Map Generation in Autonomous Racing

A Comparision of a Classic Heuristical Algorithm and Machine Leaning

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
21. 03. 2022

Abstract

- advancing technology in automation of driving and in controlled environment racing - reconstruction of abstract racing map from camera and lidar input, using slam output or using direct output - implemented in two ways a classical approch using foo bar and heuristics - and machine learning approach using mlp, cnn, etc.

Acknowledgements

Optionale Danksagungen

Table of contents

Todo list

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 [**?**]. Autonomous racing sets 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 innovation to take over the whole industry pushing progress further [**?**]. One example of such competition is **fsd!** (**fsd!**).[1] **fsd!** challanges teams across the world to build cars that can atonomously drive around fixed tracks that are defined by different colored cones. One car is racing at a time and is competing for the fastests lap rounds.

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 surrounding, to do this sensory input such as camera images, LIDAR data and odometric input from an **imu!** (**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 **slam!** (**slam!**) algorithms [**?**] and is not part of this thesis. On the other side the controlling of 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 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

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

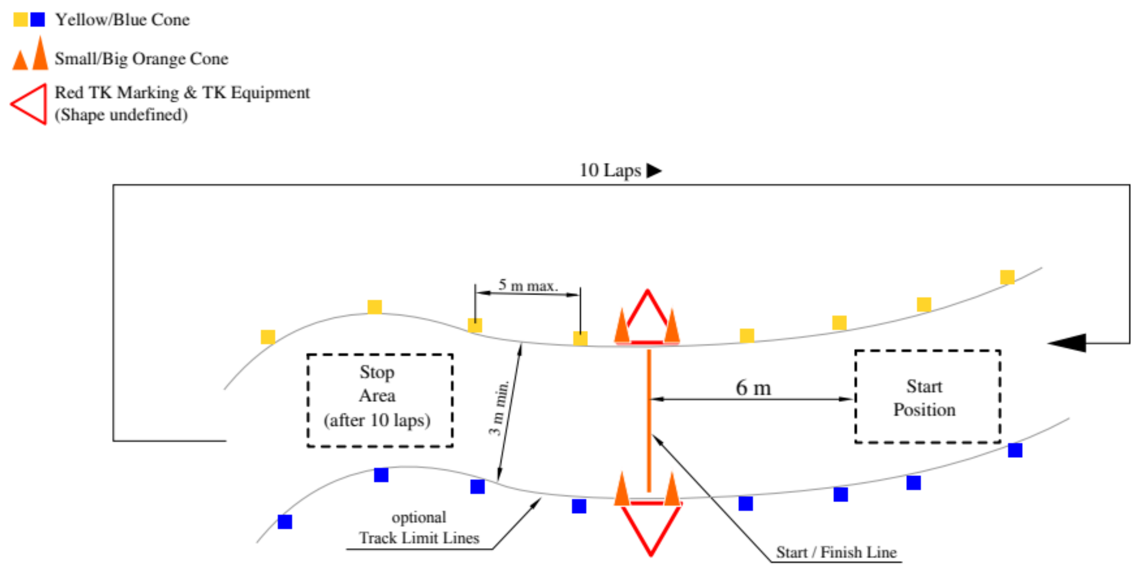

*Fig. 1.1 Layout of an **fsd!** track (Source: FSG21 Competition Handbook, p.14, "Figure 2: Trackdrive")*

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 previously worked on classical algorithm for map generation 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 which make the algorithm not yet useable in practice. For three of these problems this theses suggests an improvement. These are:

- Robustness against the incorrect detection of the color of landmarks (misdetection), missing landmarks completely (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 can't be corrected renders the resulting map completely unusable. Also, non-detections are completely ignored, with leads to problems especially in narrow curves while 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 real time

1.3 Related Work

Many works in the field of autonomous driving can be found, however, all those works focus on key aspects that differ from this thesis in one or more ways.
With regards to the classical approach to map generations several techniques have been documented. The following Papers apply a classical algorithm specifically to the Problem of autonomous racing in **fsd!**. AMZ Driverless [**?**] as well as Andresen et al. [**?**] focus on an architecture using an ordinary **slam!** in conjunction with a Delauney triangulation do to path planning. Zeilinger et al. [**?**] as well as KIT19d [**?**] use an **ekf!** (**ekf!**)-**slam!** to derive the center line for trajectory planning directly. Also, 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 **ml!** (**ml!**) to the problem of map generation and path planning in **fsd!** specifically. Dewing [**?**] used a **cnn!** (**cnn!**) to solve autonomous driving in a virtual racing game. While Dziubiński[2] documented the use of a **cnn!** for steering a toy car in free 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 [**?**]. Georgiev implemented Williams et al. [**?**] **mppi!** (**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!**.

[2]https://medium.com/asap-report/training-a-neural-network-for-driving-an-autonomous-rc-car-3906db91

To the knowledge of the author, no full **ml!** approach has been made specifically in the context of map generation in **fsd!**. Also, no comparison to a classical approach in **fsd!** has 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 basics and technical background is 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 is presented.
In the 4th chapter the approaches are evaluated and compared, and in the last chapter the results are summarized and several improvement ideas and ideas for future work are listed.

Chapter 2

Foundations and Technologies

2.1 Raceyard and Formula Student

Formular Student is global competition for building racing cars. The subclass **fsd!** is focused on autonomous driving and is spit 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, so all data about the track must be gathered and processed in real time with no prior map. Since 2005 Raceyard is the Team from Kiel for Formula Student and aims to compete in **fsd!** in the upcoming competitions.

Fig. 2.1 The T-Kiel A CE, one of Raceyards latest cars (Source: https://raceyard.de/autos/)

2.1.1 The Rosyard Pipeline

The software that is to be used in **fsd!** by the Raceyard car is called "Rosyard" which is build on the **ros!** (**ros!**) [?]. 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 feed forward fashion. The pipeline processes sensory data as input to control data that can be used to move the actuators of the the vehicle as output. The pipeline consists of five stages which are each represented by one or mode nodes plus sensory input:

1. Input/Detection: sensory input from cameras and **imu!** and preprocessing
2. SLAM: extract landmarks and locate them in a virtual map
3. Estimation: estimate centerline in the virtual map
4. Driving: given map data decide steering and velocity
5. Lowlevel: hardware controlling

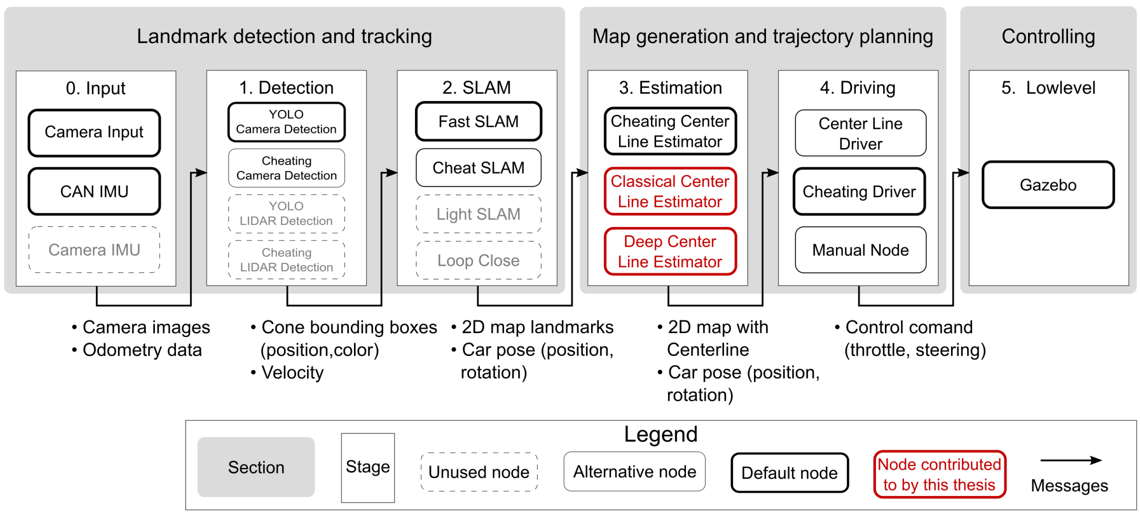

Fig. 2.2 Visualization of the Rosyard pipeline, with stages that contain one or more nodes and the topics that are published and subscribed to by the nodes, represented by messages (Source: adapted from https://git.informatik.uni-kiel.de/las/rosyard/-/blob/master/docu/images/overview.png)

This thesis focuses on the implementation of the 3rd stage. Given the landmarks located in a virtual map from the **slam!** this node should estimate the course of the track, such that the 4th stage can successfully drive the car along the track. The pipeline is fully dockerized and runs in four different docker containers: the master node 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 Leaning

Machine Leaning describes a class of algorithms that have the ability to improve automatically, this process of improving is known as learning. Three different categories of learning can be distinguished, supervised learning, unsupervised learning and reinforcement learning. 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 **nn!** (**nn!**) can be modelled 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 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 **relu!** (**relu!**) $f_{ReLU}(x) = max(a,x)$.

