions on the MCL algorithm were made. The first step was to estimate the robots pose x_t at time step t. This is done with the Monte Carlo localization as

$$p(x_t|z_{1:t}, u_{0:t-1}) \sim p(z_t|x_t) \int_{x'} p(x_t|x', u_{t-1}) p(x'|z_{1:t-1}, u_{0:t-2}) dx' \qquad (3.18)$$

where $u_{0:t-1}$ is the list of actions which were executed by the robot so far, $z_{0:t}$ is the observation sequence and $p(x_t|x_{t-1}, u_{t-1})$ is the motion model that states the likelihood of the robot ending in state x_t after executing action u_{t-1} in state x_{t-1}. $p(z_t|x_t)$ is the probability of observing z_t when staying at position x_t.
To improve the performance of the MCL algorithm, KLD sampling is applied. This leads to the fact that particles are only generated when needed and thus improve the performance. While the robot is not completely sure about its localization status new particles are generated. When it has enough knowledge about its position the particle set is kept small to increase the performance. To further improve the MCL algorithm and make it more precise laser scan matching is applied. Therefore local sensor measurements are stored in a grid map and compared during runtime. The variant which is used by Röwekämper et al. is the iterative closest point (ICP) principle. This method is used as post processing step. Thus the output of the MCL algorithm is used as input for the scan matcher. To make the localization system even more robust, beams that are not well explained by the environment are masked by integrating over the particles. This improves the localization behavior. Using the localization system as described above, Röwekämper et al. examined a robots localization accuracy in a static and dynamic environment [127]. They managed to keep the localization error low at only a few millimeters. They also state that the localization error in a dynamic environment never exceeded a translational error of 17mm and a rotational error of 0.15 degree. Although these results sound quite promising one cannot derive further informations on larger environments that are highly dynamic.

Röwekämper et al. also summarize other localization approaches which are currently researched [127]. Many of them are based on the Monte Carlo Localization method or on the Extended Kalman Filter (EKF). The most common sensors for robot localization are odometry sensors and range sensors but also cameras, RFID chips, GPS receivers and wireless receivers are gaining more and more popularity. Some localization approaches which use perspective cameras store visual features in a database and try to match them. Those features are for example so-called SIFT features [137]. Another presented method is the combination of Monte Carlo Localization with stereo cameras and SIFT features [46].

An current approach for the Extended Kalman Filter that was presented by Cho and Kim is to use chirp-spread-spectrum ranging [22]. Another method that becomes increasingly popular localizing robots by using wireless signals [44, 48]. The main advantage of this method is that it is quite cheap but it does not offer a high accuracy yet. In general it is hard to say which localization approaches are the best since all of them have advantages and disadvantages. Mostly it depends on the desired accuracy, on the performance, on the complexity or on the price.

3.2 Progress and Application of CNNs

This section focuses on the progress of convolutional neural networks and their practical applications. CNNs are a promising structure for neural networks and increasingly gain popularity. They excel in classifying images and thus also have their main application in image classification. Due to increasing popularity the CNN is a hot research topic and it is applied in various research areas. This section especially focuses on the current fields of application which are promising research topics. First feature detection on images is discussed and then a publication is discussed that detects dynamic obstacles in grid maps with convolutional neural networks.

3.2.1 Feature Detection on Images

The main application of CNNs is the detection of features within images. Therefore an given image is used as input of a neural network and sent into convolutional layers which store single features in feature maps. Those features can then be used to recognize patterns for classification.

Modern CNN detecting Tower Lighthouses

A simple and modern example is presented by Shamov and Shelest [139]. They present the main features of convolutional neural networks and show how they are nowadays applied for feature detection. Therefore they created the task of detecting tower lighthouses from a video stream. The main part of their work focuses on presenting popular activation functions which can be used for neurons. A common activation function which was already presented in Section 2.4.1 is the sigmoid function. It is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.19}$$

and offers easy computation as well as an derivative which is continuous. Another popular activation function is the hyperbolic tangent function. Like the sigmoid function it is easy to compute and offers a derivative which continuous. Another property is that it is antisymetric. It is defined as

$$\sigma(x) = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{3.20}$$

The last activation function which is presented here is the Rectifier Linear Unit (ReLU). This was also used by Shamov and Shelest to train their CNN. This function filters negatives and otherwise returns the given x value. It is defined as

$$\sigma(x) = \max(0, x). \tag{3.21}$$

By using ReLUs as activation function and a GPU for increasing the training performance, Shamov and Shelest managed to receive an accuracy of 69−73% in testing with a simple network structure [139]. They also state that this accuracy can be further improved by conducting further trainings.

This study is interesting for this thesis because it shows that modern CNNs have a lot of potential in recognizing features. It also shows that complex systems are not always necessary to train an acceptable solution.

Road Detection with CNNs

A challenge which becomes increasingly popular due to self driving cars is the detection of free road surfaces. This is needed to avoid obstacles on the road, to support path planning and enhance decision making. Especially when it comes to unclear situations like invisible lane markings a good road detection is necessary. Often simple camera images are used in combination with deep neural networks to detect free road surfaces [100, 84]. In [16], Caltagirone et al. present a road detection approach using light detection and ranging (LIDAR) devices and fully convolutional neural networks (FCN). By taking the point cloud from the LIDAR device and generating an image from the top view they managed to train a convolutional neural network. The top view was chosen because they think that it better represents the current situation than a normal camera perspective. Caltagirone et al. showed that their approach of generating top view images leads to an efficient network that has high accuracy. They managed to detect 95.32% of the road surface correctly, using the KITTI road benchmark data set [53]. Another advantage of their method is that it is fast and usable for real time applications on GPU accelerated hardware.

Both advantages of this study can be applied in this thesis. First of all, the point cloud of the particle filter could be represented from the top view on an image since this seems to achieve acceptable results. Also it is shown that fully convolutional neural networks can be applied in real time, which is also needed to estimate a robots localization status.

3.2.2 Dynamic Object Detection in Grid Maps

A new application of convolutional neural networks is the detection of dynamic obstacles in grid maps. Piewak et al. use a grid map and a deep neural network to detect whether grid cells within a map are moving or not [115]. Their difference to a normal tracking approach, like particle filters, is to use the complete map grid with a top view as input image. They also proposed an approach which is optimized for real time applications. Another advantage that occurs with the use of grid maps is that this approach does not depend on specific sensor types. It is applicable on any grid map without the need to restrict the sensor types that recorded the map. The aim of their publication is to classify every pixel within an image that represents a dynamic occupancy grid map (DOG). For better results the grid map is preprocessed. Therefore the information of the DOG is extracted for each cell. A cell contains information about its occupancy status which is between 0.0 (free) and 1.0 (blocked). It also contains a velocity information for each cell which is a vector in (x, y) direction on the map. This is received from a velocity distribution that is calculated with the help of a particle filter. The particles of the filter can be used to calculate the mean velocity $\mathbf{v}_{overall} = (v_x, v_y)^T$ as well as the variance of the x and y velocity $Var(v_x), Var(v_y)$. This can be used to calculate the Mahalanobis distance [159] m as

$$m^2 = \mathbf{v}^T_{overall}\Sigma^{-1}\mathbf{v}^T_{overall} \tag{3.22}$$

where Σ is the covariance matrix of the particle velocities. These values and a boolean variable *Occ* which indicates whether the cell is free or not are used to create five different combinations of RGB images. Other variables which are used to create RGB images are the norm of the velocity which is calculated as

$$v_{x,norm} = \frac{v_x}{\sqrt{Var(v_x)}}, v_{y,norm} = \frac{v_y}{\sqrt{Var(v_y)}} \tag{3.23}$$

and the overall variance of the velocity

$$Var_{overall} = Var(v_x) + 2Cov(v_x, v_y) + Var(v_y). \tag{3.24}$$

Based on the preprocessed images, a convolutional neural network is trained which maps each input pixel to one output pixel. For applying backpropagation an additional weight matrix C is used in the cost function $J(\theta)$. The cost function itself is based on the multinomial logistic loss and defined as

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} C(y^{(i)})1\{y^{(i)} = k\} \log Q(i, k, \theta)\right] \tag{3.25}$$

with

$$Q(i, k, \theta) = P(y^{(i)} = k|x^{(i)}; \theta) \tag{3.26}$$

and C contains a weight for each of the K classes

$$C = \left[c^{(1)}, ..., c^{(K)}\right]. \tag{3.27}$$

By applying this cost function Piewak et al. managed to classify each grid cell from the input [115]. The outcome was a grid map which contained only dynamic obstacles. They achieved a test accuracy was 97.2% and thus higher than the used baseline method for comparison. The baseline method which was used separates static and dynamic obstacles by defining a threshold on the Mahalanobis distance.

This publication is discussed in this thesis because a particle filter for robot localization performs best in static environments. The proposed method for detecting dynamic obstacles can be used to remove moving obstacles when predicting the robots location. This might yield in a better performance of the particle filter and thus increase the localization score. A problem with this method tough is that it is applied on a complete grid map. This might lead to a problem when the input grid is large because the training time will increase tremendously. It also seems like the detection of dynamic obstacles is only applicable within the trained map. Other map structures might not achieve the same results and thus every map needs to be trained on its own.

3.3 LSTMs and their Application

The next topic that is needed for this thesis are long-short term memory networks as discussed in Section 2.4.4. To create a recurrent structure that can keep up with the literature, some recent research is discussed. The most popular application of LSTM networks is sequence classification like e.g. speech recognition [120, 63]. Therefore one needs to define a vocabulary that is used in training. Also language models are trained which learn the probability distribution of word sequences [142].

Recurrent neural networks can also be used for prediction and simple classification. Those two abilities are interesting for this thesis. Prediction is a topic which can be used to predict delocalizations of a robot. Thus one can prevent or act before the actual delocalization occurs. Simple classification can also be used for state estimation of a robots localization status since one only needs to know whether the robot is delocalized or not.

This section shows how research applies LSTM networks nowadays to predict certain tasks. Therefore different fields of application are addressed like the stock market or solar power stations. Also some recent publication on classification with a LSTM network is discussed.

3.3.1 Forecasting with LSTMs

Predicting future events is a topic that fascinates human minds. If one could collect enough information to reliably forecast a future state one could solve many problems that are currently unsolved. LSTM networks make claim to predict the future. Although they might not be able to predict complex future events, they are able to use given training data and train a pattern that makes it possible to forecast events. The prerequisite for successfully conducting this task is that the data contains enough information about the desired prediction.

Solar Power Prediction

Gensler et al. wanted to see how good LSTM networks can really predict a future event [56]. Therefore they compared different types of deep learning networks on a defined task. The task was to forecast the power production of solar panels. With this task they want to provide a reliable power forecasting method that allows to efficiently operate within a solar power station. For comparison they used a multilayer perceptron (MLP) architecture which is a simple neural network that consists only of fully connected layers. As discussed in Section 2.4.1, the output for every single neuron which is used within such a network is computed as

$$out = f(\sum_{i}^{inputs} (x_i \cdot w_i + b_i)) \tag{3.28}$$

where x_i is a input of a neuron which is weighted by w_i. b_i is a given bias which is used within activation function f. Another network architecture that was used is the deep belief network (DBN). This is a method that can be separated in two steps for forecasting. The first step is to reduce the dimensionality of the input. This is done by conducting feature learning like in CNNs. Then additional layers are added to conduct the prediction of the output. Every layer consists of a two layer artificial neural network (ANN). Those two layers are used like a funnel. Thus the first ANN helps to learn better features within the data. This kind of layer architecture is also referred to as Restricted Boltzmann Machine (RBM) [128]. Often a DBN is trained like a unsupervised network at the beginning to abstract the data such that the feature set is reduced. Then an artificial neural network is appended which is trained using the labelled training set. Those two deep learning architectures are then used to evaluate a LSTM network and a LSTM network which was combined with an Auto Encoder. An Auto Encoder is a multilayer perceptron network which follows a specific network topology. To make a reliable statement Gensler et al. used five different error measurements for comparing the deep learning architectures. All of them take N samples as input and compare the measured power production x of a solar power station with its predicted power production x'. The first error measurement which was used is the root-mean-square-error (RMSE). Another error function is the average absolute deviation (AbsDev) and the mean absolute error (MAE). Also the Bias and the correlation between measured and predicted outcome was measured. The definition of the different error functions which were used are shown in Equation 3.29. In the correlation function $\bar{x}$ represents the median of the given data set.

$$\begin{aligned}
RMSE(x,x') &= \sqrt{\frac{1}{N} \cdot \sum_{n=1}^{N} (x'_n - x_n)^2} \\
AbsDev(x,x') &= \frac{1}{\sum_{n=1}^{N} x_n} \cdot \sum_{n=1}^{N} |x'_n - x_n| \\
MAE(x,x') &= \frac{1}{N} \cdot \sum_{n=1}^{N} |x'_n - x_n| \\
BIAS(x,x') &= \frac{1}{N} \cdot \sum_{n=1}^{N} (x'_n - x_n) \\
Correlation(x,x') &= \frac{\sum_{n=1}^{N} (x'_n - \bar{x}') \cdot \sum_{n=1}^{N} (x_n - \bar{x})}{\sqrt{\sum_{n=1}^{N} (x'_n - \bar{x}')^2 \cdot \sum_{n=1}^{N} (x_n - \bar{x})^2}}
\end{aligned} \tag{3.29}$$

The results of the experiment for solar power forecasting showed that in nearly all error measurements the Auto Encoded LSTM performed the best. MLP networks had a lower error in the Bias but in the overall comparison the Auto-LSTM network performed the best. This also shows that recurrent networks and especially long-short term memory networks excel at their task of predicting future events.

Prediction of the Stock Market

Another publication that uses LSTM networks for prediction is the work of Nelson et al. [106]. They used the recurrent network architecture for predicting the price movement on the stock market. Therefore they gathered data from different stocks over a period of seven years. Since the stocks have different prices and because the variance between time series needs to be stabilized they normalized the data to

$$x_i^{(1)} = \log(p_i) - \log(p_{i-1}) \tag{3.30}$$

where p_i is the price at time step i and $x_i^{(1)}$ is the first input for the LSTM network. To not only predict the data on the stock price 175 other indicators that are often used for predicting the stock market are generated and used as input to the network. The label y_i for each input sample is a binary output that indicates an increase or decrease of the closing price at the next time period. Thus the label is calculated as

$$y_i = \begin{cases} 1 \text{ , if } p_{i+1} > p_i \\ 0 \text{ , else.} \end{cases} \tag{3.31}$$

The results of this experiment were that the LSTM outperforms other prediction methods as well as a pseudo-random prediction. In general the accuracy of roughly 54.6% is chastening. This accuracy seems to be roughly the same like with random outputs. To show that it is better than random they also evaluated the output with a pseudo random number. This method yields an accuracy of $48 - 50\%$ on the same sets. Although the result is not far away from random it showed a small improvement and that LSTMs might score better accuracies when investigated a bit more. Also one has to admit that it is difficult to predict the progress of a stock without investigating the complete market. This publication is interesting for this thesis not only because of its prediction task but also on the fact that the network was trained to predict a binary output. This can also be used in this thesis to classify the position of a robot as localized or delocalized.

3.3.2 Classification with LSTMs

Another task which can be done with LSTM networks is classification of given samples. This is usually a task of MLP or CNN networks but research has also shown that recurrent network architectures can be used for classification. Especially when temporal information like the development of a sample set or the position of an object should be included.

Identifying Targets for Military Applications

A good example is a recent research that was done by Jithesh et al. [74]. They used a LSTM network in the military sector to detect targets on a high resolution range profile (HRRP) based radar. Their goal was to identify given types of targets which are recorded on a radar. Therefore three different target types were defined and used to train a LSTM network. For example, one type was a model of a military aircraft and another model a missile. The data for the HRRP radar was prepared through an electromagnetic simulation. For each target model different simulations were done and recorded. Those simulations were then labelled and trained using a LSTM network. The results of this study were astonishing. It was shown that the target models could be classified without any ambiguity. They tested 100 different profiles and all of them were identified correctly.

3.4 Pattern Recognition with Neural Networks

Pattern recognition is a popular task in visual computing. It is about detecting specific patterns within an given input. In visual computing these inputs often are images which are used to detect certain structures like humans. A lot of different approaches are presented which state to be the best pattern recognition method. Since it is impossible to compare every approach with each other it is difficult to determine the best solution for a given problem. The pattern recognition methods that are presented in this section focus on recent approaches which were done to recognize patterns in different input formats. They also rely on a neural network structure since this is a main aspect of this thesis. First a recent pattern recognition approach is presented that aims to detect bird pest. Then a SVM approach for character recognition is presented.

Detecting the Bird Pest with Pattern Recognition

For winegrowers birds are a huge problem. Since the very first beginning of wine growing it is a challenge to keep vermin away from the vineyards. To do so many different protection mechanisms were developed which aim to scare away the birds. The problem of such systems nowadays is that they are often loud, expensive and only activate in certain time periods. To improve the efficiency of such a system Dolezel et al. developed a system that detects birds [38]. Their system consists of a central control unit that controls the overall system, detection units that search for birds and scare units that are meant to scare birds off. The goal is to detect birds in specific areas and only do an frightening maneuver within this area. The part which is interesting for this thesis is the detection unit. Those units are spread all over a vineyard and record sounds. This sound is then divided into sequences of constant length so that they can be used to extract features. It is said that the feature extraction is the most important step for a good pattern recognition model. Therefore they did a linear prediction coding (LPC) [95]. LPC is a successful method for sound recognition which can approximate a given sound sample according to

$$\bar{s}(n) = \sum_{k=1}^{p} \alpha_k s(n-k) \tag{3.32}$$

where $s(n)$ is the given sound sample, p is the number of LPC coefficients, α_k is the k-th LPC coefficient and $\bar{s}(n)$ is the approximation of the given sound sample. To effectively predict bird patterns the error between the approximated and current sound has to be minimized. It is defined as

$$E = \sum_{n=1}^{N} e(n)^2 \tag{3.33}$$

where N is the number of sound samples and $e(n) = s(n) - \bar{s}(n)$. To minimize the error E one has to set the derivative to zero w.r.t. each parameter in α. Having this done one obtains the values for α_k. The advantage of the LPC approach for feature extraction is that it can be efficiently solved and that the error of the approximation can be increased with the number of LPC coefficients but stays limited. This approach is then used to generate a set of training data that was acquired by recording different bird species. The training data was then used in a deep neural network and trained on different combinations of sound sequence length and number of LPC coefficients. The network was trained with the error function

$$E_{val} = -\frac{1}{N}\sum_{i=1}^{N}[o(i)\ln(y(i)) + (1 - o(i))\ln(1 - y(i))] \tag{3.34}$$

which is believed to be far more suitable than a usual mean squared error function. Here the error function runs over N samples, $o(i)$ indicates the desired output and $y(i)$ is defined as the actual output. The result of this study was that they managed to train a neural network for pattern recognition with an accuracy of 89.6%. This is a promising result and shows that patterns can be detected with high accuracy by using neural networks. Although this study was conducted on sound signals, it is an important literature topic for this thesis because it shows off that a deep neural network can be used to detect specific patterns from a given input. For this study only a deep learning network with simple hidden layers was used. This also shows that good results can be achieved by using simple neural networks that do not contain convolutional layers.

Recognizing Characters on Licence plates with SVM

Recognizing licence plates is a job which is often still done by humans. With an increasing number of cars also more images have to be examined to determine a drivers car. Carata and Neagoe presented a method which may automate the process of detecting licence plates in images [18]. They used a pulse-coupled neural network (PCNN) for segmenting images and then applied a Support Vector Machine on the image segment to detect the characters on a licence plate. A PCNN is a neural network that is inspired by biology and based on a similar structure that was found in the visual cortex of mammals. This method is used to obtain a binary image on the given segmentation for better character recognition.

The processed image is then used by a Support Vector Machine classifier to recognize a character in every segmentation that was produced by the PCNN. SVM uses a decision hyper plane which is tried to optimize such that it separates the character classes. The results of this publication show that the combination of PCNN and SVMs for pattern recognition to a good job in detecting characters on licence plates. Depending on the SVM kernel and the input size, the accuracy varies between $92 - 96\%$. Compared to reference methods this is an improvement of roughly 15%. This research is interesting for this thesis because it is shown how patterns can easily be recognized using only black-white images for pattern recognition. Although the input image was a normal image, PCNN converted it into a binary image consisting only of black and white pixels. This knowledge can be used when it comes to designing input samples for predicting the localization state of a robot.

3.5 Improvements of the Particle Filter

This section now focuses on the particle filter and proposed improvements. Since the particle filter is one of the most popular methods for state estimation, a lot of research is done to improve its accuracy and performance. In theory the state of a complex system can be estimated correctly by sending out infinity particles. This sounds great at the first moment but with an increasing number of particles the efficiency decreases too. This is the point where many researches start their work. They try to decrease the number of particles while offering the same performance. Others try to adapt the number of particles based on the current state. If the PF is sure about its state the number of particles is decreased whereas the number is increased if the uncertainty increases. This is called adaptive particle filtering [34]. The particle filter improvements which are proposed in this section is part of recent research which claims to be different than other approaches. If promising methods are found, the state estimation of a robots localization status could be improved in further research by improving the current particle filter. First a new tracking method is presented that applies particle filters on voxels in the environment. Then a method for robust localization that uses lanes as markers is presented where the weight update step and resampling step is combined.

