AALBORG UNIVERSITY
STUDENT REPORT

Department of Architecture, Design and Media Technology

Medialogy, 10th Semester

Title:
Danger recognition and visual aid for the monitoring of wayfinding robots in obstacle avoidance scenarios

Project Period: 1 February 2018 – 30 June 2018

Semester Theme:
Master's Thesis

Supervisors:
David Meredith

Projectgroup no.
MTA181036

Members:
Sonny Glasius Polack
Viktor Schmuck

Abstract:

The course of action for object avoidance throughout the past 15 years has given birth to new possibilities for indoor navigation in crowded areas. However, the gap lies within perceiving and navigating the area in a safe manner. Robots are also becoming a larger part in people's everyday lives which provides a need for understanding the robots' interactions. However, the gap lies within task management and visualising the brains of a robot. We propose a solution for the recognition of dangerous scenarios and an untested visual aid for comprehending robot reactions to input. We can conclude that our solution is programmatically accurate but requires improvements to be applicable in a real life scenario.

Danger recognition and visual aid for the monitoring of wayfinding robots in obstacle avoidance scenarios

Sonny Glasius Polack

spolac13@student.aau.dk

School of information and communication technology

Aalborg University, Denmark

Viktor Schmuck

vschmu13@student.aau.dk

School of information and communication technology

Aalborg University, Denmark

Supervisor: David Meredith

Abstract

The course of action for object avoidance throughout the past 15 years has given birth to new possibilities for indoor navigation in crowded areas. However, the gap lies within perceiving and navigating the area in a safe manner. Robots are also becoming a larger part in people's everyday lives which provides a need for understanding the robots' interactions. However, the gap lies within task management and visualising the brains of a robot. We propose a solution for the recognition of dangerous scenarios and an untested visual aid for comprehending robot reactions to input. We can conclude that our solution is programmatically accurate but requires improvements to be applicable in a real life scenario.

On the note of mixed groups

Due to the project group consisting of people taking two different specialisations, Sonny Glasius Polack on Computer Graphics and Viktor Schmuck on Interaction, several parts of the report were only written by one of the team members. The following list describes this relation for sections which were not written together:

- Sonny G. Polack – Sections: 2.2, 3.2, and 7.2
- Viktor Schmuck – Sections: 2.1, 3.1, 4.1, 5.1, 6.1, and 7.1

Contents

1 Introduction

Development in object avoidance in general allows a new option for the development of wayfinding robots, which can be utilised in densely populated indoor environments. The applications of such a solution could be used in hospitals or airports, where users could follow a guidance robot. However, there is a gap between the perception of the environment and the navigation that has to be performed so it is optimal, safe for others in the environment, and predictable for the users following and calculating their movements based on the robot's. Robots have undeniably become a bigger part of our everyday life, and so has the need for visualising what they do, whether it is a professional wayfinding robot or a robotic vacuum cleaner, especially for novel users. The current gap lies between using visualisation as a control device and portraying the thought process of the robots. We propose to visualise inputs and monitor reactions of a robot for improving the understanding for novel users.

We propose a system based on a simple wayfinding algorithm, with the extension of continuously assessing the situation of the robot and performing programmatically evolved avoidance policies when needed. From the outlined full solution, this report documents the implementation and evaluation of the situation assessment solution, and contain a solution where users are shown a robot's POV with overlaid distance data, while having security states available to relate current state to input data from laser distance sensor. However the visual part of this project remains unevaluated.

2 Related Work

2.1 Automated Vehicle Navigation and Object Avoidance

This section describes the related work regarding the proposed techniques and improvements for the data collection and training phases of an object avoidance project. It goes into detail on how a safer and more resourceful sample collection can be conducted for such a project. Moreover, it describes the different approaches for algorithms and practices implemented for solving the static or dynamic object avoidance problem. Lastly, the considerations regarding human-robot interaction are also described.

2.1.1 Sampling and training

This section describes how the techniques for training, data collection and assumptions in this domain changed over time.

Nasrollahy and Javadi (2009) proposed an approach for creating a more efficient path planning algorithm which is also capable of avoiding obstacles in dynamic environments. Even though their results showed that with the correct assignment of penalty values and a good measurement of the status of an environment a better path can be calculated, their assumptions limit the implementation. Firstly, in their implementation two obstacles cannot have overlaps, and secondly, the robot is constantly aware of the position of all obstacles in the area, which is usually not the case in a real-life pathfinding scenario.

A different implementation was developed by Kim and Do (2012), which aimed for using a single camera for obstacle detection and avoidance during a robot's navigation. Their block based motion estimation (BBME) showed that by splitting a camera feed into blocks and performing classification on the blocks, the obstacles can be distinguished from the background of the view. Moreover, the identified obstacles' motion vector can be calculated with an algorithm that utilises a matching criterion called the sum of absolute differences (SAD). Their implementation was mostly successful in identifying the obstacles and their movement vectors, however it had limitations caused by the obstacles' distance, colour and reflected light.

As for the data collection for the training of obstacle avoidance implementations, Shrivastava et al. (2016) propose a concept with the utilisation of a standard approach, bootstrapping. Bootstrapping means that the number of training samples is accumulated by first conducting a general sample collection and then, after training, revisiting scenarios where the training presents inaccurate results (e.g. false positives). In their implementation, they used a fixed model for finding examples and trained their model on them. In the second phase, the model is set up to collect data in scenarios producing false positive results during evaluation, thus gathering "hard examples" which, based on their evaluation, leads to improved training time and efficiency as well as continuous improvements regarding the detection accuracy of the trained model.

Another aspect of data collection is the principle of keeping the robot safe while it learns to avoid collisions. This can be achieved by introducing risk-awareness in the data collection algorithm. Daftry et al. (2016) state that it is inevitable, even with a high-accuracy model that the robot encounters scenarios it cannot handle. To solve this problem, they propose the implementation of introspective behaviour, with which they trained a drone (UAV) to avoid inevitable crashes and perform emergency manoeuvres

by detecting when the inputs of the drone will result in poor trajectory estimation. With their implementation, they prevented several inevitable crashes and, just as proposed by Shrivastava et al. (2016), made the drone collect additional training samples of the ambiguous situations. Lastly, based on their evaluation results, they argue that the introspective behaviour is performing better when it is monitoring uncertainty based on raw sensory inputs rather than relying on the certainty values of trained classifiers. Similarly, Kahn et al. (2017) propose a solution for making the real-life training of robots safer while keeping their learning speed at an optimal level. Their solution enables robots to drive or fly at a higher speed during training, only preventing them from doing so if the calculated confidence value for a classification is too low, in order to prevent high-speed crashes. Their evaluation shows that in case of high certainty the robots were able to learn avoidance strategies at a higher speed while in low certainty cases a "risk-averse" strategy was chosen automatically by the robot.

Gandhi et al. (2017) describe a different approach for dynamic object avoidance by training deep neural networks on collected negative (collision involving) and positive (collision free) flying data. According to Gandhi et al. (2017), this approach eliminates the gap between a safe simulated environment and a real-world one, while establishing a controlled environment and utilising a robot prepared for collision. They gather data from different environments by flying in randomly generated directions several times in order to acquire video footage which can be used to classify safe and non-safe states and to determine what the optimal avoidance action of the used drone should be. Their solution is capable of navigating in clustered areas and narrow passages with both static and dynamic obstacles.

Instead of collecting data from the real world, Sadeghi and Levine (2016) propose a simulation-based data generation and model training approach. They use the raw camera feed of a simulated 3D environment in order to train a collision predictor with reinforcement learning. They argue that this enables the safe collection of data, while also eliminating the limitations resulting from expert-based inverse reinforcement learning, which affects the diversity and quality of the collected data. Their evaluation shows that the pure simulation-based implementation is able to perform as well as other end-to-end learning methods.

2.1.2 Algorithm overview

This section describes what methods have been used for the implementation of object avoidance models.

A solution for avoiding collisions while navigating is proposed by Minguez and Montano (2004). Their nearness diagram navigation is based on a perception-action flow process. The robot in this implementation is programmed to move towards a goal, while after each performed movement it re-analyses the input data provided by sensors and takes further actions based on these readings. Their model defines a set of scenarios that have to be identified by the robot and defines a set of preprogrammed actions the robot has to take upon classifying a scenario in order to reach its goal. Based on their evaluation, the robot was able to find a short path to its goal while being able to handle object avoidance and navigation in both wide and narrow areas.

A different approach, as presented by Large et al. (2005) uses an A* algorithm for the navigation of a robot, while utilising a nonlinear velocity obstacle (NLVO) algorithm for predicting the future position of dynamic obstacles and identify those which would collide with the robot. Based on their evaluation, the robot is capable of reacting quickly to changes in its environment and avoid risky situations real-time while it's moving towards its goal.

Ross et al. (2013) propose a solution based on expert-based learning. The implementation is based on dataset aggregation (DAgger) with which a policy is trained in order to create a model that can mimic a human pilot's obstacle avoidance behaviour to a desired level of similarity. Based on the trained model, their evaluation showed that with the use of neural networks and a DAgger based training, a single camera was enough to perform expert-based training and enable the drone to avoid stationary obstacles in a dense outdoor area.

Neural network-based solutions LeCun et al. (2005) propose a convolutional network-based solution for off-road obstacle avoidance. The system's outputs were a set of steering angles which were decided based on two cameras' feed with the use of a 6-layer convolutional neural network (CNN). The data collection was performed in a variety of off-road environments, ensuring diversity. Moreover, the data collection was performed so turning was only utilised when it was absolutely necessary, and in such cases the avoidance distance from the obstacles was consistent. The thorough data collection and the neural-network's training resulted in, based on their evaluation, a reliable obstacle avoidance model in multiple off-road outdoor environments. Moreover, the used methods ensured that manual calibration or parameter tuning was not necessary, as those happened automatically.

Wang et al. (2007) present a solution for a slightly different problem, but with a similar type of solution when addressing the problem of obstacle avoidance while flying

UAVs in a formation. Similarly to the navigation principle of LeCun et al. (2005), the drones flew to their destination in a straight line and only turned when necessary. In the implementation of Wang et al. (2007) a so-called Grossberg neural network was trained in order to identify "pop-up" (unmapped) obstacles. According to their implementation, upon identification, a buffer-zone was created around the obstacles, based on which the drones had to re-plan their path before proceeding to their destination and realigning themselves into the formation.

