UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Università degli Studi di Padova

Department of Information Engineering

Master Thesis in Control Systems Engineering

Development of vision-based soft sensing techniques with training in virtual environment for autonomous vehicle control

Supervisor: Bruschetta Mattia

Master Candidate: Matteo Grandin

1 Abstract

The goal of this master thesis is to develop an original approach to lane estimation for scaled vehicles using a front-mounted camera and convolutional neural networks. The key components of this estimation process are the fact that all the training is performed in simulation using a noisy path; and the online inference is performed on low-end hardware (Raspberry Pi 4) in an efficient and responsive way, while being very accurate. The heading error of the standard pure pursuit controller is chosen as estimation target. A clothoid based centerline has been chosen as training path for its several advantages in the analyzed scenario. Different performance metrics are evaluated and the standard deviation of the error is found to be the more effective. An analysis on the hyperparameters (image dimension, lookahead distance, training variability, and others) is performed in order to find the best combinations and evaluate the impact of each parameter. From the results in a real world scenario a very small network and image and a very high training variability resulted as the best overall combination, with the network complexity and training variability playing a major role in the accuracy of the system. The whole process is finally tested in a real life control loop achieving very good performance, allowing for precise lane tracking using delayless local estimation.

Contents

2 Introduction

This thesis project is placed in the context of autonomous driving and, more specifically, it addresses the problem of lane detection and keeping. The methods and the implementation are tailored for the specific application of the Bosch Future Mobility Challenge (BFMC), an international autonomous driving competition for 1:10 scaled vehicles, in which we competed and won as the DEI-University of Padova team.

To this day, the autonomous driving problem is a very complex problem, since it requires several components to work together, from a robust perception system, to an advanced path planner and controller capable of dealing with the uncertainties of the environment and the unpredictability of the other drivers. Several strategies have been developed, both in the literature and by various companies, in order to partially solve it. Lane detection and keeping is usually one of the most important building blocks, since it's one of the most frequent task that an autonomous vehicle must handle. It also has been tackled in many ways, although mainly using camera (or multi-camera) based strategies [1]. The most recent methods are mostly based on deep learning and convolutional networks [2]. These methods can perform very well even in challenging situations but are usually very computationally intensive and require a big amount of training data from real-world driving scenarios.

For the purposes of the BFMC, we are interested in designing and implementing a lane detection and keeping strategy that is able to perform well within the constraints of the competition; namely it should be able to run on a Raspberry Pi 4, a low-end single-board computer, and use a front-mounted camera. Moreover, it's important to notice that we are not able to access to competition environment before the actual challenge, but we are given a simulated version of the competition track where we can test our algorithms.

To address our particular instance of the lane detection and keeping problem,

we propose a strategy which is based on a camera-based estimator and a controller working together. The estimator is a convolutional neural network trained in the given simulation environment, that has as estimation target the heading error of the vehicle with respect to the centerline. The controller is a partially modified pursuit controller [3], which uses the estimated heading error as input. The controller is not object of this thesis, but have been extensively tested and improved by the work of Antonio Gallina, in his thesis, *Development of a pure pursuit lane keeping controller for a 1:10 scale autonomous vehicle*.

The main objective of this thesis is to develop a vision-based estimation strategy, capable of running on an embedded device that, together with the appropriate controller, is capable of keeping the autonomous vehicle in the lane, and tracking the center line in a gps-denied environment.

The proposed approach is based on a clothoid-based centerline interpolation for its several advantages and with the standard deviation as the performance metric of choice for its simplicity and effectiveness in our particular scenario.

The second part is dedicated to the description and implementation of the proposed approach, from the data collection process in the simulator to the dataset generation and training of the neural network. The different hyperparameters are introduced and the reasons behind their initialization are discussed.

In order to understand the importance of the various parameters, the proposed estimation method is extensively tested in a real-world scenario and an analysis on the most important hyperparameters is performed, with respect to the estimation capabilities and performance. From the analysis it emerges that the learning parameters like the number of epochs and learning rate play a major role, but once those are fixed, the amount of steering noise in the training datasets and the distance ahead of the heading error also become important.

Finally, the whole estimation and controller scheme is tested in a standard

control loop application in a real-world scenario to evaluate the tracking performance. The strategy is found to perform very well in both the analyzed scenarios, achieving perfect tracking at low speed and keeping the lane even at high speed.

3 Problem Statement

This thesis project is placed in the context of autonomous vehicles and autonomous driving: more specifically we want to target the problem of lane following in a vehicle equipped with a frontal camera. In particular, we want to estimate the heading error of the vehicle with respect to the center of the lane, using only local information about the car and the road, and with training of the system exclusively in a simulated environment. The idea behind the need of an accurate estimation of this road parameter is that it can be used in a standard control strategy to perform lane following or other driving behaviors.

The objective is to design a system which satisfy a set of reasonable constraint in a real drive situation:

- The system should be able to accurately estimate the vehicle heading error with respect to the center of the lane. The accuracy should be measurable and quantifiable using some appropriate performance metrics.

- The system should operate in a real-time environment, on low-end hardware, with low latency and high frequency. This is a strict constraint that limits the complexity of the system in order to be able to run it fast and efficiently on a Raspberry Pi 4.

- The estimation process should only use local information about the vehicle and the road, without the use of any other localization system or map; in order to increase the robustness in a loss-of-signal situation, and to mimic the way human drivers approach the same task.

- The system should operate on a real vehicle, in a real environment, with a real camera; but the training should be performed in a simulated environment. This is a reasonable constraint since it limits the amount of expensive real

world testing and data necessary to train the system.

3.1 Vehicle model and characterization of the heading error

The entire car assembly is composed of the vehicle and a frontal camera. The camera is mounted on the vehicle in a fixed position, located in an elevated point for better visibility and pointing ahead towards the road. The vehicle is a standard car with 4-wheel drive, with standard suspensions and frontal steering.

In the literature there exists several vehicle models, that span across a wide range in complexity and degrees of freedom [4]. The more accurate models are used when the vehicle is driving close to the limit conditions, for example at very high speeds and accelerations in a racing scenario; the less accurate, but simpler models can be used if the vehicle is driving at low speeds in average conditions. In this thesis we will use a simple model, and more specifically the *bicycle model* [5] to describe the vehicle, the reason behind this choice is that we are interested in developing a general framework that is focused on the vision system and basic physical quantities of the vehicle like speed and steering angle; in addition to this, in the model vehicle we were working on we were able to directly control only the steering angle and the vehicle velocity.

The Bicycle Model

The bicycle model represents the foundation for vehicle modelling, the more complete models are build on top of this model by removing or lightening some of its assumptions. The bicycle model takes a 4-wheel model and combines the front and rear wheels respectively to form a 2-wheeled model (hence the name). We can therefore deal with only 2 wheels and 1 steering angle instead of 4 wheels and 2 steering angles. The bicycle model is based on the following assumptions:

- The vehicle is operating on a 2-dimensional plane. This is a reasonable assumption, it means that the vehicle does not move in the vertical direction.
- The vehicle is a rigid body with its mass concentrated in the center of mass.
- No-slip condition: there is no lateral or longitudinal slip in the tires; therefore we can assume the velocity of the wheels acts in the same direction that the wheel is facing in.

The idea is to consider only the vehicle speed and steering angle as inputs, and use the model to estimate the x and y position as wheel as the vehicle heading direction using the model.

Using the aforementioned assumptions, it is possible to derive the following equations of motion for the bicycle model:

$$\dot{x} = v \cos\theta \tag{1}$$

$$\dot{y} = v \sin\theta \tag{2}$$

$$\dot{\theta} = \frac{v}{L} \tan\delta \tag{3}$$

with the following symbol definitions:

- θ = vehicle yaw angle, i.e. heading direction
- L = wheelbase, the distance between the wheels
- δ = steering angle
- v = vehicle speed

Pure Pursuit and Heading Error

Pure pursuit (PP) is a tracking algorithm that works by calculating the curvature that will move a vehicle from its current position to some goal position. The whole

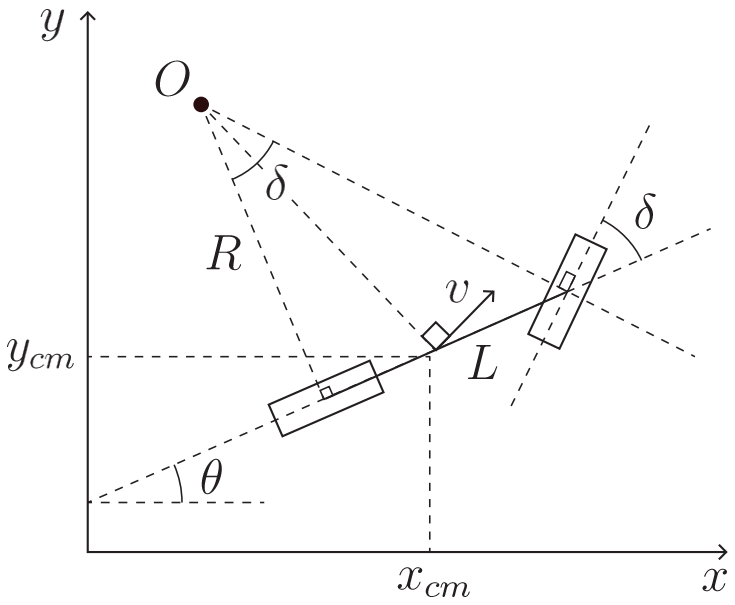

Figure 1: Bicycle model

point of the algorithm is to choose a goal position that is some distance ahead of the vehicle on the path. The name PP comes from the analogy that we use to describe the method. We tend to think of the vehicle as chasing a point on the path some distance ahead, it is pursuing that moving point. That analogy is often used to compare this method to the way humans drive. We tend to look some distance in front of the car and head toward that spot. This lookahead distance changes as we drive to reflect the twist of the road and vision occlusions[6].

The heading error (HE, α) is the key parameter to be estimated, as it is the starting point of the PP control strategy[3, 7]. The HE is defined as the angle between the vehicle heading direction and the line that connects the lookahead point and the center of rear axle of the vehicle. The lookahead point is defined as the point on the path on a fixed (or variable) distance ahead of the vehicle.

$$\delta = \tan^{-1}\left(\frac{L\dot{\theta}}{v}\right) = \tan^{-1}\left(\frac{L}{R}\right) = \tan^{-1}\left(\frac{2L\sin(\alpha)}{l_c}\right) \tag{4}$$

The distance of the lookahead point is a delicate parameter, since a big value can lead to cutting curves at low speed and a too small value can lead to unstable behaviors at higher speeds. Moreover, the accuracy of the estimation is also affected by this distance, since it becomes more difficult to identify the lines which are far ahead. For these reasons an analysis on the lookahead distance is necessary.

In the practical implementation used to generate the training datasets in the simulator, we assume to have an *optimal* path[1] that follows the center of the lane. The implementation follows the same approach explained in [6]; in particular, the basic steps of the algorithm are:

1. Calculate the closest point p_0 on the path to the vehicle position, using the standard euclidean distance.

2. Calculate the lookahead point on the path, using the lookahead distance l_c from p_0.

3. Convert the lookahead point to the vehicle coordinate system

4. Calculate the heading error

It is important to note that this process makes use of the position and orientation of the vehicle in the global frame of reference; therefore it can be used only in a simulated environment. This is indeed the algorithm used to generate the training datasets. The algorithm implementation is a little more delicate and explained in detail in chapter 6.1.

Camera

The camera is the main sensor used to perform the heading error estimation. In the model car it is positioned about 20 cm above the ground. It is oriented 20

[1]The optimality of the path will be discussed in more detail in later sections.

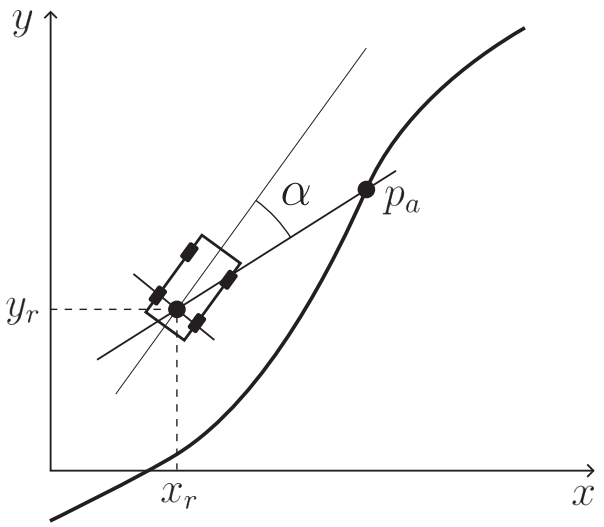

Figure 2: Heading Error

degrees downward in order to see also the closer section of the road. The camera is a standard *Raspberry Pi Camera* module. The technical specifications of the camera are the following:

- Still resolution: 8 Megapixels
- Video capture resolution: up to 1080p30, used at 640x480p at 30fps (max 90fps)
- Focal length: 3.04 mm
- Horizontal field of view: 62.2 degrees
- Vertical field of view: 48.8 degrees
- F-stop: 2.0

The native resolution used is 640x480 since it allows for a better frame rate, however the images will be further reduced in size in order to increase performance.

Figure 3: Example image from the camera

The simulated camera is mounted in the exact same position of the real one, and has been programmed to have the same field of view and frame rate.