Tracking Objects with Particle Filter and Voxels

Morales et al. presented a method for object tracking by using a 3D occupancy grid as environment representation and a particle filter based approach for detecting and tracking obstacles [102]. Therefore they took a 3D point cloud as input and converted it into a occupancy grid. To do so an empty grid is created and filled with empty voxels. Then for each voxel g a set of points P_j is searched within its neighbourhood. The size of the set P_j ($||P_j||$) then indicates the occupancy probability of the voxel. The probability also depends on the distance Z between the voxel and the camera. The probability of occupancy for voxel g can be calculated using its centered position $\mathbf{g_c} = (g_x\ g_y)^T$

$$P(occ|g) = \frac{||P_j||}{\sqrt{(u_1 - u_0 + 1) \cdot (v_1 - v_0 + 1)}} \tag{3.35}$$

with

$$\begin{aligned} u_0 &= (g_x - \frac{v_s}{2}) \cdot \frac{f_x}{2} \qquad & u_1 &= (g_x + \frac{v_s}{2}) \cdot \frac{f_x}{2} \\ v_0 &= (g_y - \frac{v_s}{2}) \cdot \frac{f_y}{2} \qquad & v_1 &= (g_y + \frac{v_s}{2}) \cdot \frac{f_y}{2} \end{aligned} \tag{3.36}$$

where v_s is a user defined size of a voxel and f_x, f_y are the focals in x, y. The voxel grid is then used for the prediction step. Therefore a particle distribution is used that is based on a motion model for each particle and also takes the time between two frames into account. While some researchers proposed a particle filter for indicating the occupancy of a voxel, Morales et al. use the occupancy probability as described above and thus only have to compute the speed of each voxel with the particle filter. The weighting and resampling step is based on the assumption that older particles represent real speed vectors with higher probability. Each voxel g holds a list of its containing particles sorted by age. When defining a maximum number of particles for a voxel τ_{max}, the resampling process only has to remove particles from this list. The main speed vector for a voxel can then be calculated with different approaches. One is the **weighted mean**. It uses the speed vectors v_x, v_y, v_z of a particle q_i and its age ψ. The age is represented by the number of cycles which the particle survived. The main vector of voxel g can now be calculated as

$$V_g = \frac{\sum_{q_i \in Q_g} q_i(v_j) \cdot q_i(\psi)}{||Q_g||} \tag{3.37}$$

where $j \in x, y, z$ and Q_g is the particle set of voxel g. The voxels are then combined into clusters which are used to track objects within the grid map. For validation they used different error measurements and configurations. The configuration changes consist of changes like the voxel size v_s, the maximum number of particles τ_{max} or the maximal speed. The result showed that a fast and powerful tool for detecting and tracking object was proposed. It can be applied for 3D point clouds in autonomous driving. The combination of particle filter and voxelization showed a good performance increase while keeping the accuracy high. This might be an interesting point for robot navigation and localization when a 3D sensor like LIDAR can be used. This might offer complete new possibilities in localization scoring and robot localization itself.

Robust Lane Localization with PF

A technique which tries to optimize the localization on lines is presented by Rabe and Stiller [118]. they proposed a method for optimizing a vehicles localization within the environment by using a particle filter and sensors like a lane detector. Their approach is to optimize the performance of the particle filter by combining the weight update step and the resampling step. Before the weight update takes place the action u_k is applied, resulting in the intermediate belief which is calculated as

$$\overline{bel}(\mathbf{x}_k) = p(\mathbf{x}_k | \mathbf{z}_{1:k-1}, \mathbf{u}_{1:k}) \tag{3.38}$$

where k indicates the current time step, $\mathbf{x}_k = (x_k\ y_k\ \theta_k)^T$ is the 2D pose of a particle, $\mathbf{z}_{1:k-1}$ is the sequence of observations and $\mathbf{u}_{1:k}$ is the action sequence that was applied. THis intermediate belief is then used to apply the sensor measurement z_k, resulting in the posterior belief

$$bel(\mathbf{x}_k) = p(\mathbf{x}_k | \mathbf{z}_{1:k}, \mathbf{u}_{1:k}). \tag{3.39}$$

After the weight update the resampling phase takes place. This is done because the weight of some particles tends to go towards zero while others get more important. In the resampling step less important particles are thrown away while more important ones are duplicated more often. This is the point where Rabe and Stiller join in [118]. They state to improve and combine the weight and update steps by applying an idea of the Kalman filter. They make the assumption that the intermediate belief $\overline{bel}(\mathbf{x}_k)$ follows a normal distribution $\mathcal{N}(\mu_p, \sigma_p^2)$. Also the observation probability is defined as a normal distribution $\mathcal{N}(\mu_m, \sigma_m^2)$. In this publication the observation is the lane-marking observation.

This is used to determine the posterior distribution which is proportional to $\mathcal{N}(\mu_c, \sigma_c^2)$ where

$$\mu_c = \frac{\mu_p \sigma_m^2 + \mu_m \sigma_p^2}{\sigma_m^2 + \sigma_p^2}, \qquad \sigma_c^2 = \frac{\sigma_m^2 \sigma_p^2}{\sigma_m^2 + \sigma_p^2}. \tag{3.40}$$

The median and variance for the intermediate distribution can be approximated as

$$\begin{aligned} \hat{\mu}_p &= \frac{\sum_{i=1}^N w_i \mathbf{x}_i}{\sum_{i=1}^N w_i} \\ \hat{\sigma}_p^2 &= \frac{1}{N-1} \frac{\sum_{i=1}^N w_i (\mathbf{x}_i - \mu_p)^2}{\sum_{i=1}^N w_i} \end{aligned} \tag{3.41}$$

where N is the number of particles and w_i are the corresponding weights for each particle. This can now be used for conducting the resampling step. This is done by shifting each particle by

$$\triangle \mathbf{x}_i = \mu_c - \frac{\sigma_c}{\hat{\sigma}_p} \hat{\mu}_p + \left(\frac{\sigma_c}{\hat{\sigma}_p} - 1 \right) \mathbf{x}_i. \tag{3.42}$$

This approach is not yet robust since only normal distributions can be modelled. This might be an issue e.g. at the initialization step where the particles are distributed based on an uniform distribution. To overcome this issue a more robust approach is presented which models the intermediate belief as a combination of an uniform distribution and a normal distribution of the form

$$\begin{aligned} g(x, W, \Theta) &= \pi_1 f_1(x, W, \theta_1) + \pi_2 f_2(x, W, \theta_2) \\ &= \pi_1 \mathcal{U}_1 + (1 - \pi_1) \mathcal{N}_1 \end{aligned} \tag{3.43}$$

with parameters

$$\Theta = \{\theta_1, \theta_2\} = \{\{\pi_1, a_1, b_1\}, \{\pi_2, \mu_1, \sigma_1\}\} \tag{3.44}$$

where $\pi_2 = 1 - \pi_1$ and

$$\begin{aligned} \mathcal{U}_i &= \mathcal{U}(x, a_i, b_i) = \begin{cases} \frac{1}{b_i - a_i}, & \text{if } x \in [a_i, b_i] \\ 0, & \text{else,} \end{cases} \\ \mathcal{N}_i &= \mathcal{N}(x, \mu_i, \sigma_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(x - \mu_i)^2}{2\sigma_i^2} \right). \end{aligned} \tag{3.45}$$

This can then be used to estimate the parameters Θ and the assignment probabilities. The result of this study shows that a lot of performance optimizations were done whereas the accuracy loss was only small and is still better than by using simple normal distributions.

The publication from Rabe and Stiller [118] shows that a lot of performance optimization can be done by combining the weight update and resampling step. Although the publication focuses on sensors for line-marking observations this also might be important knowledge for the improvement of particle filters on robot localization using 2D range sensors.

3.6 Combining Particle Filters and Neural Networks

The last research topic that is closely related to this thesis are particle filters and their combination with neural networks. This also describes the goal of this thesis. Taking a particle filter and using its information for training a neural network. However, research does not address the exact same topic. Particle filters are a popular method which is known to support various machine learning approaches and help them to solve complex tasks. Thus, literature combines particle filters and neural networks in a way such that both methods work together or improve each other. The task of using a particle filter and not combining it with a neural network but using it to train a neural network seems to be new. Although recent research does not handle the supposed combination of the used methods, it might be important to investigate how a particle filter and neural networks work together so far and what improvements is offered by this combination. Therefore, an approach for supporting a particle filter with a neural network such that less particles are needed for localization is presented. More over how an aircraft can be tracked in video frames is discussed to show how particle filters and neural networks can be combined nowadays.

Reducing Particles due to Neural Networks

Localization is an essential part of mobile robot navigation. To efficiently navigate a robot through an environment it is indispensable to have an efficient and accurate localization algorithm. In terms of efficiency Choi et al. proposed a new navigation algorithm that is based on particle filters but uses a reduced particle set by combining it with a neural network [23]. Therefore they use a radial basis function (RBF) that is used for training. Their proposed algorithm consists of six steps

1. **Initialization step**
   To combine the process they use an input vector $\mathbf{x}_k$ which is retrieved from the encoder $\mathbf{u}_k$ (e.g. odometry) and an measurement vector $\mathbf{z}_k$ for training. Then a set of particles S_k is created, containing M particles. This set contains the particle $\mathbf{x}_k^m$ and its weight $\mathbf{w}_k^m$ that incorporates with the measurement.
   $$S_k = \{\mathbf{x}_k^m, \mathbf{w}_k^m | m = 1, ..., M\} \tag{3.46}$$

2. **Prediction step**
   In the prediction step the particles are updated based on the motion of the robot. This is done by updating each particle according to the given action from the motion model. In the motion model also some random noise has to be added such that the variable of interest also simulates noise. The variable of interest is the weighted sum over all particles. The noise is often simulated with Gaussian noise. Therefore the robot is initialized at its position as $\mathbf{x}_0 = (x_0 \; y_0 \; \theta_0)^T$. During the movement of the robot the difference between the rotation is calculated as
   $$\delta\theta_k = \theta_k - \theta_{k-1} \tag{3.47}$$
   and the orientation is calculated as
   $$\theta_k = \arctan(\triangle y_k / \triangle x_k). \tag{3.48}$$

The translation of the robot is calculated as

$$\rho_k = \sqrt{\triangle x_k^2 + \triangle y_k^2} \tag{3.49}$$

resulting in the motion model with errors such that

$$\begin{aligned} \mathbf{x}_k = \begin{pmatrix} \hat{x}_k \\ \hat{y}_k \\ \hat{\theta}_k \end{pmatrix} &= \begin{pmatrix} x_k + \rho_k \cos\theta_k \\ y_k + \rho_k \sin\theta_k \\ \theta_k \end{pmatrix} \\ \hat{\theta}_k &= \hat{\theta}_{k-1} + \delta\hat{\theta}_k + \mathcal{N}(\mu_{rot}, \sigma_1 rot\delta\hat{\theta}_k) \end{aligned} \tag{3.50}$$

where σ_{rot} is the rotation in radian, $\sigma_1 rot\delta\hat{\theta}_k$ and μ_{rot} are the mean and standard deviation. The translation can have two different errors. The first is the error in the distance and the second one is the error in orientation. This results in

$$\hat{\rho}_k = \rho_k + \mathcal{N}(\mu_{dist}, \sigma_{dist}) + \mathcal{N}(\mu_{or}, \sigma_{or}) \tag{3.51}$$

where μ_x and σ_x are the means and standard deviations for the distance travel and orientation. The input of the prediction step is the particle set S_k with the latest action $\mathbf{u}_k$ and the recent observation $\mathbf{z}_k$. S_k represents the intermediate believe such that

$$\overline{bel}(\mathbf{x}_k) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k) bel(\mathbf{x}_{k-1}) d\mathbf{x}_{k-1}. \tag{3.52}$$

3. **Weight update step**
In the weight update step the weight of each particle is updated according to the observation $\mathbf{z}_k$ that was made

$$\mathbf{w}_k^m = p(\mathbf{z}_k|\mathbf{x}_k^m) \tag{3.53}$$

resulting in the posterior believe

$$bel(\mathbf{x}_k) \approx p(\mathbf{z}_k|\mathbf{x}_k^m)\overline{bel}(\mathbf{x}_k). \tag{3.54}$$

4. **Training the RBF**
Having updated the weight, it is used to for training on the RBF. Therefore the weight coefficient is calculated as

$$\pi_{i,k} = [f_i(\mathbf{w}_k^m)' f_i(\mathbf{w}_k^m)]^{-1} f_i(\mathbf{w}_k^m)' \mathbf{z}_k \tag{3.55}$$

with

$$f_i(\mathbf{w}_k^m) = \exp\left(-\frac{||\mathbf{w}_k^m - \mathbf{c}_{i,k}||^2}{r_{i,k}^2}\right) \tag{3.56}$$

where $\mathbf{c}_{i,k}$ is the i-th basis function center vector and $r_{i,k}$ the i-th basis function center width. The new weight can then be trained with

$$\mathbf{w}_k^m = \pi_{0,k} + \sum_{i=1}^{P} \pi_{i,k} f_i(\mathbf{w}_k^m) \tag{3.57}$$

where P is the depth of the radial basis function.

5. **Resampling**
   In the resampling step the particles in the set S_k are replaces by M new particles. The probability of drawing a particle is given by the distribution of importance weights from the old set. The new particle set is then normalized by its weights such that $\mathbf{w}_k^m = 1/M$.

6. **Update step**
   In the update step all necessary variables are updated. When the prior pose is given as $\mathbf{x}_{k-1} = (\hat{x}_{k-1}\ \hat{y}_{k-1}\ \hat{\theta}_{k-1})^T$ and the robots posterior pose is given as $\mathbf{x}_k = (\hat{x}_k\ \hat{y}_k\ \hat{\theta}_k)^T$, the observation can be seen as

$$\mathbf{z}_k = \begin{pmatrix} \hat{\rho}_k \\ \hat{\theta}_k \\ \hat{\phi}_k \end{pmatrix} = \begin{pmatrix} \sqrt{d_x^2 + d_y^2} \\ atan2(d_y/d_x) - \hat{\theta}_{k-1} \\ atan2(-d_y/-d_x) - \hat{\theta}_k \end{pmatrix} \tag{3.58}$$

   with $d_x = \hat{x}_k - \hat{x}_{k-1}$ and $d_y = \hat{y}_k - \hat{y}_{k-1}$. The robots pose as described in Equation 3.50 can now be rewritten into

$$\mathbf{x}_{k,ms} = \begin{pmatrix} \hat{x}_{k,ms} \\ \hat{y}_{k,ms} \\ \hat{\theta}_{k,ms} \end{pmatrix} = \begin{pmatrix} \hat{x}_{k-1} + \hat{\rho}_k \cos\hat{\phi}_k + \hat{\theta}_{k-1} \\ \hat{y}_{k-1} + \hat{\rho}_k \sin\hat{\phi}_k + \hat{\theta}_{k-1} \\ \pi + \hat{\phi}_k + \hat{\theta}_{k-1} - \hat{\theta}_k \end{pmatrix} \tag{3.59}$$

   and for each particle measurement $\mathbf{z}_k^m$ the observation can be written as

$$\mathbf{z}_k^m = \begin{pmatrix} \hat{\rho}_k^m \\ \hat{\theta}_k^m \\ \hat{\phi}_k^m \end{pmatrix} = \begin{pmatrix} \sqrt{(d_x^m)^2 + (d_y^m)^2} \\ atan2(d_y^m/d_x^m) - \hat{\theta}_{k-1} \\ atan2(-d_y^m/-d_x^m) - \hat{\theta}_k^m \end{pmatrix} \tag{3.60}$$

   with $d_x^m = \hat{x}_k^m - \hat{x}_{k-1}$, $d_y^m = \hat{y}_k^m - \hat{y}_{k-1}$ and $\mathbf{x}_k^m = (\hat{x}_k^m\ \hat{y}_k^m\ \hat{\theta}_k^m)^T$. Using three standard deviations $\sigma_{\hat{\rho}}, \sigma_{\hat{\theta}}, \sigma_{\hat{\phi}}$ for the measurement noise of a observation, one can calculate the probability $p(\mathbf{x}_k^m|\mathbf{x}_{k-1}, \mathbf{z}_k)$ as

$$\begin{aligned} p(\mathbf{x}_k^m|\mathbf{x}_{k-1}, \mathbf{z}_k) = &\frac{1}{\sqrt{2\pi}\sigma_{\hat{\rho}}} \exp\left(-\frac{(\hat{\rho}_k - \hat{\rho}_k^m)^2}{2\sigma_{\hat{\rho}}}\right) \\ &\cdot \frac{1}{\sqrt{2\pi}\sigma_{\hat{\theta}}} \exp\left(-\frac{(\hat{\theta}_k - \hat{\theta}_k^m)^2}{2\sigma_{\hat{\theta}}}\right) \\ &\cdot \frac{1}{\sqrt{2\pi}\sigma_{\hat{\phi}}} \exp\left(-\frac{(\hat{\phi}_k - \hat{\phi}_k^m)^2}{2\sigma_{\hat{\phi}}}\right). \end{aligned} \tag{3.61}$$

By applying those six steps for a localizing a mobile robot, Choi et al. managed to get an acceptable localization result by only using 40 particles [23]. Compared to other methods like Kalman filter or normal particle filter they managed to get slightly better errors with their RBF optimization. This research is especially interesting because it is an optimization method that can be applied to the normal particle filter for performance improvement. Thus the evaluation time of the localization status could also be improved.

Combining PF and NN for Aircraft Tracking

Another field of application for particle filters and neural networks is aircraft tracking. Izadkhah et al. presented a novel approach for aircraft tracking which makes use of both methods to increase the tracking accuracy [72]. Therefore they use images of flying aircrafts as input and combine particle filters and neural network to visually track the aircraft. The main problem which currently exists is the problem of loosing the aircraft while tracking is done due to changes in speed, occlusion or in light conditions like reflection. To make their approach more stable Izadkhah et al. decided to split their tracking approach into three steps. The first step is to use a particle filter for estimating the position of the aircarft within a video frame. This knowledge is then used to segment the target from its background feeding it in a neural network and to use the greedy snake algorithm to find the contour of the aircraft.

The particle filter was chosen instead of the Kalman filter because it offers lots of advantages. For example, particle filters can handle non-linear systems and thus are better to handle fast speed and orientation changes. To segment the target from its background the Epanechnikov kernel is layed over the estimated position [24]. It is defined as

$$k(x) = \begin{cases} \frac{1}{2}c_d^{-1}(d+s)(1-\|x\|^2)\text{ , if } \|x\|^2 \leq 1 \\ 0\text{, otherwise} \end{cases} \tag{3.62}$$

where c_d defines the volume in the d-dimensional region. Having the silhouette region of the target, the kernel can be applied over this. The silhouette of the aircraft is then also modified with various image processing algorithms. The result is then used to feed a perceptron neural network. The task of the neural network is to exactly split the background from the target pixels. Therefore, the output size of the video frame is the same as the input size. This allows the neural network to classify each pixel. The outcome of the neural network is a binary image where white indicates the background and black indicates the shape of the aircraft. Having an exact shape of the aircraft the Greedy Snake algorithm is applied to get the contour of the target [78].

This approach shows that a combination of different methods, especially particle filter and neural networks, can be used to solve or improve given tasks. A particle filter is used to track a target but it is not directly used to feed a neural network. Instead, the estimated pose of an aircraft is used to separate the background from the target. This is a way of improving neural networks with the tracking outcome of a particle filter. Although the particle filter is not explicitly used as input of a neural network this still gives a good example on how the particle filter is currently combined with neural networks.

Chapter 4

Concept

This chapter focuses on the concept how to train a neural network using information from particle filters as input data to evaluate the localization accuracy. To keep the outcome of this thesis transparent it is important to clearly specify every step which is conducted. At first, the particle filter is analysed to define the information which is used for training a neural network. Then it is described how data samples are classified into two classes, localized and delocalized namely, and how they are generated. The next step is to enforce delocalizations by driving the robot within an environment to receive samples that represent the situation where the robot lost its localization. Having generated enough data samples the network structure is designed and presented. It is also described how networks are trained and how the results are used for boosting the current scoring technique.

4.1 Overall Concept

To get a overall idea of the concept this section discusses the main steps which are conducted in this thesis to estimate the localization accuracy of a robot. Figure 4.1 illustrates the overall concept. A robot is simulated and randomly driven through an environment to produce data. Some of this data is collected, like the information of the laser scan, the map as well as the particle filters information. This information is then used to extract various features which are used in boosting approach for estimating the localization accuracy. The distribution of the particles is also used as input of neural networks to train the localization state of a robot. The trained neural network is then used to score the localization accuracy. It is also used as additional input for the boosting approaches to further improve the combined localization accuracy.

4.2 Particle Filter Analysis and Data Filtration

To train a network with information from a particle filter that is used for robot localization one has to define what sort of information to use and how it should be represented. This section focuses on the *what*-part, so it is described what information is used for training the neural networks.

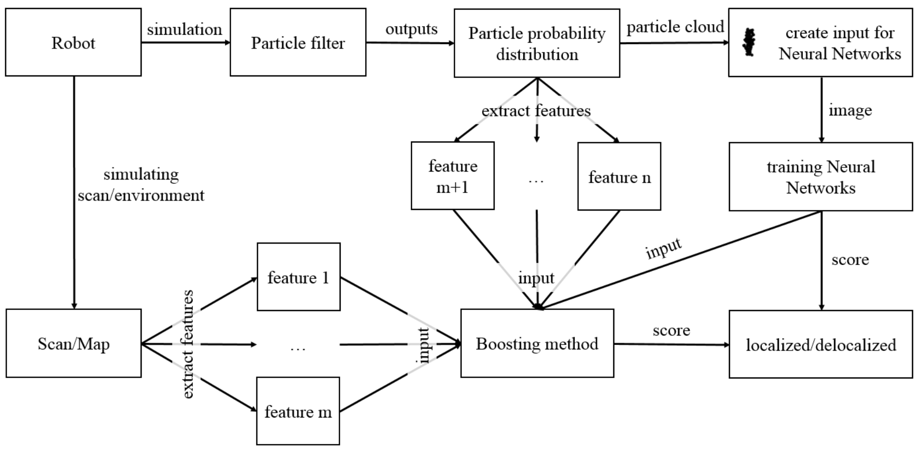


Figure 4.1: The overall concept for extracting features and estimating the localization quality

When looking at Section 2.3.4 and how the particle filter works one receives important information like the shape of the particle cloud which is used to determine the robots position. This information is used by the particle filter to estimate the robots position state. Since it is hard or often impossible to calculate complete distributions for complex systems, a limited number of particles are used which represent the complete system. The more particles are used the more accurate the estimation gets. But the performance also depends on the number of particles such that a compromise between efficiency and accuracy has to be made. An often used number of particles is around 1000 [146] because the performance and the accuracy trade-off is still acceptable.