Milde et al. (2017) discuss the use of a vision-based reactive avoidance strategy, which could identify scenarios when an avoidance manoeuvre was needed. The system, with the utilisation of neural networks was able to navigate an obstacle-filled office environment where background noise was also a problematic factor. Their implementation was a part of a proof of concept project demonstrating the increased performance of neuromorphic hardware and its applicability for neural network-based computations.

Genetic algorithm-based solutions Genetic algorithms are inspired by Darwin's theory of evolution. Some of the first attempts for solving obstacle avoidance are the ones proposed by Han et al. (1997) and Tu and Yang (2003). Their implementation describe genetic algorithms trained for tens of generations in order to teach robots to avoid stationary or dynamic obstacles. In the case of genetic algorithms, a generation consists of a population of chromosomes, which are essentially a series of commands. They describe the different variables used for creating a training algorithm such as:

- Selection - How big percent of the chromosomes make it to the next generation
- Crossover - Combining chromosomes of parents to create a new generation
- Mutation - Change in the chromosome to introduce new information to the population and prevent its saturation

Lastly, a fitness function is described by them, which is used to evaluate the generated possible solutions and with which good actions can be rewarded, bad ones penalised, and based on which the better-performing chromosomes can be selected.Tu and Yang (2003) also argue that it is better to use variable-length chromosomes, since, according to their evaluation, that type is better at finding optimal paths in dynamic environments than fixed-length ones.

Genetic algorithms, as stated by Pinto and Gupta (2015) eliminate the need for human labelling of datasets. They state that such approach may be biased by semantics and it is not scalable since the amount of data required to generalise objects, and in their case

the possible grasping locations and directions, is too high. Therefore, they propose a trial and error approach in the form of a self-supervising reinforcement learning algorithm.

Compared to reinforcement learning, inverse reinforcement learning (IRL) is not trained from just a set of actions its chromosomes can be constructed from. Inverse reinforcement learning is used with expert-based training, where the desired behaviour is taught to the robot by human input and the reinforcement learning algorithm tries to approximate the behaviour when constructing its chromosomes from single actions. One of the implementations for IRL is described by Kim and Pineau (2016). They present an approach where expert-based training in combination with IRL attempts to find a solution for socially adaptive path planning, and the generation of human-like avoidance trajectories in crowded areas. They compared their solution to a dynamic window approach (DWA). The IRL, where data is gathered from experts' interaction with the system, was able to create a socially fitting trajectory planner, as opposed to the DWA approach which, firstly, did not perform well when avoiding dynamic objects it detected, and was not suitable for creating socially adaptive paths. Similarly, Zhang et al. (2016) describe an expert-based learning where their robot is learning from demonstration. They trained their model with IRL, utilising neural network-processing in order to translate the sensory readings (video feed).

Q-learning is a sub-category in reinforcement learning. Jaradat et al. (2011) describe it as an on-line learning agent, which is trained to identify which action a robot should take given the scenario it is in. During the training, the algorithm tries and evaluates the state-value pairs and in the end establishes a policy which fits for the task it was given. Jaradat et al. (2011) describe four kinds of states: safe, non-safe, winning, and failure. The last two states indicate the end of a run, winning being a desired state and failure being a state the robot needs to avoid. In the case of the other two states the Q-learning algorithm has to learn how to act in order to achieve a winning state and strive to be in the least amount of non-safe states. Their approach to the defined states attempt to mimic human reasoning, however their implementation is built on the assumption that the robot knows the position of all obstacles while planning and modifying its path. Mnih et al. (2015) extend the usage of Q-learning by processing input data with deep neural networks and using that input for training the Q-value of the reinforcement learning algorithm. Duguleana and Mogan (2016) investigated how a path planning training can be implemented in a virtual environment by defining the underlying participants of an obstacle avoidance scenario and creating a digital representation of real world setups. They used Q-learning, utilising the 4 different states identified by Jaradat et al. (2011). Their evaluation showed that with the tested training parameters the pathfinding system was able to construct an optimal, collision-free path even with

dynamic obstacles. They also concluded that using simulated training is safer compared to real-world scenarios, however it did not allow for the training and testing of complex scenarios and its representation in relation to real-world cases is not accurate. Moreover, training with a simulated environment in their case was dependent on the accessibility of accurate global knowledge, which is often not the case in real-life scenarios.

2.1.3 Human-robot interaction

This section describes the work conducted on making the pathfinding solutions socially aware in order to not violate social rules in scenarios when they are placed in crowded environments.

Sisbot et al. (2007) state that a human-aware motion planner should take social acceptability into account when constructing a safe path for itself. This, according to their definition includes that, firstly, the robot ensures safe motion, meaning that human participants in navigation scenarios are not harmed by the robot. Moreover, the robot should have reliable and effective motion, which ensures that the robot completes its task. And lastly, that the robot is capable of a socially acceptable motion, resulting in a policy that is in accordance with the robot's goal and preferences, but which also takes the principles of human movement into account. Henry et al. (2010) expand on latter principles by stating that its elements include " "desire to move with the flow", "avoidance of high-density areas", "preference for walking on the right/left side", and "desire to reach the goal quickly" ". They also describe that humans also evaluate a path-option based on the density of people and their expected position and velocity weighed against the time and distance it would take to reach a goal in crowded areas. Therefore socially-adaptive robots should also be capable of such evaluation.

To evaluate a human-aware navigation robot, Kruse et al. (2013) in their survey investigate three aspects: comfort, naturalness, and sociability. Based on measurements, they define the distance at which a robot should be able to deviate from a path when performing avoidance, and the speed at which it can approach human obstacles during wayfinding without causing discomfort or scaring the participants. Moreover, apart from the kinematic constraints and behaviour of the robot, they found that a human-aware navigation system's natural motion is not only defined by the path it chooses, but also by its predictability. The latter can be achieved by creating a behaviour that can be regarded as smooth. Such can be achieved by constructing a policy with an avoidance trajectory and velocity profile that resembles human navigational behaviour.

2.2 Graphical User Interfaces for Artificial Intelligence and Robots

The use of graphical user interfaces (GUI) when working with AI and robots is predominantly concerned with two focus areas, namely, control interfaces, where the GUI is primarily used for task management, and visualising the thought processes of the AI, where the GUI is used for conveying the decision making process and ease of use in terms of human computer interaction. This section describes both uses of GUI in relation to AI and robots.

2.2.1 Control interfaces

Sakamoto et al. (2009) worked on a graphical user interface for novel users to allow them to easier control home robots. They argued that when implementing technological solutions for novel users the usability of the application must be high as most people are not technopaths. They used a GUI with stroke commands due to familiarity as the GUI offer hands-on feedback due to the availability of a screen. The implementation requires a camera setup that cover the entire work surface area for accurate real-time tracking of the robot. The users sketch the work surface from a real-time view of the room and can then use different stroke commands for tasks such as starting, stopping and return. Their results suggest that their implementation ease the use of home robots, however, they only pilot-tested the interface.

Liu et al. (2011) implemented a user friendly task management tool for multiple robots (RoboShop), which is a graphical interface where users can select tools for giving tasks to robots. Liu et al. (2011) argue that most current implementations require direct instruction for the robot and that they also do not take multiple robots into account. The implementation draw parallels to Adobe Photoshop where Liu et al. (2011) made it so the user can specify a task consisting of a tool, place and time where a tool can be vacuuming, the place could be living room or a section of the room and time could be during the work hours of the user. The tasks are specified in a layer system for ease of keeping track of tasks. The available tools are in a tool panel similarly to Photoshop and the location is user-specified in what would be the canvas in Photoshop. Lastly, the time is specified in a scheduler. They user-tested their implementation on 7 participants with an age range of 21-24 years old in a four part experiment.

1. Pre-Questionnaire where demographics and experience with robots were determined
2. Introduction to the user interface

3. Actual evaluation

4. Post questionnaire, which was focused on usability

In order to prevent initial bias the participants had to specify three tasks while evaluating the interface. Their results suggest that their implementation was useful for task assignment and management of multiple robots.

Srinivas et al. (2013) and Wei et al. (2016) argue that monitoring autonomous robots becomes a higher demand as the capability of the robots increase, and that the complexity of the logic should not be apparent for novel users.

Srinivas et al. (2013) worked on a graphical interface for autonomous robots with their focus being on non-expert users. They developed a touchscreen interface where users can plot points and specify actions or connections between them with functionality such as colour coding events or locations and long pressing waypoints for additional features.

Wei et al. (2016) expanded the work by allowing users to sketch the intended path of the robot based on a roadmap where users first have to specify waypoints and connectivity, meaning that if users sketch a path that is not possible for the robot, due to the roadmap, it will follow the nearest path available. However, their work is not user-tested.

2.2.2 Visualising the thought process of artificial intelligence

The black box nature of neural networks are known to be a problem when understanding how they work (Plate et al., 2000).

Methods for understanding neural networks range from a 'how' approach to a 'what' approach, where the 'how' approach works with the internal state of the network decision process and the 'what' attempts to to explain the decisions based on the inputs. One way of working with the what approach is to illustrate the decisions by plotting relational inputs on a two dimensional plot to detect if there is a relation between the different inputs.

They developed a neural network for detecting the risk of lung cancer compared to other non-smoking related cancer types and found that when using two dimensional plots it provides a quick insight to the output and, at the same time, aid the ability to understand the output of a neural network.

Chadalavada et al. (2015) Attempts to improve the relationship between humans and robots in shared work environments by communicating the intended path of a robot forklift through floor projection. They compared two scenarios:

1. Robot not avoiding collision path

2. Robot avoiding collision path

Where the robot would alternate between whether to show the intended path. Their experiment measured five parameters, communication, reliability, predictability, transparency and situation awareness and they found that when showing the intended path it increases all five. Additionally, they discovered that humans change their own intended path sooner when the robot's path is projected.

Amershi et al. (2015) are working with performance analysis of machine learning models and are implementing a visual aid for debugging during test and training. They argue that debugging is an important step in model building and that usually this is done through graphs and statistics. They suggest that debugging can be improved through analysing the behaviour of the model and find that existing tools can be cumbersome even for experts.

