Map Generation in Autonomous Racing

A Comparision of a Classic Heuristical Algorithm and Machine Learning

Alexander Seidler

Bachelor's Thesis

Department of Computer Science
Multimedia Information Processing Group
Kiel University

Advised by: Prof. Dr. Reinhard Koch
Lars Schmarje, M.Sc.

March 2022

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die eingereichte schriftliche Fassung der Arbeit entspricht der auf dem elektronischen Speichermedium.
Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegen hat.

Alexander Seidler
29. 03. 2022

Abstract

Automation is the natural direction to take on in the seek of increased safety, efficiency and passenger comfort, wherein autonomous racing sets a competition-driven framework for the exploration of new ideas. The problem of autonomous racing can be split into three main parts: landmark detection and tracking, map generation and trajectory planning, and controlling the vehicle. Map generation is a primary building block in which landmarks in a virtual space provided by a SLAM algorithm are used to create a map that can be used to determine the neccessary driving parameters. Two solutions for this problem are presented and compared in this work: an extension of a previously used classical algorithm by Vaishnav/Agrawal and a novel machine learning based algorithm. Three improvements to the classical algorithm are proposed: an improved spatial ordering, the readdition of missing points using heuristic guessing, and a filtering method based on the certainty of the detection. Even with these improvements, it is shown that the algorithm is too brittle to produce accurate results with erroneous input data. The machine learning algorithm is very error-resilient while still approximating sufficiently enough to be used in a simulated environment. Additionally, the runtime of these algorithms is shown to differ by an order of magnitude.

Acknowledgements

I wish to express my sincere thanks to my thesis advisor M.Sc. Lars Schmarje who guided me throughout this project and allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it. Also, I would like to thank Prof. Dr. Reinhard Koch for enabling me to write this thesis at the Multimedia Information Processing group in the first place.

I wish to extend my special thanks to all of my proofreaders, namely Leif Marvin, that provided me with corrections and valuable comments on this thesis.

Finally, I must express my profound gratitude to my best friend Kimberly for keeping me motivated and helping me finalize the project.

Thank you.

Table of contents

Chapter 1

Introduction

1.1 Motivation

Automation plays an essential role in the development of modern transport, as automation is the natural direction to take on in the seek of increased safety, efficiency and passenger comfort [16]. Autonomous racing provides a competition-driven framework for the exploration of autonomous driving, which incentivizes new innovations to take place. Thereby, racing often sets the starting point for new innovations that seek to revolutionize the entire industry[8]. One example of such competition is Formula Student Driverless (FSD).[1] FSD challenges teams across the world to build cars that can autonomously drive around fixed tracks which are defined by differently colored cones. One car is racing at a time and is competing for the fastest lap time.

The problem of autonomous racing in this context can be split into three main parts: landmark detection and tracking, map generation and trajectory planning, and controlling the vehicle. The first step in autonomously driving a vehicle is to generate an abstract representation of its surroundings. In order to do this sensory input such as camera images, LIDAR data and odometric input from an inertial measurement unit (IMU) is used to create and track landmarks in a virtual space and locate them relative to the vehicle. This task can be accomplished by simultaneous localization and mapping (SLAM) algorithms [20] and is not part of this thesis. Regarding the process of steering, the vehicle uses specific driving parameters such as desired velocity and steering angle to control the various actuators, e.g. motors, that move the vehicle. This problem is very similar to the controlling of non-autonomous manually driven vehicles, since the main difference is the driving parameters coming from sensors

[1]https://www.formulastudent.de/teams/fsd/

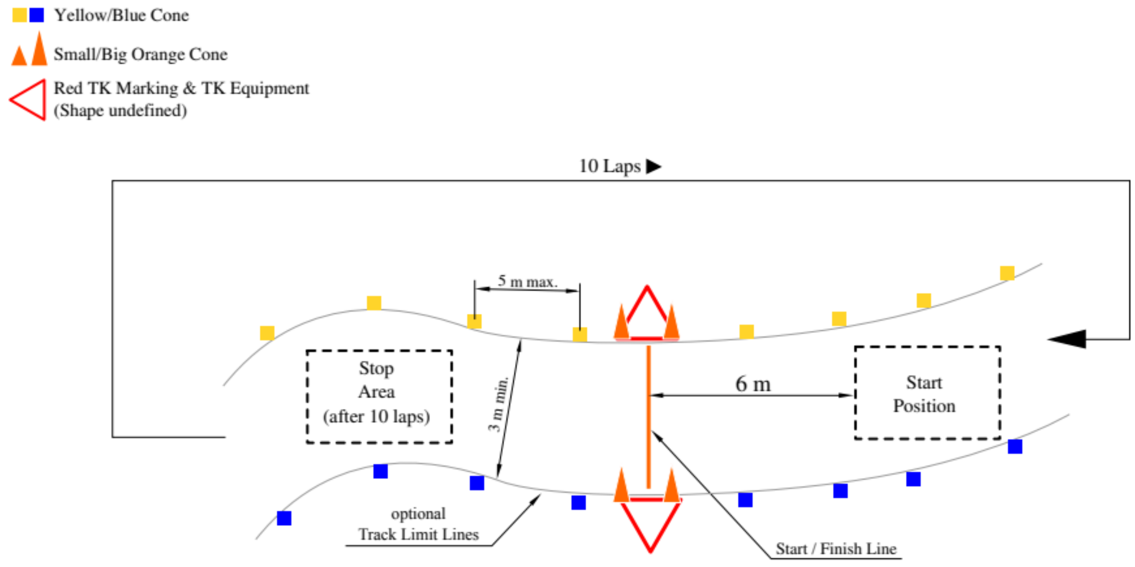


Fig. 1.1 Layout of a FSD track (Source: FSG21 Competition Handbook, p.14, "Figure 2: Trackdrive")

like the acceleration pedal and steering wheel in manual driving as opposed to the output of a processing pipeline in autonomous driving. This is also not part of this thesis. The problem that is left to solve is using the virtual space provided by the SLAM to determine the driving parameters velocity and steering angle. This problem can be split into two parts: Map generation, which focuses on transforming information about landmarks into an abstract map of the racing track and trajectory planning, which uses the abstract map to plan actions that will lead the vehicle to move along the track. This thesis looks at an extension of a classical algorithm for map generation which has been previously worked on and a novel machine learning approach to solve map generation and trajectory planning in one step and systematically compares these two approaches.

1.2 Goals

Raceyard is a team from Kiel aiming to compete in FSD and sets the framework for the implementation and application of the ideas presented in this thesis. As of writing this thesis, a simplistic classical approach to map generation is used at Raceyard which imposes several problems that render the algorithm not yet usable in practice. For three of these problems this thesis suggests an improvement. These are:

- Robustness against the incorrect detection of the color of landmarks (misdetection), missing landmarks entirely (non-detection) and detection of landmarks twice or more with one detection being at the wrong place (over-detection). Using the current approach, only some misdetections can be automatically corrected. Any misdetection that cannot be corrected renders the resulting map completely unusable. Furthermore, non-detections are completely ignored, which leads to problems especially in narrow curves. Over-detections are handled like normal landmarks leading to wrong predictions as well.
- Using the certainty the SLAM provides: The SLAM assigns covariances representing the certainty in x- and y-direction to each landmark detected, this covariance is completely ignored by the current algorithm, although it could be beneficial to use.
- Runtime: The current approach takes orders of magnitudes too long to be used in realtime.

Another goal of this thesis is providing a profound comparison of the different characteristics of the machine learning- and classical algorithms used, especially with regard to their error resilience, precision and runtime.

1.3 Related Work

Many publications in the field of autonomous driving can be found, however, each of these works focus on key aspects that differ from this thesis in one or more ways.
With regard to the classical approach to map generation, several techniques have been documented. The following papers apply a classical algorithm specifically to the problem of autonomous racing in FSD. AMZ Driverless [12] as well as Andresen et al. [2] focus on an architecture using an ordinary SLAM in conjunction with a Delauney triangulation to do path planning. Zeilinger et al. [27] as well as KIT19d [18] use an extended Kalman filter (EKF)-SLAM to derive the centerline for trajectory planning directly. Additionally, these papers do not take a look at machine learning as an alternative for path planning.

In machine learning some approaches to autonomous racing can be found, however, none of those apply machine learning (ML) to the problem of map generation and path planning in FSD specifically. Dewing [6] used a convolutional neural network (CNN) to solve autonomous driving in a virtual racing game. While Dziubiński[2] documented the use of a CNN

[2]https://medium.com/asap-report/training-a-neural-network-for-driving-an-autonomous%2drc-car-3906db91f3e,accessedon19.02.2022

for steering a toy car in open terrain without cones to mark the path.

One notable exception that applied machine learning to the problem presented in FSD specifically is the work of Georgiev [9]. Georgiev implemented Williams et al.'s [25] model predictive path integral (MPPI) in the Formula Student racing environment. MPPI uses a path integral over several possible trajectories to derive the best possible future trajectory in path planning. A neural network is used to train the parameters of the MPPI.

To the knowledge of the author, no full ML approach has been made specifically in the context of map generation in FSD. Furthermore, no comparison to a classical approach in FSD has yet been conducted. This work evaluates a modified classic heuristic Algorithm in comparison to a ML approach in the context of FSD racing.

1.4 Thesis Structure

In the following chapter, foundations and technical background are explained surrounding the two approaches and autonomous racing in general.
Thereafter, in the third chapter, the details of the classical and ML approach, as well as their implementation, are presented.
In the fourth chapter the approaches are evaluated and compared and in the last chapter, the results are summarized and several improvement ideas and suggestions for future work are listed.

Chapter 2

Foundations and Technologies

2.1 Raceyard and Formula Student

Formula Student is a global competition for building racing cars. The subclass FSD is focused on autonomous driving and is divided into different disciplines. Whereof Autocross is the most relevant for this thesis. The goal in Autocross is to drive a previously unknown track for one lap as fast as possible, thus all data about the track must be gathered and processed in realtime with no prior map. Since 2005, Raceyard participates in Formula Student for Kiel and aims to compete in FSD in upcoming competitions.

Fig. 2.1 The T-Kiel A CE, one of Raceyard's latest cars (Source: https://raceyard.de/autos/, accessed 18.11.2021)

2.1.1 The Rosyard Pipeline

The software to be used in FSD by the Raceyard car is called "Rosyard", which is built on the Robot Operation System (ROS) [23]. In ROS, processing takes place in nodes which can communicate with each other using data channels called topics. The nodes can be written in Python or C++ and are connected in a way that forms a pipeline in a feedforward fashion. The pipeline processes sensory data as input to compute data that can be used to move the actuators of the vehicle as output. The pipeline consists of five stages which are each represented by one or more nodes plus sensory input:

1. Input/Detection: sensory input from cameras and IMU, also preprocessing
2. SLAM: extracts landmarks and locates them in a virtual map
3. Estimation: estimates centerline in the virtual map
4. Driving: given map data determines the steering and velocity
5. Lowlevel: hardware controlling

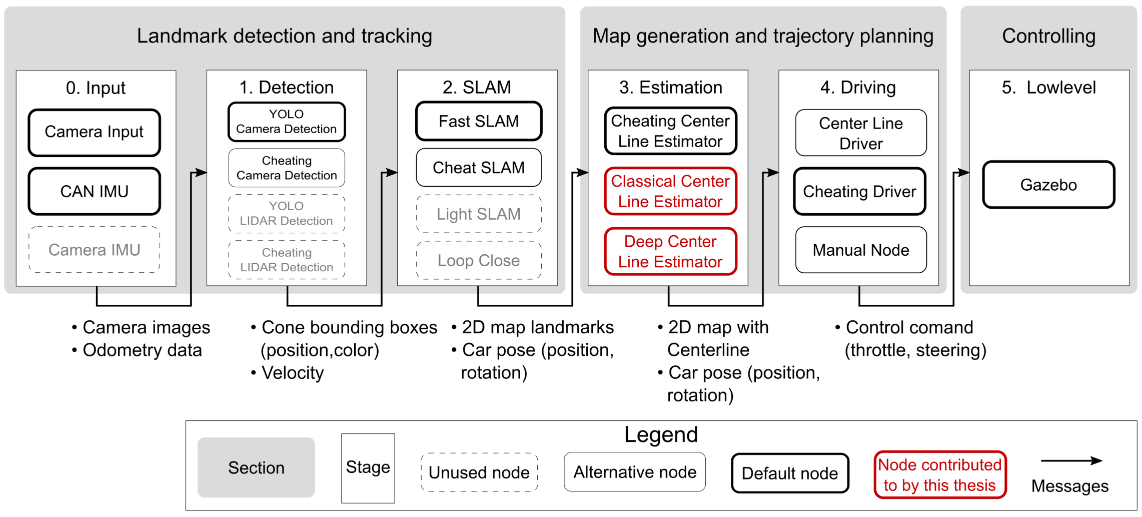


Fig. 2.2 Visualization of the Rosyard pipeline with stages that contain one or more nodes. Also visualized are the topics that are published and subscribed to by the nodes, represented by messages (Source: adapted from https://git.informatik.uni-kiel.de/las/rosyard/-/blob/master/docu/images/overview.png)

This thesis focuses on the implementation of the thrid stage. Given the landmarks located in a virtual map from the SLAM, this node should estimate the course of the track, such that the fourth stage can successfully drive the car along the track. The pipeline is fully dockerized and runs in four different docker containers: the Roscore coordinating everything in ROS, an optional visualization container, a simulation container for providing fake sensory input, and a container running all pipeline nodes.