3.2 Real track and simulation environment

The real track where we competed on, is a scaled down version of a city, with a variety of different roads, intersections, roundabouts, traffic lights, traffic signs, crosswalks, other vehicles and pedestrians. The track is roughly a square of 15 meters per side. Unfortunately we could not access the real track in the development phase, so we had to rely on the given simulated version of the track. A picture of the real track and the simulated one are shown in Figure 4.

The simulation environment is a key component of the estimation process, it is not directly related to the online estimation itself, but it is used to generate the training datasets; the accuracy of the estimation is therefore extremely correlated with the quality of the dataset and, consequently, with the quality of the simulation.

For the simulation environment, we use the *Gazebo simulator*[8], which is a

powerful simulator commonly used in robotics. The reasons behind the use of this simulator are the fact that it's open source and does not require extreme hardware to run, moreover we were able to use assets that were given to us by the organizers of the challenge. In the following are listed the main characteristics of the simulation environment:

- It is a physics-based simulator
- The simulation is modular, allowing for the use of different models
- *Gazebo* is fully integrated with *ROS*, this allows for easy access to all the sensors and actuators of the simulated vehicle and control of the simulation.
- The basic environment is composed of the model of the car and the streets, depending on the need it is possible to add other vehicles, pedestrians, and ramps.
- The car model is composed as a group of links/joints, with their respective physical properties. All the virtual sensors and actuators are mounted on the car model and are accessible through the *ROS* network.
- The virtual camera is mounted in the exact same position of the real one, and it's configured to have the same field of view and frame rate. The flow of images is easily accessible through *ROS*.

The simulator has been run first on a laptop with an *Intel Core i7-7500U* processor, an *NVIDIA GeForce GTX 950M* graphics card and 16 GB of RAM; and later on a desktop with an *AMD Ryzen 5 3600* processor paired with a *NVIDIA GeForce GTX 1080Ti* graphics card and 32 GB of RAM.

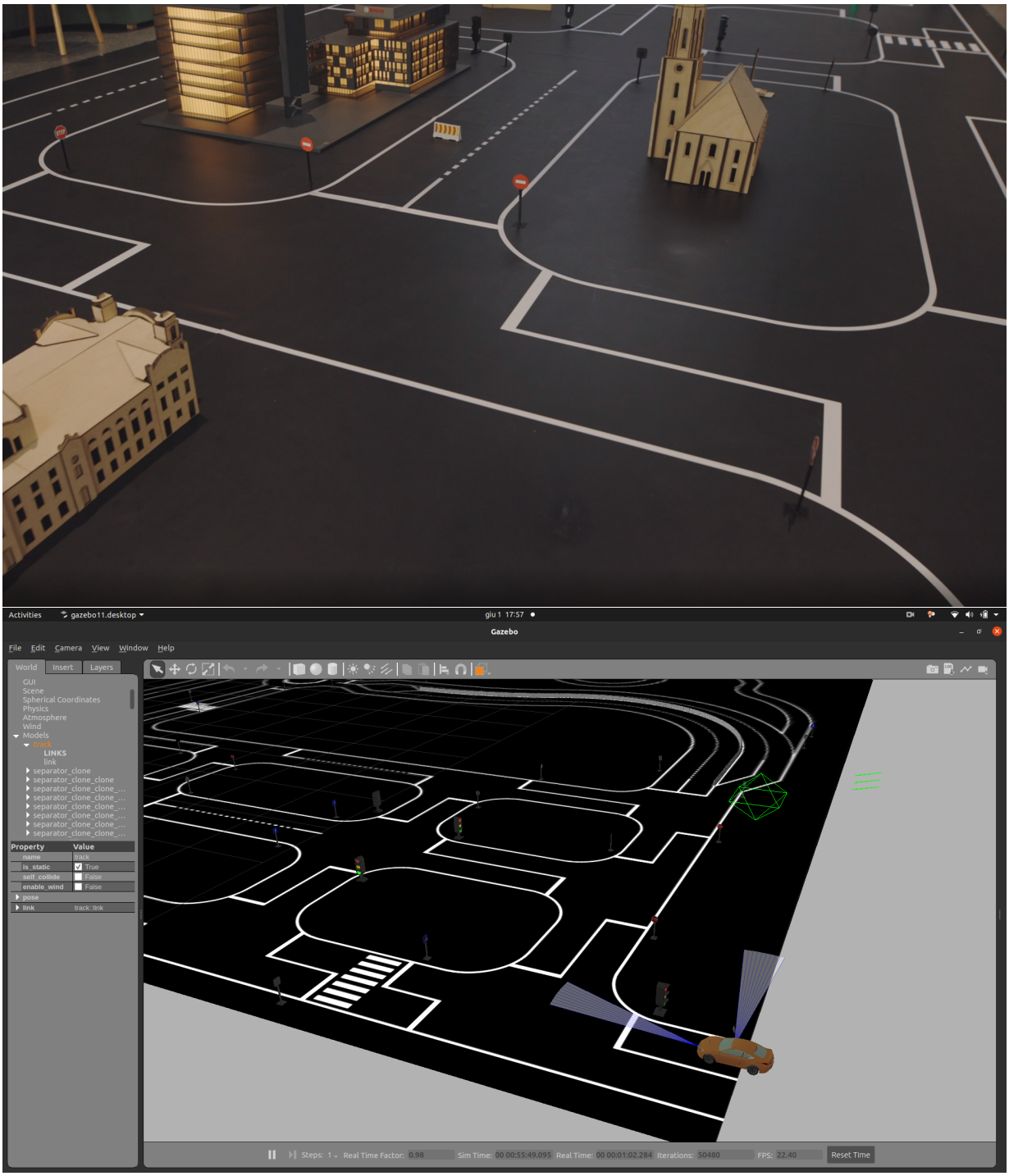

Figure 4: Competition track section and simulator image

4 Optimal path

4.1 Why do we need a path to follow?

The basic idea of having an *optimal* path is being able to localize the vehicle on the path and to sample the heading error looking at the path ahead.

Since we are only interested in estimating the heading error with respect to the center of the road it does not seem necessary to have an optimal path; however there are several reasons why one would prefer a different path to the centerline. The first one is that the centerline could be an infeasible trajectory to follow for the vehicle, with an aggressive curvature or discontinuous first derivative. The second one is that the path method is very easy to scale in situations where the centerline is not clear or defined, for example when dealing with intersections or unmarked roads. The third is that it's possible to have some degrees of control over the path, for example on the smoothness or the maximum curvature. Moreover, the path concept is useful when planning far ahead in the future or when considering obstacle avoidance. Finally, using a path is very convenient from the implementation point of view, because it decouples the camera image from the localization and path tracking.

4.2 Why clothoids

Common path-planning methods usually generate obstacle-free path, but with no or very little concern about path feasibility or optimality so that it is usually necessary to apply some kind of transform algorithm to locally smooth such a path. Various path-smoothing algorithms are proposed in the literature: cubic splines [9], intrinsic splines [10], Bezier's curves [11], quintic Bezier splines [12], and clothoids. The main advantage of clothoids over other smoothing methods in path-planning applications is linear change of their curvature, which is of great im-

portance for transportation of people or heavy and sensitive loads since it prevents abrupt changes in the centripetal acceleration and forces experienced by a vehicle increasing driving comfort. Clothoids are very attractive in path-smoothing applications as they are easy to follow because of their linearly changing curvature [13].

Clothoids also have advantages over other smoothing techniques in sense of vehicles optimal motion planning—by applying the Maximum Principle from the optimal control theory, for forward motion and differential drive vehicle, one can find that the necessary condition for trajectory to be time optimal yields clothoids [14]. Further, Boissonnat et al. [15] studied the shortest plane paths joining two given positions with given tangent angles and curvatures along which the tangent angle and the curvature are continuous and the derivative of the curvature is bounded. They showed that at a point where such a path is of class $\mathcal{C}^3$, it must be locally a piece of a clothoid or a line segment. Similarly, Fraichard and Scheuer [16] showed that with requirements of continuous curvature and bounded both curvature and its derivative, the shortest path consist of line segments, circular arcs, and clothoids [17].

4.3 Clothoid definition

A Clothoid, also known as Euler or Cornu spiral, is a plane, invariant spiral curve defined in parameter form. As explained in [18] Its curvature κ can be expressed as a linear function of its arc length s:

$$\kappa(s) = \kappa_0 + \sigma s, \quad \kappa_0,\, \sigma \in \mathbb{R} \tag{5}$$

where σ is the sharpness of the clothoid and κ_0 is the initial curvature at $s = 0$. The tangent or winding angle θ with respect to its arc length is given as:

$$\theta(s) = \theta_0 + \kappa_0 s + \frac{\sigma s^2}{2}, \quad \theta_0,\, \kappa_0,\, \sigma \in \mathbb{R} \tag{6}$$

where θ_0 is the initial tangent angle at $s = 0$. Thus, a general clothoid can be expressed in paremetric form as in [19]:

$$\mathbf{F}(s) = \begin{pmatrix} x_0 + \int_0^s \cos\left(\theta_0 + \kappa_0 u + \frac{\sigma u^2}{2}\right) du \\ y_0 + \int_0^s \sin\left(\theta_0 + \kappa_0 u + \frac{\sigma u^2}{2}\right) du \end{pmatrix} \tag{7}$$

Based on the above definition, it is possible to define the elementary clothoid segment $\mathbf{F}_{\mathcal{E}}(s)$ with $\sigma = \sigma_{\mathcal{E}} > 0$, $s \geq 0$, $\kappa_0 \geq 0$, $\theta_0 = 0$, $(x_0, y_0) = (0, 0)$ and $\theta \in (0, \pi/2]$

$$\mathbf{F}_{\mathcal{E}}(s) = \sqrt{\frac{\pi}{\sigma_{\mathcal{E}}}} \mathbf{R}\left(-\frac{\kappa_0^2}{2\sigma_{\mathcal{E}}}\right) \begin{pmatrix} \delta_c(s) \\ \delta_s(s) \end{pmatrix} \tag{8}$$

where $\delta_c(s) = C_f(\frac{\sigma_{\mathcal{E}} s + \kappa_0}{\sqrt{\pi \sigma_{\mathcal{E}}}}) - C_f(\frac{\kappa_0}{\sqrt{\pi \sigma_{\mathcal{E}}}})$ and $\delta_s(s) = S_f(\frac{\sigma_{\mathcal{E}} s + \kappa_0}{\sqrt{\pi \sigma_{\mathcal{E}}}}) - S_f(\frac{\kappa_0}{\sqrt{\pi \sigma_{\mathcal{E}}}})$. Here, $\mathbf{R}(\theta)$ is the standard planar rotation matrix, and $C_f(x)$ and $S_f(x)$ are a pair of nonnegative functions named Fresnel integrals [20]:

$$C_f(s) = \int_0^s \cos\left(\frac{\pi \xi^2}{2}\right) d\xi,\ S_f(s) = \int_0^s \sin\left(\frac{\pi \xi^2}{2}\right) d\xi. \tag{9}$$

4.4 Implementation

The context of our particular application is the following: we are given a map of a scaled city, with roads, intersections, roundabouts, crosswalks, etc... Alongside with the map we are also given a set of waypoints in the form of a directed graph. Every waypoint corresponds to an (x, y) position on the map, more specifically to point in the center of a lane. All the waypoints are placed at about $30cm$ from each other. Every waypoint also incorporate a list of edges that connects it to other waypoints, in such a way that only consecutive waypoints are connected by an edge and only if they follow they same lane, the only exceptions being the points associated to roundabouts and intersections as well as the start point and the end point.

The goal of the algorithm is to take 2 waypoints as input and return a path that connects them. To achieve this goal, we start by finding the shortest path in the graph of waypoints that connects them[2]; to do this we use the Dijkstra algorithm [21].

The second and last step to create the *optimal* path is to fit clothoid segments between the waypoints of the shortest path. Since the Fresnel integrals in equation (9) are two transcendental functions, they can't be solved analytically, however there are several approximation methods in the literature that achieves good results [17, 18, 22]. In our implementation we used the python library *pyclothoids* [23], which is a python wrapper of the C++ library by Bertolazzi [24–27]. In figure 5 it is possible to see the result of the algorithm, the path is designed to cross most of the waypoints.