They built a binary classifier and visualised its performance, i.e. classification accuracy, for all samples for both training and test sets, adding up to the full dataset. With the accuracy representation, the low scores can be related to exact samples which are problematic regarding classification. They argue that a visual aid can benefit over traditional methods by showing source and severity of errors and they found that a visual aid for model debugging is preferred over traditional methods on a at-a-glance level.

Wortham et al. (2017b) are working with real-time visualisation of a robot's AI where they take offset in naive users while demonstrating that even an abstract representation is enough to improve understanding of the behaviour of a robot.

They utilise an instinct planner which essentially monitor the execution, success, failure, error and in progress status event. Wortham et al. (2017a) are describing the functionality of the robot to fit with cases of search scenarios and the initial states are created for such a purpose and they use a graphical solution for visualising them.

They user tested the implementation in two experiments where one was a video representation of the robots and the other was for direct observation, for both experiments they tested whether having the aid of their graphical solution would increase understanding of the robot's behaviour and they found that for both experiments having a visualisation did in fact increase understanding.

3 Implementation

3.1 Danger Recognition and Avoidance Implementation

Since the navigation of the robot in a populated or even cluttered area is a complex problem, we designed a system which outlines the different required aspects of a full solution. The outlined solution addresses the following broad problem statement:

How to navigate safely from a starting point to a destination in an area densely populated by static and dynamic obstacles?

Based on the related work, there are several different technologies which can be used in similar tasks. We decided to take into consideration the resources required to set up a fully monitored environment, which on one hand could provide the robot with accurate, real-time information about the whole area at all times, however a full environment monitoring system's setup is costly, and in some cases (e.g. open areas linked to buildings) hard to achieve (Nasrollahy and Javadi, 2009). Moreover, much of the information provided by such systems is not needed about entire areas at all times, the perception of the surroundings of the robot is enough for the navigation and avoidance.

3.1.1 Full solution outline

Addressing only solutions for monitoring the surroundings of the robot and taking actions accordingly, the flow described below was designed.

The robot is situated in a starting point in an area which was previously mapped by it. The robot is continuously using simultaneous localisation and mapping (SLAM) to monitor its environment and track its own position within it with a laser distance sensor, and a single camera is mounted on its front. Given a destination, it uses an algorithm for the calculation of the most optimal path, regardless of dynamic and static obstacles on its way there. When following the initially outlined path, it monitors the laser readings and analyses the camera feed. Two machine learning algorithms, trained separately for static and dynamic obstacles, classify the current state and predict the probable states of a short upcoming time period. Compared to the states outlined by Duguleana and Mogan (2016), the 5 states descibed in Table 1 (p. 16) were defined and can be associated between the two implementations.

In case the system detects static, dynamic or both types of obstacles which need to be avoided, an avoidance policy is started, trained by a Q-learning algorithm to get past the obstacle(s). Then the previous wayfinding algorithm replans the path to the destination,

Duguleana and Mogan (2016) states	Our states
Safe	Not in view
Safe	In view, at safe distance
Non-safe	Avoidance needed
Non-safe	Dangerous
Failure	Terminal
Winning	-

Table 1: The defined 5 states and their relation to the ones descibed by Duguleana and Mogan (2016).

and guides the robot until another avoidance is needed or until the destination is reached.

The above described decision making algorithm can be seen in Figure 13 (p. 41).

As mentioned above, in order to classify a situation, two machine learning models are required. One of them classifies static obstacles, and the other handles the dynamic ones. This distinction is needed, since the avoidance policy can vary depending on the type or types of obstacles the robot encounters. To create these models, data collection, cleaning and the training of the model has to be performed. Kim and Do (2012) proposed that a single camera is enough to train an algorithm for obstacle avoidance. This data is paired with the read laser ranges, which are already monitored due to the localisation and mapping (SLAM). They (Kim and Do, 2012) also propose the splitting of the camera feed in case of their monoscopic camera. Since the robot is equipped with the same type of camera, splitting the incoming images in half can improve the classification accuracy. Moreover, to support the side-specific classification of a situation, separate models are trained for the left and the right side angles of the distances as well.

To combine the laser and camera models' predictions, ensemble methods can be used, either creating an averaged value representation of the output classes, or weighting the outputs of the models based on their evaluation accuracies. As a different approach, stacking can be used to train an overall model from the outputs of the distance/image, left/right specific models, given the final label.

For the development of the avoidance policies, the robot needs to be able to handle three distinct situations. When there are static, dynamic and both of those types of obstacles in the way of it. The policies are evolved with a genetic machine learning technique called Q-learning. The Q-learning penalises the time, the distance travelled and the distance from obstacles during the avoidance, and uses variable-length chromosomes for evolving the policy. The set penalty factors are required, so the avoidance is the most optimal in its direction and speed, while it allows the robot to stay far enough

from obstacles. The variable-length chromosomes, according to Tu and Yang (2003) are required during the evolution of the policy to create more optimal paths as the length of the actions the robot can take is not limited by the chromosome length. With fixed-length chromosomes, at a low speed, it could happen that the robot does not have enough actions to make an avoidance.

3.1.2 Final problem statement

Since SLAM, and obstacle-less navigational algorithms are implemented and already used for basic wayfinding tasks, from the described complete solution for wayfinding, we chose the next problem to solve from the navigation process, the implementation of the detection of unsafe scenarios, in order to create a base for the avoidance policy development. Moreover, with the distinction of static and dynamic obstacles, given that static obstacle detection and avoidance has been a well-researched area in the past years, we chose to address the detection of states when encountering dynamic obstacles. The final problem formulation is as follows:

How to detect the need for obstacle avoidance during wayfinding tasks having knowledge only about the surroundings of the robot when encountering dynamic obstacles?

3.1.3 Requirements

In order to create the desired solution, some setup requirements have to be met. These are described in the following sections.

Hardware To allow focusing on the machine learning's implementation, a robot kit was chosen to serve as the hardware base of the project, the Turtlebot 3 Burger (ROBOTIS, nd). This robot was designed to have an easy setup, provide a way of easily adding additional sensors. Moreover, it was created by Robotis to have direct support for the robotic operating system (ROS), allowing a relatively quick software environment setup as well.

The Turtlebot comes equipped with a light detection and ranging (LIDAR) device (360 Laser Distance Sensor LDS-01), which can take measurements multiple times a second in 360 degrees in a single height which can be used for creating SLAM maps and navigation.

Lastly, a Raspberry Pi Camera Module v1 (Raspberrypi.org, nd) was installed with a 3D printed mount on the Turtlebot, which allowed a plug-and-play installation and

which is supported by ROS and its camera calibration library (Wiki.ros.org, nd) as well. Even though the camera is capable of a video output of 1080p quality at 30 fps, due to the limited disk space, only its 640x480 mode was used.

Software In order to set up a development environment with ROS, Ubuntu was installed as an operating system in a virtual machine. The Turtlebot setup was based on the guide they provided and python and the machine learning packages (`Keras` with `Theano` backend) were installed on top of this setup.

To access the robot despite the university network's security, it was set up to connect to a VPN service, where it could connect to the host computer's `roscore`, providing a master server. With this solution, the robot could also be directly operated through ssh from the computer.

Lastly, the camera was calibrated using the script provided by ROS. This is necessary, as the camera needs a configuration file which describes how it should be rotated and what transformation it should apply to a 3D environment's representation to translate it to a 2D image.

3.1.4 Data collection

To create a machine learning model, a dataset has to be created which can be used for training and testing the designed solution.

Despite Sadeghi and Levine (2016) proposing a simulation-based data collection approach, a real-life end-to-end learning is set up. The reason behind it is that the robot is sturdy enough to get over a few crashes during the data collection. Moreover, the environment is also controlled and slow-paced, minimising the risk of harm. Due to these reasons and that a simulation-based approach cannot include natural noise and external factors of the environment.

For the implementation of the data collection, the bootstrapping approach of Shrivastava et al. (2016) is used, which collects samples of all labels in order to train the model and proposes the later addition of samples describing those labels which cannot be reliably classified. In addition, the chosen data collection approach utilises the tactics described by Gandhi et al. (2017), who primarily collected negative examples in order to make their drone recognise unsafe scenarios.

Logging implementation In order to create a database, a logging script is used, which handles reading, processing, and storing of the values from the target ROS nodes of the

```
22  class L2PC:
23      def __init__(self, csv_handler):
24          self.csv_h = csv_handler
25          self.laser_topic = '/scan'
26          self.camera_topic = '/raspicam_node/image/compressed'
27          self.laserSub = rospy.Subscriber(self.laser_topic, LaserScan, self.laser_call)
28          self.imageSub = rospy.Subscriber(self.camera_topic, CompressedImage, self.img_call)
29
30          self.laser_msg = None
31          self.camera_msg = None
32
33          self.timestamp = ''
34          self.np_ranges = None
35          self.label = '0'
36
37          self.last_call = tt()
38
39      def run(self):...
55
56      def laser_call(self, lsr):
57          self.laser_msg = lsr.ranges
58
59      def img_call(self, msg):
60          self.camera_msg = msg
61
```

Figure 1: Subscriber and callback setup script.

robot.

In order to handle the data published by the camera and the laser sensor into their nodes on the ROS master server, subscribers are set up to monitor value changes in these nodes. Therefore, the script sets up a `Subscriber` for the camera's and the laser sensor's node, with their own `Callback` functions. A callback function is triggered by the subscriber when the values published to a node change. The callbacks handle the acquisition of relevant data for the logging. In case of the laser's callback, only the ranges are kept from the `LaserScan` class's headers, while the image callback saves the full data passed to it as a `CompressedImage` class. The subscriber and callback setup can be seen in Figure 1 (p. 19).

To create a database, the `csv` package is used, which allows the writing data to a delimited file row by row. After setting up the headers, the file is closed and waits for reopening, when the script appends its rows with new recorded data.