2.2 Machine Learning

Machine Learning describes a class of algorithms that have the ability to improve automatically. This process of improving is known as learning. The most commonly used types of learning include supervised learning, unsupervised learning and reinforcement learning[7]. In supervised learning, a set of labeled data, called training data, is used to improve the parameters of the algorithm to make it predict labels better without explicit programming. Supervised learning can be used to train artificial neural networks. A neural network (NN) can be modeled as a directed graph consisting of artificial neuron as nodes and connection between neurons as edges. One example for artificial neurons are perceptrons. A perceptron is an abstract and mathematically easy to compute model of a biological neuron. A perceptron receives a number of inputs x and by using the weights of the inputs w calculates their weighed sum $z = w \cdot x$ and passes it through an activation function f. This leads to the output $y = f(z)$, which is called the activation of the perceptron. Common activation functions include linear $f_{linear}(x) = a \cdot x$ for some factor $a \in \mathbb{R}^+$ (commonly 1) and rectified linear unit (ReLU) $f_{ReLU}(x) = max(a,x)$ [19] where ReLU can be used to introduce non-linearity.

2.2.1 Deep Learning and Multilayer Perceptions

Multiple Perceptrons can be arranged in layers to form a special kind of NN, called multi-layer perceptron (MLP). In such a layer, a perceptron may only have connections to perceptrons in the preceding layers. A layer that has the maximum number of connections to the previous layer, such that each neuron is connected to every neuron in the previous layer, is called a fully connected layer. An MLP consists of an input layer, an output layer, and a variable number of so-called hidden layers in between the input and output layer. By having at least two hidden layers, the decision boundary of an MLP can take an arbitrary form, allowing it, in theory, to solve arbitrarily complex problems as opposed to a single perceptron, which can only solve linearly separable problems [14]. In recent years, research primarily focuses on networks with an even greater number of hidden layers. Such networks with a large number of layers are called deep networks. Since deep networks mostly use non-linear activation functions, the optimal weights cannot be found analytically. Therefore, other algorithms for learning must be used, which are known as deep learning. One of those algorithms is

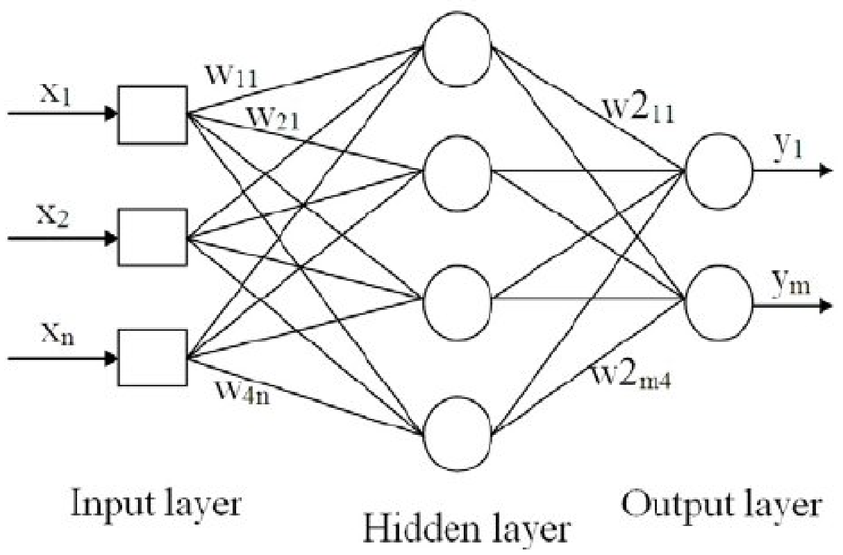


Fig. 2.3 A schematic diagram of a Multi-Layer Perceptron (MLP) neural network. (Source: Figure 5, An Oil Fraction Neural Sensor Developed Using Electrical Capacitance Tomography Sensor Data, Khursiah Mokhtar, 2013)

backpropagation, which uses gradient descent to learn the weights as an optimization problem of the weights with respect to the desired output. Usually, deep networks do not contain only fully connected layers, because these layers would introduce unnecessary complexity in the number of trainable parameters and through many consecutive activations, the effect of errors is increasingly amplified or diminished by the intermediate non-linear activation functions, thus creating either a vanishing or exploding effect which hinders efficient learning[10].

2.2.2 Convolutional Neural Networks

In CNNs, the concept of MLPs is extended by adding convolutional- and pooling layers in a neural network. Convolutional layers allow for processing a large number of inputs while not imposing a huge number of learnable parameters as a fully connected layer would. As a result of having this property, convolutional layers are ideal for processing images, as even small images e.g. a 32x32 RGB image already has 3072 inputs. A convolutional layer uses a number of weights matrices, called kernels, of a fixed small size (e.g. 5x5). These kernels are convolved across the width and height of the inputs, meaning the dot product of the filter and a specific local region is computed for each input, thereby computing a two-dimensional map of that kernel. The weights of the kernels can be learned using backpropagation, while certain hyperparameters must be set when designing the NN. One of such parameters is the size and number of kernels used. Another hyperparameter is by how many pixels the kernel is "moved" after each calculation, thereby skipping pixels as center for the kernel.

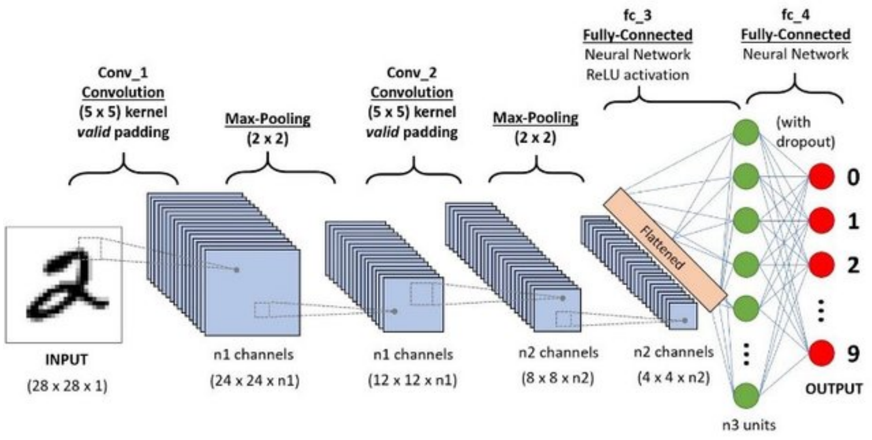


Fig. 2.4 Architecture of a CNN. An image as input is fed through multiple convolutional layers and pooling layers. The output of these Layers is flattened and fed into fully connected layers to compute the output. (Source: Deep Learning model-based Multimedia forgery detection, Pratik Kanani, 2020)

This hyperparameter is called stride. Around the edges, the input needs to be padded (usually with zeros), so that the edges of the input can be processed as well. A pooling layer reduces the number of inputs by partitioning the input along the width and height into equally sized chunks (e.g. 2x2) and computing an output for each of these chunks. Some commonly used pooling is max pooling, calculating the maximum of its inputs, and average pooling, calculating the arithmetical mean. Often, convolutional- and pooling layers are succeeded by fully connected layers which are then used to compute the final output of a network.

2.3 Discrete Curvature

Discrete curvature applies the concept of curvature from a continuous curve to a discrete curve called a polyline.

A polyline is a series of line segments and is determined by a sequence of points $(P_0, \ldots, P_n)$ $n \in \mathbb{N}$ where each line segment connecting a pair of adjacent points $[P_i, P_{i+1}]$ $i \in \mathbb{N}_{<n}$ forms a vertex in the polyline.

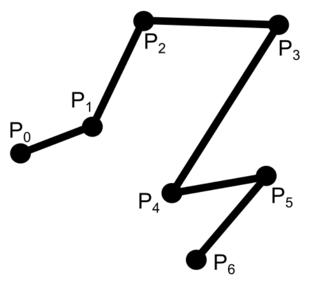


Fig. 2.5 A polyline over the vertices P_0 to P_6

In the continuum the curvature κ in a point of a differentiable curve is defined by the radius of the osculating circle in that point. This definition, however, is not useful to determine the curvature in a (discrete) polyline, given its non-differentiable nature. All straight segments would have a curvature of 0 while the curvature in the edges would diverge to infinity. A new definition must be used to determine the curvature of a series of line segments, which can then in turn be used to approximate this series. A different definition can be derived from the quotient of the circular angle φ and the arc length s:

$$\kappa = \frac{d\varphi}{ds}$$

Using this idea, we can define the curvature from a point A, a heading $\vec{h}$ in that point and a point B as the reciprocal of the radius of the circle passing through A and B and being tangent to $\vec{h}$ in A.