[2]We restrict ourselves to the case where at least one path exists between any two waypoints, which was always true in our scenario, except if we considered the end point as starting point

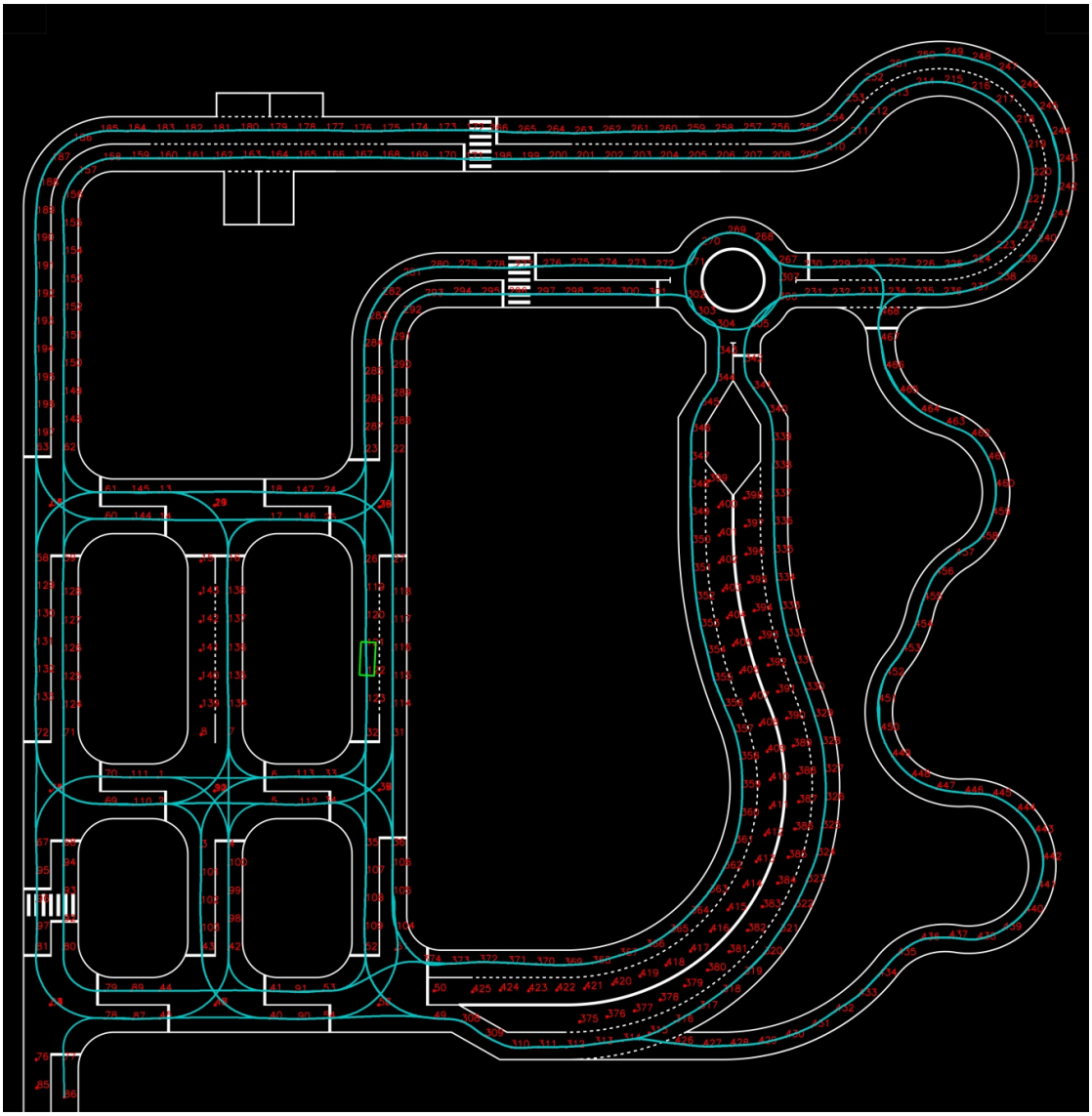

Figure 5: Map and waypoints, with the optimal path passing through most of the waypoints

5 Performance metrics

Since we are dealing with a complex estimation problem, it is very important to define a set of appropriate performance metrics in order to reliably evaluate the quality of the estimation. This is a difficult task since the problem of the estimate of the heading error with respect to the center line is strictly related to the problem of the road estimation itself.

In this section we will analyze a set of metrics found in the literature, and we will propose a metric that we believe is more appropriate for our problem.

It is very important to make an initial distinction between two, very related, but different, approaches to the performance evaluation of lane detection and tracking systems: the first one consists in comparing the estimation capabilities of the whole systems (estimation *and* control) by means of evaluating the tracking performance in a real or simulated scenario. The second one consists in evaluating only the estimation performance on some kind of ground truth data. The first approach is more realistic, but it has some big drawbacks: it is very difficult to perform a fair comparison between different systems because it is hard to replicate the same conditions in every test; moreover a systematic analysis of a big parameter space with many possible variations is often infeasible within the testing time constraints. Lastly, a very important issue is that testing tracking performance in a real scenario doesn't give an accurate picture in terms of disturbances, or in other words, in situations where car is relatively far from the center line, or in some very rare or extreme configurations, since a good tracking systems is always close to tracking target in terms of both position and orientation. The second approach is more specific to the estimation problem, and therefore less applicable to a wide range of systems. However, it has the advantage of being performed offline on prerecorded videos, guaranteeing a fair comparison between variations, and it is also possible to perform systematic sweeps over the parameters space in

order to find the best configuration. In addition to this it's possible to evaluate performance on edge cases in a fair and systematic way. For these reasons in this thesis we will focus on this second approach, analyzing the whole tracking performance as a final step.

5.1 Metrics from the literature

In the survay paper of autonomous driving systems[28] we can see that in the analyzed methods, to evaluate performance, visual inspection of lane estimation is deployed in all studies. However, this is only a qualitative metric. In order to quantify the performance of a lane estimation technique, one of the most common metric is ego-vehicle position localization within the lane [1, 29, 30]. Lower error in ego-vehicle localization is a result of more accurate lane detection process itself. This is because ego-vehicle localization is computed using the estimated lane positions in the proximity of the ego-vehicle. The method we propose will be strictly related to this concept. Another key metric proposed in [28], which is not very common in the previous literature, but very important for the implementation side of the problem, is the computational efficiency of the algorithm. Since we are dealing with a real-time problem, and relatively low computational power, this metric is very important in our application.

Other, more quantitative, proposed metrics are the following:

- Lane markings related metrics: this set of metrics, despite being useful in some applications, has been discarded since the estimation target doesn't directly depend on the markings themselves, and could potentially be used in a scenario where there are no markings at all, or very non-standard ones. In addition to this, one should consider the additional complexity, and the appropriate way of generating the estimate of the lane markings, which is not a trivial task and not the object of this thesis.

- Pixel level accuracy: this metric is very widely applied in the context of vision based estimation, it consists in classifying each pixel of the image as either part of the lane or not. It is a very common in deep learning approaches [31], firstly because it's very straightforward to implement using convolutional filters and secondly because most of the training and evaluation datasets are annotated pixel by pixel. This metric is usually evaluated by means of a ratio between the number of correctly classified pixels and the total number of pixels. This metric is not very useful in our case, since we are interested in the heading error, however it is a very interesting possible extension of the proposed method.

There are several other metrics in the literature that are of little interest in our particular application.

5.2 Proposed metric

Since in our particular application we are interested in the heading error, which is a scalar quantity, a very natural choice for a performance metric is the squared error of the single sample with respect to the perfect ground truth, that can extended to evaluate a set of samples using the standard deviation (STD) of the errors. The main issue with this approach is how to retrieve the ground truth. In the simulated environment it is very easy since we have access to the precise absolute position and orientation of the car. In the real case it is more challenging, but with the experimental setup we used, explained in chapter 7.1 we were able to retrieve a precise position and orientation, therefore we have an accurate ground truth that can be used to evaluate the estimation performance and to compare different methods. In addition to the STD, the mean absolute error can also be considered with the same rationale, which is less dependent on outliers.

Another considered metric is the continuity and smoothness of the output

sequence, meaning that a sequence of consecutive frames should generate a continuous smooth output sequence of heading errors, since we are not expecting any sudden changes in the road ahead or in the car position. This metric becomes particularly important if we plan to use a control system on top with some parameters that are dependent on the variation of the heading error, like, for example, a PID controller with a derivative term. The continuity metric is calculated as the standard deviation of the sequence of differences between consecutive samples, with respect to the ground truth.

Lastly, we can also consider the computational efficiency of the algorithm and the generalization capabilities of the system from the simulated environment to the real one, which is a very important aspect of this estimation method.

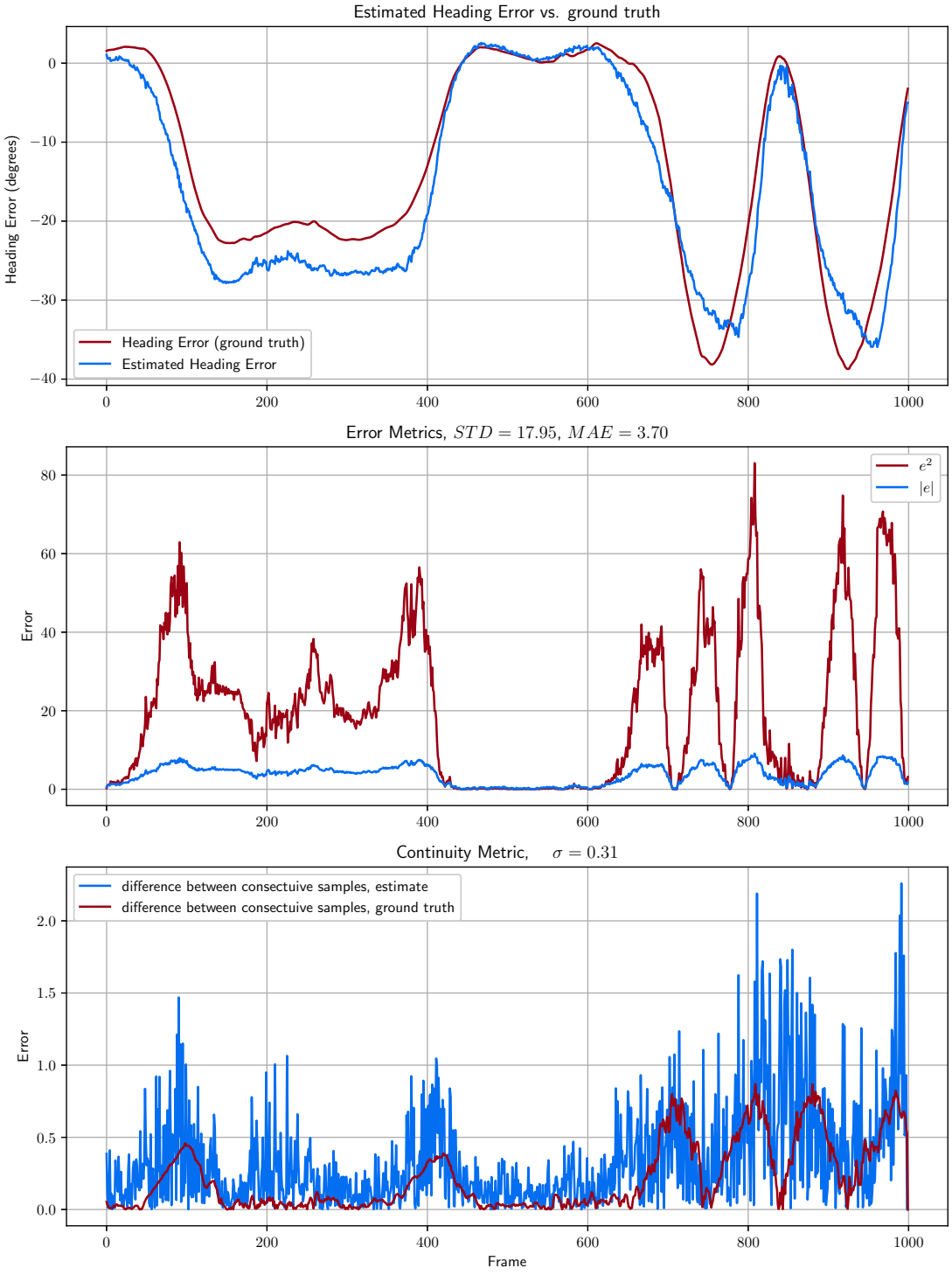

Figure 6: Performance metrics example

6 Proposed estimation method

In this chapter we will present the proposed method for the estimation of the heading error. From a general point of view the process can be subdivided in 5 main steps that will be described in detail in the following subsections:

1. **Data collection**: this step consists in collecting, in the simulated environment, the raw data that will be used to generate the datasets.

2. **Heading error calculation**: the second step is to generate the ground truth for the previously generated datasets. It's important to separate this step from the previous one, in order to allow a more general approach to the problem.

3. **Dataset generation**: this step consists in generating the datasets that will be used to train the network, it is when data preprocessing and augmentation are applied: two key components of the proposed method.

4. **Network training**: this step consists in training the network using the previously generated dataset. The network architecture will be discussed, together with the training process itself and the hyperparameters used.

5. **Online estimation**: this is the final step, where all the previous steps are combined in order to accurately estimate the heading error in real time, on the real vehicle, from the camera frames.

6.1 Data collection

The main idea of data collection is to use the *Gazebo* simulator to generate an arbitrarily large set of images and the corresponding ground truth labels. During the developing process 2 main methods were designed in order to solve the task. The original method was tailored to the specific aspects of the challenge, and was therefore designed to solve not only the data collection problem but also any other aspect of the challenge that could be tested in a simulated environment, such as the high level logic, the control system, the interface with the additional simulated sensors, etc. The second method was designed after the challenge to be specific with respect to the heading error estimation problem. The second method heavily simplifies the interface and the complexity required to control the car in the simulator, and allows to generate the dataset with a much finer level of control. In order to study the influence of the dataset collection parameters on the final performance, in this thesis we will use and focus mostly on the second method, however the original method will also be described since it shares most of the ideas.

Original method

The Gazebo simulator is very convenient for developing since every simulated model and sensor information is published on a *ROS* topic, and the car can also be controlled by publishing steering and throttle commands on their respective topics. This is very useful since it allows to use the same code for both the simulation and the real vehicle, where we also implemented a *ROS* network.

The main steps of the original method are the following:

- A training path is generated for the car to follow, this path is based on the aforementioned Clothoid interpolation and travels across all the different sections of the map in order to allow the car to encounter a good variety of situations.