To regulate how often the script records data, the script uses a custom `run` function, which sets the run rate of its loop to 4 times a second. After every quarter of a second, when the loop is done with processing and saving the gathered data, it uses `sleep` to wait until the next iteration can be started. The subscriber callbacks work asynchronous to this behaviour, meaning that they can be called by the subscribers multiple times before the handling of the read data begins again in the next iteration. Each iteration sets a new timestamp as well, so the saved images and the entries in the database can later be related to each other. The `run` function can be seen in Figure 2 (p. 20).

Since the images are published to the node as `CompressedImage` classes, the image

```
def run(self):
    r = rospy.Rate(4)
    while not rospy.is_shutdown():
        if not self.laser_msg == None:
            if not self.camera_msg == None:
                self.timestamp = dt.now().isoformat()
                self.laser_process()
                self.img_process()
                # print('Label:', self.video_label)
                # print('PROCESSED at: '+str(tt()))
                self.generate_entry()
            else:
                rospy.logwarn('No cam input')
        else:
            rospy.logwarn('No laser input')
        r.sleep()
```

Figure 2: The run function of the logging script.

processing first needs to take care of decompressing the image. After the image data is converted into a `numpy array` from its compressed string format, it can be loaded as an `opencv image` by decoding the received string. To save the image, it has to be compressed with an `opencv` method, then encoded and converted into a string to be written into a `png` file. The code for image processing can be seen in Figure 3 (p. 20).

To process the ranges, the received array of 360 range values have to be formatted. Firstly, since the LDS outputs 0.0 when it cannot detect anything within its set viewing distance (3.5 meters by default), these values are reset to be a uniform value out of the

```
def img_process(self):
    msg = self.camera_msg
    png = self.from_msg(msg)
    path = '../img/' + self.timestamp + '.png'
    with open(path, 'wb') as image:
        image.write(png)
        print('Saved png')

def from_msg(self, msg):
    if isinstance(msg, CompressedImage):
        # Decompress message.
        msg = self.from_raw(msg.data)

    # Convert ROS Image to OpenCV image.
    bridge = CvBridge()
    img = bridge.imgmsg_to_cv2(msg)

    # Convert to PNG, highest compression
    compression = [cv2.IMWRITE_PNG_COMPRESSION, 9]
    png = cv2.imencode(".png", img, compression)[1].tostring()

    return png

def from_raw(self, raw):
    # Convert to OpenCV image.
    nparr = np.fromstring(raw, np.uint8)
    img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)

    # Convert to ROS message.
    bridge = CvBridge()
    msg = bridge.cv2_to_imgmsg(img)

    return msg
```

Figure 3: The script used for preprocessing images.

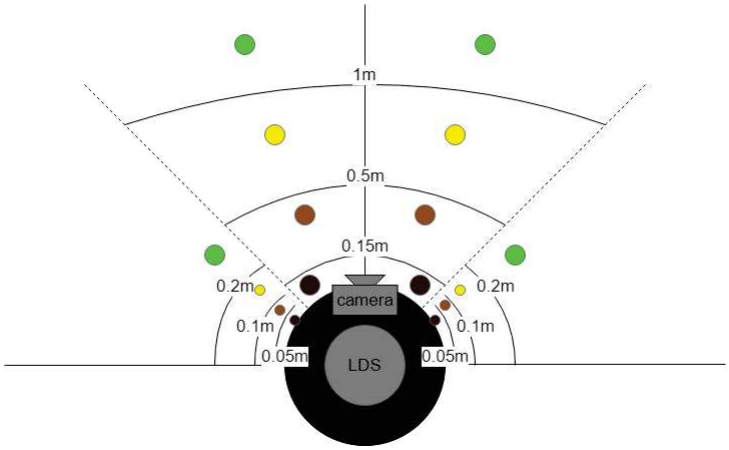

Figure 4: The assignment of labels in relation to the distances for the inner and outer angles. The colour coding of the states' circles is as follows: Green – 'In view, at safe distance', Yellow – 'Avoidance needed', Brown – 'Dangerous', Black – 'Terminal'. The 'Not in view' state is not represented.

maximum reading distance to avoid confusion. Moreover, the rear range readings are discarded before the front-facing 180 degrees are split into left-right, then inner-outer, equal areas. The four constructed arrays are then used to create a new `numpy array`, which can be sent to the `entry_generation` part of the script.

To assign labels, the laser ranges were used. Four thresholds were set up to distinguish the desired five labels. The thresholds have a difference depending on whether the label assignment was based on the inner ranges or the outer ones. For each side, the code looks through the inner ranges first, searching for values within the set boundaries, and if it did not find any value out of the checked 45 degrees which would classify as closer than the 'Safe – Within view' label's threshold, then it checks the outer ranges with their thresholds. The inner ranges' checking is of higher importance, since if an obstacle needs to be avoided in the robot's heading direction, it usually requires a quicker response due to the forward speed of the robot. However, if an obstacle is dangerously close or approaching from a side, eventually resulting in a crash, the robot should be able to issue an avoidance policy. The relation of range thresholds and labels for the inner and outer degrees can be seen in Figure 4 (p. 21). Lastly, the final label is decided based on which side concluded a more dangerous scenario describing the timestamp. The script for label calculation can be seen in Figure 5 (p. 22).

After an image has been saved for a timestamp and the ranges have been split, then a label has been calculated, the information is assigned to a dictionary variable, which is used to create a new array of data that is appended to the database file. The script used for appending the database can be seen in Figure 6 (p. 22).

```
def label_calc(self, l_r):
    label = 0
    terminal_range = [0.15, 0.05]
    unsafe_range = [0.5, 0.1]
    avoid_range = [1, 0.2]
    max_view_range = 3.5
    for range_i in self.np_ranges[0][l_r]:
        if range_i <= terminal_range[0]:
            label = 4
            break
        elif terminal_range[0] < range_i <= unsafe_range[0]:
            label = 3
        else:
            if unsafe_range[0] < range_i <= avoid_range[0]:
                label = 2
            elif avoid_range[0] < range_i:
                for range_o in self.np_ranges[1][l_r]:
                    if range_o <= terminal_range[1]:
                        label = 4
                        break
                    elif terminal_range[1] < range_o <= unsafe_range[1]:
                        label = 3
                    else:
                        if unsafe_range[1] < range_o <= avoid_range[1]:
                            label = 2
                        elif avoid_range[1] < range_o < max_view_range:
                            label = 1
                        else:
                            label = 0
    return label
```

Figure 5: The script used for calculating the labels for the database.

Environment and collected data For the collection of data two corridor areas were chosen at the University building. The two areas can be seen in Figure 7 (p. 23) and Figure 8 (p. 23).

The criteria for choosing a fitting area included:

- Having as low amount of static obstacles as possible – since the model should be trained for recognising the need for avoiding dynamic obstacles.
- People and objects have to pass through – to have a diverse dataset.

Dynamic obstacles have to be able to pass by the robot – so the area has to be wide enough for avoidance, but narrow enough so the robot or people have to perform

```
def append_db(self, data_dict):
    with open(self.csv_path, 'a') as csvfile:
        writer = csv.writer(csvfile, delimiter=self.delim)
        to_csv = [
            data_dict['Time'],
            data_dict['Inner_ranges'],
            data_dict['Outer_ranges'],
            data_dict['Image_name'],
            data_dict['Label_l'],
            data_dict['Label_r'],
            data_dict['Label']
        ]
        writer.writerow(to_csv)
        print('%s - Added sample with label %s' % (data_dict['Time'], data_dict['Label']))
```

Figure 6: The script used for appending the database with a new sample.

Figure 7: Area 1 used for sample collection from the robot's perspective.

avoidance instead of taking very distant trajectories.

Throughout data collection, the robot was manually controlled to patrol the chosen areas on different routes and at different speeds. This way, 300 safe walk-by samples and 300 walk-by samples resulting in a crash were recorded in both areas. Due to people and our research group walking by from different angles, on different trajectories and at different speeds, the resulting database can be used to create a general model for the classification of the outlined states.

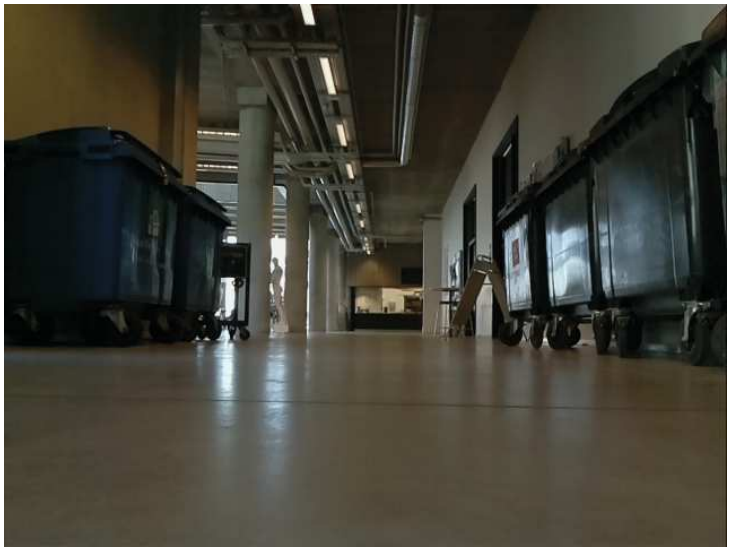

Figure 8: Area 2 used for sample collection from the robot's perspective.

3.1.5 Danger recognition implementation

This section describes the data processing and model training of the solution.

Preprocessing To create a complete dataset, the samples collected in the two environments had to be merged together. The primary reason why the data was recorded into separate files is the two-location setup of the data collection, which in itself meant two different `csv` file outputs. However, since a `csv` is produced every time the logging script is ran, when issues arose during data collection at any of the locations, the logging had to be restarted, resulting in multiple databases. Issues halting the data collection were, among others, the robot's battery being depleted, dropping WiFi connection to the building's network, and, upon reconnection, being given a new IP address by the VPN service.

After joining the separate `csv` files, the image file name column was iterated and matched to the list of images in their folder. If an image was not found in the database, then it was deleted. Images without database entries originated from failed logging attempts, where the image or the ranges were not properly received from the ROS node (primarily due to bad connection or the host machine's latency).

Following the deletion of images, they were split in half, with a 10% percent overlap in the middle for both sides.

Lastly, the nested array representation of the distances was fixed for quicker readability. To do so, the inner and outer range columns were iterated and their held values appended to a new database, which held the headers of the single degrees the measured distances can be associated with.