The information which can be observed from the particle filter are the particles. Those particles help representing the complex system and hold information about the robots position. For the state of the localization task each particle in the set contains its position (x, y) within an environment and its associated orientation θ. The observation of the particle set can be done during two stages of the particle filter algorithm.

1. After the **evidence** phase, where the particles are weighted based on the observation made. This leads to a particle set that contains not only the position and orientation but also a weight for each particle. This weight indicates how likely it is to be on the location represented by the particle.

2. After the **resampling** phase. Here a new particle set is generated, based on the weights which were calculated in the evidence phase. The weight of each particle is normalized since the probability of the evidence phase was used to generate this sample. This leads to the fact that the new particle set is designed based on the previously calculated weights.

This thesis deals with the particle set which is retrieved after the resampling step because it is easier to represent the set for neural networks when no additional weight parameter is used (see Section 4.3.1).

4.3 Generating and Separating Data

This section describes how data is generated and how it is labelled. To understand the procedure of creating a training sample one has to understand how data is generated and how it is labelled.

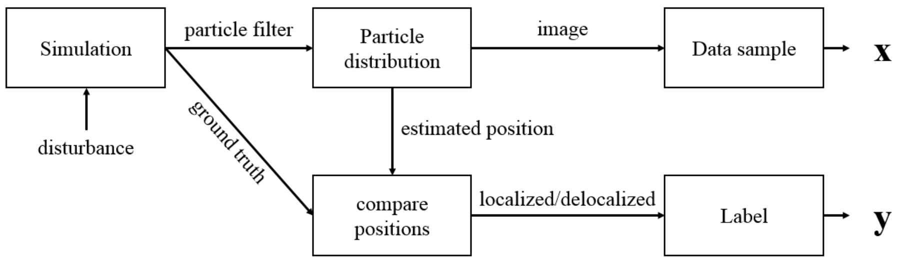


Figure 4.2: The overall procedure of generating a data sample and labelling

Figure 4.2 shows the overall procedure of generating a data sample and finding the correct label. To create a data sample one needs the information from the particle filter which is used to create an training image x. This image is labelled by comparing the exact position, retrieved by the simulation and the estimated position, retrieved by the distribution of the particles. By determining how different those two positions are one can classify the robots localization state and thus also create a label y for the data sample. The exact procedure of creating and labelling data samples is shown within the next sections.

4.3.1 Data Generation

This section focuses on the question about *how* the data should be represented. This is an important step since some kind of neural networks work better with a specific format for the data samples. For instance convolutional neural networks are designed for pattern recognition on images and therefore work best if the input is an image. For recurrent networks it is important to keep track of the sequence order since it is important to train samples in correct order. To satisfy both network types, CNN and RNN, the particle set is used to generate an image which is labelled with the time step.

To create an image one needs to represent the particle set in a proper way. To do so, the position of each particle is placed within an empty image grid and represented as black dot. The density of the dots represents the likelihood of a position, the image represents the probability distribution. Since the particle can be spread all over the map grid, the problem might occur that an environment that is used to classify the robots position status does not have the same size as the environment which was used for training. Also, it might become very expensive to train a neural network when the environment is large because the image would then have to represent the complete grid. Using large images which represent the whole grid also leads to the issue that the particle set is hard to localize since it might only be a small black cloud in a large environment. To overcome these problems a filter is applied. When creating an image only the immediate surrounding of the particle set is used, instead of the complete map grid. This part holds the most particles and contains the most relevant information which is used for training. To restrict the image size on the estimated location, one also has to define the location of the immediate surrounding. In this thesis the center of gravity of the particles is used to determine the center of the image. Therefore the mean over all particles is calculated as

$$\mathbf{m} = \frac{1}{N} \sum_{i}^{N} \mathbf{x}_i \tag{4.1}$$

where N is the number of particles within the set and $\mathbf{x}_i$ is the position vector containing the particles position in the environment $\mathbf{x}_i = (x_i, y_i)^T$. This mean is usually also used as the stated robots position which further motivates this choice. Since the particles indicate a possible position, they will occur closely together when the robot is well localized. Thus the mean over all particles is a good approximation of the robots position. When the shuttle looses its localization the particle set will start to diverge but the mean still indicates where the most information can be found. Assumed that two particle cluster form on completely different locations in the environment, the mean might point to a location which does not hold any information at all. This is not an issue because such separated clouds indicate a delocalized robot and thus it might also be learned by the network. For additional information, each image is rotated by the mean of the orientation of all particles. This leads to the fact that every image is aligned based on their main orientation. This might also reveal information about the localization state of a robot.

To keep the images small enough such that a neural network can be trained efficiently, the image has to be cut out on a stated position. This position is the mean of the particle set. The size of an image can be chosen individually. In this thesis a size of 36×36 pixel is used. This also has the advantage that the trained network can be validated using environments of different size. Figure 4.3 shows some examples of generated images, stored at different time steps t. The first three images illustrate a particle set of a localized robot and the fourth image is an example of a delocalized robot. Those images are only samples and do not imply that all other images look the same.

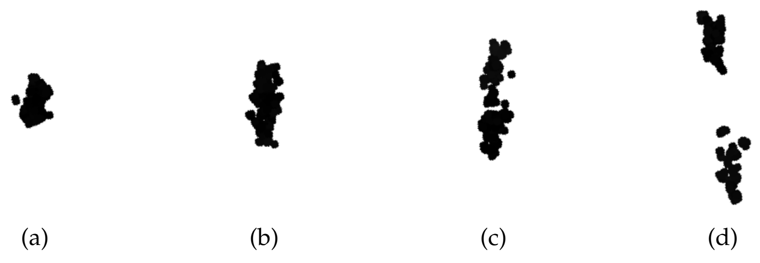


Figure 4.3: Example images generated for training a neural network. Each time step is done in a frequence of 5Hz. (a) Localized particle set at time step t. (b) Localized set at time t+40. (c) Localized set at time step t+90. (d) Delocalized image at time step $t + 110$.

The transformation and inflation of particle cluster may indicate the localization state of the robot. Thus a neural network can be trained to detect the patterns within the transformations. If that is the case and the particle set is inflating, the main task for a neural network is to learn at which state the robot becomes delocalized.

To generate enough distinctive data samples the robot has to drive randomly within a predefined environment. When creating training data for neural networks only one environment is created. This is done because it should also be evaluated if the form of the particle cloud changes with the environment structure or not. To receive many different training samples the robot is driven randomly through the environment such that it is unlikely to drive the exact same path twice. This leads to different robot locations within the map and thus also to different data samples.

4.3.2 Data Separation

Having created a big data set which can be used for training one also needs to separate the training set into localized and delocalized samples. When it comes to training a neural network one needs a huge amount of data samples such that the weights of the network are adjusted for many given samples. A human expert might encounter some troubles when labelling every sample by hand because it is time consuming.

To label data samples automatically one needs to observe the robots position which is estimated by the particle filter and the ground truth. In a real world environment it is difficult to observe the ground truth since one needs to determine the correct position at any stage of the robots movement. An approach for that would be a high precision motion capture system [103] which allows to analyse the robots position with high accuracy. Since those systems are expensive and because it is difficult to set them up for big environments, this solution was not an option in this thesis. Instead, a simulation software is used. This simulation software imitates an specified environment and simulates the robots hardware. The simulation software which is used is an adoption of Stage [57] and allows to simulate different kinds of robot hardware devices with defined errors. The advantage of such a simulation is that it can observe the ground truth of a robots position. This can then be used to compare the actual robot position with its stated one.

To label the dataset one can now define a threshold distance d. When observing the robots motion one can then calculate the distance between desired and stated position. This distance is then checked with the threshold and a sample is labelled in a way such that

$$y_i = \begin{cases} 0 \text{ , if distance } < d \\ 1 \text{ , else.} \end{cases} \tag{4.2}$$

When it comes to the question how the threshold distance should be chosen, some problem occurs. If the threshold is chosen too low data samples are marked as delocalized while they are still valid. This leads to the problem that a neural network might not find a solution because delocalized samples look too similar with localized samples. If the threshold is selected too high some delocalized positions might be labelled as correct location. Another problem with high threshold is that it becomes harder to generate delocalized sample data since it is difficult to produce a lot of incorrect data samples which have a high distance to the ground truth.

4.4 Enforcing Delocalizations

Using the simulation software above one can now use a robot to drive randomly in an environment and observe its positions. However, a good implementation of a particle filter will not lead to an localization error if every measurement is reasonably.

One possible solution to delocalize a robot is to move the robot to a stated position. This method is related to the kidnapped robot problem where one moves the robot without telling him how he moved. This solution is not usable for this thesis because one needs to observe the delocalization over a time period. The particle filter adopts its system based on the previous distribution. If one would randomly reposition the robot the previous distribution would be useless.

Therefore one needs a solution that can produce delocalizations by changing the input for the particle filter over a time period. The best way how this can be achieved is to simulate sensing errors in the observation and transition model. There are many different ways to produce sensing errors for a robot. The most common solutions are errors in odometry and dynamic obstacles in the laser scan. The simulation software which is used can simulate both, odometry errors and scan obstacles. This thesis focuses on only one error type since this allows to draw conclusions based on the selected error type. For the current situation it seems more important to focus on dynamic obstacles since the implementation of this task should be used in highly dynamic environments. To do so, the simulation is encouraged to randomly insert dynamic obstacles into the laser scan such that the particle filter has to account these differences when calculating the new particle distribution. This is also a realistic case in a real world since it is known that the robot looses its localization while driving through a group of dynamic obstacles, like a group of persons.

4.5 Preparing Network Structures

Having created a training set for neural networks one can start defining a network structure which can be used for training. The problem with defining a single network structure is that one cannot generalize the outcome of the network. Thus several network structures need to be defined and trained. When using particle filters as input it is hard to specify the best network type and structure since it is not clear which information is extracted by a neural netowrk for estimating the robots localization accuracy. A possible network structure is a convolutional neural network which is used for pattern recognition within an image. Another possibility is that a recurrent network might learn the transformation of the particle cloud over a certain time period. To evaluate all possibilities, both network structures are designed. A CNN for feature extraction and a recurrent network structure for learning the transformation over time. For learning a recurrent network a long-short term memory (LSTM) structure is used since a better performance is expected [67]. Additionally a combination of both, CNN and LSTM, is trained. This is called a Long-Term Recurrent Convolutional Network (LRCN). By using this network structure it is evaluated if the advantages of both networks can be combined.

For each different network type (CNN, LSTM and LRCN) three different network structures are created and trained. At first a simple model with only one or two type layers is created. Then a larger model and a complex model is designed and trained. Those three structures are chosen to evaluate whether a simple network structure leads to underfitting or a complex structure leads to overfitting. Then, for each network structure different parametrizations are tried and evaluated. Therefore parameters like the learning rate or the learning rate policy are adopted to see if the results can be improved by using different learning rates. This results in a total of nine networks to be trained with different parametrizations.

In the following sections different network structures are modelled and the reasons for the different network types are discussed in more detail.

4.5.1 Training a CNN

A convolutional neural network has the advantage of detecting translated features by using feature layers. Those feature layers are the heart of CNNs. The reason why this network type is used in this thesis is because it is tried to extract feature information from the form of a particle cloud to detect delocalizations. To train a meaningful network one needs to have enough training samples and a network structure that fits to the purpose. However, creating a big set of data samples has some difficulties. Also finding the right structure is not so easy if one has few research topics to rely on. This section now focuses on those two problems. First, the difficulties with creating big data sets are discussed. Then three network structures that are used for training a CNN are proposed.

When generating data samples with a robot that randomly drives through an environment some problems occur. The first problem is to find situations where the robot looses its localization. While the robot is driving in the environment one needs to enforce the robot to loose its location by itself. It is hardly foreseeable when the robot starts to delocalize and also it does not occur so often. Due to the issues it is rather seldom to create delocalized images compared to localized ones. This leads to the issue that a lot more positive data samples are generated. In general, roughly 80% are positive data samples and only 20% represent delocalized states. If this data set is used to train a neural network, it would already achieve an accuracy of 80% by classifying every state as localized. This does not reflect any success of such a network structure. To overcome this issue one could randomly delete positive data samples until an acceptable ratio of localized and delocalized images is achieved. This would be about 50/50 or maybe 60/40. Another issue with the big data set is the correlation of samples [59]. When using ordered training data one obtains highly correlated examples for whole mini batches. This leads to the problem that the network might not extract relevant features. To overcome this issue training data should be randomly shuffled. This leads to uncorrelated data which usually performs better. Different random orderings do not perform completely the same but only vary in a small factor that is negligible.

Having created a working training set one needs to define possible network structures. To get a good comparison of the network and to determine the rough complexity of the structure that fits the best, this thesis trains three different network structures with increasing complexity.

The simplest network structure is shown in Figure 4.4a. It consists of one convolutional layer and one pooling layer, then merges them together in a fully connected layer which also calculates the output for classification. The next structure is shown in Figure 4.4b. It is based on the LeNet structure [3] and consists of two convolutional layers that are connected via pooling layers and two fully connected layers.

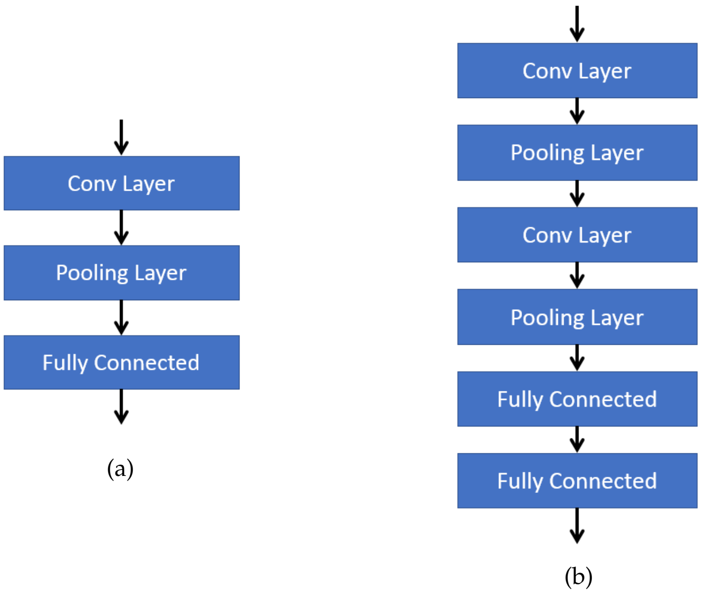


Figure 4.4: Structures of Convoutional Neural Networks. (a) Simple structure, (b) mid-complex structure

The complex network is based on a successful deep learning structure called *GoogLeNet* which was developed and published by Google [144]. Google presented a deep learning architecture which was designed for classification in the ImageNet Large-Scale Visual Recognition Challenge 2014. It differs from other network structures by its high utilization and carefully selected layers. Since Google has more computational power and because the network structure becomes quite large for learning with limited resources, the structure was only designed until the first classification output. Also the size of the local receptive fields and the number of feature maps for each convolutional layer was adopted.

Figure 4.5 illustrates the adapted GoogLeNet structure which was used for training images from the particle filter. Additionally to the simple and mid-complex structures this network also holds *concat layers*. Those concat layers take multiple layers as input and then combine them to one single output layer. This has the advantage that multiple layers can be created and learned in parallel.

To receive meaningful results all three different network structures are trained on the same data set. While changing the parametrizations, like the learning rate, on every network structure, we should be able to compare the results since the training and validation set are the same.

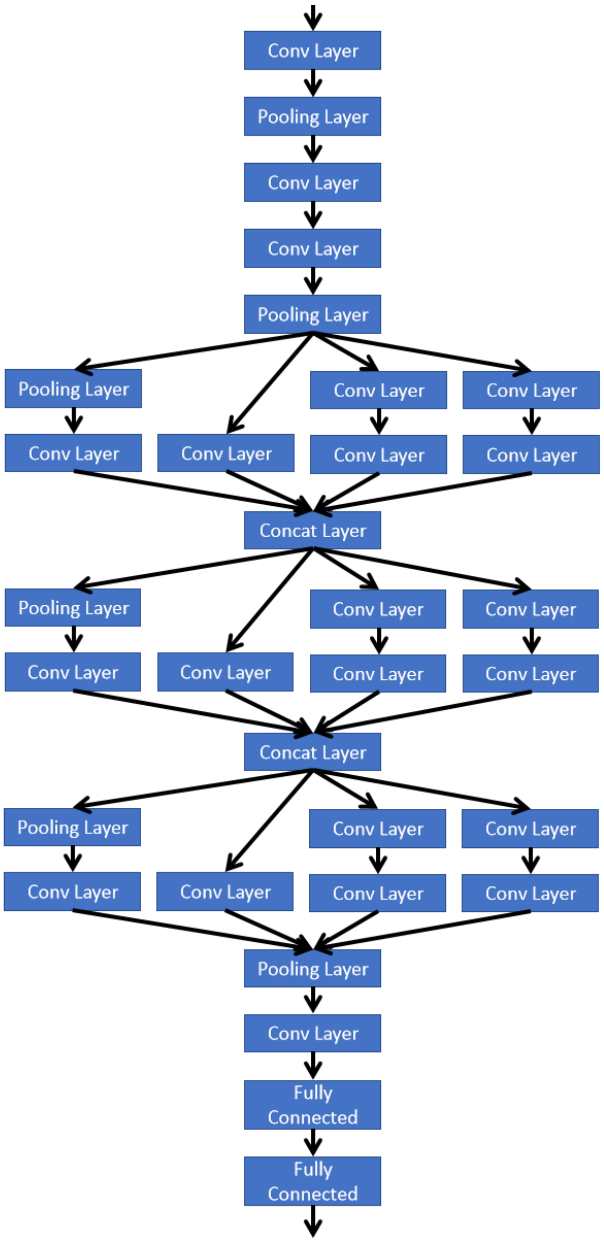


Figure 4.5: The complex network structure for a CNN leaned on GoogLeNet

4.5.2 Training a LSTM

When it comes to training a recurrent neural network one faces the same problems as before. First, the ratio between localized and delocalized training data does not fit since roughly 80% are labelled as localized. Also the data is highly correlated. The solution of randomly deleting images and shuffling them does not work for this kind of network because the network is trained with sequences. If one randomly deletes data samples the information about the sequence gets lost. The same holds for shuffling images. To overcome this issue a simple solution was found for this thesis. To keep a suitable ratio between positive and negative images, only delocalizations itself are recorded. While generating images, no image is stored directly. It is waited for a delocalization to occur which is then used to store a sequence of data samples with length s.

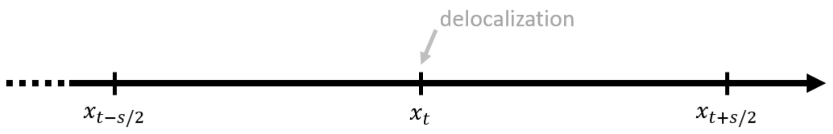


Figure 4.6: Illustration of generating sequence data sets for a LSTM. When a delocalization occurs a period from the past to the future is recorded and stored

To not only store the delocalization itself but also the inflation before the last $s/2$ samples are stored temporarily while generating the data. When the robot delocalizes, those $s/2$ images are stored as data samples and the next $s/2$ images are being recorded as well. The future is recorded since the development of the particle cloud might reveal further information about the localization accuracy. This leads to the result that roughly 50% correct images are recorded before the delocalization took place and roughly 50% incorrect images are created after the delocalization was detected. Figure 4.6 illustrates the method. At time step t a delocalization occurs so samples from $x_{t-s/2}$ to $x_{x+s/2}$ are generated.
Using this data would train a recurrent network to only detect starting delocalizations. It would not be possible to determine the normal localized state where the robot does not indicate to get lost. To solve this issue some random sequences of correct driving behaviours are also stored such that about 60% positive data samples are generated.

Another issue which has to be faced is the correlation of the data samples. LSTM networks need the information of its predecessor and thus also some kind of correlation. But it should also be possible to randomly shuffle input sets for recurrent networks. To do so, this thesis uses some sort of sequence shuffling. Instead of shuffling all the images randomly sequences of a certain length are hold together and those sequences are shuffled randomly.

Again, three different networks are created to determine how deep a LSTM structure should be. Figure 4.7 illustrates all three network structures. The LSTM layer consists of a defined number of LSTM units. At first there seems to be not so many differences but also the depth of each layer is different. With increasing network complexity the number of LSTM units for each layer increases too. This results in a high difference of the network structures since the simple network only uses a few LSTM units whereas the complex LSTM layers use more units that are learned.