- The car is controlled with a PP controller at a fixed speed.
- The point ahead for the controller is selected by finding the closest point of the vehicle on the path and then choosing the point on the path at a fixed distance ahead of it. This way of selecting the point ahead is very convenient and easy to implement and has the advantage of always finding a unique point, but is not conformed to the common definition of point ahead for the PP controller.
- While the vehicle is travelling along the path, Gaussian noise is injected into the steering in order for the car to slide left and right. This is done in order to generate more variety in the training dataset. The noise injection is a key aspect of the estimation method and has a big impact on the final performance, as it will be discussed in chapter 7. The idea is that if the camera only sees the road ahead always perfectly centered, later, after the training, when facing the road at a different position and orientation, slightly off center, the network will not be able to estimate the heading error for lack of examples in the training dataset. The amount and characterization of the noise injected will be subject of a parametric analysis.
- While the car is travelling, the camera is recording images at a fixed rate of 30 fps and saving them alongside their corresponding ground truth heading error in a compressed file for later use.

The 2 main advantages of this method are the flexibility and the wide variety of possible scenarios. The main disadvantages are the complexity of the implementation, the lack of precise control over the noise injected and the fact that the heading error distance is fixed and cannot be changed after the dataset has been generated.

New method

The new data collection method is designed to solve the previous method main issues: reducing the code complexity, fine controlling the noise injected and generating the heading error independently of the images. To achieve this, we apported a series of modifications:

- The entire interface with the simulated car is removed, the vehicle is moved around the simulated environment by teleporting it to the new desired position an orientation using a *ROS* service. In this way we are able to directly control the variance of the noise both in the car orientation and lateral position with respect to the lane center by sampling from 2 Gaussian distributions; the change effect can be seen in image 7. In this way we remove any bias that could be introduced by the PP controller.

- The map and the path are simplified to a single loop designed to match the real track we built in the university's laboratory for the real-life tests. This allows to isolate only the lane detection problem, and it's a practical test bed for performing systematic tests and evaluate their performance.

- To disconnect the heading error calculation from the image generation, the images are saved alongside the respective car position and orientation, and the heading error can be calculated offline for any distance using the more accurate method described in the following section.

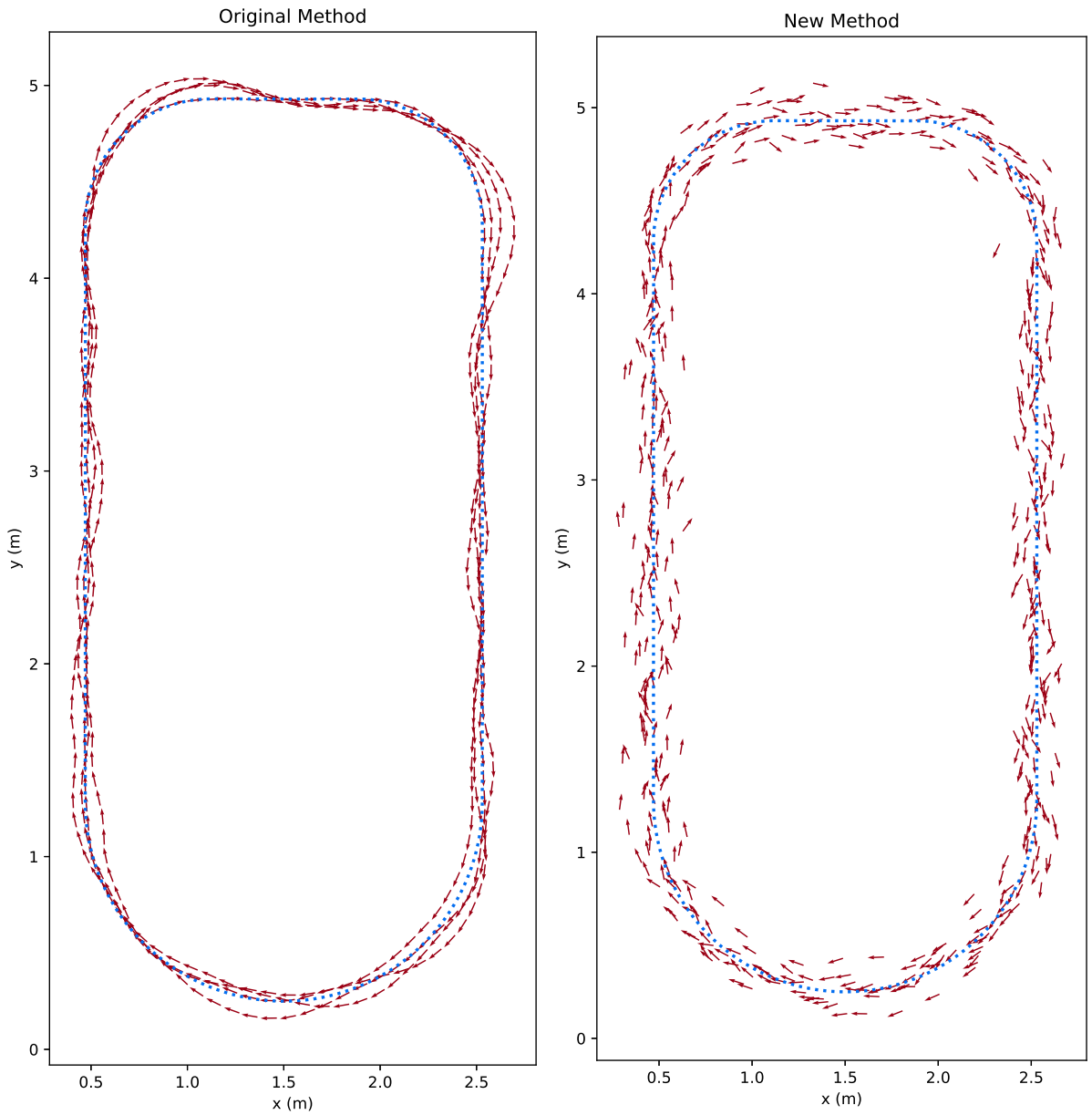

Figure 7: Original method and new method

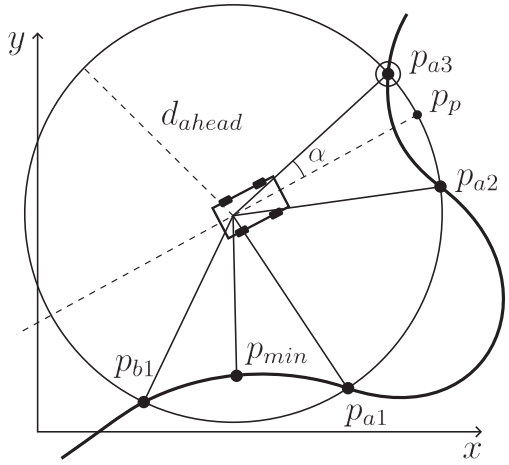

Figure 8: Heading error calculation

Heading error calculation

In order to precisely calculate the heading error an algorithm has been developed that is a direct implementation of the common heading error definition in the PP controller [3]. The algorithm works by first finding the (normally 2) points at the distance d_{ahead} required, and then selecting the one that is coherent with the car orientation. The algorithm is robust with respect to the 2 possible edge cases of finding a single point or no point at all, which means the vehicle is too far from the path and the algorithm returns the closest point; or the one where there are ≥ 2 points at the same distance, in this case the algorithm returns the further in the path. The algorithm is implemented in python and a pseudocode for it can be seen in Algorithm 1. A graphical intuition of the algorithm can be seen in Figure 8.

Algorithm 1 Calculate Heading Error

procedure CALCHE, INPUTS : $x, y, \phi, path, d_{ahead}$

  $thrsh \leftarrow$ distance between points in $path$

  $p_{car} \leftarrow$ create point (x, y)

  $p_0 \leftarrow$ projection of p_{car} on $path$

  roll $path$ to make p_0 first

  $path_a, path_b \leftarrow$ split $path$ in half

  $D_a \leftarrow |\|path_a - p_{car}\| - d_{ahead}|$ #vector of distances ahead

  $D_b \leftarrow |\|path_b - p_{car}\| - d_{ahead}|$ #vector of distances behind

  $i_a \leftarrow \text{argmin } D_a,\ i_b \leftarrow \text{argmin } D_b$

  $min_a \leftarrow D_a[i_a], min_b \leftarrow D_b[i_b]$ #minimum distances

  if $min_a \geq thrs$ **or** $min_b \geq thrs$ **then** #car too far from path

    $p_{HE} \leftarrow p_0$

  else #select the further point

    $p_a \leftarrow path_a[\max(\text{where}(D_a < thrsh))]$

    $p_b \leftarrow path_b[\min(\text{where}(D_b < thrsh))]$

    $p_p \leftarrow p_{car} + d_{ahead}(\cos\phi, \sin\phi)$ #project car heading

    $p_{HE} \leftarrow$ closest point to p_p between p_a and p_b

  $\phi_{ref} \leftarrow \text{atan2}(p_{HE}[y] - p_{car}[y], p_{HE}[x] - p_{car}[x])$

  return $\phi_{ref} - \phi$

6.2 Dataset generation

With dataset generation we mean the process of converting the collected raw data into the set of couples of input images and labels that will used for training. In this section we will discuss the 2 main steps for the images to be ready for training: image preprocessing and image augmentation.

Image preprocessing

Image preprocessing is applied both to the images collected in the simulator and to the images processed in real time for online estimation. The 3 main goals of the preprocessing are reducing the amount of information in the image to the minimum required for the task at hand, standardizing the images to make the real and simulated data more comparable, and reducing the computational cost. The preprocessing main steps are the following:

- **Gray scale conversion**: all the images are converted from RGB to gray scale. This is done because the color information is not relevant for the task at hand (and the images are already mostly black and white) and because it reduces the computational cost.

- **Crop**: the images are cropped to remove the top section. The reason for this is the fact that the top section of the image contains the sky and the horizon and the very far away section of the road, which does not contain any relevant information. The percentage of the image kept is a hyperparameter that can be tuned. The influence of this parameter will be discussed in the Results chapter (7).

- **First resize**: the images are resized to double the final size. In this case to a square of 64x64 pixels. This is done to reduce the computational cost of the algorithm and to preserve only the most predominant features of the image.

The reason for the square is that is very convenient for the convolutional neural network to have the same number of pixels in the height and width of the image.

- **Canny edge detection**: the images are processed with the Canny edge detector [32], which is a very common edge detection algorithm. Edge detection is the key component for making the images more comparable between the real and simulated data, since it removes any brightness and contrast variations that are present in the real images, and outputs a binary image. The Canny algorithm has 2 parameters that can be tuned to filter the edges by strength and by length.

- **Blur**: the images are blurred with a Gaussian filter. This is done to smooth the binary image, since the neural network performs worse with binary images. This is also done to prepare the images for the next step.

- **Second resize**: the images are resized to the final size. In this case to a square of 32x32 pixels. This very small size has been chosen because in this way only the very predominant features of the image are preserved, and the neural network can learn to recognize them more easily and reliably. Not to mention that the computational cost is drastically reduced with this small size.

The preprocessing steps are shown in Figure 9.

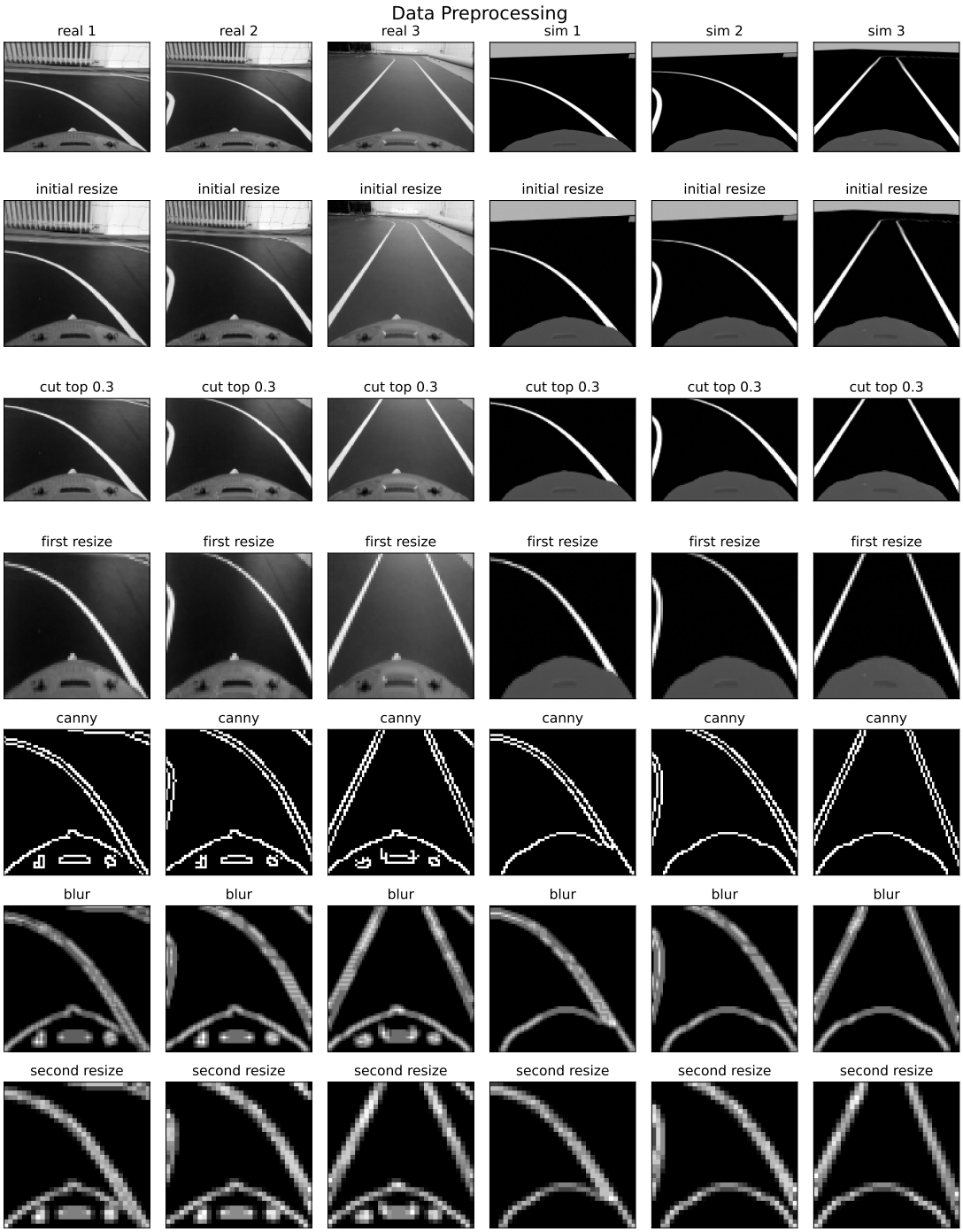

Figure 9: Preprocessing steps

Image augmentation

Image augmentation is a technique that is used to artificially increase the size of the dataset, in this case by slightly changing the images in the datasets. The main goals of the augmentation are increasing the amount of images, and making the simulator images more similar to the real images.