2.2.1 Deep Learning and Multilayer Perceptions

Multiple Perceptrons can be arranged in layers to form a special kind of **nn!**, called **mlp!** (**mlp!**). In such a layer a perceptron may only have a connection to perceptrons in the directly succeeding layer. A layer that has the maximum number of connections to the previous layer, such that each neuron is connected to each neuron in the previous layer is called fully connected layer. A **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 2 hidden layers the decision boundary of a **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 [**?**]. In recent years the research primarily focuses on networks with an even greater number of hidden layers. Such networks, with a big number of layers are called deep networks. Since Deep networks most often use non-linear activation functions the optimal weights cannot be found analytically, other algorithms for learning must be used, called deep learning. One of those algorithms is backpropagation which uses

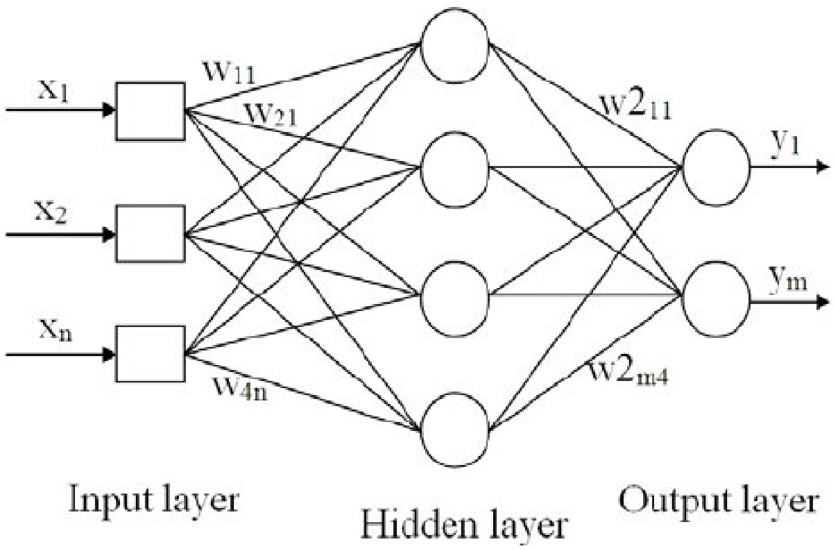

Fig. 2.3 A schematic diagram of a Multi-Layer Perceptron (MLP) neural network. (Source: Figure 5, An Oil Fraction Neural Sensor Developed Using Electrical Capacitance Tomography Sensor Data, Khursiah Mokhtar, 2013)

gradient descent to learn the weights as an optimization problem of the weights in respect to the desired output.

2.2.2 Convolutional Neural Networks

In **cnn!**s the concept of **mlp!**s is extended by adding convolutional and pooling layers in a **nn!**. Convolutional layers allow for processing a big number of inputs while not imposing a huge number of learnable parameters as a fully connected layer would. 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 inputs width and height, 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. 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 equal size chunks (e.g. 2x2) and computing an output for each of these chunks. Some commonly used pooling is max pooling, calculating

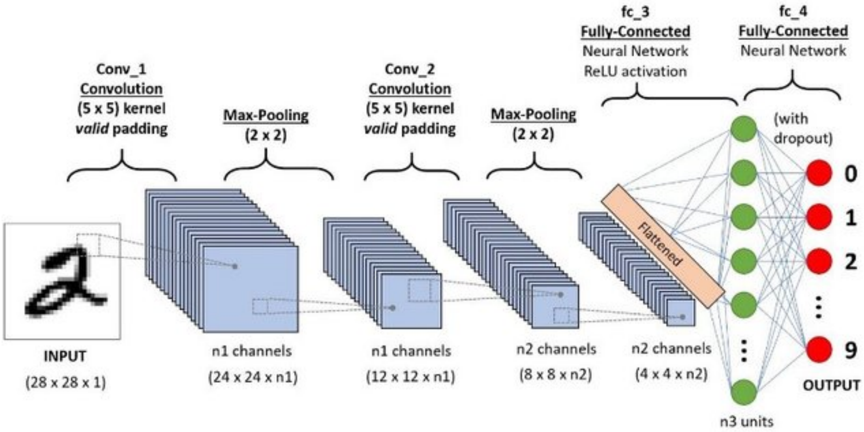

Fig. 2.4 Architecture of an ***cnn!****, an image as input is fed through mutiple convolutional layers and pooling layers. The output of these Layers is flattened and fed into fully connected layers to compute the output. (Source: Deep Learning model-based Multimedia forgery detection, Pratik Kanani,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

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 $[P_i, P_{i+1}]$ $i \in \mathbb{N}_{\leq n}$ forms a vertex in the polyline.

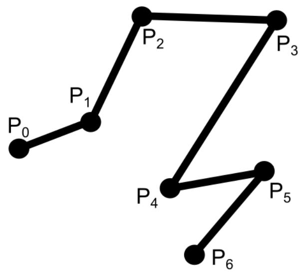

Fig. 2.5 A polyline over the vertices P_0 to P_6

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 though A and B and being tangent to $\vec{h}$ in A.

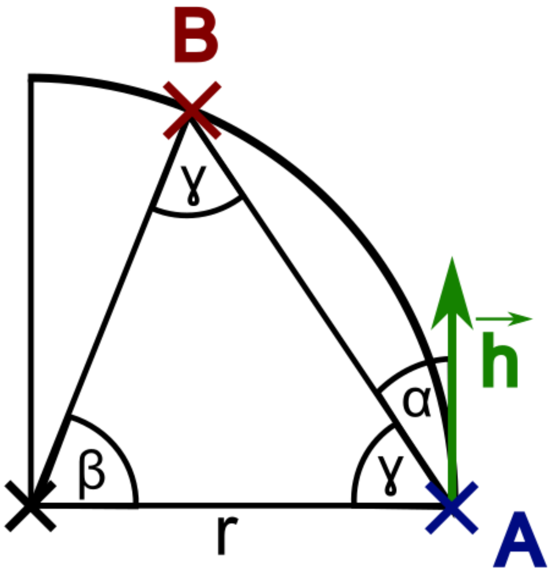

Fig. 2.6 Points A with heading $\vec{h}$ and B in circle with radius r, implying a curvature in point A of $1/r$. The circle center and B and A form an isosceles triangle with base angle γ and vertex angle β

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

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

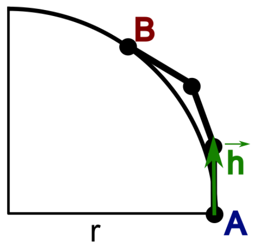

Fig. 2.7 Example curvature of $1/r$ approximating a polyline leading from A to B, the circle corresponding to the curvature has the radius r