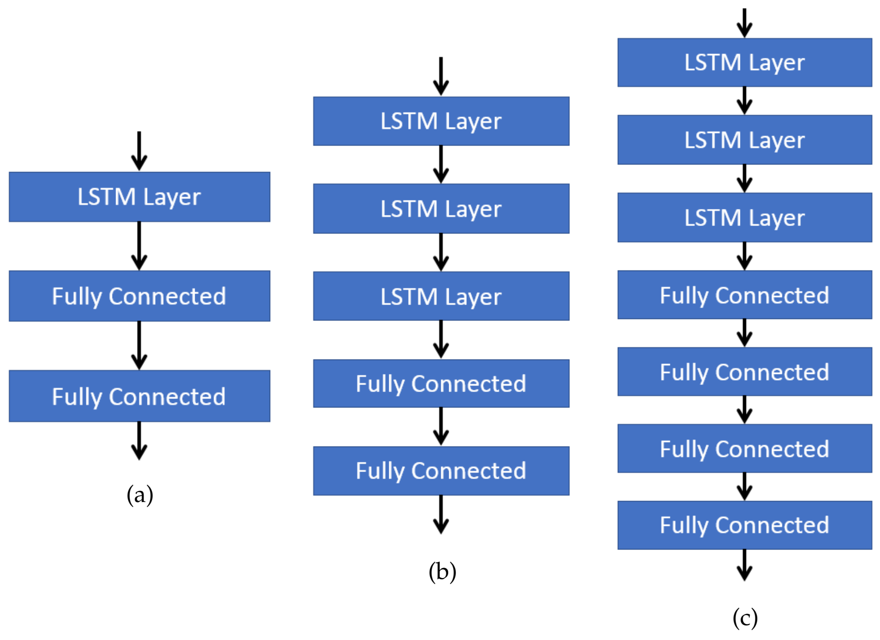


Figure 4.7: Defined structures of LSTM Networks. (a) Simple structure, (b) mid-complex structure, (c) complex structure

4.5.3 Training a Combination LRCN

The third network type which is used in this thesis are long-term recurrent convolutional neural networks [39]. They are a combination of the previously designed network types. For generating training data the same issues like for LSTM networks hold. Therefore some sequence shuffling and delocalization recording has to be done, like above. The reason why this network type is also evaluated is because it is tried to combine the advantages of both network types into one network. Of course it is not said that both network types perform well or that any of the previously presented networks can be used for state estimation on robot localization scoring, but even if none of those networks find an acceptable solution it might still be possible that a LRCN increases the efficiency by picking the best of both network types.

To validate the network type three different network structures are designed, again. Those network structures vary in complexity and thus are suitable for further comparison. The layers of a LRCN network are a combination of convolutional and recurrent layers. By changing the local receptive fields in convolutional layers and by adopting the number of LSTM units for the LSTM laye, one can easily increase the complexity without adding a ton of new layers. Figure 4.8 shows the network structures which are used for learning the classification of particle filters with LRCN networks. They do not only vary on the depth of the network but also on their single layer definitions.

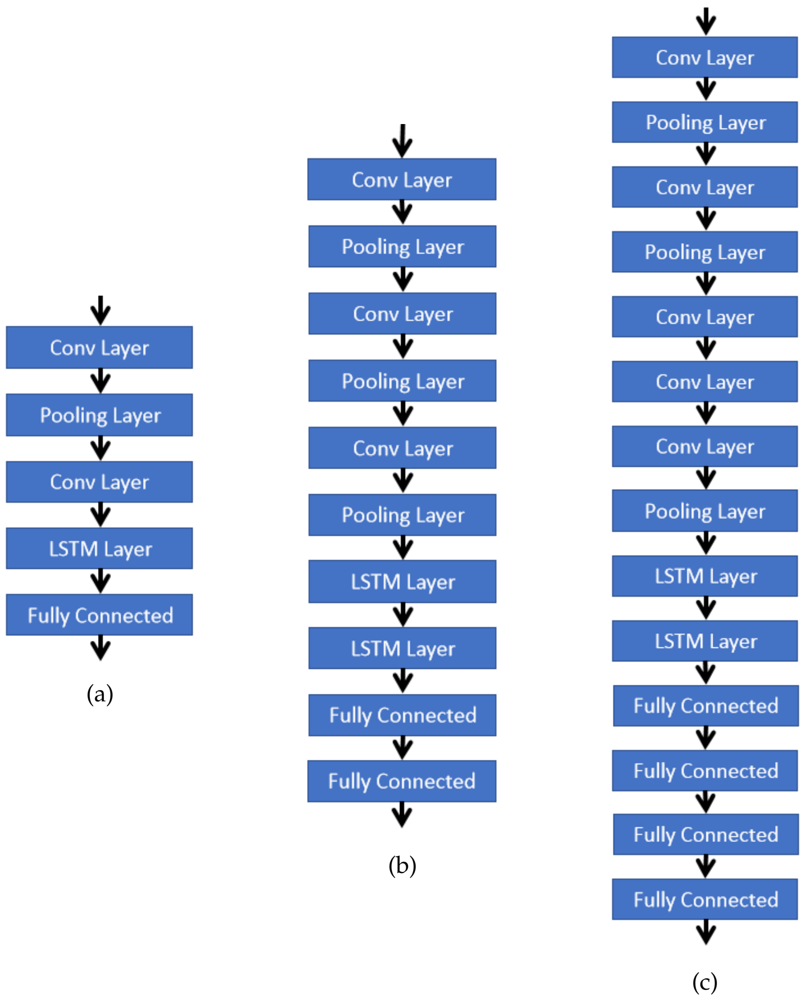


Figure 4.8: Defined structures of LRCN Networks. (a) Simple structure, (b) mid-complex structure, (c) complex structure

4.6 Validating networks

To determine whether a neural network succeeds in classifying information from particle filters one needs to define how the success is measured. This section shortly discusses how network types and structures are validated and how they are compared.

To validate a network one needs to define some kind of measurement which is not only meaningful but can also be used to compare different networks. The best measurement to do so is the **accuracy**. Accuracy is a value that describes how good a network fits to a given data set. If a classification task with two classes is done it can be easily computed by comparing each result with the desired output. Having two classes, one can assign to a class either the term *positive* or *negative*. To validate a data sample it is run through the network structure and the output is measured. The output is then assigned to one of the two classes, depending on the result. By comparing the assigned class with the expected label class one can now determine whether a class was correctly assigned or not. If both, the result and the label match, then the outcome is *true*. When a data sample is classified as positive and the expected class is also positive the data sample is *true positive*. If the two classifications do not match the outcome is *false*.If the data sample is classified as negative and the expected result is positive the outcome is *false negative*. By counting the results one can now determine the accuracy *acc* as

$$acc = \frac{t_p + t_n}{t_p + t_n + f_p + f_n} \tag{4.3}$$

where t_x indicates a correct classification and f_x an incorrect one. To normalize the classifications and show only the ratio of true and false classified samples, one can represent the measurements as percentage v_x such that

$$\begin{aligned} v_{tp} &= \frac{t_p}{t_p + f_n}, v_{fn} = \frac{f_n}{t_p + f_n}, v_{tp} + v_{fn} = 1 \\ v_{tn} &= \frac{t_n}{t_n + f_p}, v_{fp} = \frac{f_p}{t_n + f_p}, v_{tn} + v_{fp} = 1. \end{aligned} \tag{4.4}$$

Using the accuracy and its true-false ratios one can validate the network structure by testing it with specific data sets. This thesis uses three different data sets to evaluate the trained networks. The first data set is the training set which was used to train the neural networks. Out of this one can evaluate how well networks were trained on the given data set. The next set is a validation set which is recorded within the same environment. The robot randomly drives through the map and collects new data samples. These samples are then used to generate a new accuracy. Thus one can determine if a network was too well fitted on the training data. The third data set is generated in a completely different environment. This is done because it should be evaluated whether the particle clusters form a specific shape within a specific environment or if it could be generalized after learning a network in only one environment.

4.7 Boosting with AdaBoost and SVM

After training and validating the different network structures boosting is done. With this method it is tried to improve the accuracy of the current localization scoring algorithm. Therefore features are selected which currently calculate the accuracy of the robots localization and also new additional features may be added. Then a bunch of feature samples is generated and labelled for classification. Therefore the same distance *d* for defining a delocalized robot is chosen. This feature set is then used to learn an optimal weight distribution such that the classification of the robots location status is improved. This is done using the boosting algorithm AdaBoost and additionally a Support Vector Machine. Both algorithms are provided by the OpenCV library and can be used for classification tasks on two labels. After boosting the features one receives an accuracy as described above. It is then evaluated in this thesis whether this accuracy can be boosted by adding an additional feature which was learned with neural networks. This additional feature is the predicted class of the neural neural network that performed the best i.e. that had been rated the best based on its accuracy on the training and validation sets.

After adding the predicted class to the boosting algorithms the accuracy is again optimized. Both outcomes can then be evaluated and compared to determine whether a localization scoring method using features can be boosted by using predictions from neural networks.

Feature Selection

A task which has to be done before boosting is to select meaningful features that represent the localization status of a robot. In general it is hard to specify which features might be important for localization scoring. To do a primitive and simple evaluation one can search various possible features that might be useful for scoring the accuracy of a robots localization. By using a threshold distance d as above the robots localization status can be determined and every feature can be inspected on its own. Therefore n feature estimations are recorded and labelled to the correct class. Those recordings are then split up into a localized X_q and delocalized X_p set. These sets are then represented as a discrete distribution over k bins. Each bin contains a number of samples which lie within that bin. A discrete probability distribution P is then represented as the number of samples within that bin divided by the total number of samples n

$$P(i) = \frac{|bin(i)|}{n}. \tag{4.5}$$

The same holds for the distribution Q. Having two discrete distributions P, Q for the localized and delocalized set one can calculate the Kullback-Leibler divergence [116] as

$$D(P||Q) = KL(P, Q) = \sum_{i}^{k} P(i) \times \log \frac{P(i)}{Q(i)} \tag{4.6}$$

where k is the number of bins. The KL-divergence is only defined if $\forall i : Q(i) = 0 \rightarrow P(i) = 0$ applies. If $P(i) = 0$ the contribution of the i-th bin is also 0. It is defined that $D(P||Q) \geq 0$ for all distributions and $D(P||Q) = 0$ if $P = Q$.

Both distributions can then be observed and their KL divergence indicates their similarity. The higher the score the lower is their similarity. If they are completely different they indicate an excellent feature that holds a lot information about the localization status. If both distributions are similar it can be assumed that only little information is offered by this feature.
After evaluating a bunch of possible features one can then select the best results and use them to apply a boosting algorithm. This boosting algorithm then tries to optimize the features for a good state estimation. Of course different approaches can be done and a lot of improvements for these boosting algorithms are possible but this thesis only focuses on the improvement of boosting algorithms by using predicted network outcomes. Therefore a complete optimization of boosting methods is not conducted to receive comparable results.

Chapter 5

Implementation

To fully understand the results which are presented in this thesis one needs to have a basic understanding on the implementational part. Thus, the implementation of the concept is presented in this section to make the results that are presented in this thesis reproducible. First the setting of software and hardware components that are used is presented and the environmental setup for training and validating data is shown. Then it is shown how data samples are separated and labelled. Another step that is discussed in this chapter is the generation, training and validation of data samples as well as the extraction of relevant features for boosting the result of the neural networks.

5.1 Setting and Versions

This section presents the setting that is used in this thesis to create, train and validate data samples as well as neural networks. Therefore, the software and hardware components are shown. In this thesis two different hardware systems were used. The first was chosen for simulating a mobile robot and all its required components. Therefore, the simulation was done on a hardware that is also sold for industrial mobile robots by incubedIT GmbH[1].

Table 5.1 shows the main specifications of the machine that is used for mobile robot simulation. Generating data samples and validating them is implemented as a ROS node and thus also done on this machine. The simulated robot works on a simple platform that does not offer GPU acceleration. Since the used *Caffe* framework for training neural networks allows the use of GPU to reduce the computational training time a second machine is used. The second hardware is responsible for creating and training neural network structures. Table 5.2 shows the specifications of the second machine.

Although the implementation and evaluation was done on the proposed systems it might not be necessary to follow the exact same hardware specifications or software versions. Changes in Software like the operating system can lead to faults. Since this thesis was only done on the defined setting it is not guaranteed to work for other settings too.

[1]http://www.incubedit.com/solutions/smart-shuttles/

Processor	Intel Core i5-2510E
Hertz	2,5 GHz
RAM	4GB DDR3
Graphics Card	None
OS	Ubuntu 14.04 LTS
ROS Version	Indigo
OpenCV Version	3.3
GCC Version	4.8.5

Table 5.1: Specifications of the hardware and software setting for simulating a mobile robot

Processor	Intel Core i7-3612QM
Hertz	2,1 GHz
RAM	8GB DDR3
Graphics Card	Nvidia GeForce GT630M
OS	Ubuntu 16.04 LTS
OpenCV Version	3.3
Caffe Version	1.0.0
CUDA Version	7.5.17
Python Version	2.7.12
GCC Version	5.4.0

Table 5.2: Specifications of the platform for training neural networks

5.2 Environmental Setup

To see the potential results of this thesis it is also necessary to understand the environmental setting that was used for the mobile robot simulation. Basically, two different environments are used for training and validating. Both environments are designed for a mobile robot with certain difficulties that are believed to cause a localization problem. For example, a difficulty for the particle filter localization is a long corridor as discussed in Section 1.3. The first environment which is used is shown in Figure 5.1. It uses the map difficulties as described above and also contains walls that are not parallel to each other. This might also lead to delocalizations since the particle filter might change its opinion about its current location and rotate a bit so that the rotated wall is matched instead of the other one. The first environment is used for generating the training set. Therefore the robot is randomly driven around and data is generated based on its movement. While the robot is driving through the environment it is sensing its environment and avoiding obstacles so that no collision occurs. To validate a network structure this environment is also used but with a completely new generated dataset for validation such that the evaluation is persistent against overfitting.
The second environment shown in Figure 5.2 is only used for validation. The data generated from a simulated robot while driving around is used to validate a network structure in an independent environment. This is done to validate the network in other maps and to evaluate whether the network can be used in different kinds of environments or if it has to be trained on every

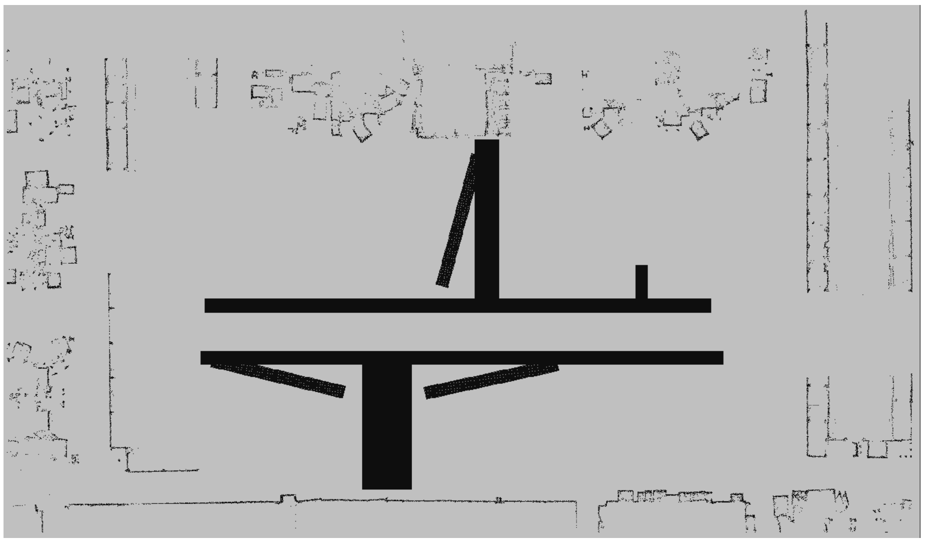


Figure 5.1: Environment for training and validating neural networks

new environment.

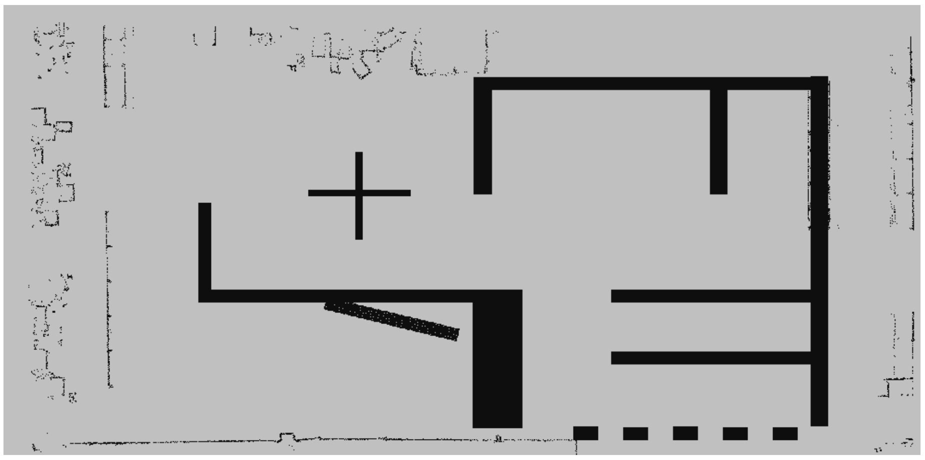


Figure 5.2: Environment for validating neural networks

5.3 Separating Samples

Before generating data samples that can be used for training and validating one has to define the state on which a robot is delocalized. As discussed in Section 4.3.2 it is necessary to define a threshold distance d which is used as boundary to label data samples.

Finding an optimal separation threshold is not trivial if the distance is not restricted to defined limits. Sometimes restrictions can be done by a user who e.g. states that the robot may never be off the route by more than 50 centimeters or 40 degree in orientation. This thesis is not restricted to user limitations and thus it is tried to find an optimal threshold distance. The threshold distance defines the allowed error in translation and rotation. The translation is given in meters while the allowed rotation error is given in radians. To do so the current scoring algorithm is used to identify a good threshold. This current implementation was done by employees of incubedIT who extracted features from the map grid and the laser scan which are used for scoring the accuracy. Therefore a simulated mobile robot is driven through an environment and different threshold distances between $0.5 \leq d \leq 1.5$ are used to classify the localization score of a state, where d is increased by 0.25 for every step. For each threshold distance about 15.000 samples were generated with the current scoring algorithm to receive an comparable result. One has to remember that the existing scoring algorithm is only a rough guideline since it is optimized in this thesis and does not work reliable. The outcome for each threshold distance are then two discrete distributions for localized and delocalized samples. The distributions are calculated by allocating each percentage of the localization score to a bin. This bin is then normalized over the number of samples to receive a discrete distribution, like $P(i) = \#bin(i)/n$ where i is the score, $\#bin(i)$ is the number of samples within bin i and n is the total number of samples of the distribution. Those distributions are then analyzed and compared to each other. An interesting thing for comparing them and getting their similarity would be to represent them via a normal distribution. However, this does not necessarily represent the distribution correctly since it is unlikely that every feature follows a normal distribution. To overcome this issue and to evaluate the similarity of both distributions, the Kullback-Leibler divergence (KLD) is used [116]. For two discrete distributions $P(x), Q(x)$ the Kullback-Leibler divergence is given as

$$D(P||Q) = KL(P,Q) = \sum_{i}^{k} P(i) \times \log \frac{P(i)}{Q(i)} \tag{5.1}$$

where k is the number of bins that are used in the distribution. The discrete distributions give the propability of bin i. This is calculated by using the number of samples that are within the bin and divide it by the total number of samples n. The KL-divergence is only defined if $\forall i : Q(i) = 0 \rightarrow P(i) = 0$ applies. If $P(i) = 0$ the contribution of the i-th bin is also 0.

threshold	*#bin*	$D(P\|\|Q)$
0.5	100	0.634
0.75	100	0.704
1.0	100	0.782
1.25	100	**0.874**
1.5	100	0.832

Table 5.3: Kullback-Leibler divergence for localized and delocalized samples on different threshold distances

Table 5.3 shows the result of the calculated Kullback-Leibler divergence on different threshold distances. *#bin* indicates the number of bins that were used. Since the scoring algorithm returns its result in percent each bin was allocated to a percent value so that a total of 100 bins were used. $D(P||Q)$ indicates the

KL-divergence of both distributions P, Q and thus also shows how similar those distributions are. P is in this case the distribution of delocalized samples and Q the distribution of localized samples. The outcome of the KLD can then be used to compare the similarity of the distributions. The higher the result, the more different the distributions are.

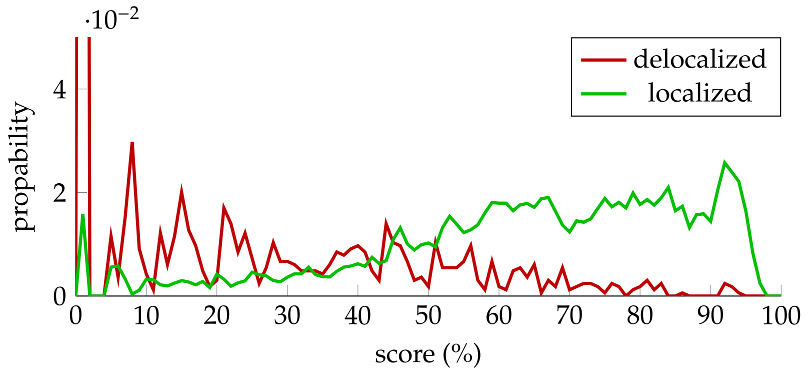


Figure 5.3: Separating delocalized and localized data samples with threshold distance of 1.25 meter and an allowed rotation difference of 1.25 radian

The result was that higher threshold distances improve the separation of the localized and delocalized curve. The optimal separation was found at 1.25 meters for translation and 1.25 radians for rotation. When setting the threshold distance one has to be aware that only a small number of delocalized samples are found if the threshold is set too high. This makes it hard to produce enough delocalized data samples for training. A threshold distance of 1.25 is on the upper limit of generating enough delocalized samples. It is still possible to collect enough samples but it is already very time consuming.

The final threshold distance was chosen to be 1.25 meters, respectively 1.25 radians for the orientation, since this offers a good data separation while producing enough delocalized data samples. Figure 5.3 shows the separation of the data samples with threshold 1.25 based on the current scoring algorithm.

5.4 Implementing Required Components

Having defined the setup and threshold distance for data separation one can start defining the components. To solve the problem of localization scoring with neural networks the components are split up into two phases.

1. **Training Phase**
   In the training phase a large data set is generated. Therefore, the corresponding node needs three different inputs. The first is the *particle cloud* which was generated by the particle filter. This input is used to create an image that contains the individual particles. The next input is the *robots pose* which was estimated based on the particle cloud. The third input is the *ground truth* which is received by the simulation observation and used in combination with the robots pose to label the image.