We start by noticing that the entire estimation problem of the heading error from a frontal image is symmetric with respect to the vertical axis. This means that we can just flip the image horizontally and invert the sign to the heading error to get a new valid sample for free. Aside from this symmetry, the data augmentation process can be summarized in the following steps:

- **Resize**: the images are resized to 4 times the final size to a 128x128 square. The size is small for faster processing.

- **Random ellipses**: random ellipses are drawn on the image, with random positions and orientations and different shades of white. The idea behind this is to simulate strong light reflections, which are not present in the simulated environment. The ellipses also partially cover the image and can be considered a random erasing technique [33], which is a common data augmentation technique to reduce overfitting and increase robustness to occlusions.

- **Dilation/erosion**: images are dilated or eroded (with a 20% probability for each one) with a random kernel. This is done to simulate the effect of having wider or narrower lane markings.

- **Preprocessing**: this is the preprocessing step, as described in the previous section.

- **Random shift**: the images are randomly shifted of a small amount of pixels across the vertical direction to account for possible camera misalignment.

- **Noise**: the images are randomly corrupted with Gaussian noise, the variance of the noise is a hyperparameter that can be tuned. This is done to simulate the general robustness of the neural network to noise.

The augmentation steps are shown in Figure 10.

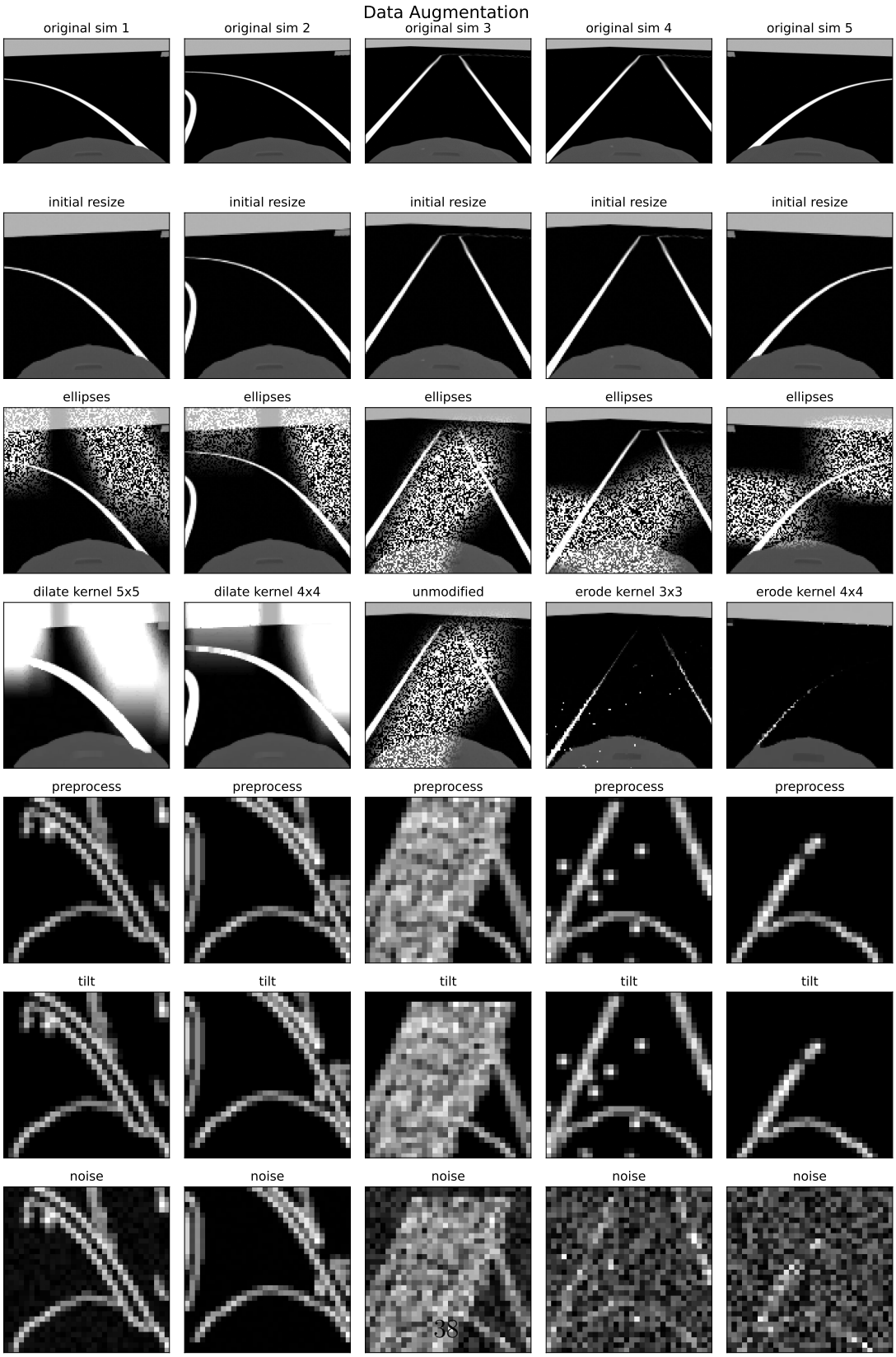

Figure 10: Augmentation steps

6.3 Neural network training

In this section we will discuss the neural network architecture and the training process. When thinking about the design of the neural network one must take into account the fact that it will have to run on a real-time embedded system, and not a cluster of powerful GPUs. This means that the neural network must be as small as possible, while still being able to perform well. Another important requirement is to being able to generalize well from the simulated data to the real data.

Network Architecture

In order to keep up with the performance required by the real-time constraints, the neural network has been designed with very few layers and very few parameters. The network architecture is inspired by the LeNet-5 [34] architecture, which is a very simple convolutional network that was originally designed for handwritten digit recognition. The design has been improved with the addition of batch normalization [35] and dropout [36]. The network is structured as a sequence of convolutional layers, followed by a sequence of fully connected layers; more specifically, the layers are the following:

- **1st 2D convolution**: 4 filters, with a kernel size of 5x5 and a stride of 1. It converts the original 32x32 image to 28x28.
- **ReLU**: activation function.
- **Dropout**: 30% probability.
- **Pooling**: max pooling with a kernel size of 2x2 and a stride of 2. It converts the 28x28 image to 14x14.
- **Batch normalization**: normalization of the output of the previous layer.

- **2nd 2D convolution**: 16 filters, with a kernel size of 5x5 and a stride of 1. It converts the 14x14 image to 10x10.
- **ReLU**: activation function.
- **Dropout**: 30% probability.
- **Pooling**: max pooling with a kernel size of 2x2 and a stride of 2. It converts the 10x10 image to 5x5.
- **Dropout**: 30% probability.
- **3rd 2D convolution**: 32 filters, with a kernel size of 5x5 and a stride of 1. It converts the 5x5 image to 1x1.
- **ReLU**: activation function.
- **Flatten**: flattens the 1x1 image to a vector of 32 elements.
- **1st fully connected**: 16 neurons.
- **ReLU**: activation function.
- **2nd fully connected**: output layer, 1 neuron.

The network takes as input a 32x32 grayscale image and outputs a single value representing the heading error. The pytorch implementation can be found in the appendix.

Training

In this section the training process will be described in detail. The main steps are the following:

- **Dataset**: the dataset is split into a training set and a validation set, with 80% of the data reserved for training and 20% for validation. The data order is shuffled to reduce correlation between datapoints.

- **Loss function**: the loss function is the mean squared error (MSE). Some tests were performed with other losses such as the L_1 loss and the Huber loss, but the MSE was found to be far more effective.

- **Optimizer**: the optimizer used is the standard Adam optimizer, with a learning rate of 3×10^{-3}. The learning rate is a very delicate parameter and will be subject of a further analysis.

- **Training**: the training is performed for 200 epochs, with a batch size of 2^{16}. The number of epochs and the batch size are 2 hyperparameters that can be tuned, and will also be the subject of a further analysis.

- **Validation**: the validation is performed at the end of each epoch, the training algorithm keeps track of validation loss across the epochs, and the model with the lowest loss is the one that gets saved at the end. This is done to prevent overfitting. It is important to notice that the validation step is performed on the 20% section of the training set and not on real-world datasets; the real datasets are used only for testing and comparing different models.

The training is performed on a single NVIDIA GTX 1080 Ti GPU, using the pytorch framework.

Online estimation

In order to run on the real car, the network is converted from a pytorch model to an Onnx model so that it can be used directly with the Opencv library, and we don't need to install the heavy pytorch library on the embedded device (a

Raspberry Pi 4). From this point onward the online estimation is very straightforward: the image from the camera is preprocessed and fed to the network, which estimates the heading error. The only difference is that the image is flipped, and the network is run twice, once with the original image and once with the flipped image. An average between the measures is then taken. This can be done thanks to the symmetry of the problem and increases the accuracy of the estimation. Performance-wise, the network runs at 250-300 fps on the Raspberry Pi 4, which is more than enough for real-time estimation, and the bottleneck is actually the camera frame-rate. The high efficiency of the architecture would probably work fine on an even lower-end device.

7 Results

In this chapter the results of the estimation system will be presented. The following are the main topics that will be covered:

- **Experimental setup**: the vehicle assembly, the real track and the localization system we used to test the system.
- **Evaluation datasets analysis**: the evaluation dataset generation process, and the differences between the generated sets.
- **Network analysis**: layer activations and convolutional filters visualizations.
- **Hyperparameters study**: hyperparameters exploration methods and results.
- **Control loop application**: test of the estimation system in full control loop scheme.

7.1 Experimental setup

In this section we will explore the experimental setup we used to collect the real world data. The real vehicle model and assembly will be explained alongside with the localization system we used to log its position and orientation, and the track where we run the tests.

Scaled vehicle assembly

In figure 11 the scaled vehicle assembly can be seen. The vehicle is a 1/10 scale of a real car. In the picture we can see the car with the main motor and the steering servo. The car is powered by a 2S LiPo battery. On top of the chassis we can see the refractive markers that are used by the Vicon system.

Real track and Vicon localization system

In order to get the accurate position and orientation of the vehicle, we use a Vicon system. This system consists in a set of cameras that are placed around the track, and that are able to track any object that has a set of markers attached to it. The system operates at a nominal frequency of 100 Hz, with some negligible packet loss, and return the measurements with a variable latency from 150 to 200 ms. The Vicon system operate as a ros node, so we can easily get the data reading the corresponding topic.