Models As described before, the left and the right sides of the samples are handled separately by the system, therefore separate models were trained for them, although with the same model structure.

The key difference is between the classification approach of the laser ranges and the side-specific images. For the classification of the distances, a general, 2 hidden layer, deep neural network was written. In its hidden layers, the activation was set to "tanh", a re-scaled alternative of the regular sigmoid function. These layers also allow a 50% dropout rate and lead into a dense output layer. A representation of the deep neural network can be seen in Figure 14 (p. 42) and Figure 15 (p. 43). This model was trained on 80% of all the collected samples, which training set was further split into a training and a validation set in an 8:2 ratio. The training was compiled with mean squared error loss calculation

and ‘adadelta’ optimizer, a learning rate method based on gradient descent. Moreover, it was allowed to run a maximum of 50 epochs in order to avoid overfitting, however it never reached this hard limit due to the implemented early stopping monitoring the value loss between epochs. The training data was used in 25 sample batches for the training.

For the classification of the camera images a convolutional neural network was designed modified from a self-driving car navigation training, used by NVidia, described by Bojarski et al. (2016). Their model is based on a three-camera setup, therefore it is not fully applicable for the setup of the Turtlebot, however it was followed for the processing and training of images. The model consists of 5 convolutional feature map layers, which are flattened and passed lead into 3 fully connected layers of decreasing node counts. A representation of the convolutional neural network can be seen in Figure 16 (p. 44) and Figure 17 (p. 45). The model was compiled to calculate mean squared error loss, using the ‘Adam’ optimizer, another gradient descent based learning rate method commonly used in image classification tasks. For our implementation, we trained the same neural network for both sides of camera recordings, 20.000 samples per epoch taken from the training set of the respective side’s images, and another 500 samples taken from the remaining entries in the training set for validation. To avoid overfitting, a hard limit of 10 epoch were set, as well as an early stopping, which monitored the value loss between epochs. Since the training of a simple epoch took over 8 hours, model checkpoints were implemented, which saved the calculated weights after each epoch.

The training of the convolutional neural networks took almost 30 hours each on an NVidia GeForce GTX 950M video card. The implemented early stopping detected no significant improvement during the training of the left and right sides after the 4th and 3rd epochs respectively. The weight calculation for the deep neural networks was quicker, even with all the training data (roughly 30.000 samples) used in each epoch. The training time for these models were around 10 minutes, and since the early stopping could not detect significant improvement, the trainings were shut down after the 7th and 11th epochs for the left and right sides of the laser distance values respectively.

3.2 Visualising Robot Interactions

Since the intent was to use the robot in a populated area and use cases could require novel users to oversee the robot and report unwanted interactions or errors to developers, we designed a tool that does not only provide users the option to monitor but also to interact with the robot. The ideal solution addresses the following broad problem statement:

How can a GUI be designed to aid novel users in understanding robot interactions

while being able to intervene and provide base instructions?

The related work shows a variety of solutions for both monitoring and task management and as mentioned earlier, we do not take a fully monitored environment into account, meaning that our visual tool will only utilise the input coming directly from the robot.

3.2.1 Ideal solution

An ideal solution is constructed from the notion that users should be able to provide the robot with a task and understand the interactions it has while performing it. This does not mean that users will have to understand the nature of a neural network but rather understand the input that the robot receives and its following actions. The visualisation tool will have to provide users with the following:

- Planned path from point A to point B
- Feedback from object avoidance
- Notification of safety states
- Point of view (POV) of the robot
- Distance calculations

Showing the planned path requires a map which can be obtained from the SLAM map the robot creates at runtime. Preferably the map would be pre-planned as the path could update faster. The feedback from the object avoidance can be implemented by adding the new path calculated from the robot when making an avoidance while maintaining the original path thereby showing users that the robot deviated from the path making it obvious that an avoidance occurred. The notification of the safety states of the robot is derived from Wortham et al. (2017a) and it is believed that it will improve the understanding of the interactions. The point of view of the robot is believed to influence understanding as users tend to not understand robot behaviour if they cannot see it. Lastly, the distance values are presented so that users can detect that objects are indeed too close or getting closer as the values change.

The ideal solution consist of a window where the screen is split in two and on the right side the map is complete with the option of clicking on it anywhere, this way giving a task for the robot to move to the desired location. The map then shows the optimal route between the current location of the robot and the destination and users have the option

of deciding whether the robot needs to take a detour by assigning intermediate locations. The left side of the screen is split in two, where the top shows the possible safety states through colour codes and these colours are related to the path as the robot travels from point A to B. Lastly, the bottom right side shows the point of view, where the distance measurements of the laser distance sensor are overlaid in the image and colour coded relative to the measured distance. This provides users with the notion of objects being too close to the robot.

3.2.2 Final problem statement

Since the solution will only use two outputs from the robot concerning avoidance, namely, current safety state and distance measurement, the problem formulation can be altered to the following:

How can a robot's interactions be communicated through a GUI using only the safety states and distance measurements the robot acquires at runtime?

3.2.3 Iterations

To create a solution that can fulfil the problem statement we decided to use Qt Creator from Qt Company, which is an Integrated Development Environment (IDE) for creating GUIs. Qt was chosen as it can work with ROS, which makes it suitable for making a visualisation tool that can operate at real time with data provided by the robot. Furthermore, it supports shader effects, which makes it valid for video manipulation for the distance measurement effect described earlier.

However, due to unforeseen events, the ROS dependencies could not be installed. As a result, we had to reconfigure the design since we could not communicate with the robot at real time.

The new solution was to make a playable demo for testing the concept of users having an improved understanding of the robot interactions based on inputs. This meant that instead of a real time video stream, we would have to embed a video into the GUI and instead of live data being streamed the solution, we had to use a `csv` file for reading the distance and safety state data.

Qt Creator works in multiple ways, and the chosen base for the solution was to use `QML`, which is a programming language where GUI elements are described directly in code. `QML` works with both `Javascript`, `python` and `C++` and is a `json`-like tool used for the creation of dynamic, animated application interfaces.

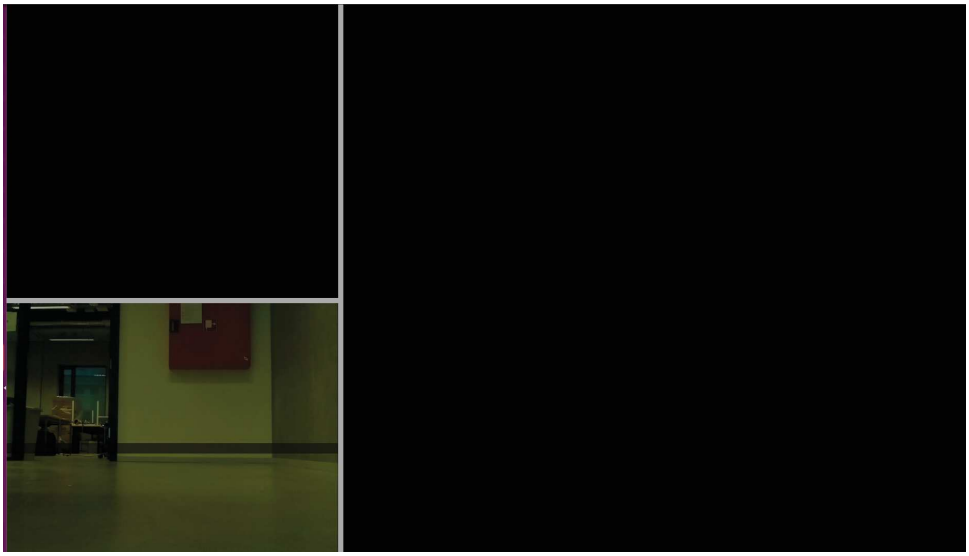

Figure 9: Implementation of the visualiser in Qt Creator. As it can be seen, the only thing implemented is a video player which shows the robot's POV there is a black semi-transparent line through the image which acts as a placeholder for the distance marker. However, the implementation of the shader to handle the distance values failed due to unfamiliarity with `QML` and the implementation using that IDE was discarded.

For the implementation in Qt Creator, we chose to follow the ideal solution where the screen is divided into three windows and the bottom left corner is showing the POV of the robot. An image of the created screen can be seen in Figure 9 (p. 28).

Unity3D was chosen as a tool for the implementation due to familiarity with the game engine. The implementation in Unity can be divided into three parts:

- SLAM map
- State changer
- Robot's POV with overlaid distance data

Unity does not originally support ROS integration and therefore the general idea of implementing a playable demo of the visualisation tool still applies.

The visualisation tool is implemented using `Unity UI`, where a canvas act as a container for all elements. The implementation of the SLAM map is achieved by creating a raw image and attaching a `Video Player` component, where a pre-recorded video of the map is playing at runtime.

The state changer part of the implementation consists of two images, where the opacity is 50% when the state is not active, and 100% when it is, and the data used for determining

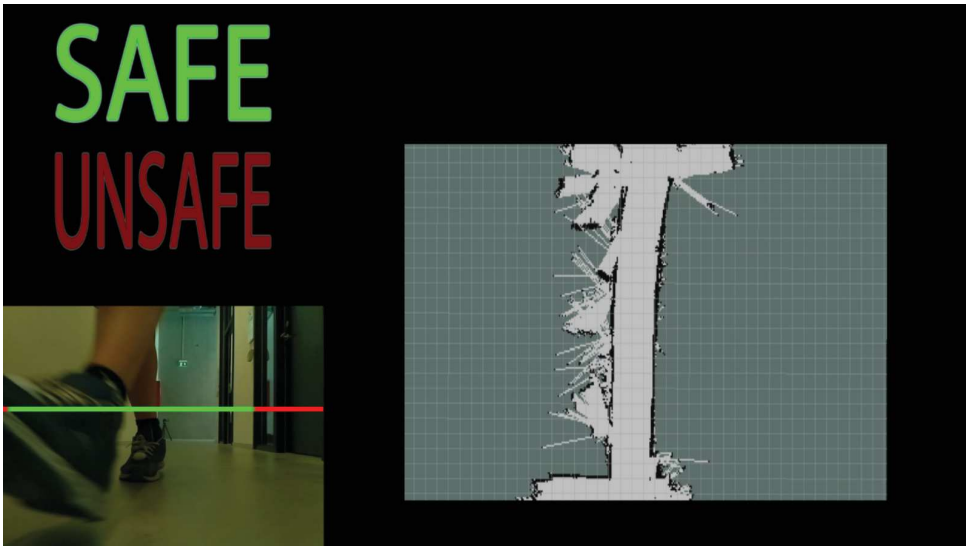

Figure 10: Visualisation tool complete with SLAM map, state changer and video feed with overlaid distance data.

which state is active is read from a `csv` file containing the state labels for each frame corresponding to the video feed of the SLAM map. Similar to the SLAM map, the robot's POV also consists of a raw image with a video player component attached. The difference lies in the overlaid distance values. An image of the implementation can be seen in Figure 10 (p. 29).