By letting the robot drive randomly in the environment and continuously updating the input one can generate a set of images. This set of images is then prepared and fed into a neural network for training. The result of this phase is a neural network structure with its trained weights.

2. **Validation Phase**
The validation phase is done after training. It is done to validate a network or to estimate a robots localization state in real time. This phase also generates an image based on the particle cloud but does not necessarily need to know the *ground truth*. The ground truth is only needed during the validation phase when the estimated class of an image has to be compared. If validation is not done one could simply deactivate the validation step and only use it for state estimation. This makes is applicable for real world applications since the ground truth can usually not be observed. To validate a created image the previously trained network is loaded and used to classify the image. This classification step is rather fast such that it can be computed on a robot while it is driving through an environment.

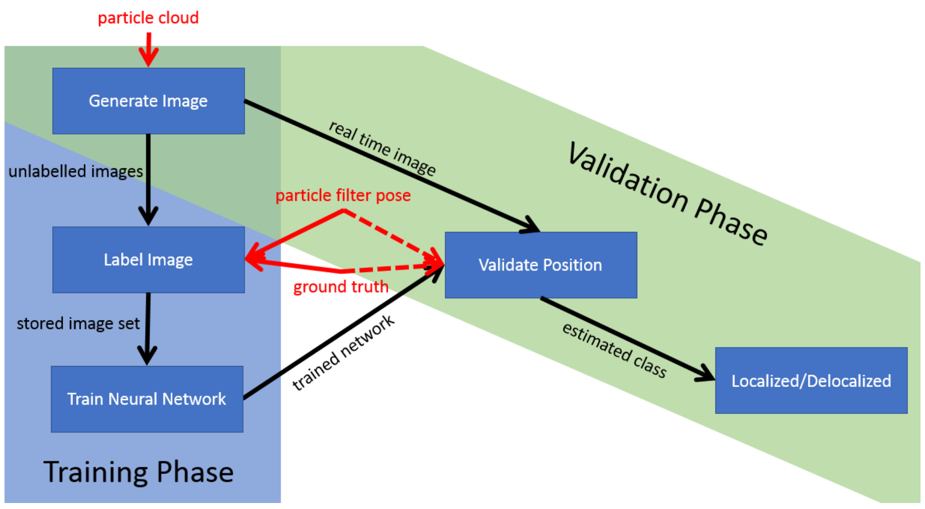


Figure 5.4: Component Structure for training and validating localization states

Figure 5.4 illustrates the two phases. The image generation can be shared by both phases. The labelling step is only done in the training phase because the it is not needed while validating. In the validation step the trained network is used to classify an image and thus estimate the robots localization state. If the ground truth can be observed within this phase, the estimated class can be compared to the desired class. This allows to measure the accuracy of the used neural network. This step does not depend on the ground truth which can only be received in the simulation and thus is also applicable in real world environments for estimating a robots localization state. The red arrows indicate the inputs for a certain steps. Those inputs are implemented using callback functions for certain topics. The red dotted line indicate an optional input for the validation phase.

The next sections focus on the single steps within the phases to get a better understanding about the image generation and on the implementation of the two phases.

5.4.1 Generating and Labelling Data

To generate images that are used for training and validating neural networks one needs to subscribe to the **particle cloud** topic that is published by the particle filter node. This particle cloud is then extracted and converted into an image. The procedure of extracting a particle cloud and converting it to an image that is used for training is described in this section. Figure 5.5 shows the particle cloud in the environment and how it is extracted.

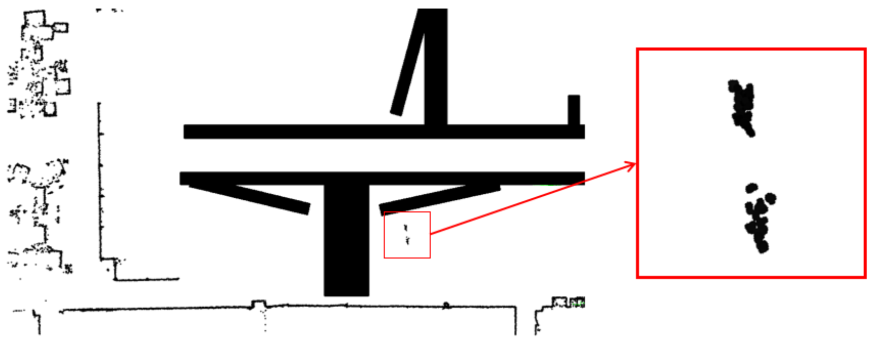


Figure 5.5: Extracting a particle cloud from the map and converting it into a training sample for neural networks

Whenever a new particle cloud is received it is used to generate an image. Therefore the position of every particle is normalized by the mean of its positions

$$\mathbf{p}_i = p_i - \bar{\mathbf{p}} \tag{5.2}$$

where $\mathbf{p}_i$ indicates a particle vector containing the position $p_i = (x_i \; y_i \; \theta_i)^T$ and $\bar{\mathbf{p}}$ is the mean vector over all particles, calculated as

$$\bar{\mathbf{p}} = \frac{1}{N}\sum_{i=0}^{N} \mathbf{p}_i \tag{5.3}$$

where N is the number of particles. The normalized particles are then drawn into the image around its center. Since the position of a particle is given in meters one does not know where to place each particle within the image grid. Thus a scaling factor f is needed so that the image is filled reasonably with particles. This factor can be seen as zoom parameter since it defines how big the field of view is on the image. Based on the image size and the wished detail of the particle cloud the zoom factor can be higher or lower. For this thesis $f = 10$ was used. After that the image is rotated by $\bar{p}_\theta$ radian and cut into a square with a predefined size $s \times s$.

For training a neural network the image needs to be labelled too. This is done by storing the image with an label flag such that the image name is *labelID_imageID.png*. To label the image the **ground truth** topic and the estimated robots pose is needed. The estimated pose is retrieved by the **particle filter pose** topic which states the robots position. The label ID is then defined by the Euclidean distance between the two positions, the difference in their rotation and the threshold distance d. As determined above, the threshold distance is chosen to be 1.25 within this thesis. The Euclidean distance for ground truth $\mathbf{x}_g = (x_g\ y_g\ \theta_g)^T$ and stated position $\mathbf{x}_s = (x_s\ y_s\ \theta_s)^T$ is calculated as

$$ed(\mathbf{x}_g, \mathbf{x}_s) = \sqrt{(x_g - x_s)^2 + (y_g - y_s)^2} \tag{5.4}$$

and the difference in orientation is simply calculated as $||\theta_g - \theta_s||$ where the norm also respects a possible overflow in rotation (from 2π to 0 or vice versa). The label l_i for particle i is then calculated as

$$l_i = \begin{cases} 0\text{ , if } \ d(\mathbf{x}_g, \mathbf{x}_s) \wedge ||\theta_g - \theta_s|| < d \\ 1\text{ , else.} \end{cases} \tag{5.5}$$

5.4.2 Training Data

Training a neural network is done separately to the ROS structure. The Caffe framework offers a binary that can be used for training neural networks (see Section 2.5). To prepare the data set for training the programming language **Python** is used. First, all created image paths within a given directory are loaded

```
import glob
dir_list = glob.glob(path/to/dir/*.png)
```

Depending on the network structure, the image list is either shuffled in sequences of length s_l or totally random. Then for each image the label is extracted from its name

```
import re
first_num = re.findall(r'/\d_', f)
label = (int) (re.findall('\d', first_num[0])[0])
```

This assumes that the image is named correctly and that the path to the directory does not contain a number. Then a text file as described in Section 2.5.2 is generated which contains a list of image paths and its associated label. This is then used to create a HDF5 data file for the Caffe framework. This is then added as input layer to the given network structure which is defined in the *prototxt* file

```
layer {
  name: "localization"
  type: "HDF5Data"
  top: "data"
  top: "label"
  hdf5_data_param {
    source: "path/to/hdf5/file_list.txt"
    batch_size: 100
  }
}
```

Listing 5.1: Integrating the HDF5 data set as input layer into a network

The network structure is then included in the solver file which is used to start training. The result of the training is a file with the ending *.caffemodel* that contains all trained weights for the given network structure.

5.4.3 Validating Data

To validate the network and use it for state estimation on live data the created *.caffemodel* file needs to be loaded into a ROS node. This node also creates an image as described above but does not store it. Instead it uses the image to propagate through the trained network and so to classify the robots localization status. Listing 2.15 shows an example implementation for loading a pre-trained network and classifying a generated image.

If the network is applied in real world environments the resulted label can be used to estimate the localization state. Depending on the accuracy of the trained network the estimation is more or less applicable. To evaluate how accurate the trained network is one needs to calculate the difference between the actual and desired position as in Section 5.4.1. Note that this can only be done in a simulation environment and that it is not applicable in real world estimations since the ground truth is needed for this task.

5.5 Feature Extraction

Having trained and validated a neural network structure one can now try to boost this structure. To do so one needs to find valuable features that can be used for boosting. Also a new component is needed that extracts features and uses them for boosting. This section now handles the structure of the new boosting component and evaluates a handful features that can be used for boosting.

5.5.1 Adding a Boosting Component

To further improve the localization scoring accuracy the state estimation of a trained neural network is used in combination with various features. This combination is inserted into the AdaBoost algorithm provided by OpenCV. It is also tried to boost the localization scoring with an implementation of Support Vector Machines, also offered by OpenCV.

To conduct boosting one has to adapt the predefined component structure and add the new component as illustrated in Figure 5.6. This component

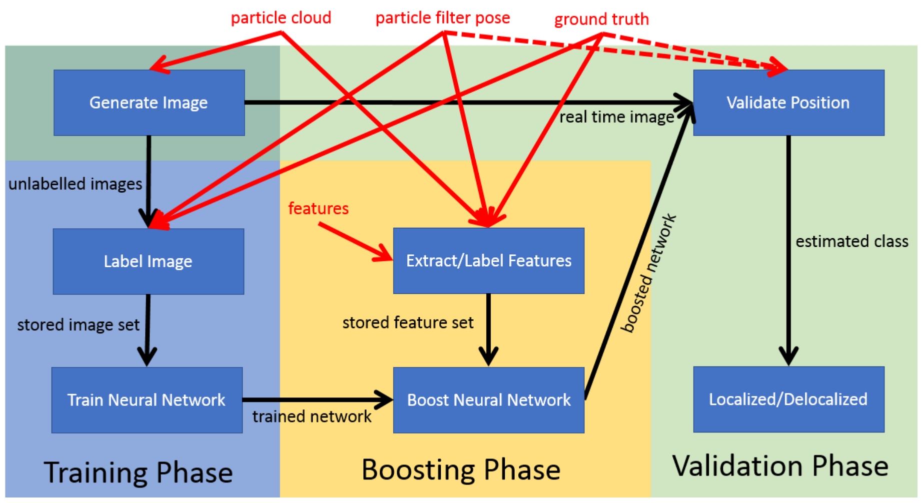


Figure 5.6: Extension of the Required Component Structure

simultaneously collects extracted features while training data is generated. Those features are labelled like the generated image and stored within a *.csv* file. Then the AdaBoost and SVM algorithm are applied on the feature set to find the right feature weights. For comparison within this thesis the boosting algorithms are applied twice. At first only the extracted features are boosted to get a reference for comparison. Then the estimations from the neural network are added as an additional feature. This new feature space is then boosted again and can be cross checked with the reference. Thus it can be evaluated whether the old localization approach can be boosted with the additional neural network feature.

5.5.2 Finding valuable Features

Having prepared the component for boosting one needs to find valuable features that can be used. Usually it is hard to specify which features are important since one does not know if a specific feature holds enough information about the localization state. To overcome this issue one could just add a whole bunch of features and boost them no matter whether they are relevant or not. This might lead to an acceptable result but unnecessary features can increase the training error of boosting algorithms. In this thesis features are examined on their relevance and promising features are extracted for boosting. As discussed in Section 4.7 the relevance of features for this thesis is determined by comparing the Kullback-Leibler divergence of localized and delocalized sets [116]. Therefore the threshold distance d is used to separate the feature set into localized and delocalized sets. By comparing the distributions of the feature sets one can check if valuable information is available that can be used for classification. When applying this method one has to be aware that the threshold distance d was computed with the help of the current localization scoring approach. This method does not work reliable and has some scoring errors. Thus it does not perfectly separate the data set and some information might be misinterpreted, leading to normal distributions that are more similar. Also, boosting algorithms may cope with seemingly irrelevant data and combine them with other features to conduct a better optimization [130]. Thus the selection of features is done generously.

To find meaningful features a total of 33 features are analyzed. These features either correlate to the particle cloud or to comparisons of map and scan points. To gain a rough understanding every feature is shortly presented. The first seven features are gained from the particle cloud and its computed center of gravity. This center is calculated as the mean over all (x, y) positions. The center is then used to calculate different distance measurements. Feature number 8-14 are also created from the particle cloud and technically the same but instead of using the center of gravity a circle is layed around the cloud to to form a hull. The next 15 features are received from the map and laser scan. The last two features use particle filters information. Since a particle cloud may form more than one cluster when the uncertainty of a position increases, the number of clusters found in the particle cloud is also given as feature. When calculating the estimated position of the robot the mean of the biggest cluster is used. Therefore it might also be interesting how the biggest cluster is shaped. This is represented in the last three features. Those features are the variances taken from the covariance matrix of the biggest cluster.

1. Center of gravity **maximum distance:** contains the distance from the particle to the center of gravity which is furthest away.
2. Center of gravity **mean:** holds the mean over all particles. It is calculated as the sum divided by the number of particles.
3. Center of gravity **mean absolute deviation:** this is similar as above but calculates the mean over the same samples again
$$MNAD = \frac{1}{N}\sum_{i=1}^{N}|x_i - \mu| \qquad (5.6)$$
where μ is the mean over distances and N is the number of distances.
4. Center of gravity **median:** sorts the distances of all particles and then selects the one in the middle of the array.
5. Center of gravity **median absolute deviation:** this is a more robust measurement than the simple median. It subtracts the median from every sample and then calculates the median again over the absolute result
$$MDAD = median(|x_i - median(X)|). \qquad (5.7)$$
6. Center of gravity **minimum distance:** contains the distance from the particle to the center of gravity which is the closest.
7. Center of gravity **standard deviation:** is the deviation from the normal distribution and defined as
$$\sigma^2 = \frac{1}{N}\sum_{i}^{N}(x_i - \mu)^2. \qquad (5.8)$$

8. Circle **maximum distance:** is the radius of the circle. This also represents the distance to the particle that is furthest away.
9. Circle **mean:** holds the mean over all particles. It is calculated as above.
10. Circle **mean absolute deviation:** this is also calculated like with the center of gravity in Equation 5.6
11. Circle **median:** is the median of all particle distances to the circle center.
12. Circle **median absolute deviation:** this feature makes the median more robust and is calculated as in Equation 5.7
13. Circle **minimum distance:** the distance between the circle center and its closest particle.
14. Circle **standard deviation:** the standard deviation of the distances between the circle center and the particles.
15. Point **distance:** is defined as the average distance of a scan point to a map line.
16. Point **fitting:** represents in percent how well the points fit to the found map lines.
17. Point **inlier:** states how many points are in front of detected map lines.
18. Point **quality:** represents the quality of single map points that were matched on detected map lines.
19. Raycasting **inlier:** states the percentage of scan points that are within a certain range from the map point.
20. Raycasting **inlier percentage:** is the percentage of scan points that were found before an obstacle occurred in the map.
21. Raycasting **matching percentage:** states how many scan points exactly match to the map.
22. Raycasting **outlier percentage:** given the percentage of scan points that were found behind map obstacles.
23. Raycasting **quality:** states how well the scan matches to the map after applying raycasting to each point. The result of this raycasting step is a number of inliers, outliers and matching points.
24. Angle **inliers:** states how many scan lines are found that are nearer that the found map lines. This is done because the map contains only total blocked points. If a line is detected behind this blocked wall it can be assumed that the scan does not fit correctly. the inlier feature is normalized by the number of found lines.
25. Angle **quality:** states how well a laser scan fits onto a map after rotating the scan point a bit. The quality is measured by searching for Hough lines and then matching them against each other after rotating the scan.

26. Line **angle:** represents the average angle between matching line. Therefore Hough lines are detected in the scan and in the map and then matched against each other. The difference to the angle of the lines is then taken as feature.

27. Line **distance:** is the average distance of scan lines to the nearest map line.

28. Line **fitting:** defines how many scan lines fit to map lines. This is normalized and given in percentage.

29. Line **length**: returns the average line length based on the found lines within the scan points.

30. **Number of clusters**: represents the number of found clusters within the particle cloud. This is used because the particle cloud may form multiple clusters when getting delocalized.

31. main cluster **variance x**: The variance of the main cluster in x direction. The main cluster is the biggest cluster which is also used to estimate the robots position. This value is received from the covariance matrix of the main cluster.

32. main cluster **variance y**: The variance of the main cluster in y direction.

33. main cluster **variance z**: The variance of the main cluster in z direction.

To determine which features are important the robot was randomly driven around in a simulation environment and the features were recorded. After generating roughly 100.000 feature samples every feature is analysed on its own. Therefore the discrete distributions P, Q for both classes are used to calculate the KL divergence. The result of this analysis is shown in Table 5.4.The feature number correlates to the numeration above. The unit column states the unit of measurement, *#bin* is the size of bins that were used for the discrete distribution and $D(P||Q)$ is the Kullback-Leibler divergence. By observing the KLD a general assumption can be made whether a feature is important for boosting or not. It is defined that $D(P||Q) \geq 0$ for all distributions and $D(P||Q) = 0$ if $P = Q$. This means, the higher the result of the KLD the more different the distributions are. When extracting the features that should be used for this thesis it was also taken into account that features that do not seem to correlate with the task may reveal information for boosting algorithms [130]. This is sometimes achieved by combining different features and reweighing them. Thus, the feature selection was done generously and only features were excluded which seem to hold very few no information.

The extracted features which are used for boosting are highlighted in Table 5.4. The number of bins depends on the measurement unit and on the discrete distributions. When using percent as measurement KLD uses 100 bins, one for each percent. For degree the number of bins is 360, one for every degree. For meters the size of a bin is $1cm$. Therefore a feature that contains its samples between a range of $0.0 - 1.0$ meters has 100 bins. The selection of features is based on the discrete distribution of the samples. Those distributions are

Feature No	**unit**	*#bin*	$D(P\|\|Q)$
1	meters	350	0.265
2	meters	150	0.313
3	meters	100	0.314
4	meters	150	0.296
5	meters	50	0.321
6	meters	100	0.153
7	meters	100	0.293
8	meters	300	0.286
9	meters	200	0.294
10	meters	50	0.281
11	meters	200	0.279
12	meters	50	0.260
13	meters	150	0.159
14	meters	50	0.285
15	meters	400	0.086
16	%	100	0.157
17	%	100	0.118
18	%	100	0.035
19	%	100	0.183
20	%	100	0.046
21	%	100	0.166
22	%	100	0.045
23	%	100	0.098
24	%	100	0.315
25	%	100	0.035
26	degree	360	0.105
27	meters	400	0.042
28	%	100	0.033
29	meters	250	0.086
30	amount	5	0.102
31	meters	100	0.403
32	meters	100	0.050
33	meters	150	0.147

Table 5.4: Kullback-Leibler divergence for all features. Gray background indicates an extracted feature that is used for boosting, selected by $D(P||Q) \geq 0.08$

used for determining the Kullback-Leibler divergence which is again used for comparing the localized and delocalized feature sets. From the definition of the KLD one knows that the higher the value the less similarities are shared between both distributions. Features that seem to hold not much information are removed. Since no feature is completely equal one could argue to consider all found features but some features are nearly identical and might only increase the boosting error. To extract relevant features the threshold for selecting a feature was set to $D(P||Q) \geq 0.08$. This value was chosen so that at least 75% of the detected features are used for boosting. This threshold results in 26 features that are used for boosting.

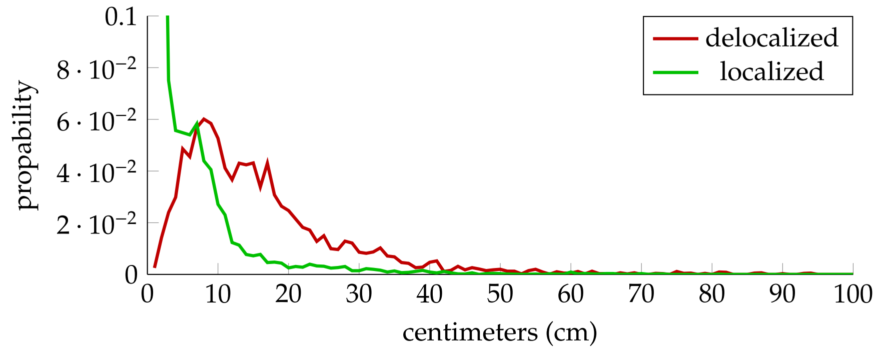


Figure 5.7: Example of good feature separation using the existing scoring algorithm and threshold distance 1.25 on feature number 31

An example for a good separable feature is shown in Figure 5.7. When looking at this extracted feature one can see that the discrete distribution does overlap but the probabilities are well separated. Also one has to remember that the used scoring algorithm is not optimized and might not lead to an optimal result. Figure 5.8 shows an example of a bad separated feature so that those results can be compared. Both discrete distributions have their most parts near to 100%. The main difference is a small number of delocalized samples around 0% but this also is not significantly high. Thus this example does not indicate a significant difference. Due to the similarity of the distributions this feature will most likely not reveal any information about the localization state.