Using this method we can calculate the average curvature of the curve that is tangent in A to $\vec{h}$ and passing though 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 seemingly a map is 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 simultaneous. While an exact solution is often not possible / or desirable computation cost wise, several methods exists 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 [**?**] which is based on particle filters. In FastSLAM particles are used as potential positions for the agent, at each time step 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 time step the particle with the biggest weight is guessed as the actual current position and reported as such. This leads to the problem of jumping 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 jumps in a non-continuous way. 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 jumps the landmarks cannot be associated to the previous landmarks because the transformation is non-continuous. The output of FastSLAM is the incrementally build 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

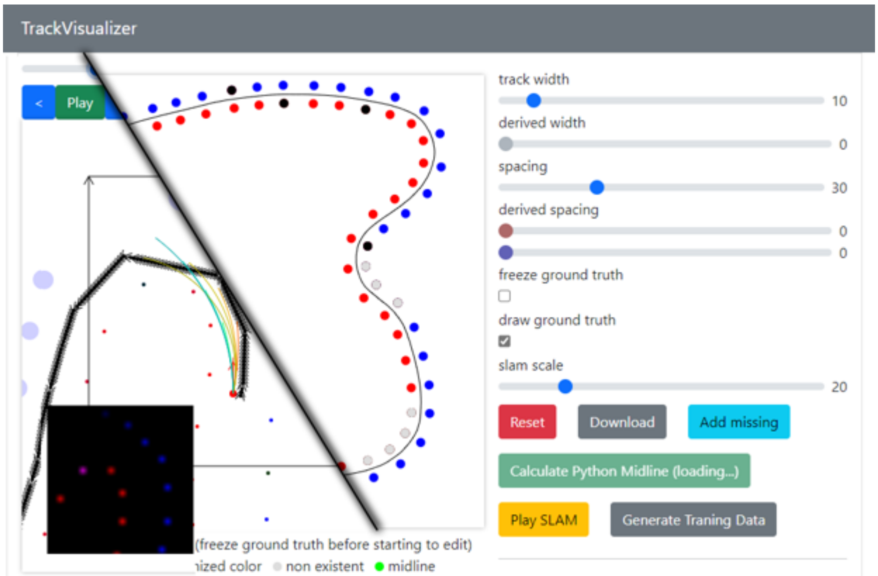

Fig. 2.8 Screeshots of the prototyping environment coded using web technologies that was used to develop and test the implementation of this thesis, in green a button can be seen that invokes python code in the browser to calculate the centerline of the currently drawn or loaded track

For developing and test 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 to use a dedicated server run 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 [**?**] which allows the execution of python code directly within a browser using WebAssembly. As Opposed to other systems Pyodide doesn't cross compile python to JavaScript but uses a python runtime to execute python code on demand. Also, many of the most used scientific python libraries, e.g. NumPy, SciPy, Pandas and Mathplotlib are supported out of the box, which makes it useable 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 [**?**][**?**]. Adding Pyodide to the dev 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 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, while

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

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

in higher dimensions. Next, the data is sorted using a naive closest neighbor sorting, and 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 closest point to it on the other side.

This partly naive approach leads to problems when used with artificially constructed data that has anormalies in it or when used with sfimulated or real input data as well.

The following picture shows application of current algorithm on artificially created data, that contains some mis-detections and non-detections.

For better visibility we choose Red to symbolize yellow cones, blue for blue cones, black represents cones with unknown/uncertain color so misdetections, Grey represents non-detections. The black line is the grounth truth of the centerline that was used to generate the data. The green line represents the centerline that is calculated by the algorithm. This exmaple illustrates some of the problems the current implementation has: It complete ignores nondetections, which leads to big deviations from the ground truth centerline when non-detections accumulate in a corner, as seen in the upper right corner. Misdetections lead to strange behavior, the reclassified cones causes the calculated centerline to deviate to the side in direction of the reclassified cone.

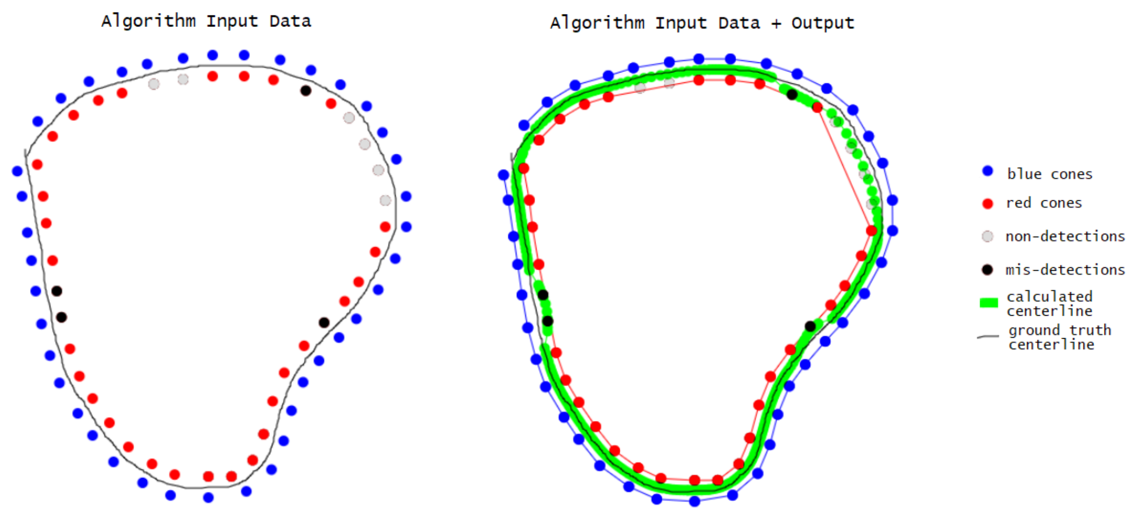

Fig. 3.1 Application of the unmodified algorithm of Vaishnav/Agrawal to artificially created data that has some non-detections in the upper right corner and center, and some misdetections where no color was asigned spread across the track, the application shows that even slight imperfections in the input 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 closest point respectively until all points are used. This approach, however, leads to errors especially when parts of the path are close together. Especially in those erroneous cases one can observe that the correct path is the shortest possible path though the point cloud.

This means the Problem of finding a path to a given pointcloud can be modelled as the

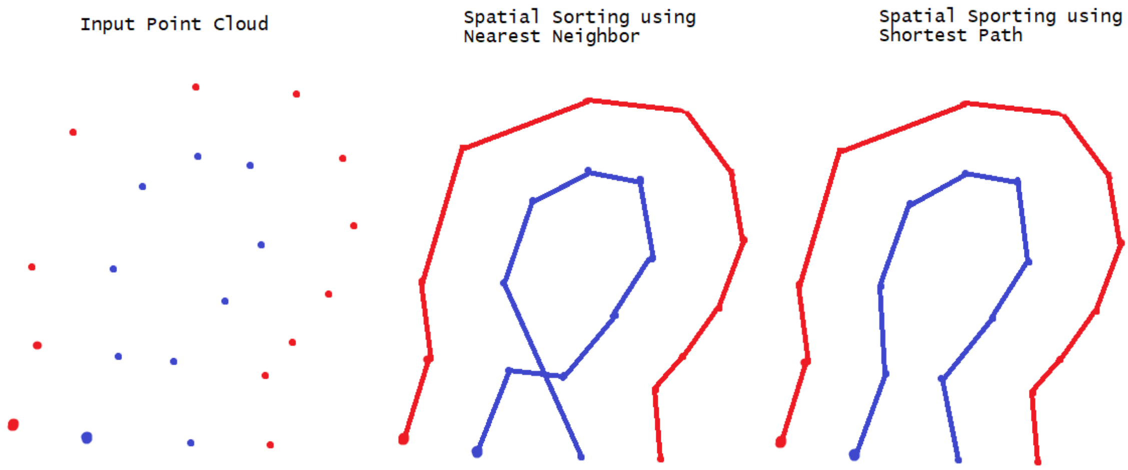

Fig. 3.2 Example for an erroneous spatial ordering: Given the pointcloud on the left, the pointclouds are sorted once according to the nearest neighbor algorithm starting at the marked larger point, and once according to the shortest path overall, the shortest path is the correct ordering

tsp! (**tsp!**). Given that the **tsp!** is NP-hard[**?**] it cannot be solved exactly while being efficient enough to be used with a larger number of points in realtime. The Algorithm of Christofides and Serdyukov was the ideal solution, leading to a better solution than a naive approach, while stil having an acceptable complexity of $O(n^2 * log(n))$[**?**]. This meant that using Christofides algorithm instead of nearest neighbour 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 privously not accounted for at all. Giving the following scenario 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 at all. This leads to many non-detections on that side of the track while the other side can still be detected. This improvements 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 for each cone