In order to show the overlaid distance data we first had to calculate which angles were overlaid on the Pi Camera's field of view. This was done by identifying the amount of horizontal view angles (54 degrees) of the Pi Camera and relating them to the LDS's angles. Since the distances are indices from 0 in the full array, and we measured a 5 degree clockwise offset between the centre of the distance sensor and the camera's image, the indices from 336 until 359 and 0 till 32 are taken from the distance readings. Next, we create a simple shader for locating the texture coordinates, and applying a colour to them using a texture script to attach to a `line renderer` component of an empty `GameObject` in Unity. However, even though everything is implemented, the `line renderer` does not update its colour values. The possible corrections for this issue are covered in the discussion (see Section 7.2, p. 35). Unfortunately, this also means that there is no evaluation of the implementation, also due to the fact that the robot could not accurately communicate its recognised states.

```
234 def load_model(path, w_path, cnn):
235     with open(path) as m_json:
236         model = model_from_json(m_json.read())
237     if cnn:
238         model.compile(loss='mean_squared_error', optimizer=Adam(lr=1.0e-4))
239     else:
240         model.compile(loss='mean_squared_error', optimizer='adadelta', metrics=['accuracy'])
241 
242     # / reading the weights of the stored model and adding them to the compiled model
243     model.load_weights(w_path)
244     return model
```

Figure 11: The script of the load_model function.

4 Evaluation

4.1 Testing of Danger Recognition

To test the danger detection's classification accuracy and map points of possible improvement, two different evaluations are performed. One, taking the created test set of the full dataset of over 9000 samples and using the built-in evaluation functionality of `Keras`, and another one, using the loaded and weighted model in ROS, classifying the situation of the robot in a real-life experiment.

4.1.1 Programmatic evaluation

The programmatic evaluation has over 9000 samples due to the 80-20 split of the full dataset. For the DNNs classifying the left and right side data feed of the laser distance sensor, the data has to be reloaded straight from the dataset's `csv` file. Then the structure of the model needs to be loaded from a `json` file, compiled and the calculated weights need to be assigned to it. All this is handled by the `load_model` function, as it can be seen in Figure 11 (p. 30). Once the data is organised into the proper shape and the model is prepared for evaluation, the `evaluate` method of `Keras` can be used, which takes the features of the dataset, uses the model to create a classification and compares it to the true label.

To evaluate the CNNs for the two sides, the same number of samples are used, however when loading the data from the database file, the images need to be read, resized, and their colour coding converted from RGB to YUV before they can be used with the loaded model and weights. For the loading and preprocessing of the images, the same `utils` script's methods were used as during the training of the model, and for the model preparation, the `load_model` method was called with a different signature. To evaluate the model, the same evaluate method was used as for the DNNs.

```
206 class IoHandler:
207     def __init__(self):
208         self.prev_label = 0
209         self.print_counter = 0
210         ts = dt.now().isoformat()
211         self.csv_path = '../csv/'+ts+'-qtdump.csv'
212         self.delim = ';'
213         if not os.path.isfile(self.csv_path):
214             self.create_db()
215         self.model = None
216         self.load_model_weigths()
217 
218     def create_db(self):
219         with open(self.csv_path, 'wb') as csvfile:
220             writer = csv.writer(csvfile, delimiter=self.delim)
221             header_array = [
222                 'Time',
223                 'Label',
224             ]
225             for deg in range(-26, 26):
226                 header_array.append(str(deg))
227             print('Header length: %d' % len(header_array))
228             writer.writerow(header_array)
229 
230     def append_db(self, data_dict):
231         with open(self.csv_path, 'a') as csvfile:
232             writer = csv.writer(csvfile, delimiter=self.delim)
233             to_csv = [
234                 data_dict['Time'],
235                 data_dict['Label']
236             ]
237             for distance in data_dict['Ranges']:
238                 to_csv.append(distance)
239             writer.writerow(to_csv)
```

Figure 12: The script used for logging distance values and labels during the evaluation.

4.1.2 Real-life evaluation

The real-life experiment required the model and weights to be loaded in a ROS script, on a different python version. Since the output structure of the saved model does not support the loading of a model exported under `python 3.6` into a script running with `python 2.7`, instead of using the `load_model` method, the same model creation methods were used as described in section 3.1.5 (p. 24). After the building and compilation of the neural networks, the weights can be loaded and a transformed logging script is initialised, holding the timestamp, label and 26 distance values for each side from the `inner_ranges` arrays. This log file is used to provide simulated live data for the implementation of the visualisation part of the project. The script for logging can be seen in Figure 12 (p. 31).

Apart from saving the full images, the script has to handle the left-right splitting and preprocessing in order to provide the correct input shape for the model. In order to produce the correct shape for the laser distances, after populating the left and right sides' lists with the distance values, they have to be converted to `numpy arrays`.

To get the current label describing the situation of the robot, the prepared `numpy arrays` are used in the `prediction` method of `Keras`, which gives an array of class probabilities, therefore `argmax` has to be used to get the label the model is most confident

in. Deciding which label should be the final one is done by taking the highest label from the four models' classifications.

During the evaluation the robot, while being manually controlled through the built-in phone-based teleoperation, patrols the corridor it was trained in, while a single test participant walks up to it from the front performing three types of actions:

- Avoiding the robot at a safe trajectory
- Nearly hitting the robot, but avoiding it
- Bumping into the robot

These actions are repeated multiple times by the participant, trying multiple incoming speeds, angles and sides. During the experiment the robot is also travelling at various velocities and on different paths along the corridor.

5 Results

5.1 Testing of danger recognition

The built-in `evaluation` method of `Keras` showed that three of the models achieved an around 75% classification accuracy when tested on over 9000 samples. Moreover, the left-side laser distance classification showed the highest, 96% prediction accuracy. The calculated evaluation accuracies are presented in the following table:

Model	$accuracy * 100$
Left CNN	74.236%
Right CNN	74.238%
Left DNN	96.816%
Right DNN	79.344%

The real-life evaluation of the model showed less promising results and revealed unexpected issues with the integration of the models.

Firstly, since the computer used as the host of the ROS environment was not powerful enough, the reading and handling of distance and image data was slower than expected due to the number of ROS launch files running on the system, taking away from its computational power.

Secondly, the script used to integrate the trained models and use them for classification could not utilise the GPU of the computer because its dual-gpu setup did not allow the detection of the NVidia graphics card in the Ubuntu Virtual Machine, and therefore the machine learning libraries ran by the script could only use a single core of the CPU. This made the classification slower, between 70-250ms per model, for all models, reducing the intended 4 processed frames per second to only 1.

Moreover, the classifications for all models in all tested situations showed that the robot did not experience any of the unsafe states, which makes the reliability of the programmatic evaluation questionable and fosters investigation of the reason for high accuracy values in light of the unreliability of the model during the real-world evaluation.

Lastly, due to the poor results of the model-integrated system's test, the test participant did not perform the outlined actions the planned number of times. Because of the poor performance in a known environment, the testing of the danger recognition was not extended, to be performed in an environment it was not trained in, either.

6 Conclusion

6.1 Danger recognition

The final problem statement for the danger recognition part of the designed avoidance solution stated as follows:

How to detect the need for obstacle avoidance during wayfinding tasks having knowledge only about the surroundings of the robot when encountering dynamic obstacles?

The designed solution for this problem presented the setup of a multi-sensor system relying on laser distance measurements and a camera feed, both monitoring the surroundings of a robot.

In order to recognise situations where the robot should perform obstacle avoidance, a dataset was accumulated of dynamic obstacles causing danger-state changes when approaching the robot in various ways. The dataset was then used to create a four-model, neural-network-based implementation to use both sensory inputs for the classification of a situation.

Even though the programmatic evaluation of the designed solution showed good results, the real-life testing highlighted major problems with it. Therefore, the solution in its current state is unable to detect the need for obstacle avoidance. The possible ways to improve the danger recognition are outlined in section 7.1 (p. 34) of this report.

7 Discussion

7.1 Danger recognition improvements

In order to improve the data collection and the training of the models, several steps should be considered. Firstly, the amount of collected samples and the classification accuracy of each label should be compared. If for most evaluated samples of a label the classification is not correct, as proposed by Shrivastava et al. (2016), the sample base of the label has to be enriched.

To better combine the predicted labels, an ensemble method could be used to take the classification certainty and evaluated accuracy of the model into account instead of always choosing the most dangerous classification, which, if one of the models is less accurate, could lead to an overly-cautious danger detection. A better way of combining the outputs of the models would be to utilise stacking, training a 5th model on top of the 4 initial ones, utilising the predicted labels, certainties and evaluated accuracies of the models in order to have a single label as the output of the stacked model's classification. Moreover, in order to predict what situation the robot will be in, the model outputs with or without the aforementioned improvements can be used to train a long short-term memory (LSTM) model. This improvement possibility is based on the research done on activity recognition from video footages with convolutional neural networks by Donahue et al. (2014), who propose the combination of CNN based feature extraction and the use of recurrent neural networks, specifically LSTM, creating a so called long-term convolutional network (LRCN or, as later coined, CNN-LSTM). Lastly, the parameters used for the creation of the implemented neural networks should be revisited and optimised with hyperparameter optimisation. In particular, the effect of changing the number of hidden layers in both networks, and the node counts of the densely connected layers should be investigated as well as how the deep neural network's classification accuracy improves with different dropout rates.