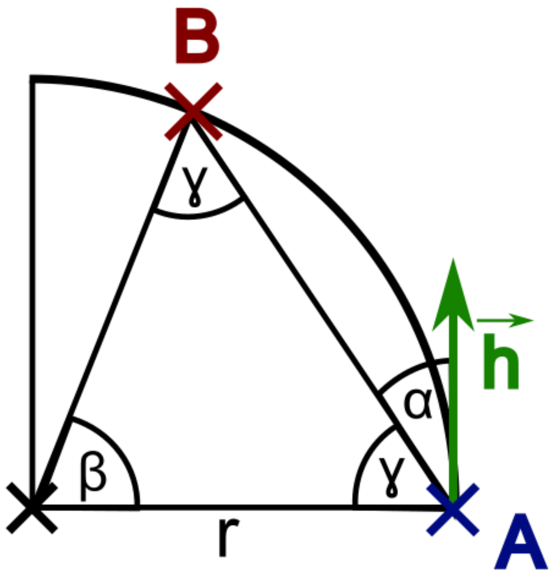


Fig. 2.6 Points A with heading $\vec{h}$ and B in circle with radius r, implying a curvature in point A of $1/r$. The circle center and B and A form an isosceles triangle with base angle γ and vertex angle β

Now, we can calculate the curvature κ as the reciprocal of the radius of this circle as follows: Since $\vec{h}$ is tangent it follows:

$$\gamma = 90^\circ - \alpha$$

and

$$180^\circ = 2\gamma + \beta$$

thus

$$(1)\beta = 2\alpha$$

Generally, the length of the secant of a circle $s := |\vec{AB}|$ can be calculated as $s = 2r \cdot sin(\frac{\beta}{2})$. Together with (1) we can derive

$$\frac{1}{r} = \frac{2sin(\alpha)}{s} = \frac{2sin(\measuredangle(\vec{AB},\vec{h}))}{|\vec{AB}|} = \kappa$$

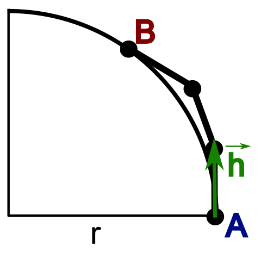


Fig. 2.7 Example curvature of $1/r$ *approximating a polyline leading from A to B, the circle corresponding to the curvature has the radius r*

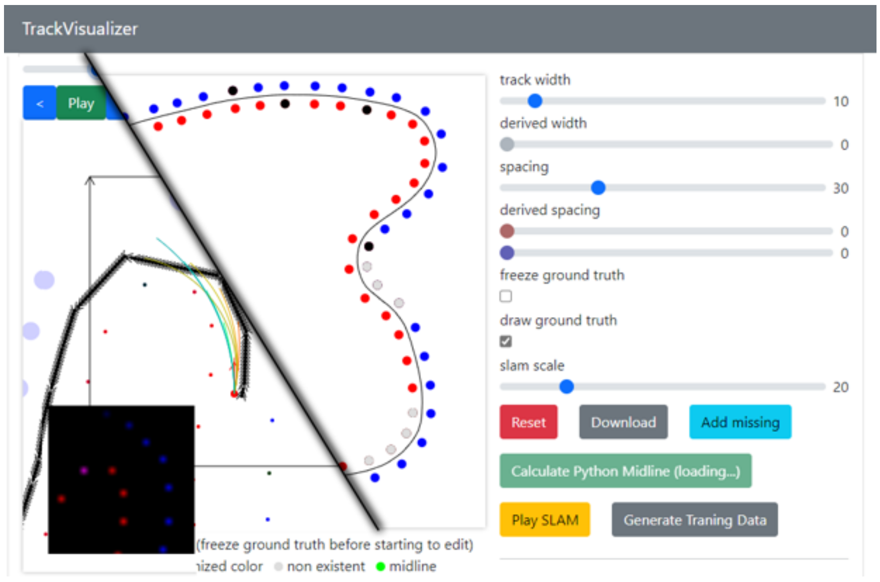
Using this method, we can calculate the average curvature of the curve that is tangent in A to $\vec{h}$ and passing through B, which approximates the polyline connecting these points using the points A, B and the heading $\vec{h}$, which can be derived from A and the next point after A leading to B. Doing this for differently distant points B on a polyline gives us a suitable approximation for the course of a polyline starting from point A. While this neglects the shape of the polyline completely, which fails to detect S-curves between point A and B, it does, however, impose no problem if we choose a fairly small distance between point A and B such that the variance of the curvature for intermediate points is non-significant.

2.4 Simultaneous Localization and Mapping

SLAM algorithms solve the chicken-and-egg problem localizing an agent in a map and mapping the environment surrounding an agent. Since for localization a map is seemingly needed and for creating a map of the surrounding the position of an agent needs to be known, the natural solution is to solve both simultaneously. While an exact solution is often not possible or desirable computation cost wise, several methods exist that can approximate the problem. These approximations for example use EKF, graphs, or particle filters. The SLAM used as input for the approaches in this thesis is an implementation of FastSLAM [17] which is based on particle filters. In FastSLAM, particles are used as potential positions for the agent, at each timestep a weight is assigned to the particles according to their likelihood of being consistent with the sensed nearby landmarks. Next, new particles are created according to the spatial distribution of weight, thereby converging to the actual position. In any timestep

the particle with the biggest weight is guessed as the actual current position and reported as such. This leads to the problem of discontinuity in the virtual space when the particles diverge to two or more different positions and the previous most likely position becomes less likely than another distant position. When this occurs, the generated map along the estimated position jump in a discontinuous manner. This also imposes the problem that landmarks cannot be identified consistently across time, since every particle keeps track of its own landmarks and once the estimated position shifts discontinuously, the landmarks cannot be associated to the previous landmarks because the transformation is discontinuous.
The output of FastSLAM is the incrementally built map of landmarks in relation to the estimated position of the agent. The landmarks have an uncertainty in the x- and y-dimension associated with them in form of a covariance matrix. This can later be used to filter for accidental detection of landmarks.

2.5 Development Environment Used



track width 10
derived width 0
spacing 30
derived spacing 0
0
freeze ground truth ☐
draw ground truth ☑
slam scale 20

Fig. 2.8 Screeshots of the prototyping environment coded using web technologies that was used to develop and test the implementation of this thesis. In green, a button can be seen that invokes Python code in the browser to calculate the centerline of the currently drawn or loaded track.

For developing and testing the implementation of the approaches, web technologies were used as the development environment, as this allows for fast prototyping and easy building of a visual interface and visual output. Additionally, this makes the prototypes easily sharable as they can be hosted on a web server and be accessed via browser. Specifically the JavaScript model-view-viewmodel framework vue.js[1] was used. Since the main source code of Raceyard as well as the previous algorithm is written in Python, the ability to run Python code was crucial for the development as well. While one possibility was the usage of a dedicated server running Python code with specific parameters that reports the result back to the web application, a web integrated solution would be more desirable.

2.5.1 Pyodide

Pyodide is a port of CPython to WebAssembly [5] which allows the execution of Python code directly within a browser using WebAssembly. As opposed to other systems, Pyodide does not cross-compile Python to JavaScript but uses a Python runtime to execute Python code on demand. Further, many of the most frequently used scientific Python libraries, e.g. NumPy, SciPy, Pandas and Mathplotlib are supported without extension, which makes it usable for many Python scripts without modification. The non-native execution, however, comes at a performance cost of running at about 2x to 10x slower than native Python, depending on the amount of C code used in packages [26][11]. Adding Pyodide to the development environment allowed the web application to be served completely statically, which meant that it could be published on a static website hosting service such as GitHub Pages[2].

[1]https://vuejs.org/
[2]https://dsalex1.github.io/BachelorThesisRaceyard/

Chapter 3

Methods

3.1 Classical Approach

3.1.1 Basis - Master Project by Vaishnav/Agrawal

The basis for the classical approach is the master project of Ashok Vaishnav and Akshay Agrawal in 2021[1]. It provides an implementation of the 3rd step of the Rosyard pipeline: Given the position of cones estimated by the SLAM, it calculates the centerline which forms the path for the driver in the 4th step to follow along. Two different scenarios need to be distinguished: In the first lap, no information about the track is known, and such the track must be navigated while simultaneously gathering information about the track to create a map that can be used in later laps. After the first round is completed, data about the track is available, so more detailed trajectory planning and navigation is possible, which allows for planning further ahead when driving. The basis for this approach primarily looks at the second case, where data about the whole track is available.

Diverting from the most optimal data the SLAM can provide, there are 3 different types of anomalies the projects looks at. These are missing cones (non-detections), misidentified cones (misdetection) and a shuffled point cloud. A shuffled point is defined as the cones in the data structure provided by the SLAM not being spatially ordered along the track. Two of these problems, misdetections and shuffled point clouds, are mitigated, yet not solved as seen later, by preprocessing the data. The preprocessing consist of a reclassification using a Support Vector Machine[4], which is a model that uses supervised learning to linearly separate data. By using a radial basis function, the input is mapped into a higher dimensional space, which allows a nonlinear separable problem in 2D to be solved linearly in higher

[1]https://git.informatik.uni-kiel.de/las/rosyard/-/blob/center_line/src/rosyard_pipe_3_estimation/Centerline_Estimation.pdf

dimensions. Next, the data is sorted using a naive closest neighbor sorting and is reorientated by comparing x coordinates of the first 2 points in the resulting dataset. After preprocessing the data is interpolated using a b-spline with the dataset as control points. The centerline is retrieved by calculating the midpoint of every point of one side and the point closest to it on the other side.

This partly naive approach leads to problems when used with artificially constructed data that has anomalies in it as well as when used with simulated input data.

The following picture 3.1 depicts the application of the current algorithm on artificially created data that contains some misdetections and non-detections. For better visibility, red was choosen to symbolize yellow cones, blue for blue cones, black represents cones with unknown/uncertain color, thus marking misdetections; grey represents non-detections. The black line is the ground truth of the centerline that was used to generate the data. The green line represents the centerline that is calculated by the algorithm. This example illustrates some of the problems the current implementation has: It completely ignores non-detections, which leads to heavy deviations from the ground truth centerline when non-detections accumulate in a corner, as seen in the upper right corner for example. Mis-detections lead to strange behavior. The reclassified cones cause the calculated centerline to deviate to the side in direction of the reclassified cone.

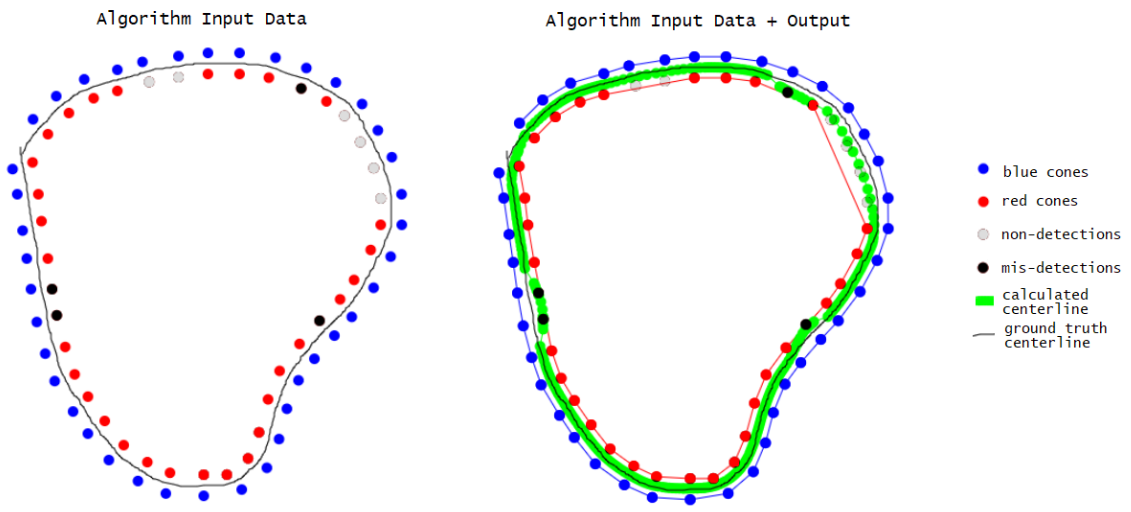


Fig. 3.1 Application of the unmodified algorithm of Vaishnav/Agrawal to artificially created data that has some non-detections in the upper right corner and center, and some misdetections where no color was assigned spread across the track. The application shows that even slight imperfections in the input data lead to an unusable centerline

This discrepancy to an ideal detection was mitigated using several improvements over the current algorithm.

3.1.2 First Improvement - Better spatial Ordering

Given a point cloud of unsorted points, we need to find the continuous path that is best described by these points. Previously, the nearest neighbor algorithm was used: Starting at an arbitrary point, it continued the path to the next respective closest point until all points are used. This approach, however, especially leads to errors when parts of the path are close together. In those erroneous cases, one can observe that the correct path is the shortest possible path through the point cloud as can be seen in figure 3.2.