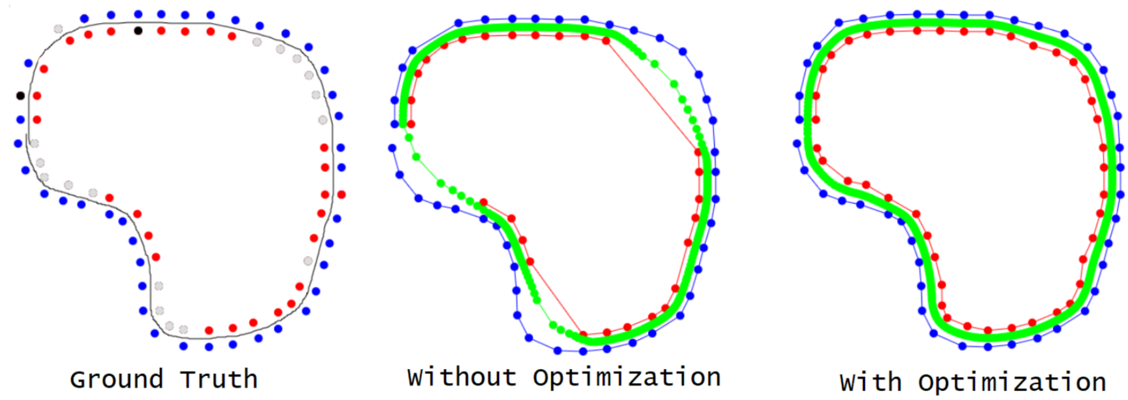

Fig. 3.3 non-detections, and their handling using the old approach and the new approach)

whether it has a cone roughly on the other side of the track that corresponds to it. And if not, 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 2 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 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 take a look at B: First, the bisecting angle $\alpha = \measuredangle ABC$ and the bisecting line b to α is

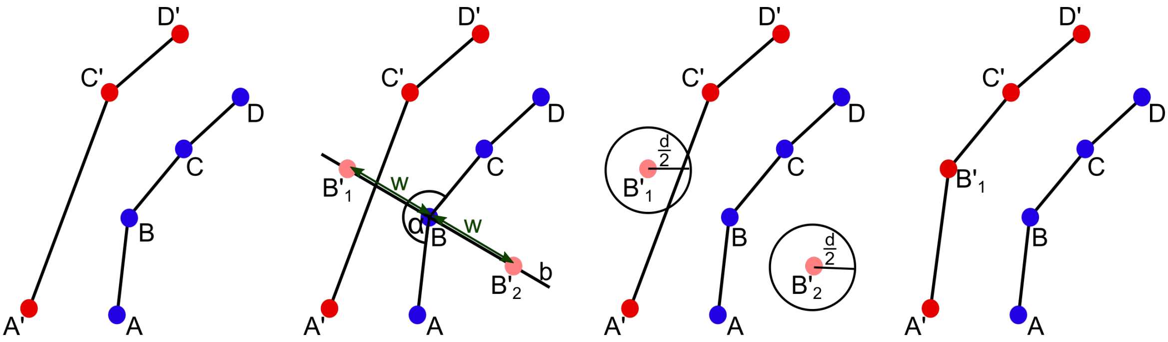

Fig. 3.4 Illustration of the guessing of missing cones where a cone is added

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 thrid 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

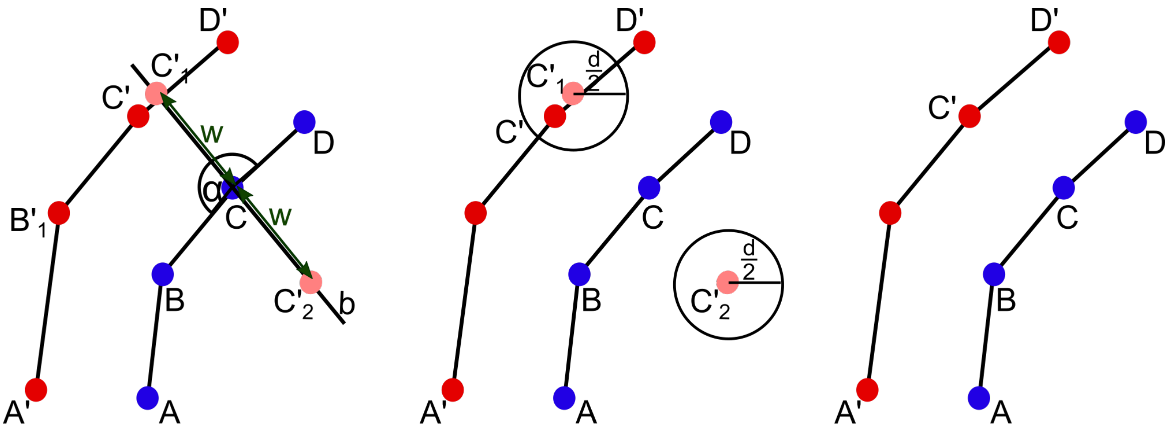

Fig. 3.5 Illustration of the guessing of missing cones where no cone is added

found in $Cones_B$ around the proposed points C_1' and C_2', and such, no point is added.

3.1.4 Third Improvement - Covariance Filtering

The thrid improvement that proved itself useful especially when used with simulated data instead of artificially created data, is the incooperation 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 threshhold based filter before passing

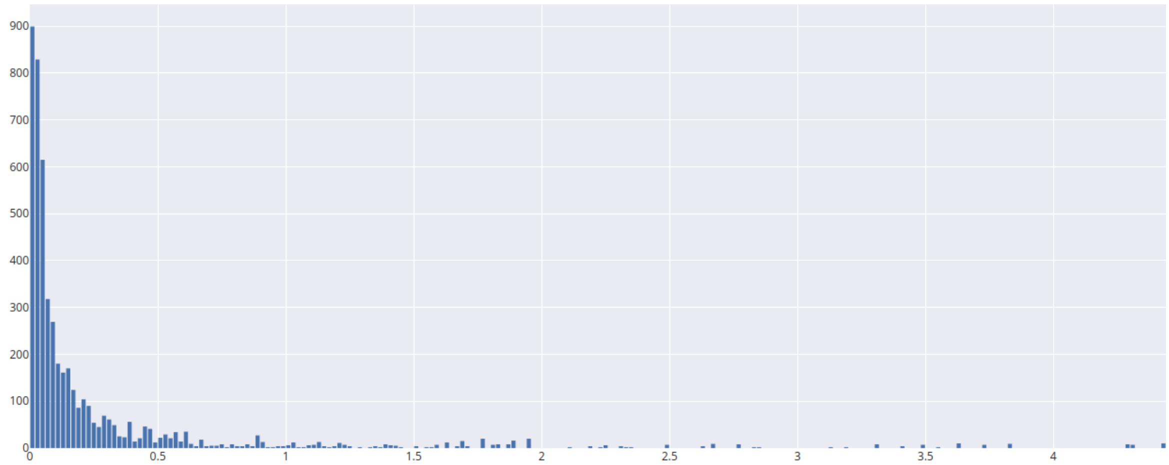

Fig. 3.6 Distribution of uncertainty in landmark detection over some simulated track drives

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, in opposite to than the overall certainty of the position, we can simplify the covariance matrix into a single scalar uncertainty value c by summing over the absolute value of its entries $c = \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|$.