To improve on the integration of the created solution, a different host machine should be used which is capable of handling the data read and written to the ROS nodes for better update rates. The used computer should be set up so the machine learning libraries can use the GPU for classification. Moreover, the start-up time of the solution could be lowered if the model structures would be exported from a `python 2.7` version, in which case they could be read from `json` files without the python-version related errors, as it was also implemented with the `load_model` method in case of the evaluation of the models. In addition, if the visualisation gets the information directly from the ROS nodes instead, the logging of distance values and saving of images becomes redundant, making

the classification of situations faster. Lastly, the image splitting and preprocessing part of the data preparation should be refactored and optimised for better performance, resulting in better recognition frame rates.

Improving the programmatic evaluation can be done by calculating a label-frequency metric describing the training and test sets, which can highlight the labels in need of more samples for a more reliable recognition rate. In the aforementioned task, the creation of confusion matrices as the bi-product of the built-in evaluation can serve as another guide, since with them the labels with low classification accuracies can be identified. Moreover, the built-in evaluation could be performed with a more controlled test split, holding a set, equal amount of entries for all labels.

The real-life evaluation procedure can be improved by involving more participants and different places, where the robot was not trained. Moreover, the robot should create a SLAM map before the testing and use the 'gmapping' algorithm with SLAM to automatically navigate to its set destination while assessing the situation with the improved danger recognition solution.

7.2 Visualisation Tool Improvements

After analysing the problems with the current implementation, the following points of improvement can be outlined.

In order to test if the visualisation tool can improve the understanding of robot interaction for novel users, the overlaid distance data need to be transparent. This means that the colours corresponding to the two states need to update while using the tool. In regards to the state data, more samples for the robot would fix the issue of the states not updating. At the current implementation level, updating the right colour values works, but the resulting colour is missing. This could be because the material is baked and never receives an update call. Another possible solution would be to create a new material each update frame and assign the correct texture to it, then apply it to the `line renderer`. However, further research needs to be conducted specifically into possible solutions to solve this issue.

Another point of improvement is to be able to communicate with the robot, getting real-time data from the robot is crucial for visualisation tool to work in a proper use case. This would not only allow implementation of controls for users to interact with the robot, but also to incorporate an accurate SLAM map. To achieve this we can use a ROS library for Unity, however, this needs to be investigated further.

If the distance data is correctly implemented we can evaluate the solution with novel

users. Such a test would be a comparison between the presence and the lack of a visual aid, meaning that participants would try either options. This would provide grounds for a comparison when questions are asked about the robot's behaviour at given time intervals e.g. during a successful or failed evasion.

References

Amershi, S., Chickering, M., Drucker, S. M., Lee, B., Simard, P., and Suh, J. (2015). ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 337–346, New York, NY, USA. ACM.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. (2016). End to End Learning for Self-Driving Cars. *arXiv:1604.07316 [cs]*. arXiv: 1604.07316.

Chadalavada, R. T., Andreasson, H., Krug, R., and Lilienthal, A. J. (2015). That's on my mind! robot to human intention communication through on-board projection on shared floor space. In *2015 European Conference on Mobile Robots (ECMR)*, pages 1–6.

Daftry, S., Zeng, S., Bagnell, J. A., and Hebert, M. (2016). Introspective Perception: Learning to Predict Failures in Vision Systems. *CoRR*, abs/1607.08665.

Donahue, J., Hendricks, L. A., Rohrbach, M., Venugopalan, S., Guadarrama, S., Saenko, K., and Darrell, T. (2014). Long-term Recurrent Convolutional Networks for Visual Recognition and Description. *arXiv:1411.4389 [cs]*. arXiv: 1411.4389.

Duguleana, M. and Mogan, G. (2016). Neural networks based reinforcement learning for mobile robots obstacle avoidance. *Expert Systems with Applications*, 62:104–115.

Gandhi, D., Pinto, L., and Gupta, A. (2017). Learning to Fly by Crashing. *arXiv:1704.05588 [cs]*. arXiv: 1704.05588.

Han, W.-G., Baek, S.-M., and Kuc, T.-Y. (1997). Genetic algorithm based path planning and dynamic obstacle avoidance of mobile robots. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 3, pages 2747–2751 vol.3.

Henry, P., Vollmer, C., Ferris, B., and Fox, D. (2010). Learning to navigate through crowded environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 981–986.

Jaradat, M. A. K., Al-Rousan, M., and Quadan, L. (2011). Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1):135 – 149.

Kahn, G., Villaflor, A., Pong, V., Abbeel, P., and Levine, S. (2017). Uncertainty-Aware Reinforcement Learning for Collision Avoidance. *arXiv:1702.01182 [cs]*. arXiv: 1702.01182.

Kim, B. and Pineau, J. (2016). Socially Adaptive Path Planning in Human Environments Using Inverse Reinforcement Learning. *International Journal of Social Robotics*, 8(1):51–66.

Kim, J. and Do, Y. (2012). Moving Obstacle Avoidance of a Mobile Robot Using a Single Camera. *Procedia Engineering*, 41:911 – 916.

Kruse, T., Pandey, A. K., Alami, R., and Kirsch, A. (2013). Human-aware robot navigation: A survey. *Robotics and Autonomous Systems*, 61(12):1726 – 1743.

Large, F., Laugier, C., and Shiller, Z. (2005). Navigation Among Moving Obstacles Using the NLVO: Principles and Applications to Intelligent Vehicles. *Autonomous Robots*, 19(2):159–171.

LeCun, Y., Muller, U., Ben, J., Cosatto, E., and Flepp, B. (2005). Off-road Obstacle Avoidance Through End-to-end Learning. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'05, pages 739–746, Cambridge, MA, USA. MIT Press.

Liu, K., Sakamoto, D., Inami, M., and Igarashi, T. (2011). Roboshop: Multi-layered Sketching Interface for Robot Housework Assignment and Management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 647–656, New York, NY, USA. ACM.

Milde, M. B., Blum, H., DietmÃijller, A., Sumislawska, D., Conradt, J., Indiveri, G., and Sandamirskaya, Y. (2017). Obstacle Avoidance and Target Acquisition for Robot Navigation Using a Mixed Signal Analog/Digital Neuromorphic Processing System. *Frontiers in Neurorobotics*, 11.

Minguez, J. and Montano, L. (2004). Nearness diagram (ND) navigation: collision avoidance in troublesome scenarios. *IEEE Transactions on Robotics and Automation*, 20(1):45–59.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533H.

Nasrollahy, A. Z. and Javadi, H. H. S. (2009). Using Particle Swarm Optimization for Robot Path Planning in Dynamic Environments with Moving Obstacles and Target. In *2009 Third UKSim European Symposium on Computer Modeling and Simulation*, pages 60–65.

Pinto, L. and Gupta, A. (2015). Supersizing Self-supervision: Learning to Grasp from 50k Tries and 700 Robot Hours. *CoRR*, abs/1509.06825.

Plate, T. A., Bert, J., Grace, J., and Band, P. (2000). Visualizing the Function Computed by a Feedforward Neural Network. *Neural Computation*, 12(6):1337–1353.

Raspberrypi.org (n.d.). Camera Module - Raspberry Pi Documentation. `https://www.raspberrypi.org/documentation/hardware/camera/README.md`.

ROBOTIS (n.d.). ROBOTIS e-Manual. `http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`.

Ross, S., Melik-Barkhudarov, N., Shankar, K. S., Wendel, A., Dey, D., Bagnell, J. A., and Hebert, M. (2013). Learning monocular reactive UAV control in cluttered natural environments. In *2013 IEEE International Conference on Robotics and Automation*, pages 1765–1772.

Sadeghi, F. and Levine, S. (2016). CAD2rl: Real Single-Image Flight without a Single Real Image. *CoRR*, abs/1611.04201.

Sakamoto, D., Honda, K., Inami, M., and Igarashi, T. (2009). Sketch and Run: A Stroke-based Interface for Home Robots. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 197–200, New York, NY, USA. ACM.

Shrivastava, A., Gupta, A., and Girshick, R. B. (2016). Training Region-based Object Detectors with Online Hard Example Mining. *CoRR*, abs/1604.03540.

Sisbot, E. A., Marin-Urias, L. F., Alami, R., and Simeon, T. (2007). A Human Aware Mobile Robot Motion Planner. *IEEE Transactions on Robotics*, 23(5):874–883.

Srinivas, S., Kermani, R., Kim, K., Kobayashi, Y., and Fainekos, G. (2013). A graphical language for LTL motion and mission planning. In *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 704–709.

Tu, J. and Yang, S. X. (2003). Genetic algorithm based path planning for a mobile robot. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 1, pages 1221–1226 vol.1.

Wang, X., Yadav, V., and Balakrishnan, S. N. (2007). Cooperative UAV Formation Flying With Obstacle/Collision Avoidance. *IEEE Transactions on Control Systems Technology*, 15(4):672–679.

Wei, W., Kim, K., and Fainekos, G. (2016). Extended LTLvis motion planning interface. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 004194–004199.

Wiki.ros.org (n.d.). camera_calibration/Tutorials/MonocularCalibration - ROS Wiki. `http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration`.

Wortham, R. H., Theodorou, A., and Bryson, J. J. (2017a). Robot Transparency: Improving Understanding of Intelligent Behaviour for Designers and Users. In *Towards Autonomous Robotic Systems*, Lecture Notes in Computer Science, pages 274–289. Springer, Cham.

Wortham, R. H., Theodorou, A., Bryson, J. J., and web support@bath.ac.uk (2017b). Improving robot transparency:real-time visualisation of robot AI substantially improves understanding in naive observers. In *IEEE RO-MAN 2017*, Pestana Palace Hotel. University of Bath.

Zhang, H., Han, X., Fu, M., and Zhou, W. (2016). Robot Obstacle Avoidance Learning Based on Mixture Models. *Journal of Robotics*, 2016:14.