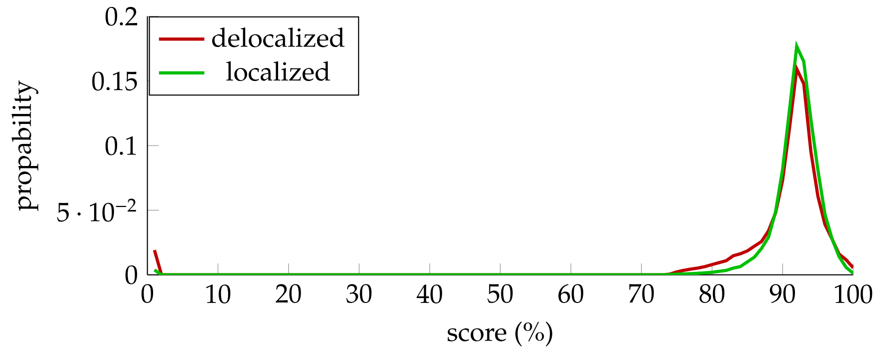


Figure 5.8: Example of bad feature separation using the existing scoring algorithm and threshold distance 1.25 on feature number 18

By analysing some features that were identified during the implementation phase one can extract features that can be used for boosting a neural network. This thesis extracted 26 out of 33 features and uses them as input for a reference boosting. After that the outcome of a trained neural network can be added as additional feature and boosted. This result can then be compared to the reference outcome. Since boosting algorithms might find information which is not revealed by comparison of normal distributions the selection of features was done generously and thus also seemingly unimportant features were selected.

Chapter 6

Evaluation

This chapter focuses on a sound evaluation of the previously presented concept. Therefore three different network types are analysed to extract a single feature that estimates the robot localization state. The first network type which is evaluated is a Convolutional Neural Network. This uses feature layers to detect patterns in the shape of the particle cloud. Then it is evaluated whether temporal information can be extracted from the particle filter to estimate the robots localization state by applying Long-Short Term Memory networks. The last network type is a combination of the two previously used types and is called Long-term Recurrent Convolutional Network. The proposed neural network structures are trained to determine the needed complexity of the networks. Therefore, three networks are designed for each network type that have different network structures with different complexity. By evaluating the different structures one can determine how complex a network has to be designed to fulfill its task of estimating the robots localization state. To evaluate a network different validation sets are used to determine the classification accuracy. As stated in Section 4, the sets are generated by letting a robot drive randomly through an given environment. The first set that is evaluated was used to train the neural network. This shows information on the overall network performance and how good the neural network was trained based on the given data. Then a second set is used to validate the network structure. This set was recorded on the same environment as the neural network was trained on. The last set is also a validation set which was recorded in a completely different environment. Using these three sets one can answer three main questions that occur for the trained networks

1. How well does the network structure perform on the given training set? Does it find valuable information that can be used to train the network?
2. Does the neural network indeed train the robots localization state or does it extract information that is irrelevant for this task? For instance does it train the driven path of the robot?
3. Is the shape of the particle cloud influenced by the shape of the environment? Is the training needed to be done for every single environment?

By answering these questions one can also answer two out of three questions that were presented in the problem definition (see Section 1.4). The third question from Section 1.4 concerning feature boosting can be answered by using the best network structure that was found as additional feature.

To further improve the localization state estimation this thesis combines different features that hold information about the robots localization state. This is done because it is assumed that a combination of features perform better than a single extracted feature like the outcome of the neural network. Therefore, various features that are extracted from the particle filter, the environment, the 2D laser scan and the robots pose are used as described in Section 5.5.2. Those features are then boosted by using AdaBoost and Support Vector Machines. This is done by taking all found features and training them once with the neural network outcome as additional feature and once without. This allows us to determine whether the neural network outcome indeed imrpoves the localization state estimation.

6.1 Evaluating Convolutional Neural Networks

The first network type that is evaluated are Convolutional Neural Networks. Those network types use convolutional layers that hold different feature maps and aim to detect trained features on an image. In this section three CNNs with increasing complexity are analysed as presented in Section 4.

Simple Convolutional Neural Network

The first Convolutional Neural Network only consists of a simple structure. Only one convolutional layer is used that is concatenated with a pooling layer and a fully connected layer. Figure 6.1 shows the loss while training the neural network. One can see that at the beginning the loss was a bit higher and while training it did not decrease very much. For training a total of 100.000 iterations were done to see if the neural network can further decrease its loss.

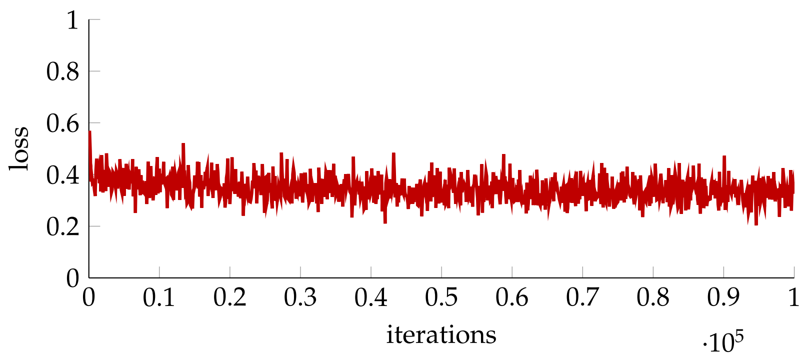


Figure 6.1: The loss while training a simple Convolutional Neural Network structure

Having trained the simple CNN structure it was evaluated by measuring the accuracy of the three validation sets. Table 6.1 shows the result of the evaluation. The *Environment* column indicates the validation set that was evaluated.

Environment	tp	tn	fp	fn	accuracy
training set	64 411	62 360	12 693	10 588	84.52 %
same environment	59 192	60 229	14 770	15 807	79.62 %
new environment	48 386	53 059	21 940	26 613	67.63 %

Table 6.1: Result of the simple convolutional neural network

The next four columns show the number of correct and incorrect classified samples where positive samples indicate that they were classified as localized and negative samples were classified as delocalized. The accuracy was calculated as discussed in Section 4.6. By looking at the accuracy of the validation sets one can see that the network scored a quite good training accuracy of 84.52%. This shows that relevant information could be extracted by the convolutional neural network since the ratio of localized and delocalized samples is about 0.5. Validating the neural network with samples that were randomly recorded within the same environment yields an accuracy of 79.62%. By comparing this result with the training set one can see that it is roughly 5% lower than the training accuracy. This might be due to the fact that the validation set contains samples that were not used for training. The accuracy of the second set is also high and shows that the robots localization state can still be estimated efficiently even if it was driven on different routes within the same environment. This leads to the conclusion that this CNN structure did indeed learn the localization state of the robot and not the path it has driven while collecting the data. The third set in Table 6.1 was recorded within a completely different environment and shows that the accuracy decreases roughly to 67%. From this decrease one can conclude that the new environment does not form completely the same shape of particles leading to a lower classification accuracy. Although the accuracy is still higher than the sample ratio, it shows that not all shapes that were trained within an environment are applicable to a different environment. This might be due to the reason that the shape of the particle cloud depends on the structure of the environment. If the new environment has many different scan- and environment shapes this might also lead to a different distribution of the particle cloud. There seem to be still some general particle formations that occur in both environments but a lot of information is lost due to the different shapes.

Mid-Complex Convolutional Neural Network

This next network structure is more complex than the previous Convolutional Neural Network. It uses multiple convolutional layers and also more fully connected layers. The exact structure of the network can be seen in Section 4. Figure 6.2 shows the loss while training the neural network. One can see that the loss decreases over a period of roughly 60.000 iterations. The total number of iterations that was used for training was 100.000.

Table 6.2 shows the result of the mid-complex network structure. When looking at the accuracy of the training set one sees an astonishing accuracy of over 96%. This shows that the training samples were nearly perfectly separated into the correct classes. The result on the validation set that was recorded in the same environment but different paths are used. It shows that the accuracy

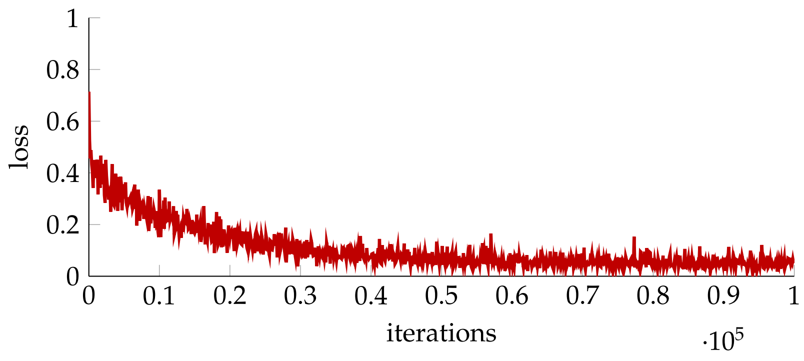


Figure 6.2: The loss while training a mid-complex Convolutional Neural Network structure

Environment	tp	tn	fp	fn	accuracy
training set	72 050	72 656	2 343	2 949	96.47 %
same environment	59 655	65 220	9 779	15 344	83.25 %
new environment	48 284	51 113	23 886	26 715	66.27 %

Table 6.2: Result of the mid-complex Convolutional Neural network

decreases by more than 13%, indicating a network that was slightly overfitted. Although the accuracy of the second validation set is not as high as for the training set, it is still a very good result and shows by comparing it to the training set that not the path was trained but the localization state of the robot. The accuracy of the third validation set again decreases by roughly 17% to 66.28%. Since this is the set that was recorded in a new environment one can determine that the network structure was only trained for one environment and does only hold general information for other environments but the overall accuracy is still ok.

Complex Convolutional Neural Network

The last convolutional network structure is designed to be very complex. This is done to see whether a complex network structure is more efficient than simpler ones. It uses many convolutional layers and is leaned on the network structure of GoogLenet [144]. The exact structure of the network can be seen in Section 4. Figure 6.3 shows the loss while training the neural network. One can see that the loss keeps decreasing to the end but at roughly 85.000 iterations the decrease in loss is not so significant anymore. The total number of iterations that was used for training was 100.000.

Table 6.3 shows the evaluation result of the complex convolutional network structure. Compared to the training accuracy of the mid-complex CNN it did not increase. This leads to the assumption that this network structure is too complex to help to solve the problem of estimating a robots localization state. The complex network structure still managed to yield a high accuracy but since a less complex network scored a higher accuracy it is not necessarily useful to

Environment	tp	tn	fp	fn	accuracy
training set	66 342	70 490	4 509	8 657	91.22 %
same environment	56 584	65 482	9 517	18 415	81.38 %
new environment	39 337	53 749	21 250	35 662	62.06 %

Table 6.3: Result of the complex Convolutional Neural network

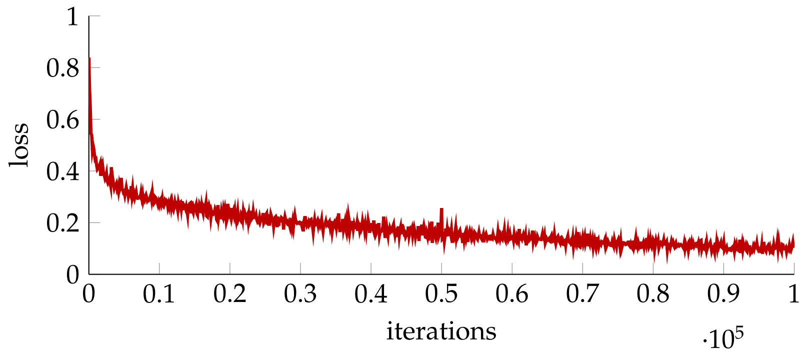


Figure 6.3: The loss while training a complex Convolutional Neural Network structure

construct such complex systems. This is due to the fact that the image samples that were trained do not hold a lot of information compared to full RGB images of e.g. humans. Only a small amount of information is given by the particle cloud. More complex images might need many more layers because they hold various features that have to be detected. This is not the case for this thesis and thus simpler convolutional networks seem to perform the best. The accuracy of the validation set that was recorded in the same environment but not used for training shows that again the network is a bit overfitted but managed to train the correct information such that the localization state of the robot is estimated. The validation within a different environment shows that it does not hold a lot general information that can be applied to different environments. its accuracy of roughly 62% shows that only little information is detected within a new environment.

6.2 Evaluating Long-Short Term Memory Networks

This section focusses on evaluating the recurrent LSTM networks which were trained to find out whether the temporal development of a particle cloud reveals information about the robots localization state. To find a proper network complexity three different network structures with increasing complexity are evaluated. To get an accuracy that can be evaluated efficiently the localized set and delocalized set was limited to about 75.000 each such that a sample distribution of roughly 50:50 is created. This allows to make more precise statements since both sets contain the same number of samples.

Simple Long-Short Term Memory Network

The first structure which is evaluated is the simplest structure. As presented in Section 4 it consists of one single LSTM layer and two fully connected layers. By looking at the loss function in Figure 6.4 one can see that the loss could only be minimized within the first 20.000 iterations. Nevertheless, a total of 80.000 iterations were done during training with the hope to further minimize the loss.

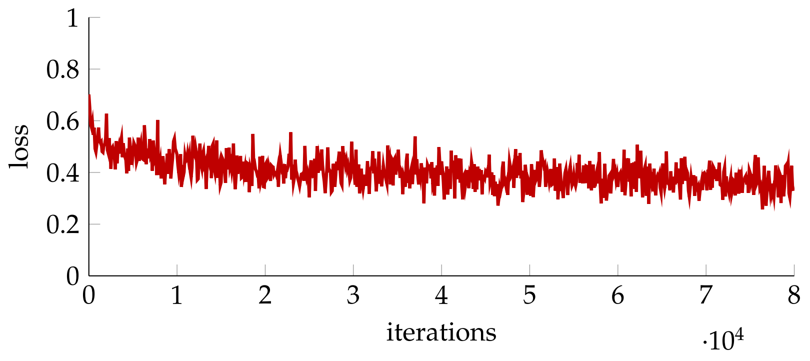


Figure 6.4: The loss while training a simple LSTM Network structure

Table 6.4 shows the results for a simple LSTM. The training accuracy is with 84.42% lower than the accuracy of a convolutional neural network but the validation accuracy on the same map can easily compete with them. This indicates a neural network that is not overfitted at all and extracted some useful temporal information for the recurrent network. The validation accuracy for the different environment again is significantly lower but not completely random. This leads to the assumption that some core information was extracted while training a recurrent network. As for CNNs it seems like the particle cloud has different shapes within different environments. Thus some information is only learned within the environment that was used for training. So far the usage of sequential neural networks is not an improvement compared to CNNs.

Environment	**tp**	**tn**	**fp**	**fn**	**accuracy**
training set	60 659	65 964	9 035	14 340	84.42 %
same environment	57 476	65 769	9 230	17 523	82.16 %
new environment	53 043	48 640	26 359	21 956	67.79 %

Table 6.4: Result of the simple LSTM network

Mid-complex Long-Short Term Memory Network

The next structure which is evaluated for training temporal development of the particle cloud is a bit more complex. It uses more LSTM layers that are connected with fully connected layers at the end. For the exact structure of the network, see Section 4. The loss for the mid-complex LSTM structure is shown in Figure 6.5. It continuously decreases over time for about 80.000

iterations. The complete training of the network structure was done in 100.000 iterations. Table 6.5 shows the outcome of the mid-complex LSTM structure.

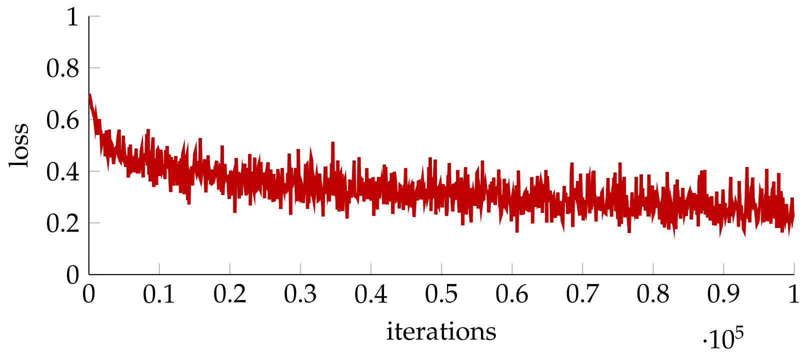


Figure 6.5: The loss while training a mid-complex LSTM Network structure

Environment	**tp**	**tn**	**fp**	**fn**	**accuracy**
training set	69 025	61 403	13 596	5 974	86.95 %
same environment	64 286	58 690	16 309	10 713	81.99 %
new environment	59 627	41 415	33 584	15 372	67.36 %

Table 6.5: Result of the mid-complex LSTM network

It can be seen that the network performs well and reaches a training accuracy of 86.95%. Also the validation set that was recorded in the same environment reaches a high accuracy of roughly 82%. This indicates a network that does not overfit. Although the accuracy in another environment decreases for over 14% it still detects a lot of useful information. By comparing the number of false positives and false negatives one can see that the detection of delocalized states was not trained as well as the detection of localized states. Delocalized states were misclassified twice as often as localized states. Depending on the requirements of the network this can be good or bad. When the user likes to detect all delocalized states it is better to reduce the false positives. On the other side the user could like to detect no false negatives so that the robot does not unnecessarily believe to be delocalized. The optimal solution is to reduce both, false positives and false negatives, but if this is not possible or the user only likes to optimize one set this can be also done by adapting the network structure. Also the number of classified training samples play a role in this case. If the training set consists mainly of localized samples the network tends to perform better in detecting localized states. So by varying the network structure and the ratio of training sample states on can shape the network for the users requirements.

Complex Long-Short Term Memory Network

The last LSTM structure that is evaluated is the most complex one. As described in Section 4 it consists of multiple LSTM layers followed by multiple fully connected layers. To find out how well the complexity of this network fits the problem for this thesis, it is compared to the other network types. Figure 6.6 shows the loss function that was recorded during training. It can be seen that the loss was continuously decreased for about 60.000 iterations. The complete training phase was done in 100.000 iterations.

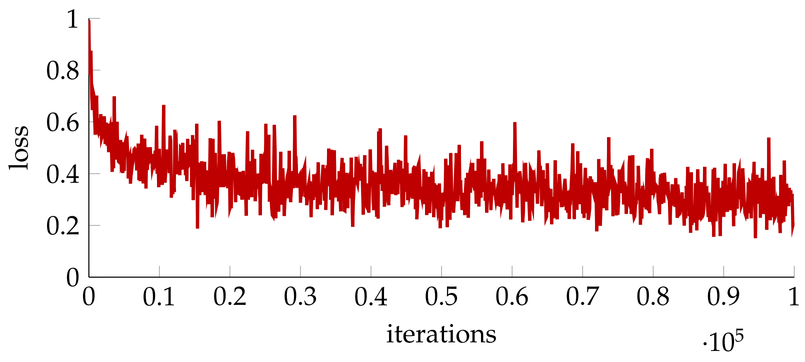


Figure 6.6: The loss while training a complex LSTM Network structure

Environment	**tp**	**tn**	**fp**	**fn**	**accuracy**
training set	64 318	66 816	8 183	10 681	87.42 %
same environment	59 549	65 093	9 906	15 450	83.10 %
new environment	47 527	51 972	23 027	27 472	66.33 %

Table 6.6: Result of the complex LSTM network

Table 6.6 shows the validation results on the complex LSTM structure using three different validation sets. The training itself yields an accuracy of 87.42% and is an acceptable result. Also the validation set that was recorded in the same environment scores a high accuracy of more than 83%. Although it seems to work well when applied in the same environment, it looses many percent when it is tested for an new environment. This problem was also detected on the previous network structures. Thus one can also assume that the complex LSTM network needs to be trained on its own for every environment.

6.3 Evaluating Long-term Recurrent Convolutional Networks

The last network type that is evaluated in this thesis is the Long-term Recurrent Convolutional Network. This combination of convolutional and LSTM layers yields to take advantage from both layer types to further improve the localization score. To find an acceptable network structure that can be used for training three different networks with increasing complexity are trained.

Simple Long-term Recurrent Convolutional Network

The first LRCN network structure that is evaluated is a simple one. It consists of only two convolutional layers that are concatenated with a LSTM layer and

a fully connected layer. The exact structure is presented in Section 4. Figure 6.7 shows the loss function during training. It can be seen that the loss rapidly increases at about iteration 22.000. This is an increase due to the stochastic gradient descent. It moved out of a local minima and tried to find a better solution. One can also see that indeed the loss was minimized a little bit after the local minimum was escaped.
The results on the trained network are shown in Table 6.7. It can be seen that the

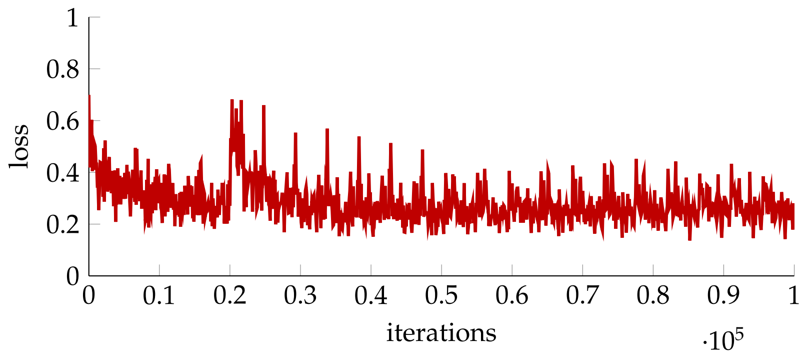


Figure 6.7: The loss while training a simple LRCN structure

Environment	tp	tn	fp	fn	accuracy
training set	66 792	66 300	8 699	8 207	88.73 %
same environment	60 400	63 417	11 582	14 599	82.55 %
new environment	55 034	49 284	25 715	19 965	69.55 %

Table 6.7: Result of the simple LRCN

combination of a CNN and LSTM network already yields a good performance when trained on a small network structure. It performs better than the other simple network structures on their own and scores an training accuracy of 88.73%. Also the validation accuracy on the same environment is with roughly 82% close to its training outcome and indicates a well fitted network. It can also be seen that the ratio of correct classified localizations and delocalizations is nearly equal. This means that both classes were trained to be detected correctly.