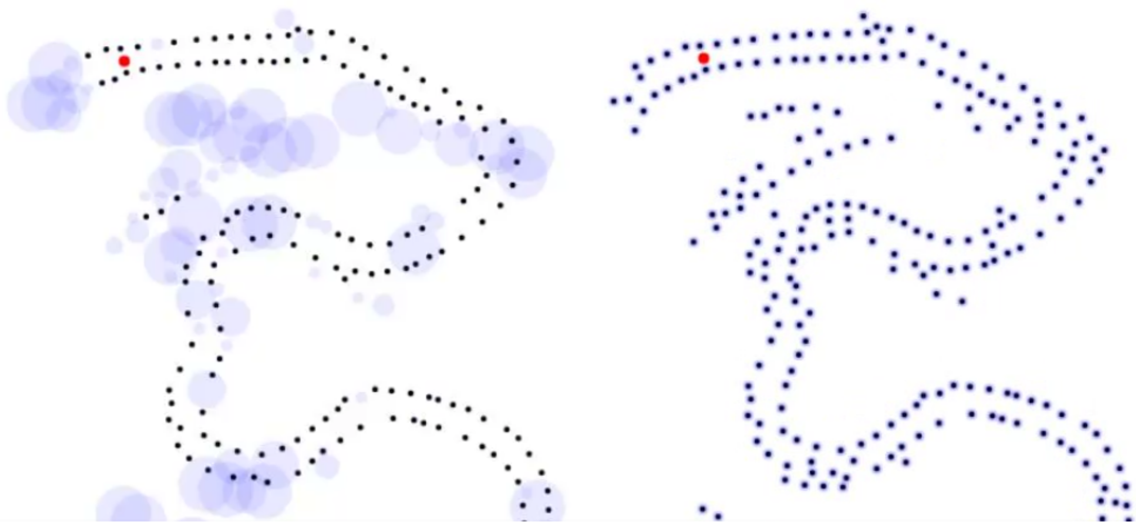

Fig. 3.7 Simulated track driving with different threshold filters, points left after filtering are marked as a black point, points filtered are vizualized as light blue circle with a radius proportional to the uncertainty, left side $c_\theta = 0.1$, right side original unfiltered data

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

$$c_\theta = 0.05$$

was found to be 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, while also the history of these local predictions can be later used to reconstruct the overall map. The local track surrounding the driver, especially in the direction of driving, can be modelled using the centerline alone, given that the track width is constant, furthermore the course of the centerline can be modelled using discrete curvature, since we can assume that certain parts of the track have a constant curvature. This can be illustrated by taking a look at the course of a typical **fsd!**track [3]: it consists of straight parts with approximately curvature 0 and curves which are distinct 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 differently distant points can be used that describe the course of the track up to an increasing distance, as seen later for example, five curvatures that estimate the course to a point 2m to 10m along in the direction of driving in 2m steps.

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, thereby making it difficult to use a standard fully connected **nn!**, since the number of input neurons would need to be fixed.

3.2.2 Modelling 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, 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 prepocessing the input data before fed to the **nn!**. In the concrete implemention some parameters were choosen heuristically and later verified to suffice experimentally.

Sourrounding the driver a with a sample radius $TSP = 8m$ a square patch of space is used to

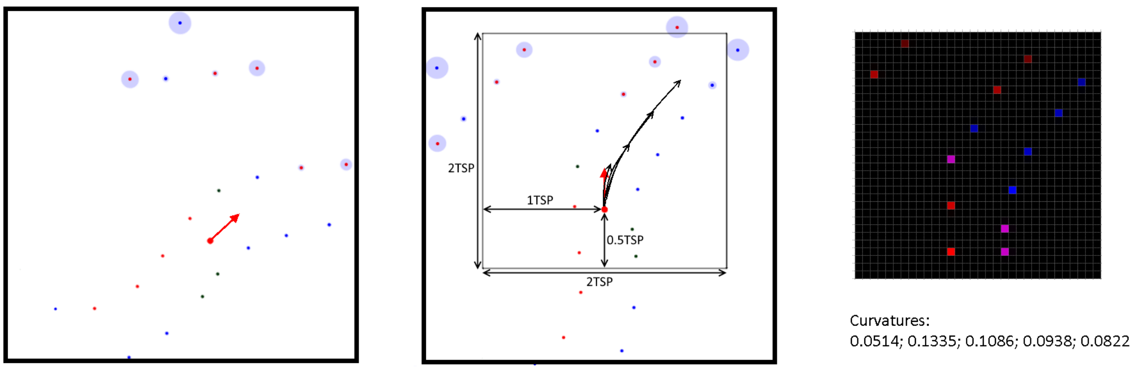

Fig. 3.8 Preprocessing of the cone data for the ***cnn!****. The car is represented by the larger red circle with its heading as arrow, first the map is rotated and moved to the local coordinate system of the car, a region according to TSP and $Car_{position}$ is selected and transformed into an Image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

generate in input for the **nn!**. inside this square the driver is centered vertically and horizonally offseted such that the drive 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 $1.5TSP$ in front, $1TSP$ to either side, and $0.5TSP$ 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 $0.5m/pixel$, considering the track width of at least $3m$ according to the **fsd!** rules[4] while keeping the number of inputs small. 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 arbitrary 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, to further flatten the distribution it is squared and lastly inverted along the x axis. This lead to a much flatter

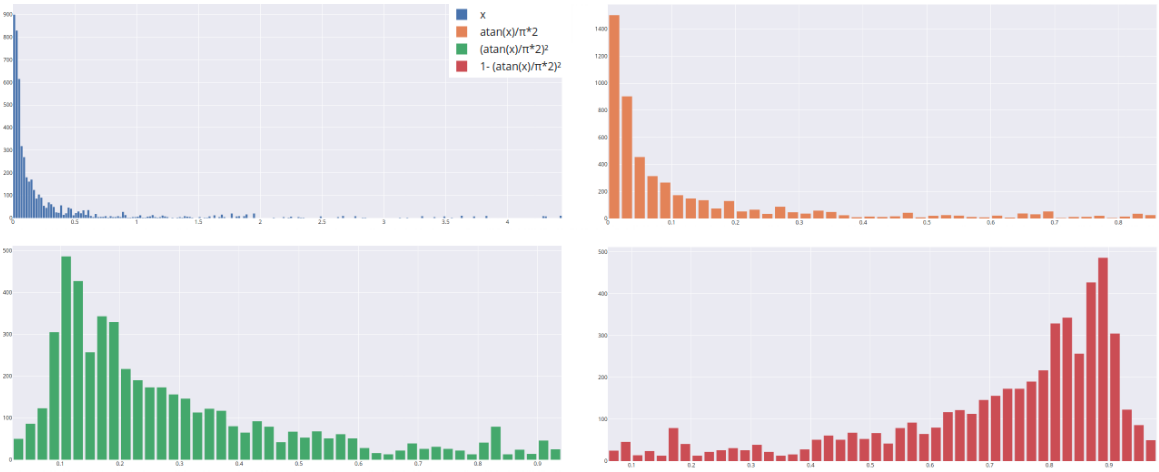

Fig. 3.9 Distribution of uncertainty of landmarks under some transformations: blue identity, yellow $x \mapsto atan(x)/2\pi$, green $x \mapsto (atan(x)/2\pi)^2$, red $x \mapsto 1-(atan(x)/2\pi)^2$

distribution that is bounded 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!**[**?**] than the very sharp 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 though, the discrete curvature from the current position on the centerline with a heading that is tangent to the centerline in that point, and a point on the centerline that is $2m,\dots,10m$ further away on the centerline respectively. Using the raw curvatures, however, 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