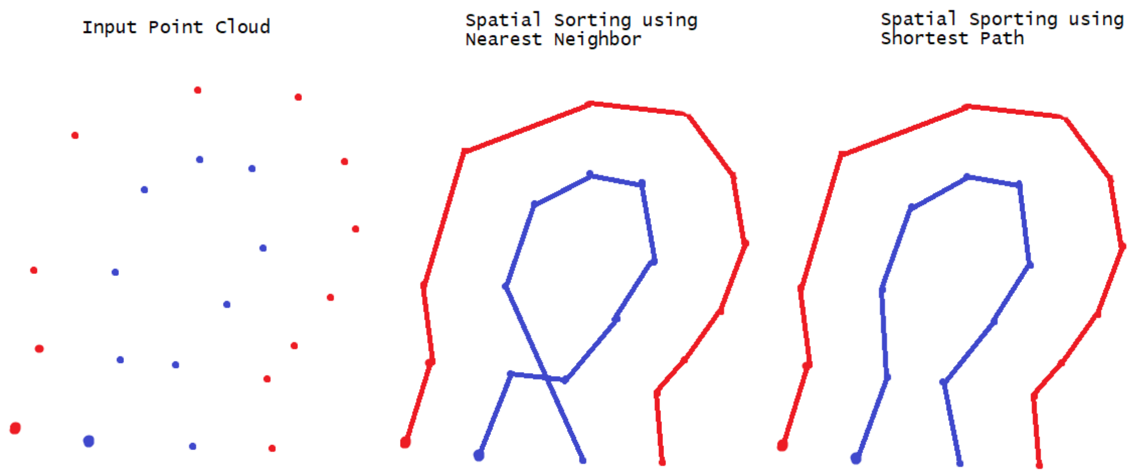

Input Point Cloud
Spatial Sorting using Nearest Neighbor
Spatial Sporting using Shortest Path

Fig. 3.2 Example for an erroneous spatial ordering: Given the point cloud on the left, the point clouds are sorted once according to the nearest neighbor algorithm starting at the marked larger point, and once according to the shortest path overall, the shortest path is the correct ordering

This means the problem of finding a path to a given point cloud can be modeled as the traveling salesman problem (TSP). Given that the TSP is NP-hard[1] it cannot be solved exactly while being efficient enough to be used with a larger number of points in realtime. An approximative algorithm such as the algorithm of Christofides and Serdyukov was an ideal solution, leading to a better result than a naive approach, while still having an acceptable complexity of $O(n^2 * log(n))$[3]. This meant that using Christofides algorithm instead of nearest neighbor would lead to a better result while still having a manageable runtime.

3.1.3 Second Improvement - Guessing Missing Points

The second improvement looks at non-detections, which were previously not accounted for at all. Given the following scenario, as illustrated by 3.3, the previous algorithm would not be able to detect the track at all: Especially within sharp corners, it is possible that one side of the track cannot be seen by the camera of the racing car. This leads to many non-detections on that side of the track while the other side can still be detected. This improvement detects these situations and guesses the positions of the non-detections to readd them thereby mitigating the non-detections. This approach guesses cone positions by checking whether each cone

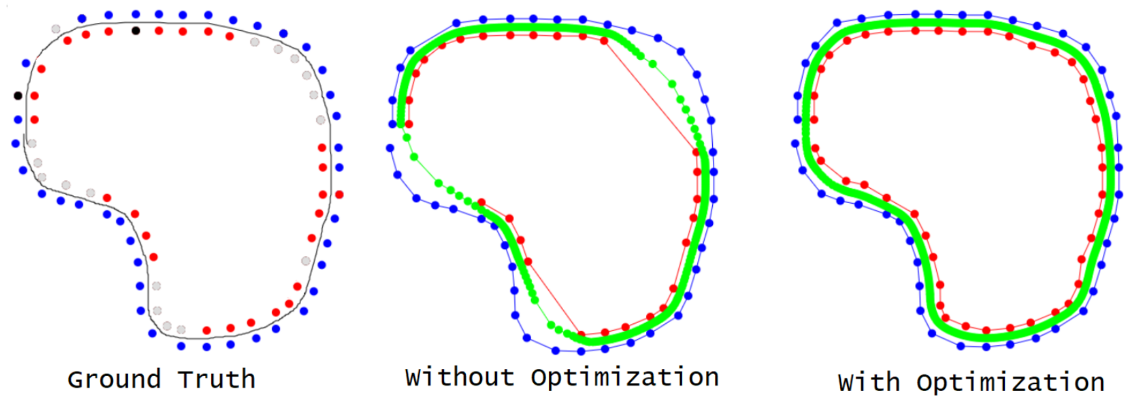


Fig. 3.3 non-detections, and their handling using the old approach and the new approach

has a corresponding counterpart roughly on the other side of the track. If that is not the case, the algrithm adds it where the corresponding cone would be expected. The estimated position of the corresponding cones can be calculated using the spatially sorted point clouds $Cones_B = (b_0, ..., b_n)$ and $Cones_A = (a_0, ..., a_m)$ for some $n, m \in \mathbb{N}$, the median track width w and the median distance between cones d. d can be calculated as the median over distances of neighboring cones $|\overline{a_i a_{i+1}}|$ and $|\overline{b_j b_{j+1}}|$ for $i < n, j < m \in \mathbb{N}$; w can be calculated as the median over the distance between each cone and the closest point on the other side, $|\overline{a_i c(a_i)}|$ and $|\overline{b_j c(b_j}|$ for $i < n, j < m \in \mathbb{N}$ where $c(a_i)$ is the closest cone in $Cones_B$ to a_i and $c(b_i)$ the closest cone in $Cones_A$ to b_i. Given that the track width and maximum cone distance are fixed along the track according to the FSD rules[2] and outliers are ignored by using the median, this yields values close to the true width and distance.

The following is repeated for $Cones_B$ and $Cones_A$ respectively. For simplicity we only take a look at $Cones_A$. For each consecutive three points in $Cones_A$, (a_{i-1}, a_i, a_{i+1}) the bisecting line of the angle between $\overline{a_{i-1}, a_i}$ and $\overline{a_i, a_{i+1}}$ is formed. With a distance of w to a_i this leads to two points on the bisecting line that could correspond to a_i. If within $\frac{d}{2}$ of one of those two points a point in $Cones_B$ is found, nothing is done. If not, the point that has the least distance to an existing point in $Cones_B$ is added.

[2]https://www.formulastudent.de/fileadmin/user_upload/all/2021/rules/FSG21_Competition_Handbook_v1.0.pdf, p.14

This is illustrated in the following example 3.4 where (A,B,C,D) are 4 consecutive points in $Cones_A$ and (A',B',D') are the points in $Cones_B$ that are closest to (A,B,D) respectively. We

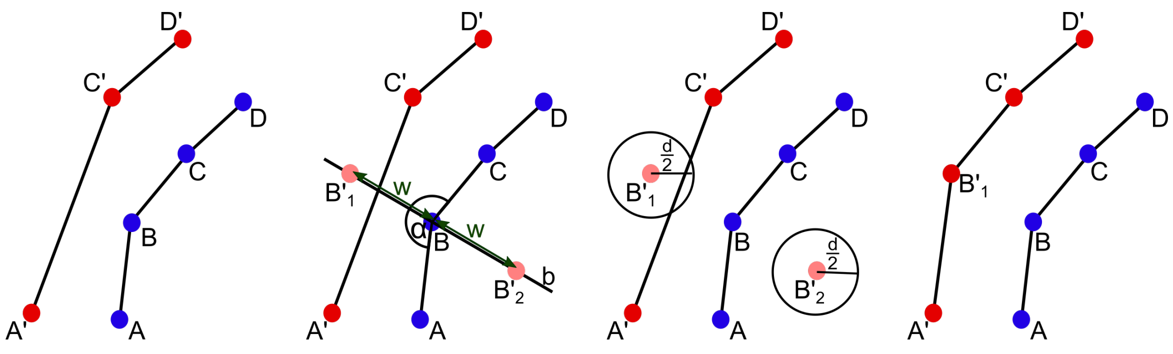


Fig. 3.4 Illustration of the guessing of missing cones where a cone is added

take a look at B: First, the bisecting angle $\alpha = \measuredangle ABC$ and the bisecting line b to α is formed. Now, on the line b with a distance of w to B two potential points are found B'_1 and B'_2. In the third step, no point in $Cones_B$ is found that is within a distance of $\frac{d}{2}$ of either point. Thus, the point that is closest to any point in $Cones_B$, B'_1, is added to $Cones_B$. In the following example the same procedure is repeated around point C. This time, however, there is a point found in

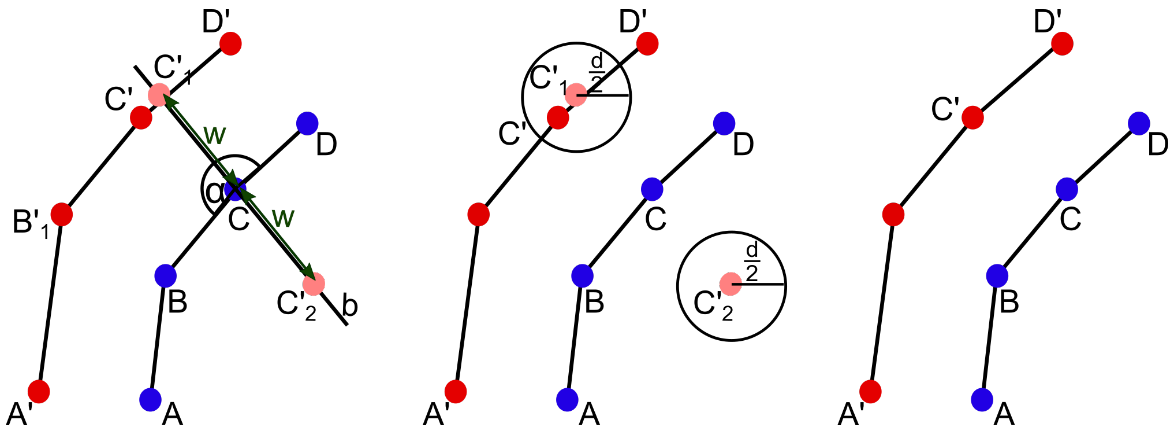


Fig. 3.5 Illustration of the guessing of missing cones where no cone is added

$Cones_B$ around the proposed points C'_1 and C'_2, and as a result, no point is added.

3.1.4 Third Improvement - Covariance Filtering

The third improvement has proven itself useful especially when used with simulated data instead of artificially created data, is the incorporation of the covariance the SLAM provides for each detected landmark. While the previous algorithm used all landmarks, the quality of the input data can be vastly improved by applying a threshold-based filter before passing

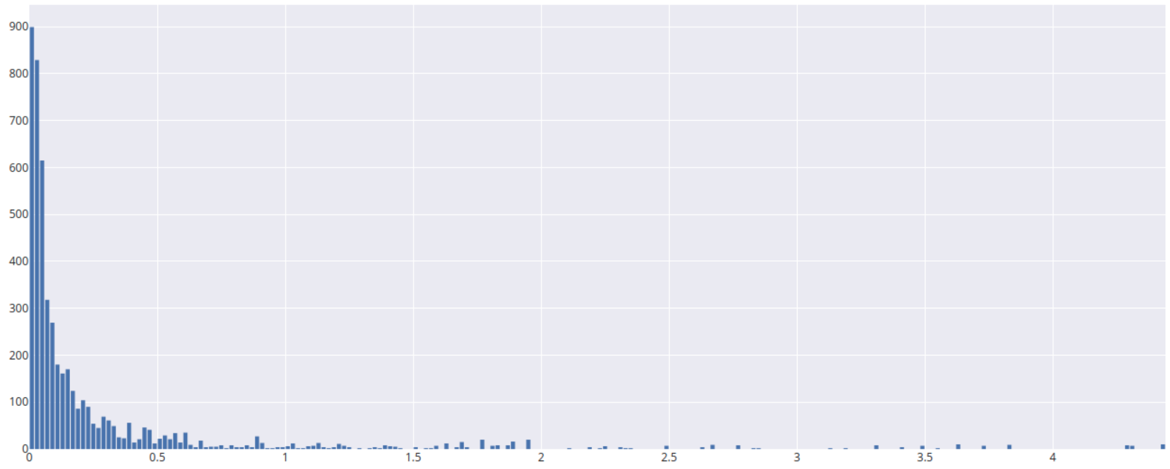


Fig. 3.6 Distribution of uncertainty in landmark detection over some simulated track drives

the data to the centerline algorithm. The covariance matrix A of a landmark is a 2x2 square matrix over the real numbers and describes the variance in the x- and y-dimension. Since the spatial orientation of the variance is not important in our case, as opposed to the overall certainty in the position, we can simplify the covariance matrix into a single scalar uncertainty value c by summing over the absolute value of its entries $c = \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|$.