To test the vehicle we designed a simple track with 3 straight lines connected by 3 turns with 2 different radii of curvature. The track is 12 meters long, and the straight lines are 2 meters long. The track is shown in figure 12. The track is made of a black surface, and the lines are designed with white tape. The tape is 2 cm wide and the distance between the lines is 38 cm. The track can be parameterized by the following equations:

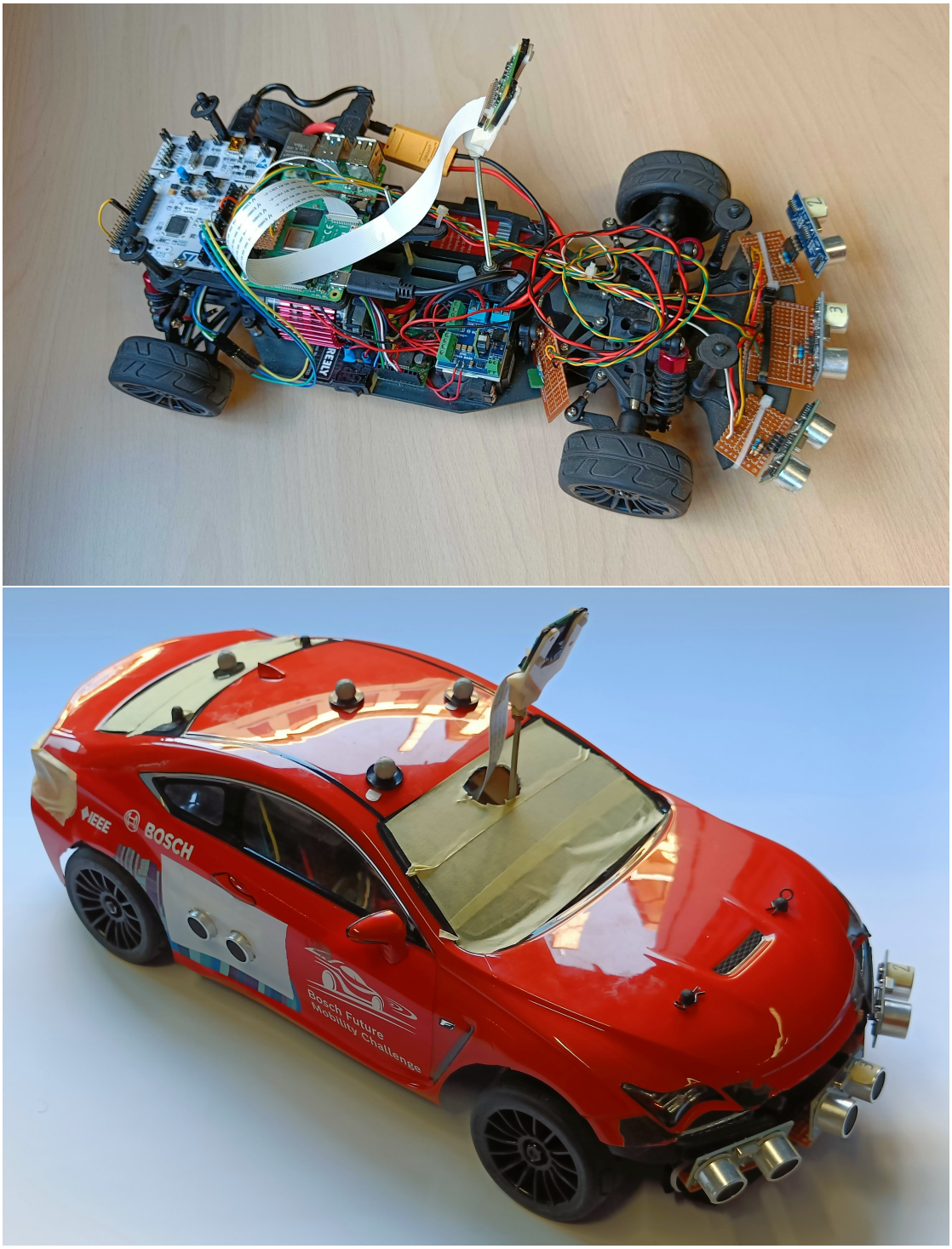

Figure 11: Scaled vehicle assembly

$$\mathbf{r}_{12}(\theta) = \begin{bmatrix} x_c + R\cos\theta \\ \big(m + R_c\big) + R\sin\theta \end{bmatrix} \qquad \theta \in [0, -\pi]$$

$$\mathbf{r}_{23}(\lambda) = \begin{bmatrix} x_c - R \\ \lambda\big(m + R_c\big) + (1-\lambda)\big(m + R_c + L\big) \end{bmatrix} \qquad \lambda \in [0, 1]$$

$$\mathbf{r}_{34}(\theta) = \begin{bmatrix} \big(x_c - R + r\big) + r\cos\theta \\ \big(m + R_c + L\big) + r\sin\theta \end{bmatrix} \qquad \theta \in \left[-\frac{\pi}{2}, -\frac{3\pi}{2}\right]$$

$$\mathbf{r}_{45}(\lambda) = \begin{bmatrix} \lambda\big(x_c - R + r\big) + (1-\lambda)\big(x_c + R - r\big) \\ m + R_c + L + r \end{bmatrix} \qquad \lambda \in [0, 1]$$

$$\mathbf{r}_{56}(\theta) = \begin{bmatrix} \big(x_c + R - r\big) + r\cos\theta \\ \big(m + R_c + L\big) + r\sin\theta \end{bmatrix} \qquad \theta \in \left[-\frac{3\pi}{2}, -2\pi\right]$$

$$\mathbf{r}_{61}(\lambda) = \begin{bmatrix} x_c + R \\ \lambda\big(m + R_c + L\big) + (1-\lambda)\big(m + R_c\big) \end{bmatrix} \qquad \lambda \in [0, 1]$$

With $\mathbf{r}_{ij}(\cdot)$ representing each section of track, and with the symbols descriptions and numerical values summarized in Table 1.

The same exact has been reconstructed in the simulation environment, so that we can compare the real and simulated data.

Figure 12: Test track

Sym	Description	Value
x_c	x coordinate of axis of symmetry	1.5 m
m	Margin from the edge	0.3 m
R_c	Big radius of the center of the lane	1.04 m
w	Width of the lane	0.38 m
R	Big radius of the track	$\left\{R_c - \frac{w}{2}, R_c - \frac{w}{2}\right\}$
r_c	Small radius of the center of the lane	0.665 m
r	Small radius of the track	$\left\{r_c - \frac{w}{2}, r_c - \frac{w}{2}\right\}$
L	Length of the straight section	2 m

Table 1: Parameters of the test track

7.2 Evaluation datasets analysis

To generate the real datasets we used the same process as the one used to generate the simulated datasets. With the difference that the true position and orientation of the vehicle is obtained not by means of the simulated environment but by means of the Vicon system. Another difference is that, since unfortunately we are not able to teleport the vehicle in the real world, we had to use the original method of generating the dataset, as explained in Section 6.1. The datasets have been generated using 8 different noise level in the steering: $0, 2, 4, 6, 8, 10, 12, 14$ degrees, the reason behind this choice is evaluating the car performance in suboptimal testing scenarios. For each noise level the car performed 4 laps around the track clockwise, and 4 laps anticlockwise. The speed of the car was kept constant at $0.3\,\mathrm{m\,s^{-1}}$.

The heading errors have been computed offline using the technique explained in Section 6.1. We considered 8 values for the heading error distance to test: $0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$ meters. Figures 13 and 14 show respectively the actual path followed by the car against a reference path, and the true heading errors for each distance ahead. For each real evaluation dataset generated, the sequence of the vehicle positions and orientations has been logged and used to generate a the equivalent evalution dataset in the simulation environment. This has been done with the intent of evaluating the generalization capabilities of the proposed strategy when applied to the real world.

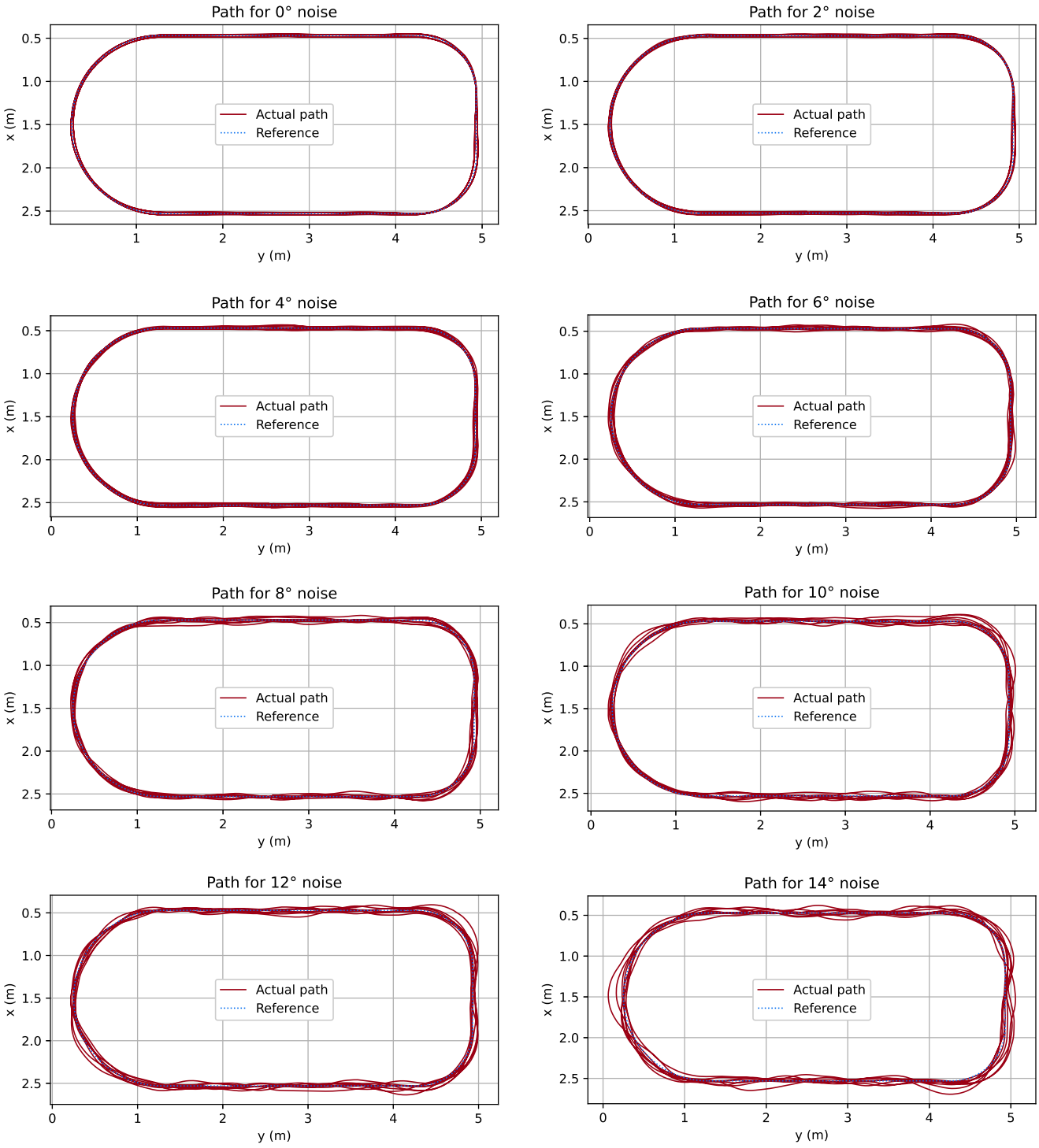

Figure 13: Evaluation datasets path and reference path

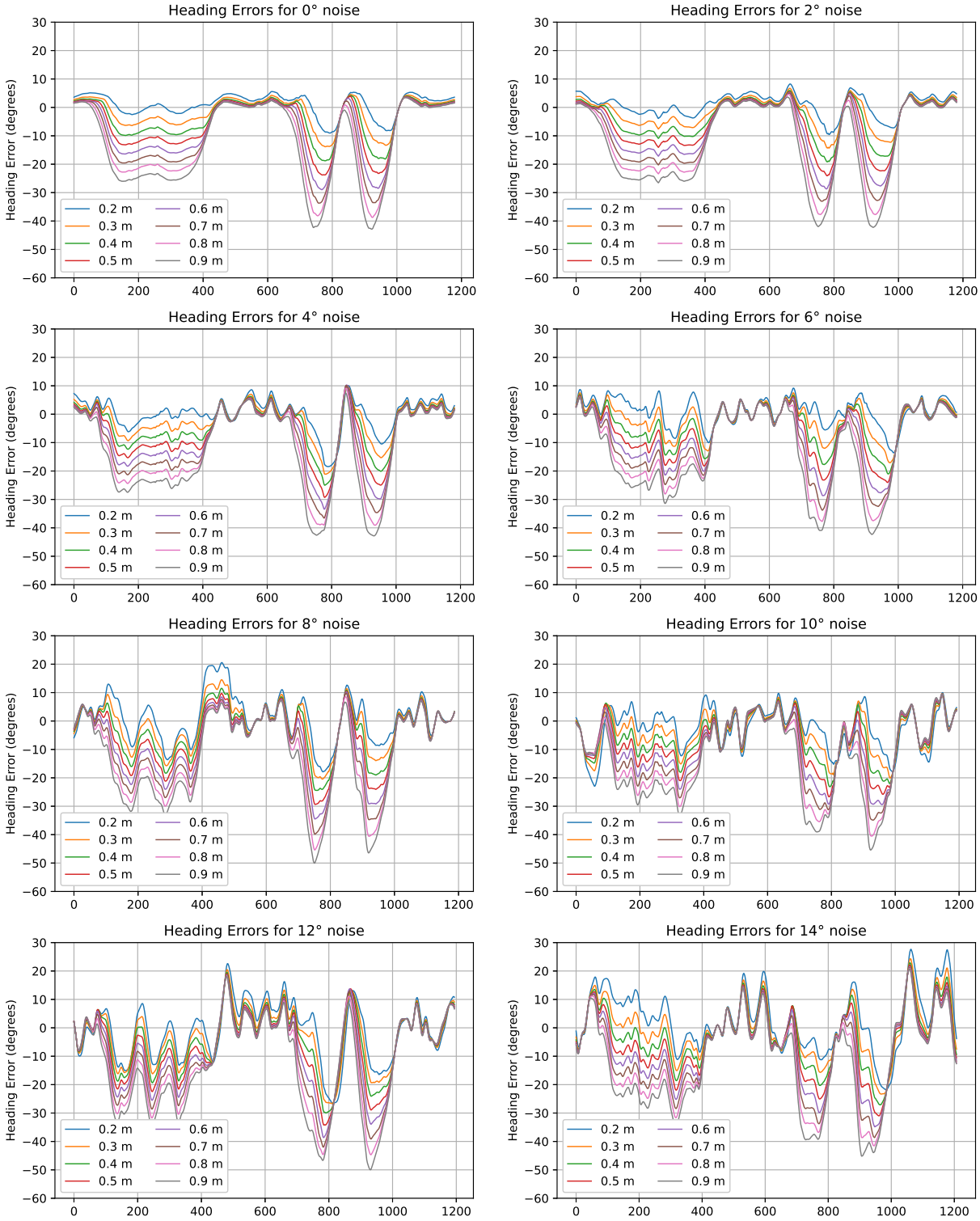

Figure 14: Evaluation datasets heading errors for the first lap

7.3 Network analysis

It's usually very difficult to analyze modern neural networks, since they are very complex, and they are composed of many layers and neurons; therefore it's very hard to identify how and where the output is generated. In our case, since the network is very simple, we can actually have an idea of the inner workings by looking at the weights, and in particular at the convolutional kernels.