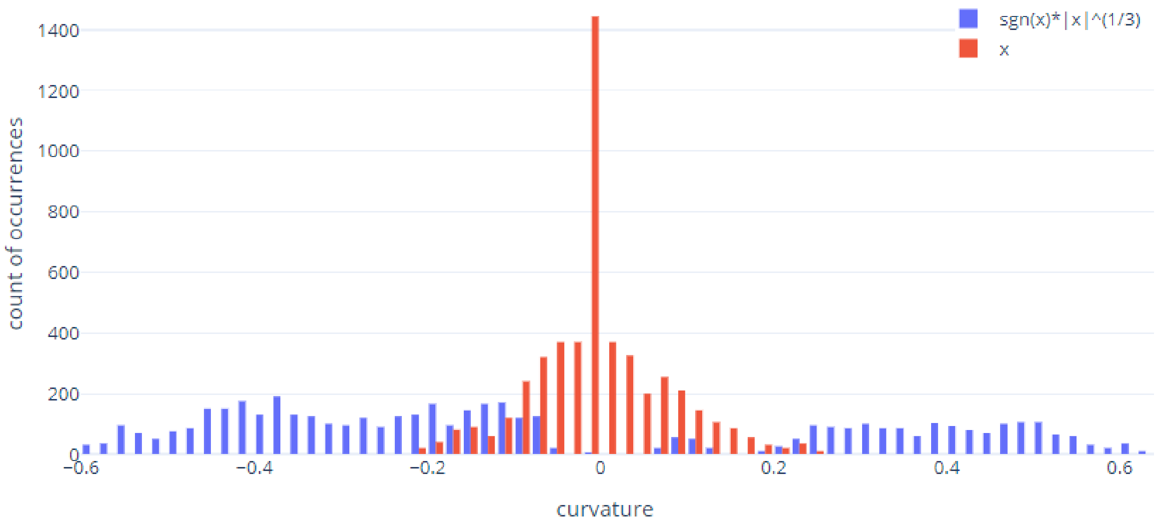

*Fig. 3.10 Distribution of curvatures in **fsd!** tracks 2019 with transformation: blue identity, red* $x \mapsto sgn(x) \cdot |x|^{\frac{1}{3}}$

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[**?**] and AlexNet[**?**] architecture was used. 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 classication as used in LeNet-5[**?**] and AlexNet[**?**]. .

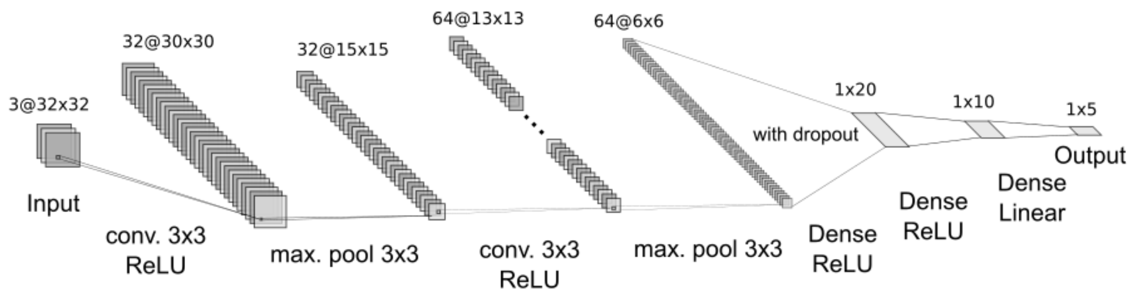

*Fig. 3.11 Architecture of the **nn!** used for the **ml!** algorithm, 2 convolutional layers with 3x3 kernels and subsequent max pooling with 3x3 kernel respectively, 2 dense layers with **relu!** activation function with 20 and 10 neurons respectively and dropout in the first layer and the output layer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

3.2.3 Training

For the training data from simulated drive-troughs were used to generate one training sample per frame in the data. The original unaugmented data was used from simulated tracks from tracks that were used in the **fsd!** competition of the last years.

Chapter 4 440

Discussion 441

4.1 Evaluation 442

4.1.1 Classical Approach 443

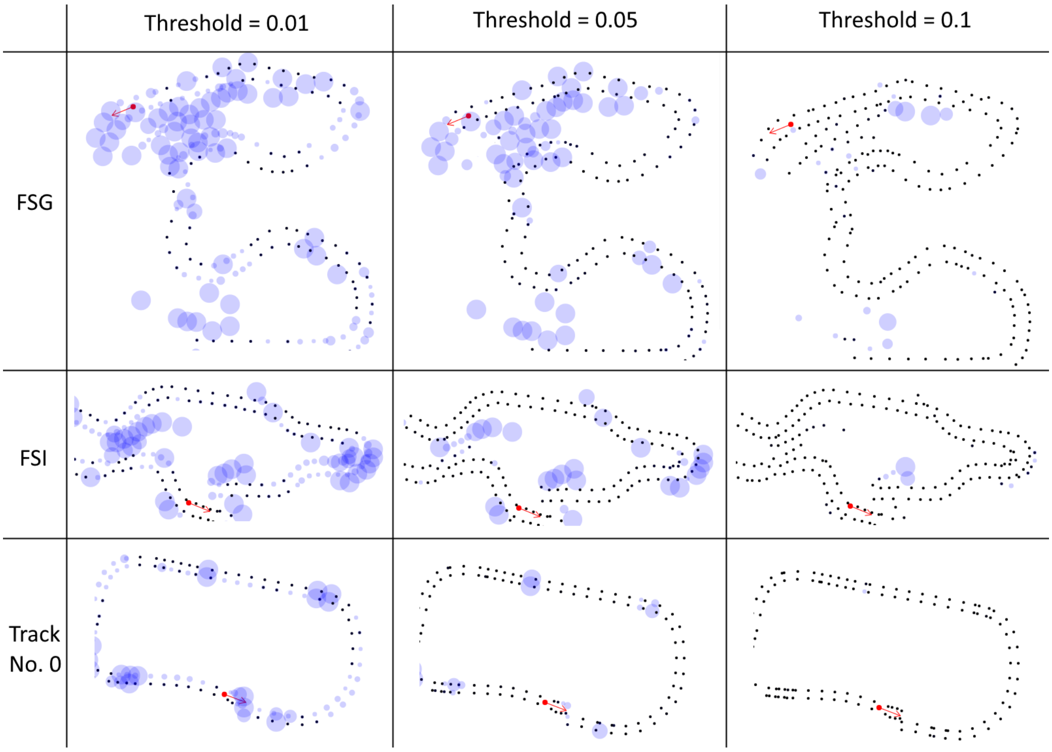

Fig. 4.1 The resulting landmarks after filtering using different certainty thresholds, $c_\theta = 0.01$, $c_\theta = 0.05$ *and* $c_\theta = 0.1$, *on three different* ***fsd!*** *tracks "FSG", "FSI" and "Track 0",* $c_\theta = 0.05$ *has the best results.*

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 with a smaller threshold of $c_\theta = 0.01$ too many landmarks are filtered such that the resulting map is incomplete. With a larger threshold of $c_\theta = 0.1$ too little noise is filtered out resulting in over-detections in the map.

A problem that occurred in developing more advanced means of filtering noisy data is the inability to track landmarks across frames. While it should be in theory 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 continues transformation, given small enough time steps between frames. This spatial matching, however, is not practical for performance reasons, as it would be computationally expensive. Also, the implementation of the particle **slam!**s uses different particles to determine the current position in the virtual map of which one is chosen with the highest certainty 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 non continuously 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 new from one frame to another.

Analysis - Deviation From gt! (gt!)

In the following all example figures will use the same structure/color scheme. Each figure consists of two parts: the input for the classical algorithm on the left for clarity and the input with 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 yellow cones were replaced by red ones for better legibility. Very large light blue circles represent cones that were filtered out by the covariance filter, the radius of the circle represents the uncertainty of a particular cone. Black dots represent cone misdetections where no color was detected for certain. Grey dots represent cones that are not detected at all, nondetections. The output of the algorithm is vizualized in 3 parts: The large green dots represent the points of the calculated centerline, the light blue and light red small points represent the two sets of cones that are the result of preprocessing the cones, namely spatial ordered and readded missing cones. The arrows connecting these three parts mark the order these lists of points are in contained in the output arrays. This ordering can be used to see the orientation (clockwise/counter-clockwise) of these lists.