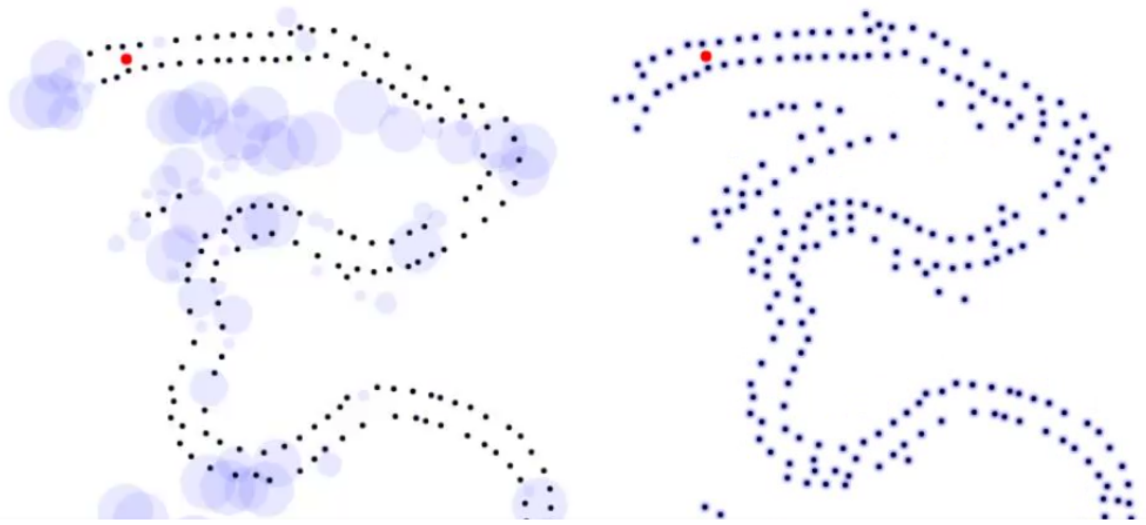


Fig. 3.7 Simulated track driving with different threshold filters. Points left are marked as a black point after filtering. Points filtered are visualized as light blue circle with a radius proportional to the uncertainty. Left side: $c_\theta = 0.1$, right side: original unfiltered data

By analyzing the distribution of uncertainty over simulated testing courses, heuristically, a threshold value of

$$c_\theta = 0.05$$

was found to be the most useful. This value, however, is very likely to change depending on the specific inputs provided to the SLAM algorithm, and will likely need to be determined experimentally, since the ideal threshold is a direct consequence of the covariances of the landmark detection, which is a direct consequence of the implementation of the SLAM as well as the input provided to it.

3.2 Machine Learning Approach

3.2.1 Idea and Input/Output Design

The problem of generating the centerline can be solved by abstracting to the problem of deciding the immediate next actions the driver can take on. The history of these local predictions can also be used to reconstruct the overall map later. The local track surrounding the driver, especially in the direction of driving, can be modeled using the centerline alone, given that the track width is constant. Furthermore, the course of the centerline can be modeled using discrete curvature, since we can assume that certain parts of the track have a constant curvature. This can be illustrated by considering the course of a typical FSDtrack [3]: It consists of straight parts with an approximatate curvature of 0 and curves consisting of distinctive parts of a track with a constant curvature. To improve the expressiveness of a single curvature value describing the local future course, several curvatures derived from points of varying distance can be used to describe the course of the track up to an increasing distance. As seen in the evaluation later, five curvatures that estimate the course to a point 2 m to 10 m in 2 m steps along the track work well. This leads to a simple yet expressive output format of five real numbers that describe the course of the track that is immediately ahead of the driver.

The input parameters are the cones that surround the driver and are immediately ahead. Here, one can notice that the measurement of curvatures are invariant under translation along the track, e.g. a medium sharp right-hand curve yields the same curvature values regardless of its position in the track, if we set the position of the car and its heading as the starting point for measuring the curvature. That is, if we assume that the cars heading points in the same direction as the centerline, but as we will see later, deviations from this are only beneficial in

[3]https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf, p.130

correcting the driving to align back with the centerline. This means we can pass the input to the neural network with positions in the local coordinate system of the car and eliminate thereby two additional input parameters: the position and heading of the car. With regard to the NN architecture, the varying number of cones that are nearby lead to a varying number of inputs that need to be considered, hence making it difficult to use a standard fully connected NN, since the number of input neurons would need to be fixed.

3.2.2 Modeling as Image Regressing Problem Using an CNN

This leads to the idea of utilizing a convolutional neural network. Since the area that needs to be considered is fixed and the curvature of a given set of points is invariant under translation the representation of the input as image was ideal. Also, the certainty as well as the color of the cone can be represented in the hue and brightness of a pixel. This concludes the idea for preprocessing the input data before feeding it to the NN. In the concrete implementation, some parameters were chosen heuristically and later verified to suffice experimentally.

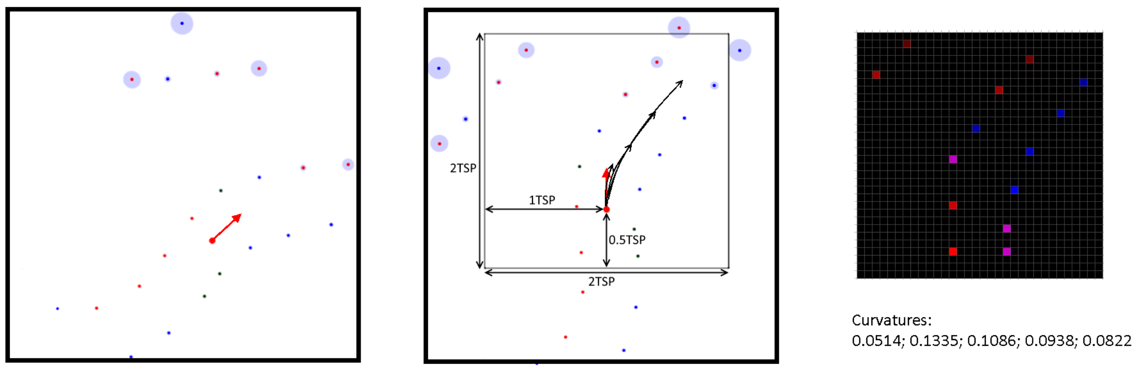


Fig. 3.8 Preprocessing of the cone data for the CNN. The car is represented by the larger red circle with its heading as an arrow. First, the map is rotated and moved to the local coordinate system of the car, a region according to TSP and $Car_{position}$ is selected and transformed into an image of size $Image_{size}$ with the certainty transformed into the brightness of the corresponding pixel. The curvatures in 2 m,... ,10 m are also shown as arrows in the center picture and numerically below the right picture

Surrounding the driver a with a sample radius $TSP = 8m$ a square patch of space is used to generate the input for the NN. Inside this square, the driver is centered vertically and horizontally offset such that the driver is in the middle of the lower half of the square. Formally, if the square starts at $(0,0)$ and has size $(1,1)$ the cars position is $Car_{position} = (0.5, 0.25)$. This meant cones $1.5 \cdot TSP$ in front, $1 \cdot TSP$ to either side, and $0.5 \cdot TSP$ behind for context are considered for estimating the curvatures. An image size of $Image_{size} = 32$ was chosen, because it gives a reasonable accuracy of $0.5\frac{m}{pixel}$, considering the track width of at least $3m$ according to the FSD rules[4] while keeping the number of inputs small. The transformation of the cone data into an image can be seen in figure 3.8. The distribution of the certainty in cone detections posed another problem when transforming the certainty to a lightness value, since the distribution is very sharp around 0 and the uncertainty can take on arbitrarily large values, the distribution needed to be transformed to fit the lightness range of $[0,1]$. To map the distribution from $[0,+\infty[$ to $[0,1]$, the arctangent is used. The result is squared to further flatten the distribution and is subsequently inverted along the x-axis (figure 3.9).

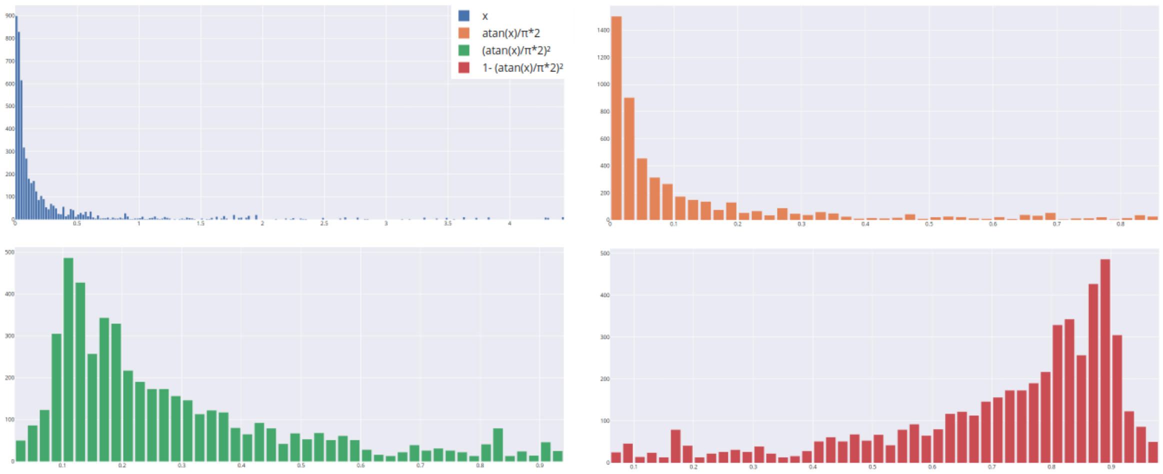


Fig. 3.9 Distribution of uncertainty of landmarks under some transformations: blue identity, yellow $x \mapsto atan(x)/2\pi$, green $x \mapsto (atan(x)/2\pi)^2$, red $x \mapsto 1-(atan(x)/2\pi)^2$

This lead to a much flatter distribution that is bound in $[0,1]$ using the transformation $x \mapsto 1-(atan(x)/2\pi)^2$, which makes the uncertainty much easier to be picked up on by the NN[22] than the considerably sharper distribution it had to begin with. The desired output, and such the labels for the training data, are calculated using the provided ground truth centerline data for simulated tracks. For each frame in a simulated drive through, the discrete curvature from the current position of the car is used as training label: On the centerline the curvature is calculated using the heading that is tangent to the centerline in that point and a point on the centerline that is $2m,\ldots,10m$ further away on the centerline respectively. Using solely the raw curvatures is problematic as well, since the distribution is fairly dense around zero while being very sensitive to small deviations from zero. To mitigate this, the desired output was transformed using a polynomial redistribution. For the data of the tracks of the last FSD competition, a transformation of $x \mapsto sgn(x) \cdot |x|^{\frac{1}{3}}$ made the distribution most uniform as seen in Figure 3.10.