Mid-complex Long-term Recurrent Convolutional Network

To observe how complex the network structure can be to solve the task of estimating a robots localization state, a more complex LRCN is constructed. It consists of multiple convolutional layers, two LSTM layers and two fully connected layers as described in Section 4. The loss function illustrated in Figure 6.8 shows that the main decrease of loss was reached within the first 25.000 iterations. While training was done for a total of 100.000 iterations the loss could not be decreased any further.
In Table 6.8 the results of the trained neural network are shown. While the simple LRCN structure scored an accuracy of over 88%, this structure only reached an accuracy of 88.13%. Also the other two validation accuracies are

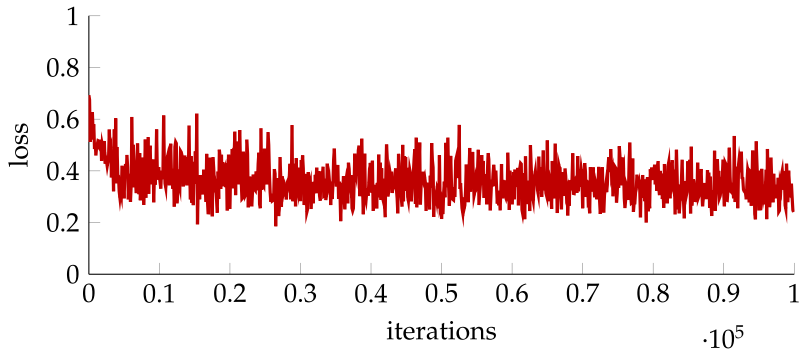


Figure 6.8: The loss while training a mid-complex LRCN structure

about 2% lower than the one in the simpler structure. Another noticeable point is the difference of false positives and false negatives. It can be seen that localized states were misclassified twice as often as delocalized states. This means the network was shaped to better detect delocalized states.

Environment	**tp**	**tn**	**fp**	**fn**	**accuracy**
training set	60 740	68 431	6 568	14 259	86.12 %
same environment	55 817	66 195	8 804	19 182	81.34 %
new environment	48 807	51 973	23 026	26 192	67.19 %

Table 6.8: Result of the mid-complex LRCN

Complex Long-term Recurrent Convolutional Network

The last network structure that is evaluated is a complex LRCN structure. It is designed to see whether this complex structure is capable of extracting relevant information or not. Also it is used to get a rough understanding of the needed complexity for estimating a robots localization state. During training the loss was recorded and is visualized in Figure 6.9. It can be seen that the loss converged and did not decrease anymore starting at about iteration 50.000. The complete training was continued for a total of 100.000 iterations.

Environment	**tp**	**tn**	**fp**	**fn**	**accuracy**
training set	55 901	73 236	1 763	19 098	86.09 %
same environment	48 566	70 380	4 619	26 433	79.30 %
new environment	26 610	61 705	13 294	48 389	58.88 %

Table 6.9: Result of the complex LRCN

Table 6.9 shows the result of the complex LRCN structure. Here it can also be seen that the training accuracy is lower than the accuracy of the simple LRCN structure. Thus one can assume that only a simple network structure is needed when a combination of CNN and LSTM is used to detect the localization state of the robot. Another interesting result is the ratio of false positives and false

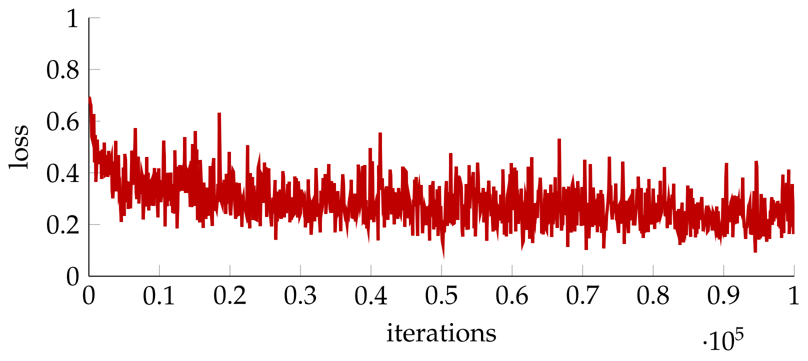


Figure 6.9: The loss while training a complex LRCN structure

negatives on the training set. It can be seen that the delocalized states were trained nearly perfectly while the classification error of false negatives is about ten times higher. Another problem that can be seen from the results in Table 6.9 is that the new environment scores only a low accuracy. This also indicates that the network might be overfitted and only trained on the structure of one environment.

6.4 Boosting Features

Having evaluated different network types and structures one can now determine the best performing network and use it for boosting. This section handles the selection of the best performing neural network and evaluates two different boosting algorithms. At first all found features from Section 5.5.2 are boosted with AdaBoost to train a base that can be used for comparison. Then the neural network feature is added as additional feature and boosted again with AdaBoost. The same two steps are then repeated on a Support Vector Machine.

6.4.1 Selecting the best performing Neural Network

As evaluated previously in this chapter there are nine trained networks which are eligible to be used as additional feature for boosting. To find the best network structure one has to define the situation in which the boosting algorithm should be used. To be more specific one has to state whether the boosted features are trained for a single environment or if the resulting feature weights should be generalized and usable for different environments. For the first case one can select the best performing network feature according to its accuracy on the training set and in the validation set that was recorded in the same environment. If it should be more generalized one has to determine the best network structure based on the validation accuracy with different environments. Since the evaluation results of the neural networks show that they perform the best when they are trained for one environment, this thesis focuses on the first possibility where it is assumed that training a network and boosting is done for every single environment.

By comparing the training accuracy and validation accuracy of the same environment one can easily determine the best performing network structure that can be used for estimating a robots localization state. Although the mid-complex CNN seems to be overfitted it still performs the best on the validation set. Therefore this neural network structure will also be used as additional feature for boosting. In practice other aspects might be important. For example for some company it might be important to minimize the number of false delocalizations while for another company a reduced number of false localizations might be in focus. Depending on the application and needs one can choose different network structures that are suitable.

6.4.2 Boosting with AdaBoost

This section applies AdaBoost on the features that were extracted and evaluates the results. First AdaBoost is used to train all features except of the neural network feature to calculate a base line for comparison. Then the neural network feature is added and boosted again. When training features with AdaBoost one needs to set the number of weak classifiers that may be used. Weak classifiers are the extracted features that are used for training and various combinations of them which are generated automatically by the OpenCV library. In Section 5.5.2 26 features were extracted and are used as weak classifiers. One could now set the number of weak classifiers to 100 and the OpenCV implementation of Adaboost automatically creates 74 new weak classifiers that are a combination of the given features. This is done to further improve the performance of the boosting algorithm. To find the best suiting number of weak classifiers one has to train AdaBoost on a different number of classifiers and then select the one with the highest test accuracy.

Evaluation of AdaBoost without Neural Network Feature

This section evaluates the the AdaBoost algorithm that is applied to the extracted features from Section 5.5.2 without the neural network outcome which was evaluated above. AdaBoost was trained with a different number of classifiers between 26 and 300 to find a proper number of weak classifiers that can be used. Figure 6.10 shows the training and testing accuracy on a different number of weak classifiers. The highest test accuracy before converging takes place is scored with 276 weak classifiers.
Table 6.10 shows the resulting accuracy that was reached with AdaBoost and 276 weak classifiers. Positive samples are samples that were classified as localized and negative samples are classified as delocalized. Overall it can be seen that the training and test accuracy are both quite high and do not differ a lot. This means that the trained weights were fitted perfectly so that no overfitting takes place.Also the validation in a different environment scores an high accuracy of 81.83%.

Evaluation of AdaBoost with Neural Network Feature

After training the baseline for comparison the output of the selected neural network is now added as additional feature. As above one has to determine the number of weak classifiers that should be used for training. Figure 6.11 illustrates the training and test accuracy of the AdaBoost algorithm on a different

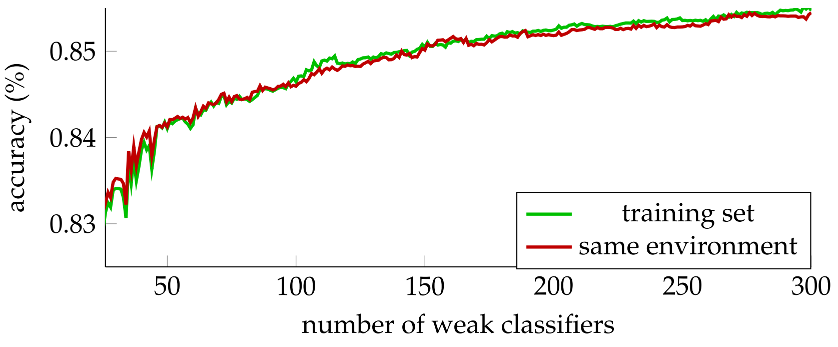


Figure 6.10: Training and test accuracy of AdaBoost without neural network feature on a different number of weak classifiers

set	**tp**	**tn**	**fp**	**fn**	**accuracy**
training set	64 258	63 883	11 106	10 731	85.45 %
same environment	64 423	63 726	11 263	10 566	85.44 %
new environment	48 935	61 540	5 898	18 626	81.83 %

Table 6.10: Boosting result using AdaBoost without the neural network feature and 276 weak classifiers

number of weak classifiers. The test accuracy starts to converge at roughly 200 iterations and reaches it maximum with a total of 229 weak classifiers.

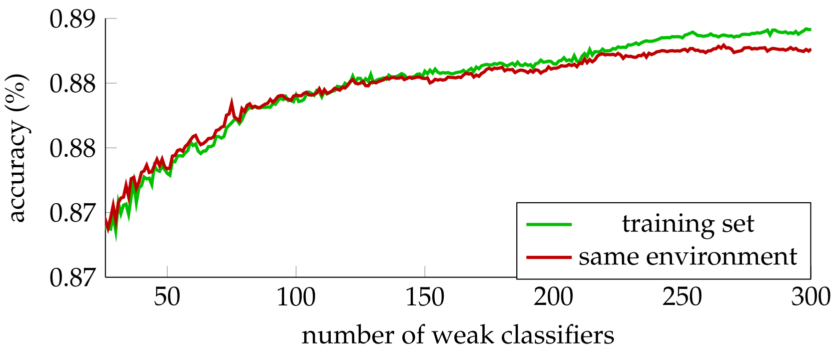


Figure 6.11: Training and test accuracy of AdaBoost with neural network feature on a different number of weak classifiers

Using this number of classifiers a test accuracy of 88.21% is reached as shown in Table 6.11. This table also shows the improvement of accuracy when the neural network output is used as additional feature. The difference of accuracy compared to AdaBoost without the neural network feature (Table 6.10) in the same environment is an increase of 2.85% in training and 2.77% in testing. When validating the boosting approach in a different environment the accuracy is still high but did not improve very much compared to boosting without neural network feature. Since the network feature does not perform well in other environments one can not expect better results than in boosting without the neural network output. This perfectly proves that the use of the neural network output as feature increases the boosting performance by roughly 3% when applied in the same environment. Also the number of weak classifiers can be reduced by 47 which increases the training performance.

set	tp	tn	fp	fn	accuracy
training set	33 633	32 563	4 886	3 888	88.30 %
same environment	33 675	32 453	5 097	3 745	88.21 %
new environment	51 550	59 988	7 450	12 011	82.62 %

Table 6.11: Boosting result using AdaBoost with the neural network feature and 229 weak classifiers

Due to the accuracy improvement and the overall high accuracy that was reached when boosting localization features with AdaBoost one can assume that this method works well on estimating the localization state of the robot.

6.4.3 Boosting with Support Vector Machines

Another method for classifying the localization state of a robot with various features as input are Support Vector Machines. SVMs are used in this thesis to find an optimal separation hyperplane that separates localized and delocalized states. As before with AdaBoost two different feature sets are used for training. At first the features as presented in Section 5.5.2 are used without the selected neural network output. This is then compared with a SVM that trains all the features including the neural network output.

Evaluation of SVM without Neural Network Feature

The first approach for finding an optimal separation hyperplane is done without the neural network feature and conducted with the library offered by OpenCV. This library offers an automatic trainer that automatically searches for the optimal separation hyperplane and also tries to detect the perfect Kernel for it. Since this SVM algorithm is well tested by OpenCV and its automatic mode decreases the computational effort this method was used for training.

set	tp	tn	fp	fn	accuracy
training set	23 027	23 099	14 291	14 572	61.51 %
same environment	23 158	23 289	14 320	14 222	61.94 %
new environment	21 735	21 970	15 639	15 864	58.11 %

Table 6.12: Boosting result using SVM without the neural network feature

Table 6.12 shows the outcome of the boosting with a SVM. It appears that this method is not applicable for this task since the testing accuracy in the same environment is with 61.94% roughly 20% lower than it is with AdaBoost. The accuracy in a different environment decreases further to 58.11%. However, since this is the base line for comparison the neural network feature is still added and evaluated with a Support Vector Machine to see if it can be improved too.

Evaluation of SVM with Neural Network Feature

To further improve the support vector machine above the neural network outcome is added as additional feature. For this case also the automatic trainer was used which searches for an optimal kernel to separate the training set.

set	tp	tn	fp	fn	accuracy
training set	15 796	31 499	5 950	21 725	63.09 %
same environment	15 943	31 727	5 823	21 477	63.59 %
new environment	22 017	21 873	15 736	15 582	58.35 %

Table 6.13: Boosting result using SVM with the neural network feature

By adding the neural network feature one can improve the accuracy of a SVM in the same environment by about 2% as shown in Table 6.13. Although this is an improvement compared to the SVM without the network feature it is still worse than the boosting approach with AdaBoost. Due to the low accuracy it can be assumed that a SVM is not suitable for solving the task of estimating a robots localization state.

6.5 Comparing Old Scoring Approach to AdaBoost

To determine if the boosting approach is an improvement compared to the old localization scoring approach, both are now shortly compared. To compare the old localization scoring approach one needs to generate samples which can be classified into two classes. In Section 5.3 the optimal threshold distance for separating two classes was already found. Since the old scoring approach returns an the quality of the robots localization in percent one needs to find a separation line which can be used to classify samples into a localized and delocalized class. Figure 5.3 shows how the two classes were separated into two discrete distributions. The optimal separation hyperplane for separating the two distributions in percent is at a score of 45%. This means that samples with a score $< 45\%$ are classified as delocalized and the others are classified as localized. By classifying the old localization scoring approach one can now collect scores and compare them to the actual localization state as discussed in Section 5.4.1. To compare the boosting approach and the old scoring method a robot is randomly driven in the training environment which was used to train the neural networks. While driving through the environment the robot classifies the old localization score and compares it to the actual label, resulting in a set of true positives, true negative, false positives and false negatives. The

Method	tp	tn	fp	fn	accuracy
AdaBoost	33 675	32 453	5 097	3 745	88.21 %
Old Scoring Approach	29 320	21 344	16 100	8 140	67.61 %

Table 6.14: Comparing the old localization scoring approachand the best performing boosting approach, AdaBoost including the neural network output in the trained environment

results for the training environment can be seen in Table 6.14. One can see that AdaBoost outperforms the old localization scoring approach by more then 20%. Thus one can conclude that the boosting approach with neural networks is indeed an improvement and can be used to score the localization state of a robot when applied in the trained environment. Table 6.15 shows the results in a new

Method	**tp**	**tn**	**fp**	**fn**	**accuracy**
AdaBoost	51 550	59 988	7 450	12 011	82.62 %
Old Scoring Approach	28 925	22 003	15 441	8 535	68.16 %

Table 6.15: Comparing the old localization scoring approachand the best performing boosting approach, AdaBoost including the neural network output in the trained environment

environment. It can be seen that no approach increases its accuracy in a different environment. While AdaBoost scores an accuracy of 82.62%, the old scoring algorithm scores 68.16%. This much lower than the boosting approach. This comparison shows that boosting also outperforms the old scoring approach in new environments.

Chapter 7

Conclusion

In this thesis an implementation of particle filter-based robot localization was analyzed for finding features of different kind which can be used to score the accuracy of robot localization. Those features were extracted by applying different approaches like statistical methods. Also features that already existed in a previously existing localization scoring approach were selected as possible feature. To determine the importance of a feature many samples were recorded and separated into a localized and delocalized class. Then the distribution of the previous localization score of these two classes was determined. Those two distributions were calculated for every feature and used to rank their importance by calculating the Kullback-Leibler divergence. Promising features were then selected to be trained on two machine learning approaches: Boosting with AdaBoost and applying a Support Vector Machine (SVM).

An additional feature was extracted by training neural networks to detect the localization state of a robot based on the shape of the particle cloud. The assumption is that the shape of the particle cloud and its temporal development bear information about the localization quality. To find the best fitting neural network three different types with increasing complexity were evaluated: a convolutional neural network (CNN), a long-short term memory network (LSTM) and a long-term recurrent convolutional network (LRCN). The first network type was used to determine whether the particle cloud holds relevant information about the localization state of a robot. This was proven to be true when the best performing CNN reached a training accuracy of 96.47% and a validation accuracy in the same environment of roughly 83%. Although this is a promising result it also showed that a CNN needs to be trained for every environment since the validation accuracy in a different environment decreases to under 70%. Training a CNN works already quite well for simple network structures. Since a binary image representing the projected sample was used for training the particle cloud only few convolutional layers were needed to extract relevant information. More complex network structures would search for complex patterns that can be found in other fields of application but not in our task. The LSTM network was used to find out if the temporal transformation of a particle cloud reveals information about the localization state of a robot. The evaluation shows that indeed a lot of information can be found by training temporal transformations. The best performing LSTM structure scored a training accuracy of 87.42% and a validation accuracy of 83.10% in the same environment. As with convolutional networks a LSTM network needs to be trained individually for every environment since the validation accuracy in different environments decreases dramatically. The last network type is the co-called long-term recurrent convolutional network (LRCN) which is a combination of CNN and LSTM. This network type was used to see whether the advantages of the different trained networks above can be combined into one single network. Although the simplest LRCN structure which was tested performed best, it could not increase the accuracy of the trained networks above. Also it could not get rid of the issue that the network needs to be trained individually for every single environment. Overall, the best performing network was a CNN with little complexity. This network was also further used as additional input for the two machine learning approaches, AdaBoost and SVM.

To find out if various features can be combined to improve the localization accuracy two different machine learning approaches were applied. Both approaches used the extracted features to train a base result for comparison. Then the output of the best performing neural network was added as additional feature and the method was trained again. The first approach that was evaluated is adaptive boosting (AdaBoost) and showed that the training accuracy can be improved from 85.45% without neural network feature to 88.30% with network feature. From scientific view this is a top result but for practical application this accuracy might still be too low. The second machine learning approach was a support vector machine. The SVM could not keep up with the results of AdaBoost and reached a training accuracy of 61.51% without the neural network as additional feature. Including the neural network output the SVM did improve its training accuracy but with 63.09% it is still a lot lower than the results with AdaBoost. In conclusio, the evaluation of different settings showed that it is possible to use information about the particle cloud for scoring the localization quality.

Chapter 8

Future Work

Having set a corner stone for estimating the robots localization state by using machine learning approaches on information provided by particle filters, further research needs to be done to improve the results of this thesis. Therefore different topics are presented which can be studied in the future.

The first topic is the triggering of errors in the particle filter. In this thesis the error of the particle filter was simulated in the laser scan by adding dynamic obstacles into the environment such that the laser model does not exactly match the environment. As mentioned in this thesis it is also possible to force errors in the particle filter by adding errors in odometry. This could be done to see if a combination of both errors lead to a better result that can be applied in practice.

The next topic is to improve the particle filters performance for better localization accuracy. Therefore different publications already exist which aim to make the particle filter more robust and more accurate. An example for this would be to apply adaptive particle filtering [50] or to detect dynamic obstacles as presented in [115] and then exclude them from the particle filter algorithm.

Another topic which can be done is to improve the neural network structures for extracting information from the particle cloud. This thesis focused on determining whether a neural network can be used to extract relevant information and if so, which network types can be applied. It was not searched for the optimal neural network and thus the accuracy of a neural network may be further improved by determining an optimal network structure.

This thesis only considered the position of particles for training neural networks. Another step might be to insert more information of the particle filter into neural networks such as the orientation of the robot. This could be done by using a color channel which indicates the robots orientation.

Also real world tests should be conducted to see how well the localization approach performs on a real robot in a real environment. Although the simulation considered several aspects of a real robot, some information might still be missing to apply the presented approach in a real world environment.

Abbreviations

Abbreviation	**Definition**
OS	Operating System
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
HMM	Hidden Markov Model
SMC	Sequential Monte Carlo
PF	Particle Filter
NN	Neural Network
SVM	Support Vector Machine
PCA	Principal Component Analysis
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long-Short Term Memory
LRCN	Long-Short Recurrent Convolutional Network
BPTT	Backpropagation Through Time
AdaBoost	Adaptive Boost
SVM	Support Vector Machine
SGD	Stochastic Gradient Descent
EKF	Extended Kalman Filter
FALKO	Fast Adaptive Laser Keypoint Orientation-invariant
OC	Orthogonal Corner
KLD	Kullback-Leibler Divergence
GPS	Global Positioning System
RFID	Radio Frequency Identification
SIFT	Scale Invariant feature Transform
MCL	Monte Carlo Localization
ICP	Iterative Closest Point
LIDAR	Light Detection and Ranging
FCN	Fully Convolutional Network
DOG	Dynamic Occupancy Grid
MLP	Multilayer Perceptron
DBN	Deep belief Network
ANN	Artificial Neural Network
RBM	Restricted Boltzmann Machine
RMSE	Root Mean Square Error
MAE	Mean Absolute Error
HRRP	High Resolution Range Profile
LPC	Linear Prediction Coding

PCNN	Pulse-Coupled neural Network
RBF	Radial Basis Function
GCC	GNU Compiler Collection
LMDB	Lightning Memory-Mapped Database
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RGB	Red Green Blue

List of Figures

List of Tables

Listings

Bibliography

[1] A. Acosta and F. Almeida. The particle filter algorithm: Parallel implementations and performance analysis over android mobile devices. *Concurr. Comput. : Pract. Exper.*, 28(3):788–801, March 2016.