While the centerline generally works really well with perfect data, it starts to produce

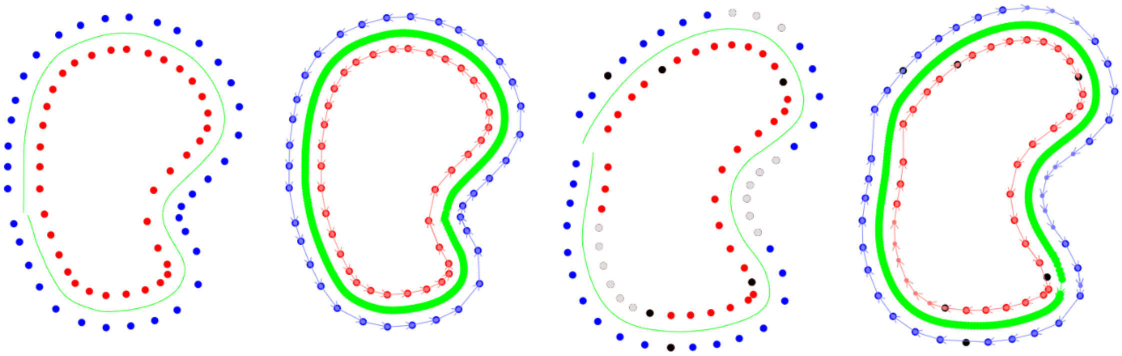

Fig. 4.2 Examples where the classical approach works very well: Unaltered artifically created data on the left and artificially created data with non-detections and misdetections with no color certainty on the right

less and less usable output with increasing errors in the input data. Artifically created data, as well as artificially created data with several non-detections and misdetections where no color is identified is handled very well.
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. Appliying the Algorithm to Simulated **slam!** data gives mixed

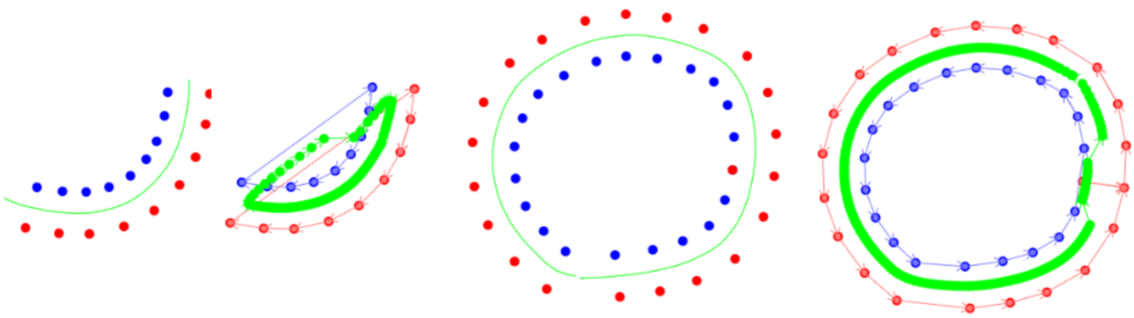

Fig. 4.3 Anormalies the classical approach can not handle: cones with misdetected certain colors on the right and non closed loops on the left

results depending on the particularities of a certain frame. 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.

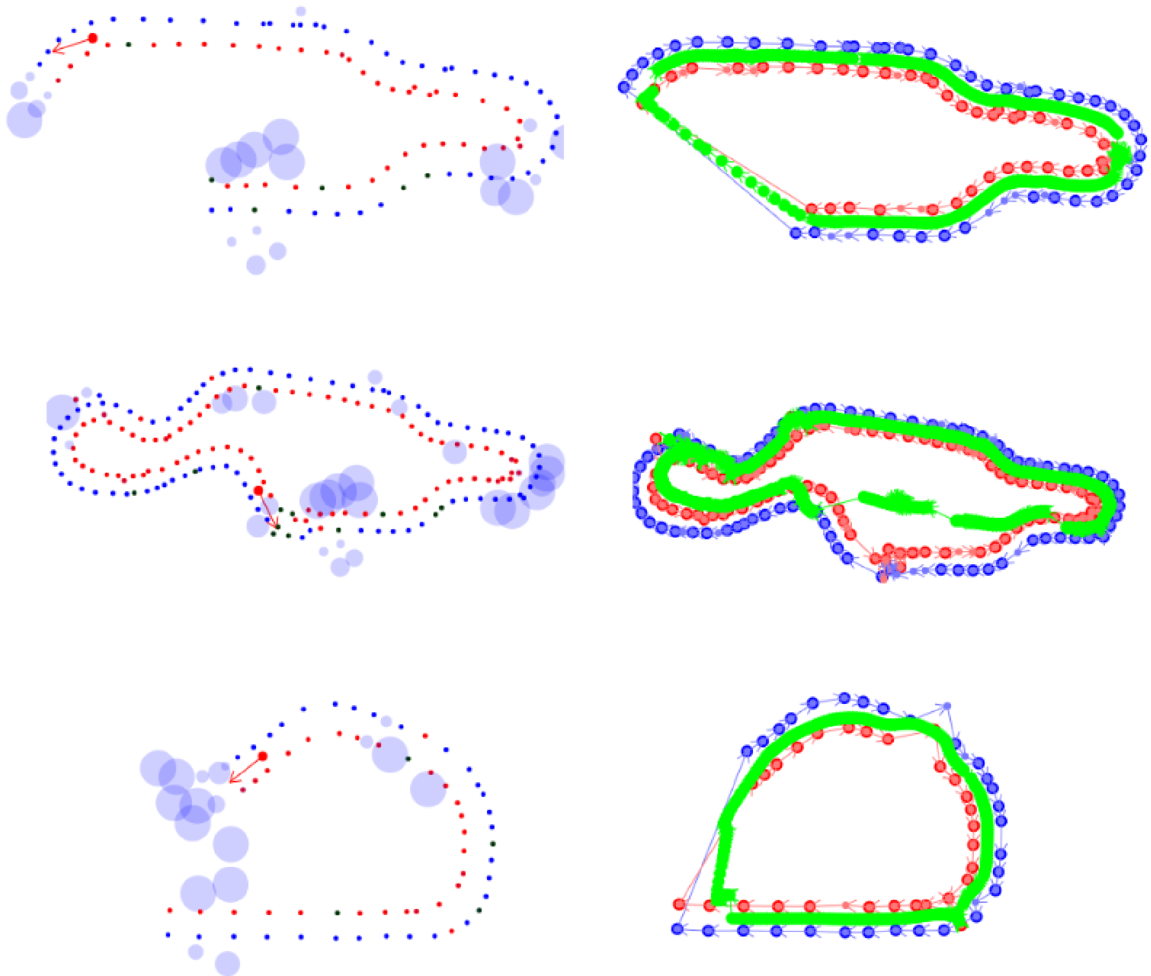

Fig. 4.4 The classical approach applied to simulated track data. In the upper example no color misdetections are present and overdetections are rare, which makes the algorithm perform well. In the middle example overdetections and color misdetections lead to a misjudgement of the orientation which breaks 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 optimizer and a learning rate of 0.001 these parameters were chosen heuristically and were proved to be succinct. The loss converged quickly and such 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