[4]https://www.formulastudent.de/fileadmin/user_upload/all/2021/rules/FSG21_Competition_Handbook_v1.0.pdf, p.14

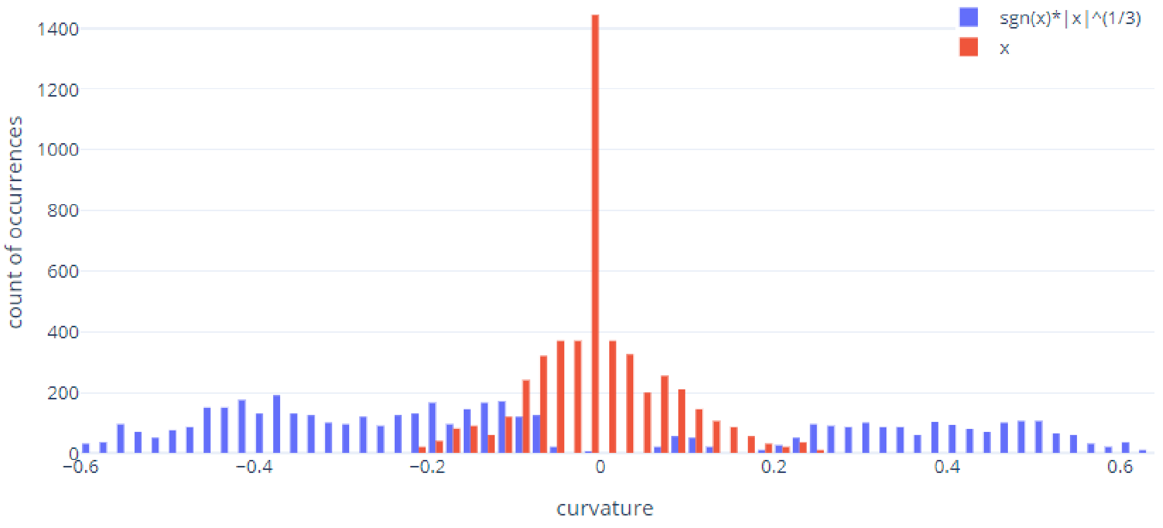


Fig. 3.10 Distribution of curvatures in FSD tracks 2019 with transformation: blue identity, red $x \mapsto sgn(x) \cdot |x|^{\frac{1}{3}}$

After these transformations, the problem is reduced to a simple image regression problem, regressing to 5 floating point numbers that correspond to a 32x32 RGB input image. To archive this a variation of the LeNet-5[15] and AlexNet[13] architecture was used an can be seen in figure 3.11. The LeNet-5 architecture was modified to fit the dimension of our input images, 32x32x3, and altered by applying more recent concepts, using max-pooling instead of average and ReLU instead of sigmoid as activation function. Lastly, the activation function of the output layer was changed to linear with 5 neurons to be able to regress data instead of classification as used in LeNet-5[15] and AlexNet[13].

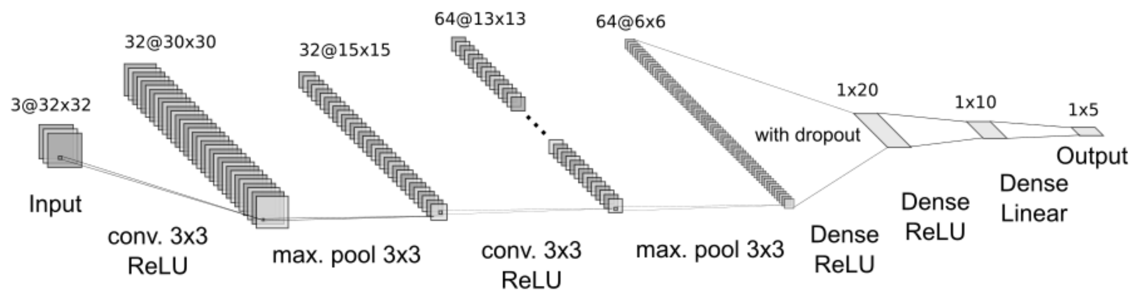


Fig. 3.11 Architecture of the NN used for the ML algorithm, 2 convolutional layers with 3x3 kernels and subsequent max pooling with 3x3 kernel respectively, 2 dense layers with ReLU activation function with 20 and 10 neurons respectively and dropout in the first layer and the output layer with 5 neurons and linear activation

3.2.3 Training

For the training data simulated drive-throughs were used to generate one training sample per frame in the data. The original unaugmented data was used from simulated tracks that were used in the FSD competition of the previous years.

Chapter 4

Discussion

4.1 Evaluation

4.1.1 Classical Approach

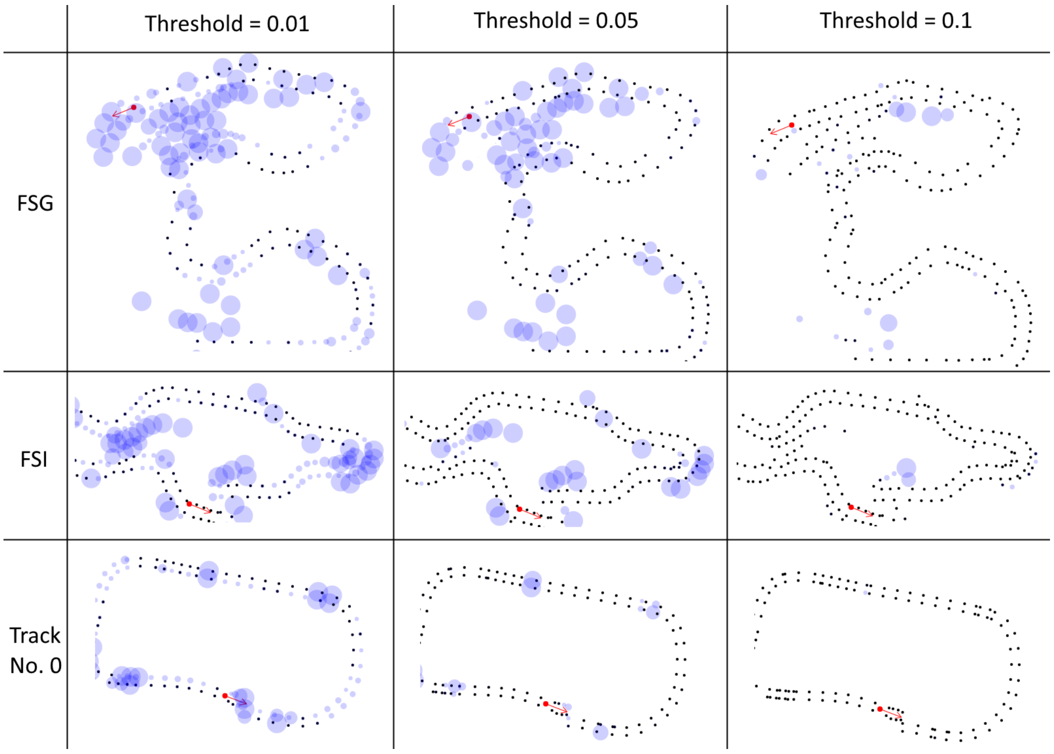


Fig. 4.1 The resulting landmarks after filtering using different certainty thresholds. $c_\theta = 0.01$, $c_\theta = 0.05$ *and* $c_\theta = 0.1$, *on three different FSD tracks "FSG", "FSI" and "Track 0",* $c_\theta = 0.05$ *has the best results.*

Anayzing the effect different covariance thresholds have on the quality of the resulting map, it is evident that $c_\theta = 0.05$ performs best, while a smaller threshold of $c_\theta = 0.01$ filters too many landmarks out resulting in an incomplete map. With a larger threshold of $c_\theta = 0.1$, too little noise is filtered out, resulting in over-detections in the map, as seen in figure 4.1.

A problem that occurred in developing more advanced means of filtering noisy data is the inability to track landmarks across frames. While it should theoretically possible to pass the information of the change in landmarks over time, the current implementation of the SLAM makes it difficult to do so since it assigns new identifiers to each landmark in each frame. Though, it would be possible to match landmarks based on their position, since the change in position approximates a continuous transformation, given small enough timesteps between frames. This spatial matching, however, is not practical for performance reasons, as it would be computationally expensive. Additionally, the implementation of the particle SLAMs uses different particles to determine the current position in the virtual map of which the one with the highest certainty is chosen to provide the landmarks in the current frame. Because every particle tracks its own landmarks, once the SLAM decides another particle has better certainty, the estimated position as well as all landmarks jump discontinuously to arbitrarily distant locations, which, in turn, makes spatial matching of landmarks impossible. The inability to track landmarks over time also means that it can not be determined which landmarks are newly added in one frame to another.

Analysis - Deviation From ground truth (GT)

In the following, the figures 4.2, 4.3 and 4.4 will use the same structure and color scheme. Each figure consists of two parts: the input for the classical algorithm on the left for clarity and the input with the overlayed output of the algorithm on the right. The thin green line represents the ground truth that was used to create artificial track data. Large red- and blue dots represent input cone positions of yellow and blue cones, respectively, where in an effort to increase legibility and visual clarity, yellow cones were replaced by red ones. Large light blue circles represent cones that were filtered out by the covariance filter. The radius of the circle represents the uncertainty of their respective cone. Black dots represent cone misdetections where no color was detected for certain. Grey dots represent cones that were not detected at all, thus marking the error of nondetection stated previously. The output of the algorithm is visualized in three parts: The large green dots represent the points of the calculated centerline, the light blue and small light red points represent the two sets of cones that are the result of preprocessing the cones, namely spatially ordered and readded missing cones. The arrows connecting these three parts mark the order in which these lists of points are contained in the output arrays. This ordering can be used to see the orientation (clockwise/counter-clockwise) of these lists.

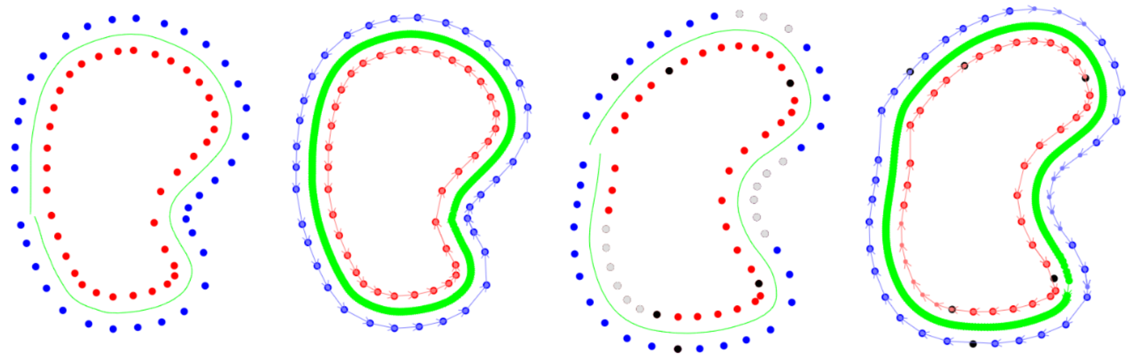


Fig. 4.2 Examples where the classical approach works well: Unaltered artificially created data on the left and artificially created data with non-detections and misdetections with no color certainty on the right

While the centerline generally works notably well with perfect data, it starts to produce less and less usable output with increasing errors in the input data. Error-free artificially created data, as well as artificially created data with several non-detections and misdetections where no color is identified, is handled well (figure 4.2).
Since no reassignment of detected colors is done and the algorithm assumes a closed loop as track, wrongly detected colors and partial tracks impose an immediate problem that is unhandled by the algorithm (figure 4.3).

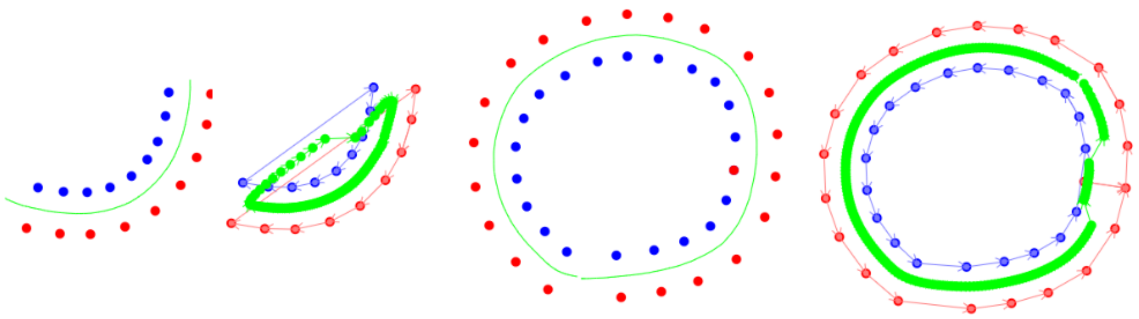


Fig. 4.3 Anormalies the classical approach can not handle: cones with misdetected certain colors on the right and non closed loops on the left

Appliying the algorithm to simulated SLAM data archives mixed results depending on the particularities of a certain frame, as exemplified in figure 4.4. More precisely, in frames where there happen to be little to no misdetected colors, the algorithm performs well, detecting the centerline perfectly except for where the misdetected colors are. However, in frames where over-detections and misdetected colors are more present, the orientation of the cones can not be determined correctly, which leads to a centerline which is unusable in major parts of the track.