Convolutional kernels

We start by looking directly at the convolutional kernels, that can be visualized as a group of 5x5 pixel images. Figure 15 shows the kernels of the first and second convolutional layers, since it becomes more difficult to understand the kernels of the deeper layers, due to the amount of linear combinations of the previous layers. As we can see, the first layers are focused on finding diagonal lines in the images, and it's coherent with what we expect given the task.

Another way to visualize the convolutional kernels is to use the technique explained in [37]; which consists in generating an image that maximize the activation of a single neuron. To generate the images we start from random noise, and we perform backpropagation and optimization over the image to maximize the activation of the neuron, leaving the network intact. This process is repeated for a fixed amount of iterations, until it converges to a local optimal image for the specific neuron. The results are shown in Figure 16. As we can see, the first layer is reacting more to basic diagonal lines. In the second layer we can see that the first neuron is starting to aggregate the basic lines into a more defined lane. In the third layer we can see how the position of the features starts to matter, and the features are more complex. This expected from theory and is a sign of a well-trained network. Not all then neurons are used, and some features are redundant, this is normal and a consequence of dropout and regularization in training. It's

important to note the fact that the image brightness has been normalized to the same value for all the images, so that it's easier to compare them, but the actual brightness of the features is important for the final value of the output.

Layer activations

Another interesting way to get insights about the network is to look at the neuron activations of the layers for some specific inputs. In order to do so 5 images have been selected from the test, picturing a very sharp left turn, a weak left turn, a straight line, a weak right turn, and a very sharp right turn. The images are shown in Figure 17, alongside with their preprocessed version. Figure 18 shows the activations of the linear layer before the output layer, for the 5 images. As we can see, the activations are very different from each other and almost all the activations seems to follow a monotonically increasing or decreasing function. This is another good sign of coherence with the task.

Figure 19 shows the activations of the convolutional layers and the pooling layers for the image of the straight section of the lane. As we can see, the different neurons segment the image in different ways, and the features are aggregated in the deeper layers. It is also possible to see how the pooling layers are reducing the dimensionality of the image and how the features becomes more and more abstract, with fewer and fewer pixels. The activations of the convolutional layers for the other images are shown in the appendix.