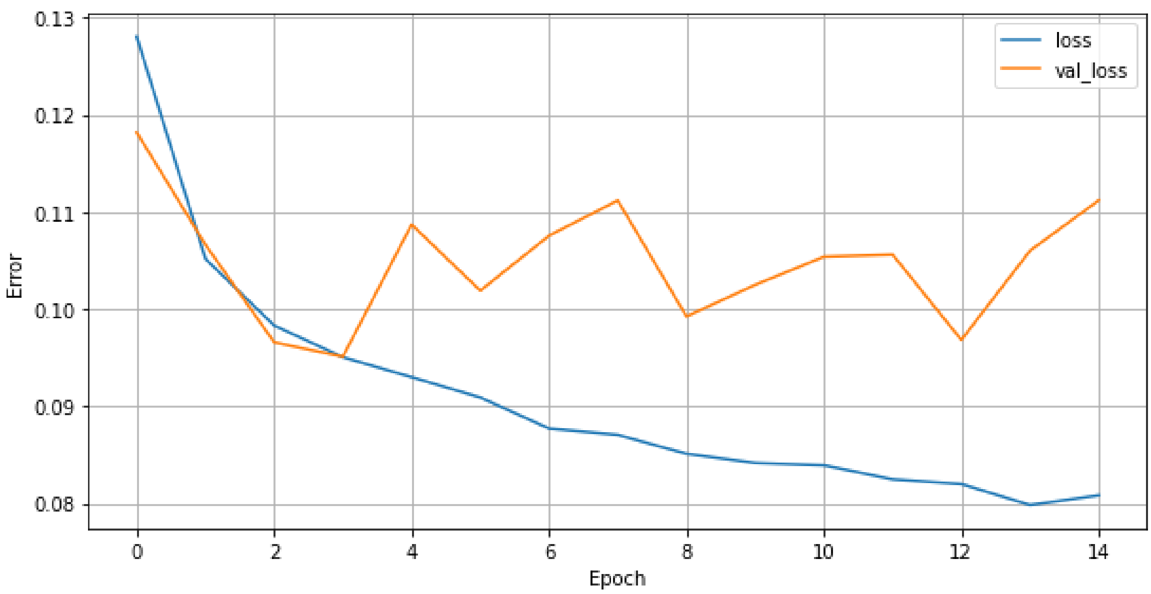

Fig. 4.5 The Training and Validation loss after 15 epochs with a batch size 2, 1400 training samples and 600 validation samples, ADAM with mean average loss, and 0.001 *learning rate*

Metric - 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:

$$steering_0 = k\kappa$$

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

Even though the neural network was trained only using training samples where the driver is centered on the track a deviation from centerline is interpreted beneficially as curve leaning

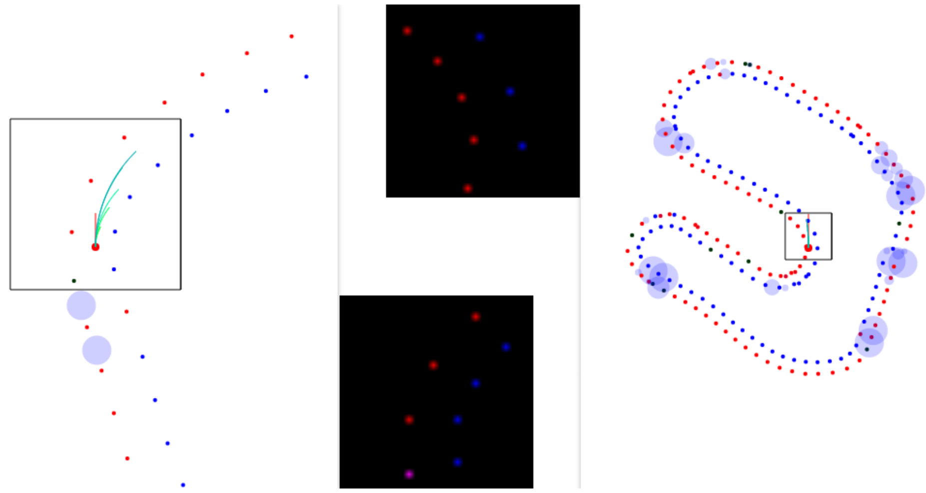

Fig. 4.6 Track 1 being driven autonomously by the machine learning approach and a simple test driver. On the right a moderate right turn can be seen along the image the ***nn!*** *sees. On the right an overview of the track can be seen while the car drives a moderate left turn along the image the* ***nn!***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

towards the opposite direction, which forms a feedback loop that keeps the car in the middle of the track.
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 round 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 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 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 compute 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 so that it can be used in the

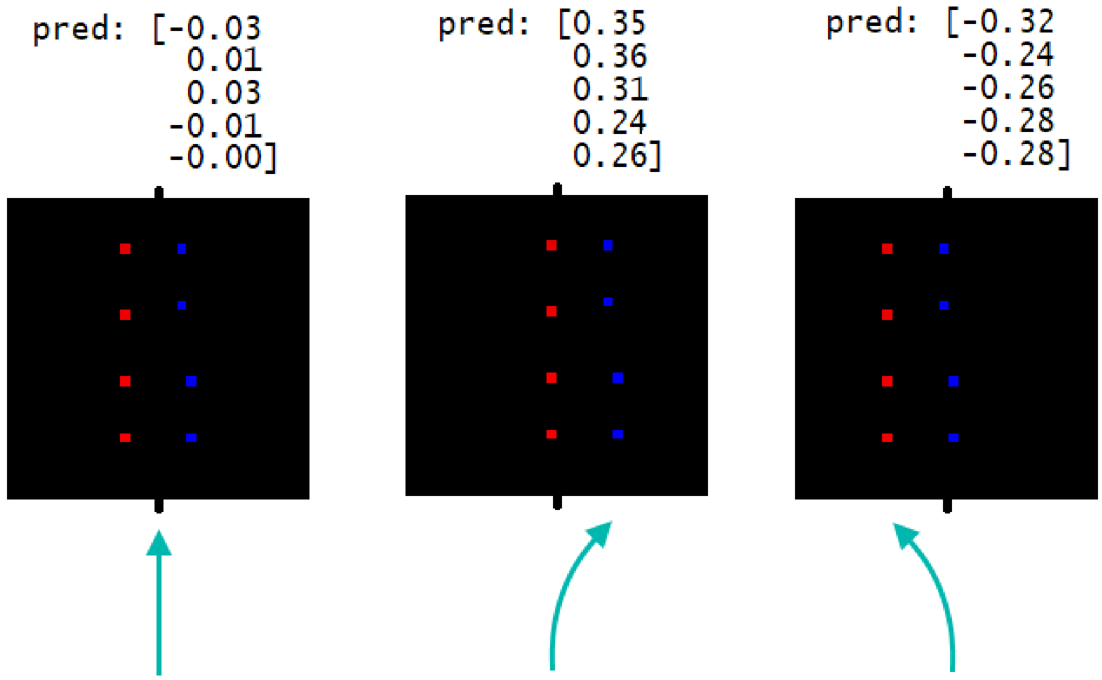

*Fig. 4.7 The same image fed into the **nn!** with three different x offsets, centered, positive offset and negative offset, the effect a deviation from the track center has on the prediction of curvatures makes the model predict a curve that lets the car return to the center in a closed feedback loop.*

first lap. Is more robust against erroneous data but produces only local 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 round.

Overall the classical approach produces precise centerline data when provided with the right data, but is so fragile that it fails to produce usable output consistently when used with simulated data, thereby rendering it - as is - useless 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 big advantage the **ml!** has over the classical approach is its runtime. An average track contains around XXXX cones which 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. The system used to measure the runtimes runs on a Intel(R) Core(TM) i7-5820K and NVIDIA GeForce GTX 980 Ti. The following runtimes were obtained over an average of XXXX frames

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

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

171 910 1948 93 622 2397 330 927 5540 50 100 185 50 101 182 50 101 269 $0.0005x^3 - 0.1368x^2 + 24.2215x - 733.9375$

Chapter 5

Conclusion

5.1 Summary

Fasse nochmal alle Ergebnisse der Arbeit zusammen. 0.5p

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 double detection 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 [**?**] as well as using different **slam!** algorithms entirely such as EKF SLAMs [**?**] or GraphSLAM [**?**]. As improvements to the input data have a positive effect along the rest of the pipeline that follows these improvements could contribute a big 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 round, therefore it cannot be used in the first round to drive along the track in the first place. 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 round as well.

Also, when used in the first round, the estimated position of the car, can be factored in to determine the importance of cones, such that potential double detections of distant track parts can be ignored this way. 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, as there is a correlation between the color of cones and the position relative to the car, given that the car has not left the track.

5.2.3 Machine Learning Algorithm

As only original unaugmented data was used, one simple way to improve 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 where the car is perfectly centered in the track. Such deviations are handled implicitly instead of handling deviations explicitly. One way of handling 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 to use 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 real life, by contributing to the pipeline one of the last essential implementations 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.

Appendix A

Abbreviations

Appendix B

TrackVisualizerJS Documentation