Appendix

A Large figures

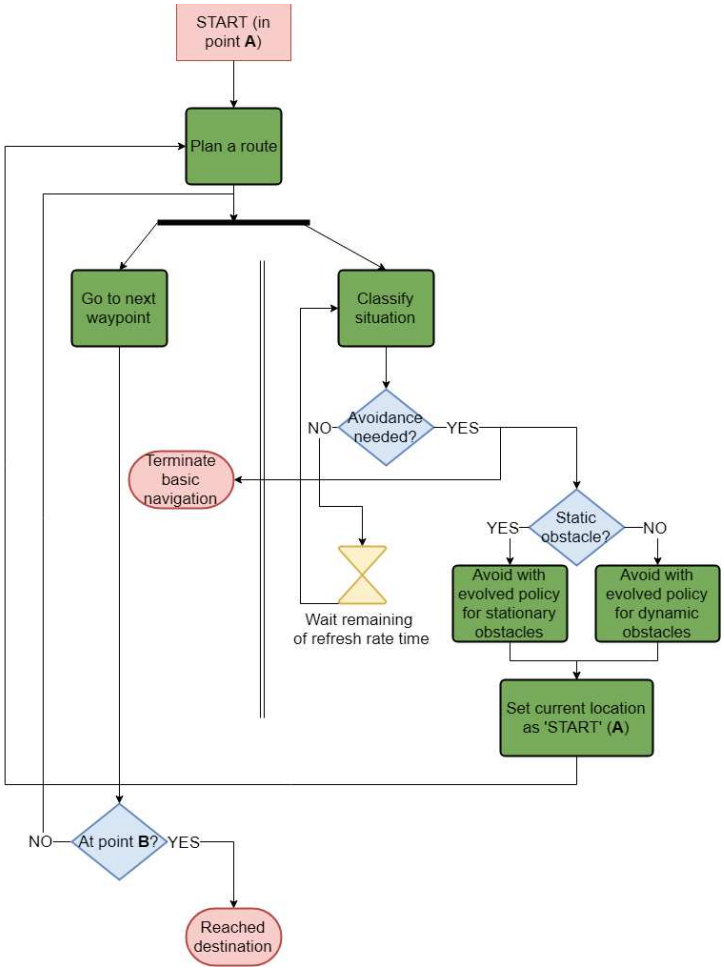

Figure 13: This figure presents the algorithm outline of the full designed solution used for static and dynamic obstacle detection and avoidance.

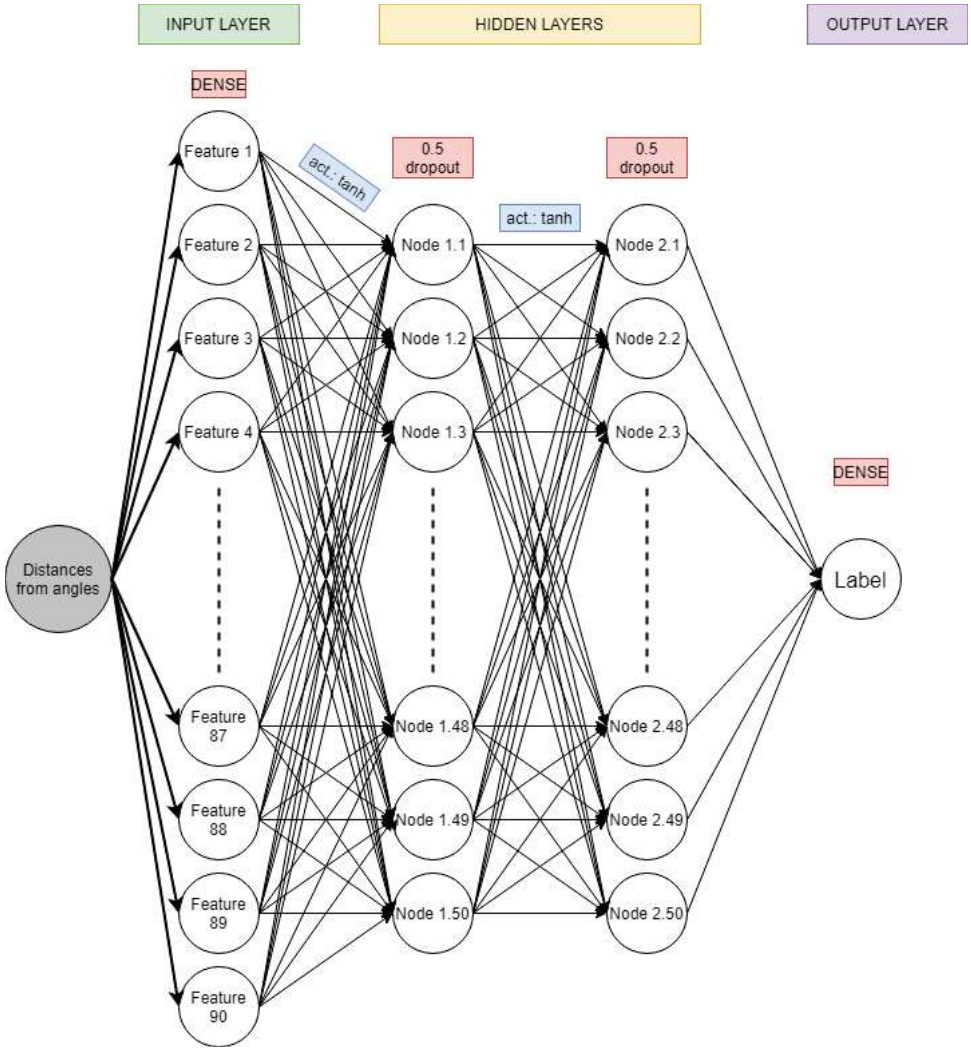

Figure 14: The graphical representation of the constructed deep neural network.

```
Building DNN model
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 50)                4500
_________________________________________________________________
activation_1 (Activation)    (None, 50)                0
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
activation_2 (Activation)    (None, 50)                0
_________________________________________________________________
dropout_3 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 51
=================================================================
```

Figure 15: The summary of the constructed deep neural network.

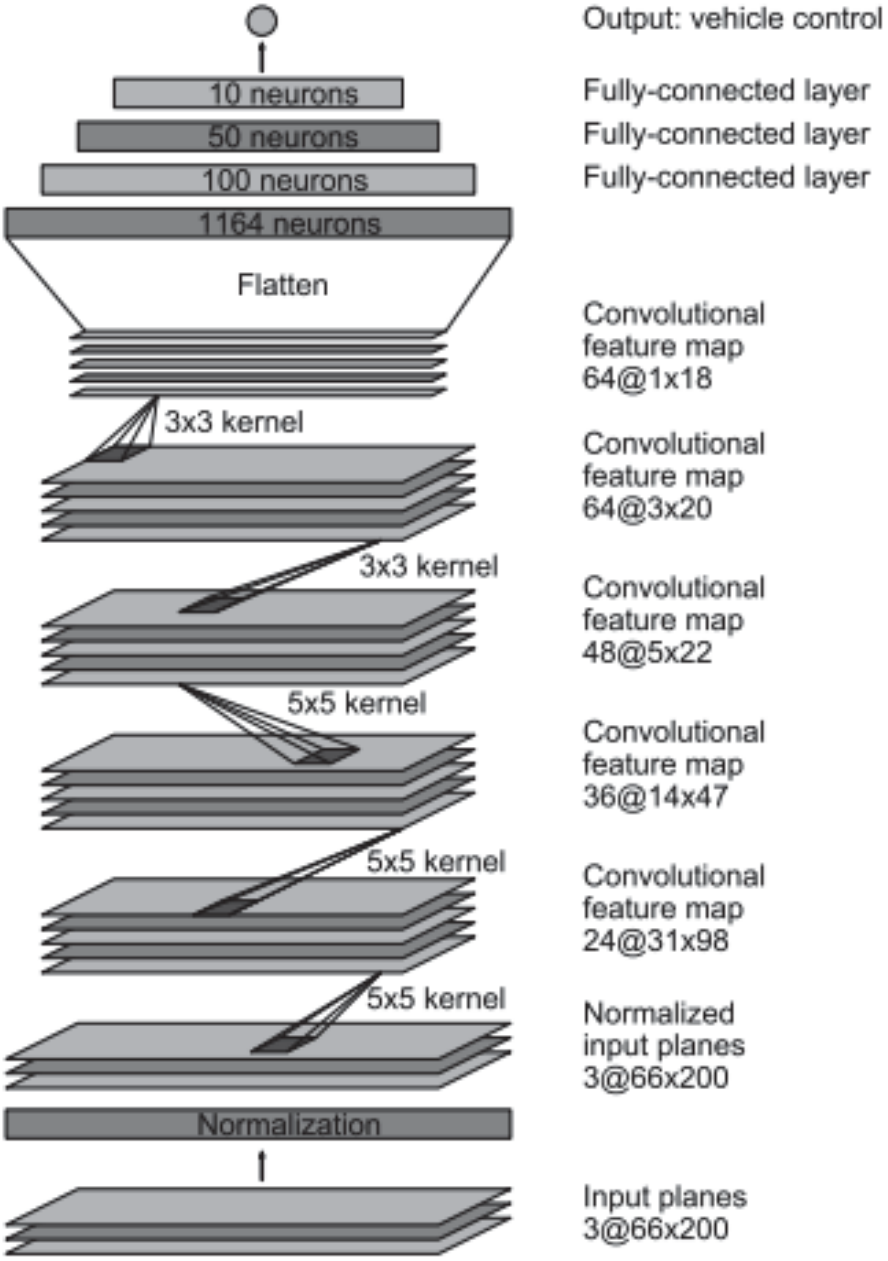

Figure 16: The graphical representation of the constructed convolutional neural network. As presented in the paper of Bojarski et al. (2016) in Figure 4 p. 5.

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 66, 200, 3)	0
conv2d_1 (Conv2D)	(None, 31, 98, 24)	1824
conv2d_2 (Conv2D)	(None, 14, 47, 36)	21636
conv2d_3 (Conv2D)	(None, 5, 22, 48)	43248
conv2d_4 (Conv2D)	(None, 3, 20, 64)	27712
conv2d_5 (Conv2D)	(None, 1, 18, 64)	36928
dropout_1 (Dropout)	(None, 1, 18, 64)	0
flatten_1 (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 100)	115300
dense_2 (Dense)	(None, 50)	5050
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11

Figure 17: The summary of the constructed convolutional neural network.

B Digital appendix

Please see the uploaded archive file.