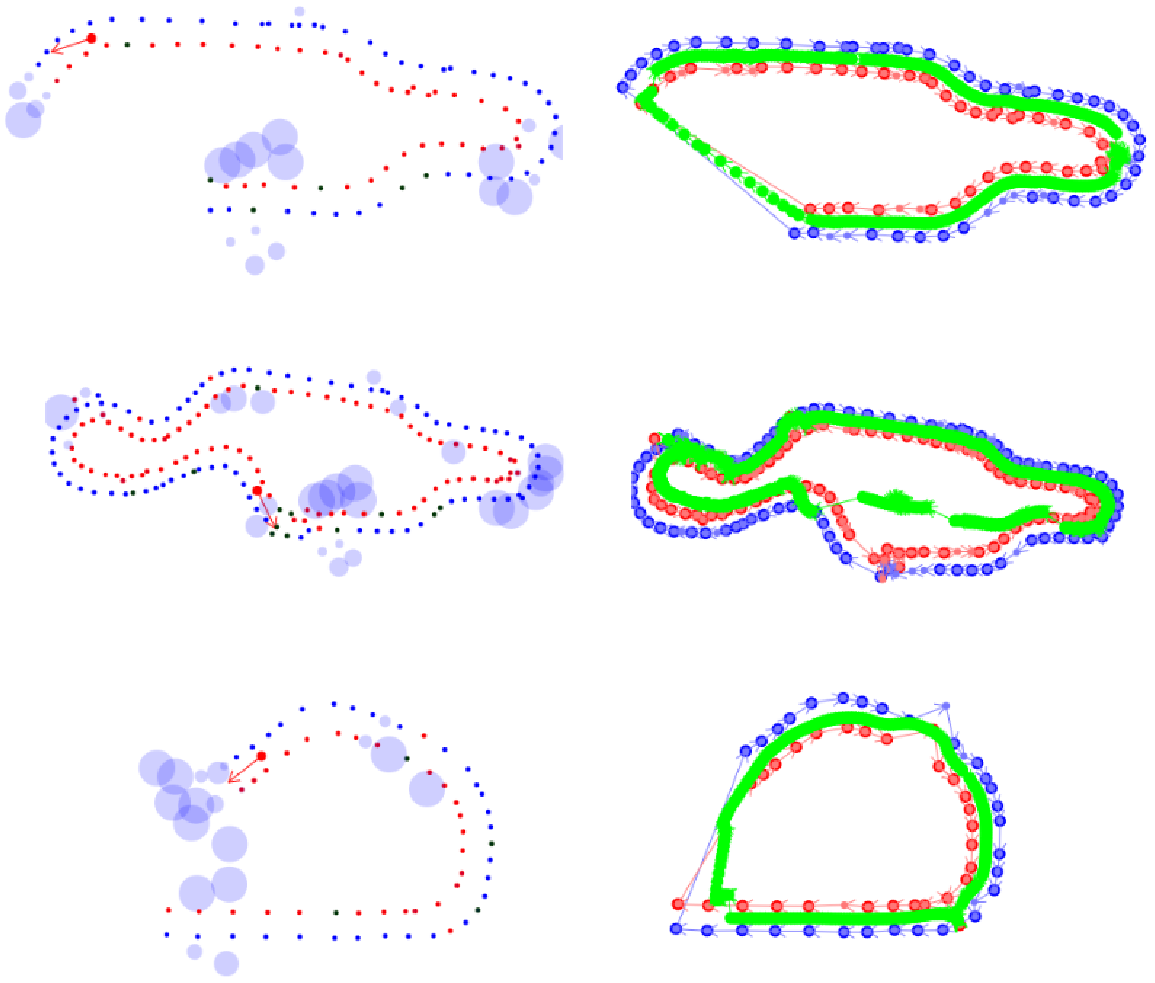


Fig. 4.4 The classical approach applied to simulated track data. In the upper example, no color midsections are present and over-detections are rare, which makes the algorithm perform well. In the middle example, over-detections and color misdetections lead to a misjudgment of the orientation, which breaks major parts of the centerline. In the bottom example, two color misdetections break the centerline at these places.

4.1.2 Machine Learning Approach

The neural network for the ML approach was trained using a mean average loss with ADAM as optimization algorithm and a learning rate of 0.001. These parameters were chosen heuristically and were proven to be sufficient. The loss converged quickly and only 15 epochs with a batch size of 2 were already enough to archive the most optimal validation loss while preventing overfitting. Additionally, a low number of training samples (2000) lead to similar results in validation as 15000 and 20000 did. These settings lead to an average test loss of 0.112.

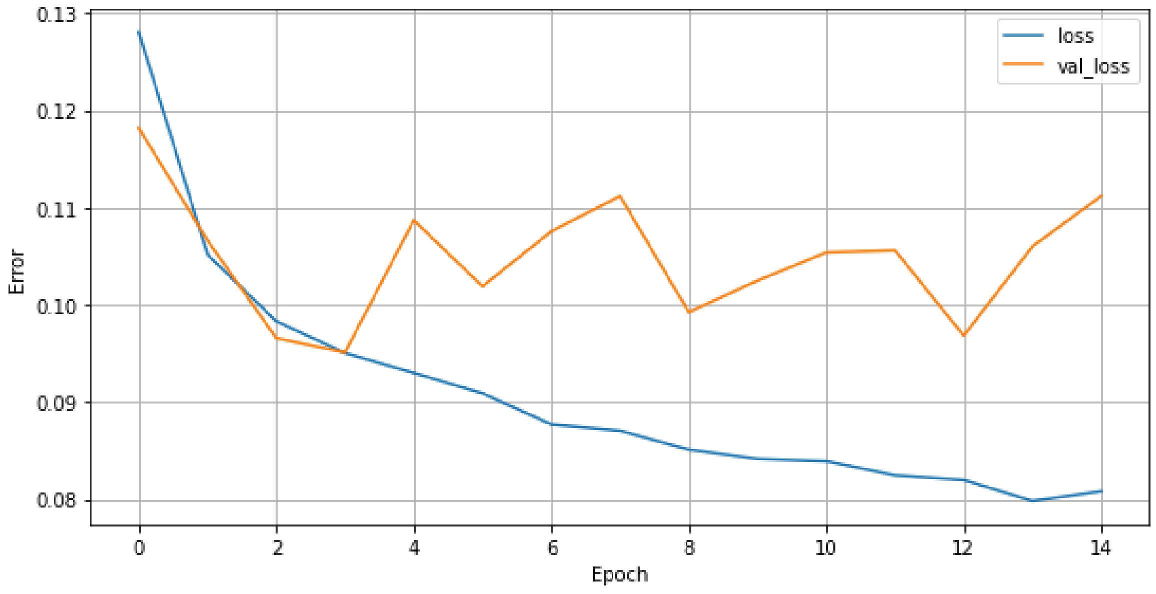


Fig. 4.5 The training and validation loss after 15 epochs with a batch size 2, 1400 training samples and 600 validation samples, ADAM with mean average loss, and 0.001 *learning rate*

Analysis - Driving Test

Since the average loss alone is not very expressive in describing the usability of the neural network, it can be evaluated by letting a driver test the curvature predictions made by the algorithm. To archive this, a simple test implementation of the 4th stage of Rosyard pipeline can be used, which applies a steering that is proportional to the estimated curvature κ. In the test implementation, a proportionality constant of $k = 240$ in addition to a low-pass filter is used to smooth out rapid changes in steering, which is realized by exponential smoothing with a smoothing factor of $\alpha = 0.8$. This leads to the overall formula for the steering for the timestep t as follows:

$$steering_0 = k\kappa$$

$$steering_t = \alpha k\kappa + (1-\alpha) * steering_{t-1}, t > 0$$

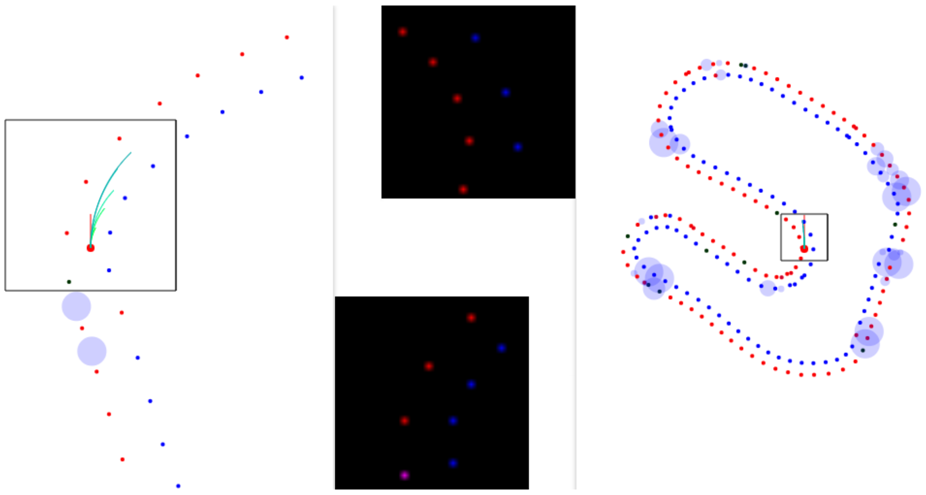
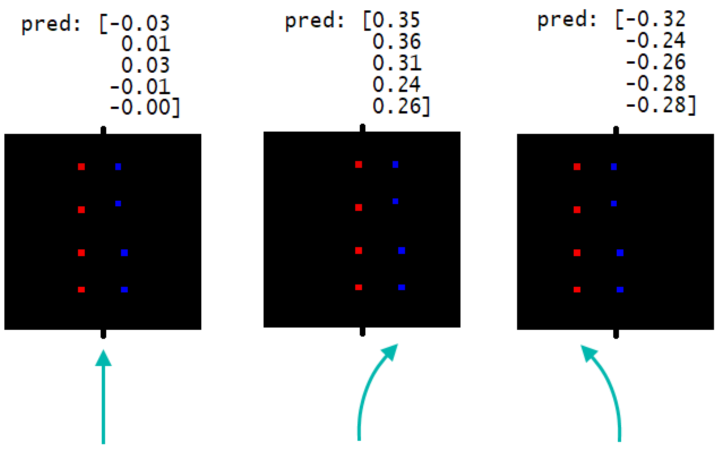


Fig. 4.6 Track 1 being driven autonomously by the machine learning approach and a simple test driver. On the right, a moderate right turn can be seen along the image the NN sees. On the left an overview of the track can be seen while the car drives a moderate left turn along the image the NN sees.

Even though the neural network was trained only using training samples where the driver is centered on the track, a deviation from the centerline is interpreted beneficially as the curve leaning towards the opposite direction, which forms a feedback loop that keeps the car in the middle of the track (figure 4.7).



```
pred: [-0.03      pred: [0.35      pred: [-0.32
        0.01              0.36             -0.24
        0.03              0.31             -0.26
       -0.01              0.24             -0.28
       -0.00]             0.26]            -0.28]
```

Fig. 4.7 The same image fed into the NN with three different x offsets, centered, positive offset and negative offset, the effect a deviation from the track center has on the prediction of curvatures makes the model predict a curve that lets the car return to the center in a closed feedback loop.