[2] Zhang Aiyun, Yuan Kui, Yan Zhigang, and Zhu Haibing. Research and application of a robot orientation sensor. In *IEEE International Conference on Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings. 2003*, volume 2, pages 1069–1074 vol.2, Oct 2003.

[3] Rashad Al-Jawfi. Handwriting arabic character recognition lenet using neural network. *Int. Arab J. Inf. Technol.*, 6(3):304–309, 2009.

[4] E. Angel and D. Morrison. Speeding up bresenham's algorithm. *IEEE Computer Graphics and Applications*, 11(6):16–17, Nov 1991.

[5] Nando de Freitas & Neil Gordon Arnaud Doucet. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[6] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, Feb 2002.

[7] B. Bach. Unfolding dynamic networks for visual exploration. *IEEE Computer Graphics and Applications*, 36(2):74–82, Mar 2016.

[8] Dana H Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

[9] F. Beaufays and E. A. Wan. Relating real-time backpropagation and backpropagation-through-time: An application of flow graph interreciprocity. *Neural Computation*, 6(2):296–306, March 1994.

[10] Vaishak Belle and Hector Levesque. A logical theory of robot localization. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 349–356, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.

[11] Jose M Bernardo and Adrian Smith. *Bayesian Theory*. Wiley Series in Probability and Statistics. John Wiley & Sons Ltd., Chichester, 2000.

[12] J. L. Blanco, J. Gonzalez, and J. A. Fernandez-Madrigal. An optimal filtering algorithm for non-parametric observation models in robot localization. In *2008 IEEE International Conference on Robotics and Automation*, pages 461–466, May 2008.

[13] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, Inc., 2nd edition, 2013.

[14] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Syst. J.*, 4(1):25–30, March 1965.

[15] Wolfram Burgard, Dieter Fox, and Sebastian Thrun. Active mobile robot localization. In *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*, IJCAI'97, pages 1346–1352, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[16] L. Caltagirone, S. Scheidegger, L. Svensson, and M. Wahde. Fast lidar-based road detection using fully convolutional neural networks. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1019–1024, June 2017.

[17] Qian Cao, Ku Wang, and Han Li. Fast and precise detection of straight line with improved hough transform. In *2010 8th World Congress on Intelligent Control and Automation*, pages 6014–6017, July 2010.

[18] S. V. Carata and V. E. Neagoe. A pulse-coupled neural network approach for image segmentation and its pattern recognition application. In *2016 International Conference on Communications (COMM)*, pages 61–64, June 2016.

[19] G. Carneiro and J. C. Nascimento. The fusion of deep learning architectures and particle filtering applied to lip tracking. In *2010 20th International Conference on Pattern Recognition*, pages 2065–2068, Aug 2010.

[20] A. Censi, L. Iocchi, and G. Grisetti. Scan matching in the hough domain. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2739–2744, April 2005.

[21] Olivier Chapelle, Bernhard Schlkopf, and Alexander Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010.

[22] H. Cho and S. W. Kim. Mobile robot localization using biased chirp-spread-spectrum ranging. *IEEE Transactions on Industrial Electronics*, 57(8):2826–2835, Aug 2010.

[23] K. S. Choi, J. W. Lee, and S. G. Lee. Navigation of a mobile robot using reduced particles based on incorporating particle filter with neural networks. In *2010 International Conference on Intelligent Control and Information Processing*, pages 128–133, Aug 2010.

[24] Chi-Yang Chu, Daniel J. Henderson, and Christopher F. Parmeter. On discrete epanechnikov kernel functions. *Computational Statistics & Data Analysis*, 2017.

[25] A. Cochocki and Rolf Unbehauen. *Neural Networks for Optimization and Signal Processing*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1993.

[26] Michael Collins, Robert E. Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Mach. Learn.*, 48(1-3):253–285, September 2002.

[27] S. Colonnese, S. Rinauro, and G. Scarano. Maximum likelihood scale parameter estimation: An application to gain estimation for qam constellations. In *2010 18th European Signal Processing Conference*, pages 1582–1586, Aug 2010.

[28] T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-14(3):326–334, June 1965.

[29] I. J. Cox, R. A. Boie, and D. A. Wallach. Line recognition. In *[1990] Proceedings. 10th International Conference on Pattern Recognition*, volume i, pages 639–645 vol.1, Jun 1990.

[30] I. J. Cox and J. B. Kruskal. On the congruence of noisy images to line segment models. In *[1988 Proceedings] Second International Conference on Computer Vision*, pages 252–258, Dec 1988.

[31] I. J. Cox, J. B. Kruskal, and D. A. Wallach. Predicting and estimating the accuracy of a subpixel registration algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):721–734, Aug 1990.

[32] Ingemar J. Cox. Blanche: Position estimation for an autonomous robot vehicle. In *Intelligent Robots and Systems '89. The Autonomous Mobile Robots and Its Applications. IROS '89. Proceedings., IEEE/RSJ International Workshop on*, pages 432–439, Sept 1989.

[33] J. F. G. De Freitas, M. A. Niranjan, A. H. Gee, and A. Doucet. Sequential monte carlo methods to train neural network models. *Neural Comput.*, 12(4):955–993, April 2000.

[34] T. de J. Mateo Sanguino and F. Ponce Gómez. Toward simple strategy for optimal tracking and localization of robots with adaptive particle filtering. *IEEE/ASME Transactions on Mechatronics*, 21(6):2793–2804, Dec 2016.

[35] Li Deng and Dong Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3&#8211;4):197–387, June 2014.

[36] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, September 1995.

[37] P. M. Djurić and M. F. Bugallo. Estimation of stochastic rate constants and tracking of species in biochemical networks with second-order reactions. In *2009 17th European Signal Processing Conference*, pages 2308–2311, Aug 2009.

[38] P. Dolezel, P. Skrabanek, and L. Gago. Pattern recognition neural network as a tool for pest birds detection. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6, Dec 2016.

[39] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Trevor Darrell, and Kate Saenko. Long-term recurrent convolutional networks for visual recognition and description. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2625–2634. IEEE Computer Society, June 2015.

[40] Ciro Donalek. Supervised and unsupervised learning. http://www.astro.caltech.edu/g̃eorge/aybi199/Donalek_Classif.pdf, April 2011. [Online; accessed 8 May 2017].

[41] Arnaud Doucet, Nando Defreitas, and Neil Gordon. *An Introduction to Sequential Monte Carlo Methods*, chapter 3, pages 79–95. Springer-Verlag, New York, 2001.

[42] Arnaud Doucet and Adam M. Johansen. A tutorial on particle filtering and smoothing: fifteen years later, 2011.

[43] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, January 1972.

[44] F. Duvallet and A. D. Tews. Wifi position estimation in industrial environments using gaussian processes. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2216–2221, Sept 2008.

[45] Austin Eliazar Eliazar and Ronald Parr. Learning probabilistic motion models for mobile robots. In *In Proc. of the International Conference on Machine Learning (ICML*, page 32. ACM Press, 2004.

[46] Pantelis Elinas and James J. Little. σMCL: Monte-Carlo localization for mobile robots with stereo vision. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.

[47] J. L. Ellis, G. Kedem, T. C. Lyerly, D. G. Thielman, R. J. Marisa, J. P. Menon, and H. B. Voelcker. The ray casting engine and ray representatives. In *Proceedings of the First ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, SMA '91, pages 255–267, New York, NY, USA, 1991. ACM.

[48] Brian Ferris, Dirk Hahnel, and Dieter Fox. *Gaussian processes for signal strength-based location estimation*, volume 2, pages 303–310. MIT Press Journals, 2007.

[49] Gernot A. Fink. *Markov Models for Pattern Recognition: From Theory to Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2014.

[50] Dieter Fox. Kld-sampling: Adaptive particle filters. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 713–720. MIT Press, 2002.

[51] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 343–349, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[52] J. Friedman, T. Hastie, and R. Tibshirani. Additive Logistic Regression: a Statistical View of Boosting. *The Annals of Statistics*, 38(2), 2000.

[53] J. Fritsch, T. Kühnl, and A. Geiger. A new performance measure and evaluation benchmark for road detection algorithms. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 1693–1700, Oct 2013.

[54] C. Galamhos, J. Matas, and J. Kittler. Progressive probabilistic hough transform for line detection. In *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, volume 1, page 560 Vol. 1, 1999.

[55] Nil Garcia. *Optimization Methods for Active and Passive Localization*. PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, F, 2015.

[56] A. Gensler, J. Henze, B. Sick, and N. Raabe. Deep learning for solar power forecasting. an approach using autoencoder and lstm neural networks. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 002858–002865, Oct 2016.

[57] Brian Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, june 2003. http://www.isr.uc.pt/icar03/.

[58] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, UK, 1989.

[59] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[60] Shixiang Gu, Zoubin Ghahramani, and Richard E. Turner. Neural adaptive sequential monte carlo. *CoRR*, abs/1506.03338, 2015.

[61] Leonidas J. Guibas, Rajeev Motwani, and Prabhakar Raghavan. The robot localization problem in two dimensions. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 259–268, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[62] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[63] T. He and J. Droppo. Exploiting lstm structure in deep neural networks for speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5445–5449, March 2016.

[64] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 190–198. Curran Associates, Inc., 2013.

[65] J. D. Hincapié-Ramos, K. Özacar, P. P. Irani, and Y. Kitamura. Gyrowand: An approach to imu-based raycasting for augmented reality. *IEEE Computer Graphics and Applications*, 36(2):90–96, Mar 2016.

[66] Geoffrey Hinton and Terrence J. Sejnowski, editors. *Unsupervised Learning : Foundations of Neural Computation*. Computational Neuroscience. MIT Press, 1998.

[67] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[68] Tomas Hrycej. *Modular Learning in Neural Networks*. Wiley-Interscience, 1992.

[69] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2003.

[70] Itseez. *The OpenCV Reference Manual*, 2.4.9.0 edition, April 2014.

[71] Itseez. Open source computer vision library. `https://github.com/itseez/opencv`, 2015.

[72] M. Izadkhah, M. Hosseini, and H. Fayyazi. Particle filter supported with the neural network for aircraft tracking based on kernel and active contour. In *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 653–658, Oct 2014.

[73] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[74] V. Jithesh, M. J. Sagayaraj, and K. G. Srinivasa. Lstm recurrent neural networks for high resolution range profile based radar target classification. In *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*, pages 1–6, Feb 2017.

[75] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[76] F. Kallasi, D. L. Rizzini, and S. Caselli. Fast keypoint features from laser scanner for robot localization and mapping. *IEEE Robotics and Automation Letters*, 1(1):176–183, Jan 2016.

[77] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[78] Naimul Mefraz Khan and Kaamran Raahemifar. A novel accelerated greedy snake algorithm for active contours. In *CCECE*, pages 186–190. IEEE, 2011.

[79] Sangjoon Kim, Neil Shephard, and Siddhartha Chib. Stochastic volatility: Likelihood inference and comparison with arch models. *The Review of Economic Studies*, 65(3):361–393, 1998.

[80] G. A. Korikar, S. K. Katragadda, S. Kesarla, J. M. Conrad, and A. F. Browne. A survey on robot localization in extraterrestrial environments. In *SoutheastCon 2016*, pages 1–7, March 2016.

[81] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press.

[82] O. Kramer. Dimensionality reduction by unsupervised k-nearest neighbor regression. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, volume 1, pages 275–278, Dec 2011.

[83] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[84] A. Laddha, M. K. Kocamaz, L. E. Navarro-Serment, and M. Hebert. Map-supervised road detection. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 118–123, June 2016.

[85] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *J. Mach. Learn. Res.*, 10:1–40, June 2009.

[86] S. Lawrence, C.L. Giles, Ah Chung Tsoi, and A.D. Back. Face recognition: a convolutional neural-network approach. *Neural Networks, IEEE Transactions on*, 8(1):98–113, January 1997.

[87] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[88] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 609–616, New York, NY, USA, 2009. ACM.

[89] J. Leonard, H. Durrant-Whyte, and I. J. Cox. Dynamic map building for autonomous mobile robot. In *EEE International Workshop on Intelligent Robots and Systems, Towards a New Frontier of Applications*, pages 89–96 vol.1, Jul 1990.

[90] R. Lienhart and J. Maydt. An extended set of haar-like features for rapid object detection. In *Proceedings. International Conference on Image Processing*, volume 1, pages I–900–I–903 vol.1, 2002.

[91] Medsker LR and Jain LC. Recurrent neural networks. *Design and Applications*, 5, 2001.

[92] J. Šíma. Training a single sigmoidal neuron is hard. *Neural Computation*, 14(11):2709–2728, Nov 2002.

[93] Steven N. Maceachern, Merlise Clyde, and Jun S. Liu. Sequential importance sampling for nonparametric bayes models: The next generation. *Can J Statistics*, 27(2):251–267, June 1999.

[94] A. Majdik, M. Popa, L. Tamas, I. Szoke, and G. Lazea. New approach in solving the kidnapped robot problem. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–6, June 2010.

[95] John E. Markel and A. H. Gray. *Linear Prediction of Speech*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[96] Jiri Matas, Charles Galambos, and Josef Kittler. Progressive probabilistic hough transform. In John N. Carter and Mark S. Nixon, editors, *BMVC*. British Machine Vision Association, 1998.

[97] J. Mazumdar and R. G. Harley. Recurrent neural networks trained with backpropagation through time algorithm to estimate nonlinear load harmonic currents. *IEEE Transactions on Industrial Electronics*, 55(9):3484–3491, Sept 2008.

[98] Nicholas Metropolis and Stanislaw M. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949.

[99] Thomas P. Minka. *A Family of Algorithms for Approximate Bayesian Inference*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001. AAI0803033.

[100] Rahul Mohan. Deep deconvolutional networks for scene parsing, 2014.

[101] C. Molter, U. Salihoglu, and H. Bersini. The road to chaos by time-asymmetric hebbian learning in recurrent neural networks. *Neural Computation*, 19(1):80–110, Jan 2007.

[102] N. Morales, J. Toledo, L. Acosta, and J. Sánchez-Medina. A combined voxel and particle filter-based approach for fast obstacle detection and tracking in automotive applications. *IEEE Transactions on Intelligent Transportation Systems*, 18(7):1824–1834, July 2017.

[103] M. Müller, T. Röder, M. Clausen, B. Eberhardt, B. Krüger, and A. Weber. Documentation mocap database hdm05. Technical Report CG-2007-2, Universität Bonn, June 2007.

[104] Radford M. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2):125–139, April 2001.

[105] Rudy Negenborn. *Robot localization and Kalman filters*. PhD thesis, Utrecht University, 2003.

[106] D. M. Q. Nelson, A. C. M. Pereira, and R. A. de Oliveira. Stock market's price movement prediction with lstm neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1419–1426, May 2017.

[107] W. L. Nelson and I. J. Cox. Local path control for an autonomous vehicle. In *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, pages 1504–1510 vol.3, Apr 1988.

[108] Richard Nock and Frank Nielsen. A real generalization of discrete adaboost. *Artificial Intelligence*, 171(1):25 – 41, 2007.

[109] J. M. O'Kane. Global localization using odometry. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 37–42, May 2006.

[110] Jason O'Kane. *A Gentle Introduction to ROS*, volume 1. University of South Carolina, 315 Main Street, Columbia, 4 2016.

[111] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[112] Sh. Ch. Pang, Anan Du, and Zh. Zh. Yu. Robust multi-object tracking using deep learning framework. *J. Opt. Technol.*, 82(8):516–527, Aug 2015.

[113] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1310–III–1318. JMLR.org, 2013.

[114] Franz Pernkopf. Computational intelligence: Teil 2. In *Signal Processing and Speech Communication Laboratory*, pages 1–36. University of Technology Graz, May 2017.

[115] F. Piewak, T. Rehfeld, M. Weber, and J. M. Zöllner. Fully convolutional neural networks for dynamic object detection in grid maps. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 392–398, June 2017.

[116] Daniel Polani. *Kullback-Leibler Divergence*, pages 1087–1088. Springer New York, New York, NY, 2013.

[117] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[118] J. Rabe and C. Stiller. Robust particle filter for lane-precise localization. In *2017 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pages 127–132, June 2017.

[119] Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[120] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[121] P. P. L. Regtien. Sensors for application in robot systems. In *IEE Colloquium on Solid State and Smart Sensors*, pages 9/1–9/3, May 1988.

[122] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[123] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[124] Reuven Y. Rubinstein and Dirk P. Kroese. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-carlo Simulation (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[125] Reuven Y. Rubinstein and Dirk P. Kroese. *Simulation and the Monte Carlo Method (Wiley Series in Probability and Statistics)*. Wiley, 2 edition, 2007.

[126] David E. Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation. In Yves Chauvin and David E. Rumelhart, editors, *Backpropagation: The Basic Theory*, pages 1–34. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[127] J. Röwekämper, C. Sprunk, G. D. Tipaldi, C. Stachniss, P. Pfaff, and W. Burgard. On the position accuracy of mobile robot localization based on particle filters combined with scan matching. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3158–3164, Oct 2012.

[128] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 791–798, New York, NY, USA, 2007. ACM.

[129] Y. Salimpour and H. Soltanian-Zadeh. Particle filtering of point processes observation with application on the modeling of visual cortex neural spiking activity. In *2009 4th International IEEE/EMBS Conference on Neural Engineering*, pages 718–721, April 2009.

[130] Robert E. Schapire. A brief introduction to boosting. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 1401–1406, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[131] Robert E. Schapire. *Explaining AdaBoost*, pages 37–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[132] Henning Scharsach. Advanced gpu raycasting. In *In Proceedings of CESCG 2005*, pages 69–76, 2005.

[133] Henning Scharsach. Advanced gpu raycasting. In *In Proceedings of CESCG 2005*, pages 69–76, 2005.

[134] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[135] Peter M. Schultheiss and Kara Wagner. *Active and Passive Localization: Similarities and Differences*, pages 215–232. Springer Netherlands, Dordrecht, 1989.

[136] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11):2673–2681, November 1997.

[137] Stephen Se, D. Lowe, and J. Little. Global localization using distinctive visual features. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 226–231 vol.1, 2002.

[138] Stephen Se, David Lowe, and Jim Little. Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *The International Journal of Robotics Research*, 21(8):735–758, 2002.

[139] I. A. Shamov and P. S. Shelest. Application of the convolutional neural network to design an algorithm for recognition of tower lighthouses. In *2017 24th Saint Petersburg International Conference on Integrated Navigation Systems (ICINS)*, pages 1–2, May 2017.

[140] Einar Snorrason. Robot localization in dynamic environments. Master's thesis, KTH, School of Computer Science and Communication (CSC), 2015.

[141] Ingo Steinwart and Andreas Christmann. *Support Vector Machines*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[142] M. Sundermeyer, H. Ney, and R. Schlüter. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):517–529, March 2015.

[143] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[144] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[145] Sebastian Thrun. Particle filters in robotics. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, pages 511–518, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[146] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[147] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust monte carlo localization for mobile robots. *Artificial Intelligence*, 128(1):99 – 141, 2001.

[148] Gian Diego Tipaldi, Manuel Braun, and Kai O. Arras. *FLIRT: Interest Regions for 2D Range Data with Applications to Robot Navigation*, pages 695–710. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[149] Lisa Turner and Christopher Sherlock. An introduction to particle filtering. *Technical Report*, pages 1–25, May 2013.

[150] T. Uchimoto, S. Suzuki, and H. Matsubara. A method to estimate robot's location using vision sensor for various type of mobile robots. In *2009 International Conference on Advanced Robotics*, pages 1–6, June 2009.

[151] W. van der Aalst, V. Rubin, H. Verbeek, B. van Dongen, E. Kindler, and C. G"unther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling*, 2009.

[152] K Vikram, Niraj Upashyaya, Kavuri Roshan, and A Govardhan. Image edge detection. *Special Issues of international Journal of Computer Science and Informatics (IJCSI)*, 2, 2010.

[153] OR Vincent and Olusegun Folorunso. A descriptive algorithm for sobel image edge detection. In *Proceedings of Informing Science & IT Education Conference (InSITE)*, volume 40, pages 97–107, 2009.

[154] J. Wang, L. Gu, B. Wang, and Y. Tian. Kalman filter method of target tracking based on los coordinate. In *2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics*, volume 1, pages 235–238, Aug 2012.

[155] X. Wang, Y. Jia, N. Xi, J. Zhen, and F. Xu. Mobile robot pose estimation using laser scan matching based on fourier transform. In *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 474–479, Dec 2013.

[156] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical Report 95-041, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[157] P. Werbos. Backpropagation through time: what does it do and how to do it. In *Proceedings of IEEE*, volume 78, pages 1550–1560, 1990.

[158] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280, June 1989.

[159] Shiming Xiang, Feiping Nie, and Changshui Zhang. Learning a mahalanobis distance metric for data clustering and classification. *Pattern Recogn.*, 41(12):3600–3612, December 2008.

[160] N. Yadaiah, R. S. Bapi, A. S. Kumar, and M. Roopchandan. State estimation of nonlinear system through particle filter based recurrent neural networks. In *2011 IEEE Recent Advances in Intelligent Computational Systems*, pages 307–310, Sept 2011.

[161] Xin Yan and Xiao Gang Su. *Linear Regression Analysis: Theory and Computing*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2009.

[162] B. Yang and H. Xiao. On large scale evolutionary optimization using simplex-based cooperative coevolution genetic algorithm. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–5, Dec 2009.

[163] J. Yoo and H. J. Kim. Utilization of unlabeled data for smartphone-robot localization. In *2016 International Conference on Electronics, Information, and Communications (ICEIC)*, pages 1–4, Jan 2016.

[164] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23*, pages 2595–2603. Curran Associates, Inc., 2010.