Notably, this allows the car to stay on the track and drive laps completely without crossing the boundaries of the track. The approximative nature of a CNN allows it to drive even using very noisy data by finding a suitable approximation for the local centerline instead of solving the centerline exactly. This allows the CNN to drive the first lap without any prior data, however, the curvatures produced by the CNN cannot be directly used to create a full centerline for the whole track, as only the local centerline directly in front of the car is approximated. This local approximation is, by definition, guaranteed to start at the position of the car, which might not be the center of the track. Lastly the CNN cannot benefit from data about the rest of the track that is not directly in front of the car.

4.2 Comparison of Approaches

The classical and machine learning approach both solve different problems and work well in their respective domain. The classical algorithm produces accurate output when used with input with enough certainty and a low level of noise. It serves the goal of computing a complete precise centerline for the whole track. However, when used with too noisy data, it fails to detect the centerline completely, which is the case with the noise level the current SLAM implementation and preprocessing provides. The machine learning approach is much more noise-tolerant and usable without any contextual knowledge about the track. Hence, it can be used in the first lap. It is more robust against erroneous data but produces only local and overall less precise centerline points. Thus, less planning ahead is possible and more work is needed to generate a complete map after completing a full lap.
Overall, the classical approach produces precise centerline data when provided with the right data, but is too fragile resulting in a failure to produce usable output consistently when used with simulated data, thereby rendering it - as is - not applicable to be used to drive the car. The machine learning approach on the other hand, while producing much less precise output, is noise resilient enough to successfully drive the car around the track.

4.2.1 Runtime

Another noteworthy advantage the ML approach has over the classical approach is its runtime. An average track contains around 200 cones of which on average half need to be processed by either algorithm to produce a centerline. The start-to-finish runtimes of a complete frame in either algorithm is compared in the following table 4.1. The system used to measure the runtimes runs on an Intel® Core™ i7-5820K and NVIDIA GeForce GTX 980 Ti. The following runtimes were obtained over an average of 50 frames:

Runtime per frame			
Track	Number of Cones	Classical Algorithm	Machine Learning Algorithm
FSD	50	171ms	63ms
	100	910ms	61ms
	185	1948ms	66ms
FSI	50	93s	63ms
	101	622ms	66ms
	182	2397ms	65ms
Track 0	50	330ms	61ms
	101	927ms	63ms
	269	5540ms	68ms

Table 4.1 Runtimes per frame for the classical algorithm and machine learning algorithm on different tracks with a different number of cones present

The runtimes of the classical algorithm imply a cubic runtime in regard to the number of cones presented. The machine learning approach implies a constant runtime. Additionally, the ML approach in the average case of about 100 cones takes an order of magnitude less time to execute per frame, while a constant execution time also helps to compute data reliably, regardless of how long the car has been running. The classical algorithm will eventually need to discard older data to maintain a constant execution time per frame. The classical algorithm is able to maintain about 1.6 frames per second (FPS) in the average case, 5 FPS in the best case and 0.5 FPS in the worst case. The machine learning algorithm was able to maintain 16 FPS consistently.

Chapter 5

Conclusion

5.1 Summary

This work proposed solutions to the problem of map generation in the context of autonomous racing at the FSD team from Kiel Raceyard. Two different approaches have been analyzed and compared. An extension of the previously used classical algorithm by Vaishnav/Agrawal was developed and a novel machine learning based algorithm was presented. Three improvements to the classical algorithm were proposed: an improved spatial ordering, the readdition of missing points using heuristic guessing and a filtering method based on the certainty of the detection. These improvements made the algorithm more robust especially against non-detections and color misdetections of cones. The algorithm is still very brittle and only produces satisfactory results if the input data is sufficiently error-free. If the input is sufficiently precise, it produces a very accurate centerline. The runtime of the ML algorithm was an order of magnitude faster than the classical algorithm, which renders the ML algorithm usable and the classical algorithm unusable in realtime. The ML algorithm has been shown to be very error-resilient while still being able to approximate the centerline well enough, such that a simple driver can follow along the track without crossing the borders.

5.2 Future Work

5.2.1 SLAM

While the SLAM provides accurate information about the landmarks in a local environment around the driver, the information provided is still noisy and drifts over time. Since the detection is not perfect, the error on the position of detected landmarks accumulates and causes the estimated position to drift from the actual position. Another problem is the doubledetection of cones when seen from a pass close by, and later drive-through. These problems could be improved upon by exploring extensions to the currently used FastSLAM [17] as well as using different SLAM algorithms entirely such as EKF SLAMs [21] or GraphSLAM [24]. As improvements to the input data have a positive effect along the rest of the pipeline folling them, an effort to increase input data accuracy could contribute a considerable part to overall system improvement.

5.2.2 Classical Algorithm

As seen in the evaluation, the classical algorithm can only be applied to find the centerline to the cones in a complete lap. Therefore, it cannot be used in the first lap to drive along the track. Changing the algorithm in a way that it can handle incomplete (and possible noisy new) data would make the classical algorithm usable for driving the first lap as well.

Additionally, when used in the first lap, the estimated position of the car can be factored in to determine the importance of cones. This way, potential double-detections of distant track parts can be ignored. Furthermore, the orientation of the cones relative to the car, being on the left or right side of it, can be used to verify the color detection of the cones. Given that the car has not left the track, the correlation between the color of cones and the position relative to the car can be used to infer the color.

5.2.3 Machine Learning Algorithm

As only original unaugmented data was used, one simple method of improving on the machine learning approach is to augment the data using mirroring and rotation. Another factor that can be used would be the deviation from the centerline, as currently only training samples are used in which the car is perfectly centered in the track. Such deviations are handled implicitly instead of handling deviations explicitly. One way of resolving those would be to augment the input data by translating them along the x-axis and altering the expected curvatures to account for the additional steering that needs to take place to return the car back to the centerline. Another possibility is to add the deviation from the centerline as an additional output parameter of the network. This way, the network learns to estimate the deviation along the future trajectory of the course.

5.2.4 Other Improvements

Another improvement could be the use of a CNN as preprocessing before passing data to the SLAM, especially for detecting the bounding boxes of cones in the image data and estimating the color right from the image data alone before passing it to the SLAM

5.3 Outlook

Overall, this work sets the first step towards driving the Raceyard race cars autonomously in the real world, by providing one of the last essential implementations to the pipeline that is needed before the first autonomous test drive can be commenced. While this thesis by no means provides an implementation that will win races, it provides many theoretical concepts for map generation, ideas for future development along a proof of concept that can very well be used in near future to drive the race car fully autonomously along a track for the very first time in real life.

References

[1] *The Traveling Salesman Problem*, pages 527–562. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-71844-4. doi: 10.1007/978-3-540-71844-4_21. URL https://doi.org/10.1007/978-3-540-71844-4_21.

[2] Leiv Andresen, Adrian Brandemuehl, Alex Honger, Benson Kuan, Niclas Vodisch, Hermann Blum, Victor Reijgwart, Lukas Bernreiter, Lukas Schaupp, Jen Jen Chung, and et al. Accurate mapping and planning for autonomous racing. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2020. doi: 10.1109/iros45743.2020.9341702. URL http://dx.doi.org/10.1109/IROS45743.2020.9341702.

[3] Nicos Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. *Operations Research Forum*, 2022.

[4] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20 (3):273–297, 1995.

[5] The Pyodide development team. pyodide/pyodide. August 2021. doi: 10.5281/zenodo.5156931. URL https://doi.org/10.5281/zenodo.5156931.

[6] A. Dewing. Now this is podracing-driving with neural networks. *Stanford CS231n Reports*, 2016. URL http://cs231n.stanford.edu/reports/2016/pdfs/100_Report.pdf.

[7] Ayon Dey. Machine learning algorithms : A review. *International Journal of Computer Science and Information Technologies*, 7(3):1174–1179, 2016.

[8] Gordon R. Foxall and Brian R. Johnston. Innovation in grand prix motor racing: the evolution of technology, organization and strategy. *Technovation*, 11(7):387–402, 1991. ISSN 0166-4972. doi: https://doi.org/10.1016/0166-4972(91)90020-5. URL https://www.sciencedirect.com/science/article/pii/0166497291900205.

[9] Ignat Georgiev. Path planning and control for an autonomous race car. *Edinburgh UG4 Project Reports*, 2019. URL https://project-archive.inf.ed.ac.uk/ug4/20191552/ug4_proj.pdf.

[10] Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/13f9896df61279c928f19721878fac41-Paper.pdf.

[11] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/jangda.

[12] Juraj Kabzan, Miguel I. Valls, Victor J. F. Reijgwart, Hubertus F. C. Hendrikx, Claas Ehmke, Manish Prajapat, Andreas Bühler, Nikhil Gosala, Mehak Gupta, Ramya Sivanesan, Ankit Dhall, Eugenio Chisari, Napat Karnchanachari, Sonja Brits, Manuel Dangel, Inkyu Sa, Renaud Dubé, Abel Gawel, Mark Pfeiffer, Alexander Liniger, John Lygeros, and Roland Siegwart. Amz driverless: The full autonomous racing system. *Journal of Field Robotics*, 37(7):1267–1294, 2020. doi: https://doi.org/10.1002/rob.21977. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21977.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012. doi: 10.1145/3065386.

[14] Alan Lapedes and Robert Farber. How neural nets work. In D. Anderson, editor, *Neural Information Processing Systems*. American Institute of Physics, 1988. URL https://proceedings.neurips.cc/paper/1987/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

[15] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[16] Jerome M Lutin. Not if, but when: Autonomous driving and the future of transit. *Journal of Public Transportatio*, 21 (1), 2018. doi: 10.5038/2375-0901.21.1.10. URL https://digitalcommons.usf.edu/jpt/vol21/iss1/10.

[17] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem. 11 2002.

[18] Sherif Nekkah, Josua Janus, Mario Boxheimer, Lars Ohnemus, Stefan Hirsch, Benjamin Schmidt, Yuchen Liu, David Borbély, Florian Keck, Katharina Bachmann, and Lukasz Bleszynski. The autonomous racing software stack of the kit19d. *CoRR*, abs/2010.02828, 2020. URL https://arxiv.org/abs/2010.02828.

[19] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017. URL http://arxiv.org/abs/1710.05941.

[20] Ashutosh Singandhupe and Hung Manh La. A review of slam techniques and security in autonomous driving. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 602–607, 2019. doi: 10.1109/IRC.2019.00122.

[21] Randall Smith, Matthew Self, and Peter Cheeseman. Estimating uncertain spatial relationships in robotics. volume 1, pages 435–461, 01 1986. ISBN 9780444703965. doi: 10.1109/ROBOT.1987.1087846.

[22] J. Sola and J. Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, 1997. doi: 10.1109/23.589532.

[23] Stanford Artificial Intelligence Laboratory et al. Robotic operating system, 2018. URL https://www.ros.org.

[24] Sebastian Thrun and Michael Montemerlo. The graph slam algorithm with applications to large-scale mapping of urban structures. *The International Journal of Robotics Research*, 25:403 – 429, 2006.

[25] Grady Williams, Paul Drews, Brian Goldfain, James M. Rehg, and Evangelos A. Theodorou. Aggressive driving with model predictive path integral control. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1433–1440, 2016. doi: 10.1109/ICRA.2016.7487277.

[26] Roman Yurchak. Performance of python code, 2021. URL https://github.com/pyodide/pyodide/issues/1120.

[27] Marcel Zeilinger, Raphael Hauk, Markus Bader, and Alexander Hofmann. Design of an autonomous race car for the formula student driverless (fsd). In *Proceedings of the OAGM and ARW Joint Workshop*, Vienna, 05 2017.

Appendix A

Abbreviations

GT	ground truth	28
EKF	extended Kalman filter	3
MLP	multi-layer perceptron	7
ML	machine learning	3
CNN	convolutional neural network	3
TSP	traveling salesman problem	17
SLAM	simultaneous localization and mapping	1
FSD	Formula Student Driverless	1
ROS	Robot Operation System	6
MPPI	model predictive path integral	4
ReLU	rectified linear unit	7
NN	neural network	7
IMU	inertial measurement unit	1
FPS	frames per second	34