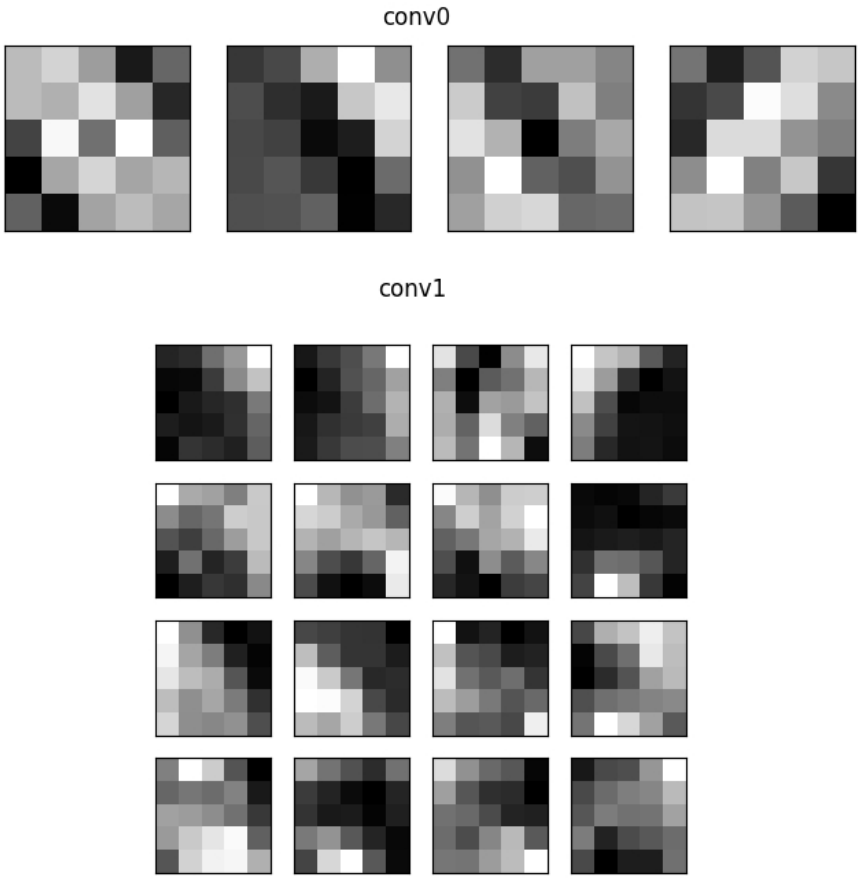

Figure 15: Convolutional kernels

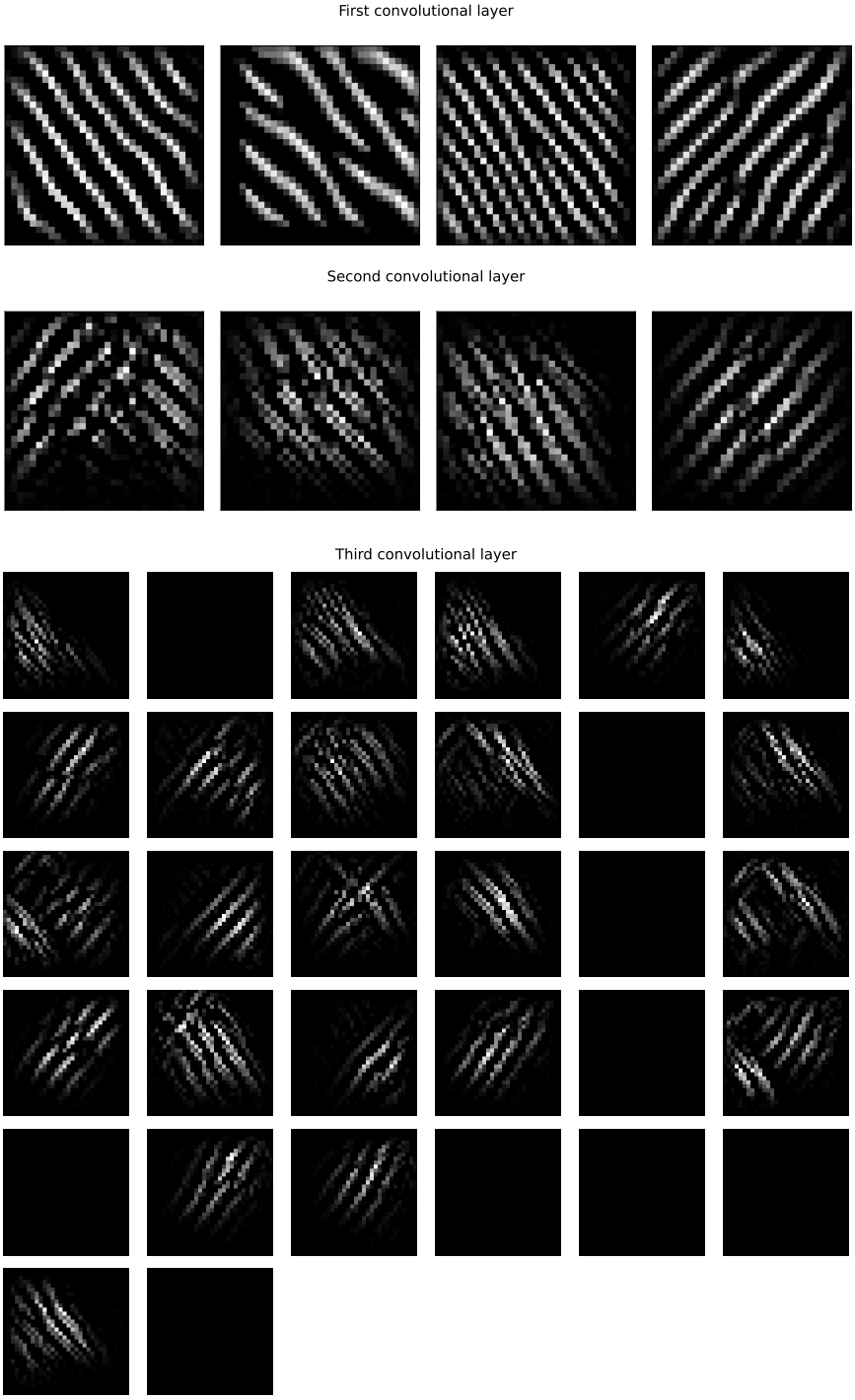

Figure 16: Convolutional kernels optimal input

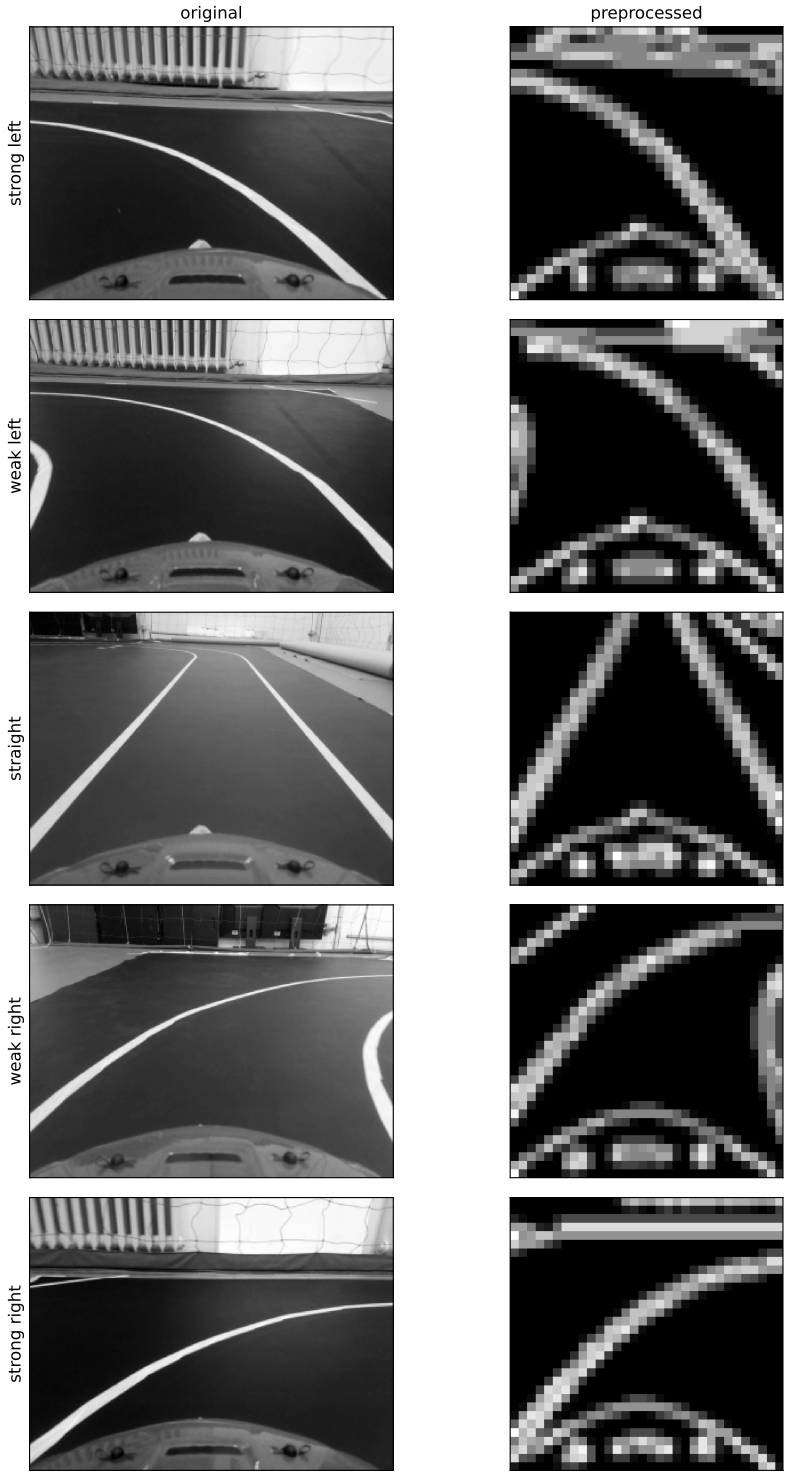

Figure 17: Different turns

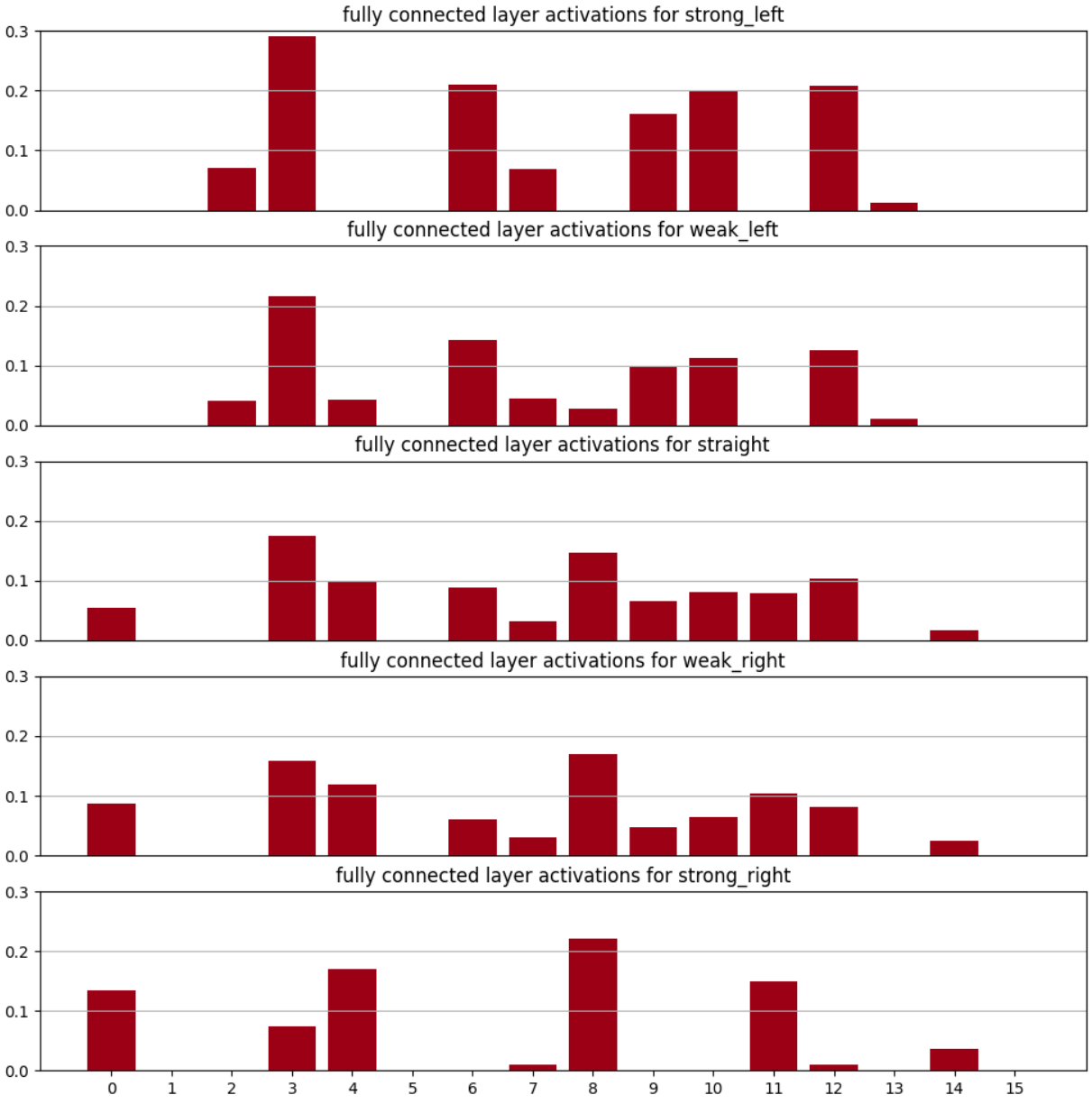

Figure 18: Activations of the linear layer

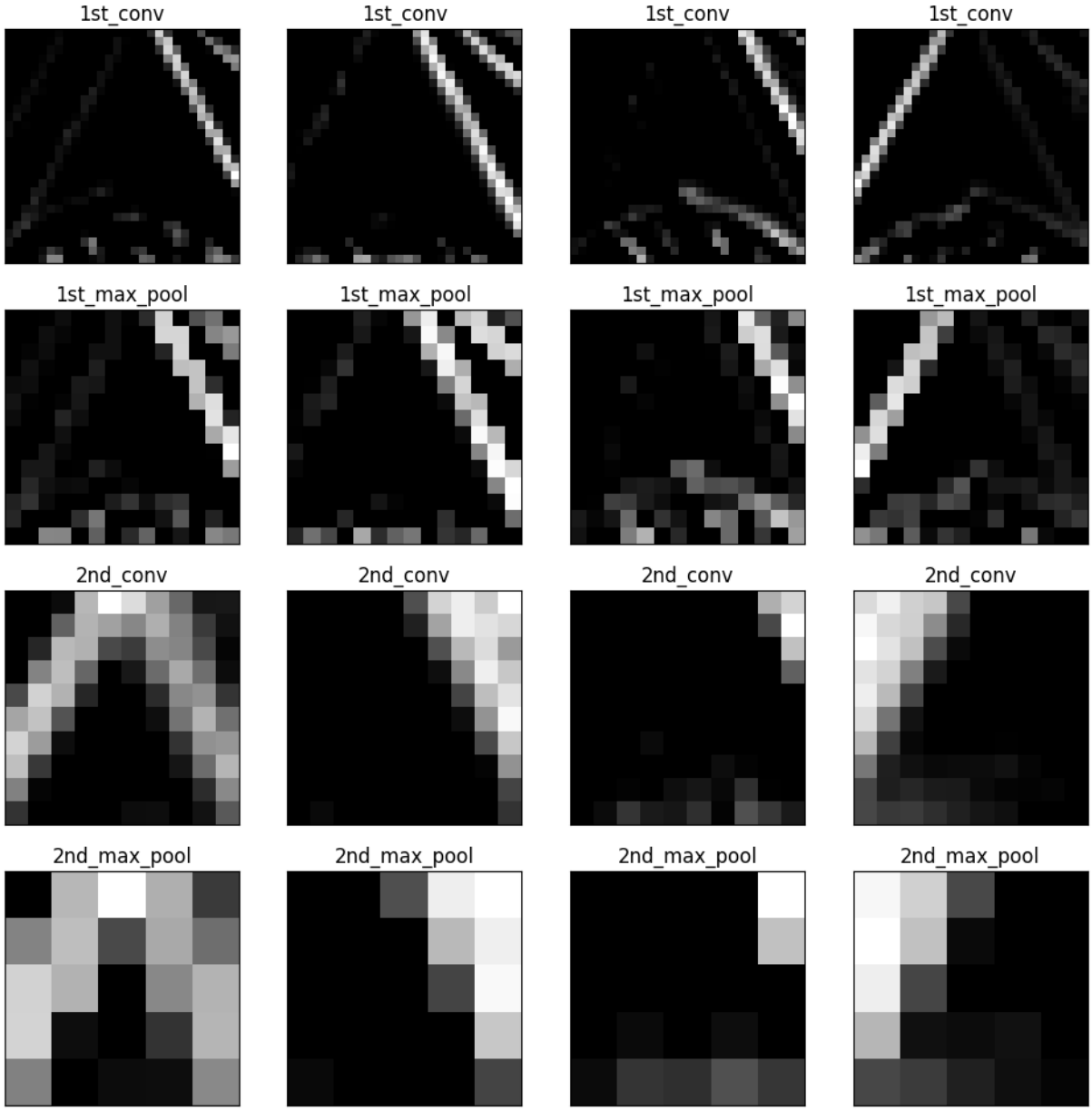

Figure 19: Activations of the convolutional layers for straight heading error

7.4 Hyperparameters study

In this section we study the effect of the different hyperparameters on the performance of the network. Since there are a lot of parameters, the space of the possible combinations is huge, and it's impossible to test all of them. Therefore, we have chosen to explore the parameters in 3 distinctive groups:

- **Learning parameters**: dropout probability, learning rate, number of epochs, $\mathcal{L}^2$ regularization.
- **Dataset parameters**: steering noise level, heading error distance ahead.
- **Preprocessing parameters**: blur kernel size, Canny, percentage of the image cut, image noise.

Methodology

For each group of parameters, a set of values for each parameter is chosen. Then, for each combination, the whole training process, as explained in Section 6.3, is performed. The combination is evaluated on 3 sets of real datasets and their simulated versions: a clean group of datasets, with a steering noise $\sigma_s \leq 4$ deg, a very noisy group of datasets, with $\sigma_s \geq 10$ deg, and finally on the set of all the datasets. The performance is evaluated calculating the standard deviation of the error between the network estimate and the ground truth, for each sample of each dataset in the set. When evaluating the performance for a single parameter a mean of the metric across all the combinations remaining parameters is calculated. When evaluating 2 parameters together, the mean is calculated across all the combinations of the remaining parameters except the 2 parameters being evaluated.

Learning parameters

The learning parameters are the most delicate ones, since they are the ones that have the most impact on the training process, and therefore on the final performance. With a bad choice of learning rate or a wrong number of epochs, the training will not converge to a good solution, resulting in extremely low estimation performance. The dropout probability and $\mathcal{L}^2$ regularization are also important, but have less impact than the previous 2 parameters, since the range of acceptable values is larger. More specifically, the dropout probability should be ≤ 0.5, and the $\mathcal{L}^2$ regularization should be ≤ 0.1. Figures 21 and 20 shows the influence of the different learning parameters on the evaluation datasets. As we can see, the learning rate has a minimum at around 5×10^{-3}. The dropout has a minimum at around 0.3, it's interesting to notice how the minimum actually is closer to 0.2 for the noisy datapoints, meaning that the network requires more complexity in order to better estimate those datasets. The $\mathcal{L}^2$ regularization has a minimum around 0.01. The STD appears to monotonically decrease with the number of epochs, but this is a consequence of taking the mean across the remaining combinations, and in particular across the learning rates, in this case more epochs can actually compensate for a too low learning rate. Figure 22 shows the STD for the different values of learning rate and number of epochs. As we can see, the STD is very low at around 200 epochs.

From the generalization point of view, it's possible to see how the STDs in simulation and in the real case are very similar; meaning that the network is able to generalize well to the real world, but also that it would have been possible to find the best learning parameters by means of simulation only.

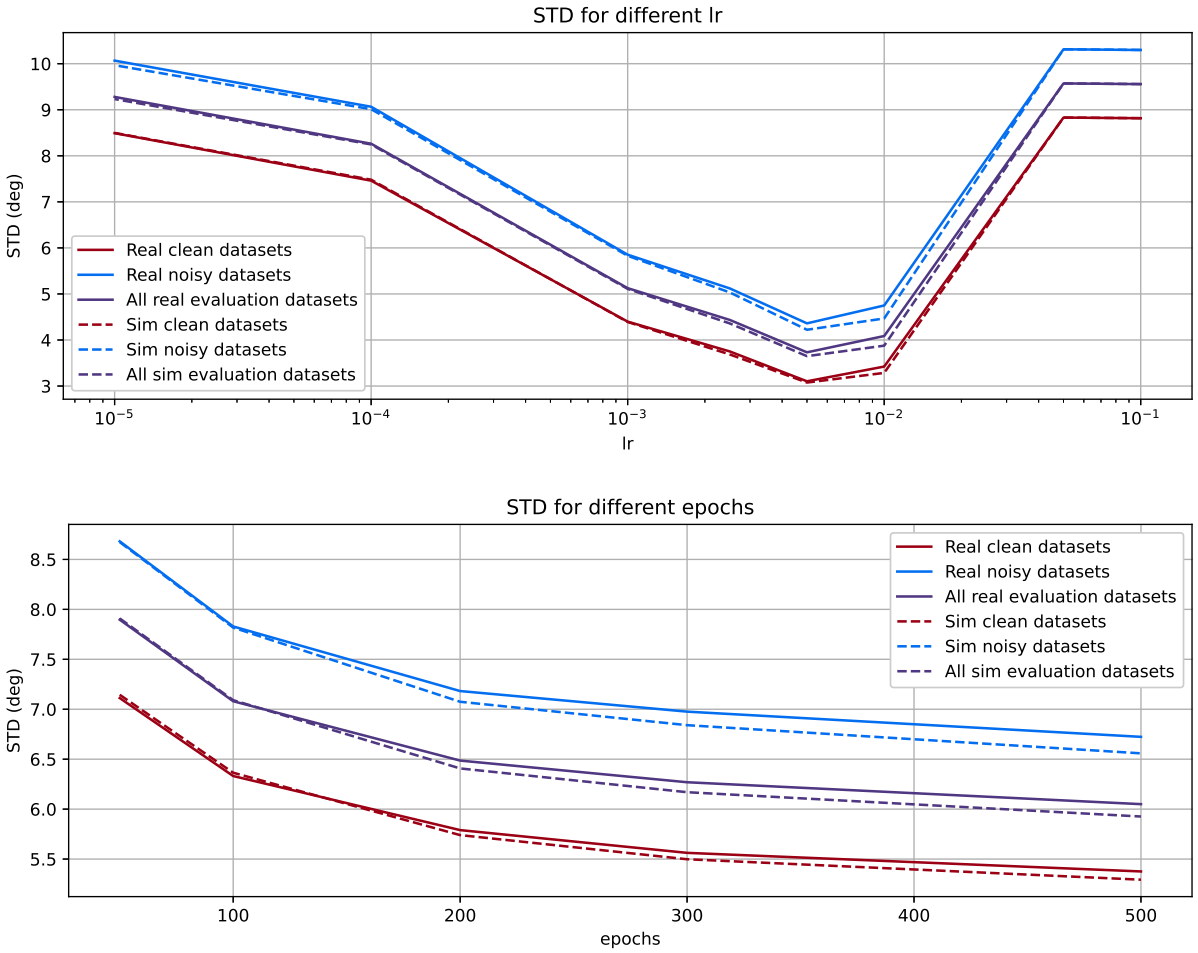

Figure 20: STD for different learning parameters (2)

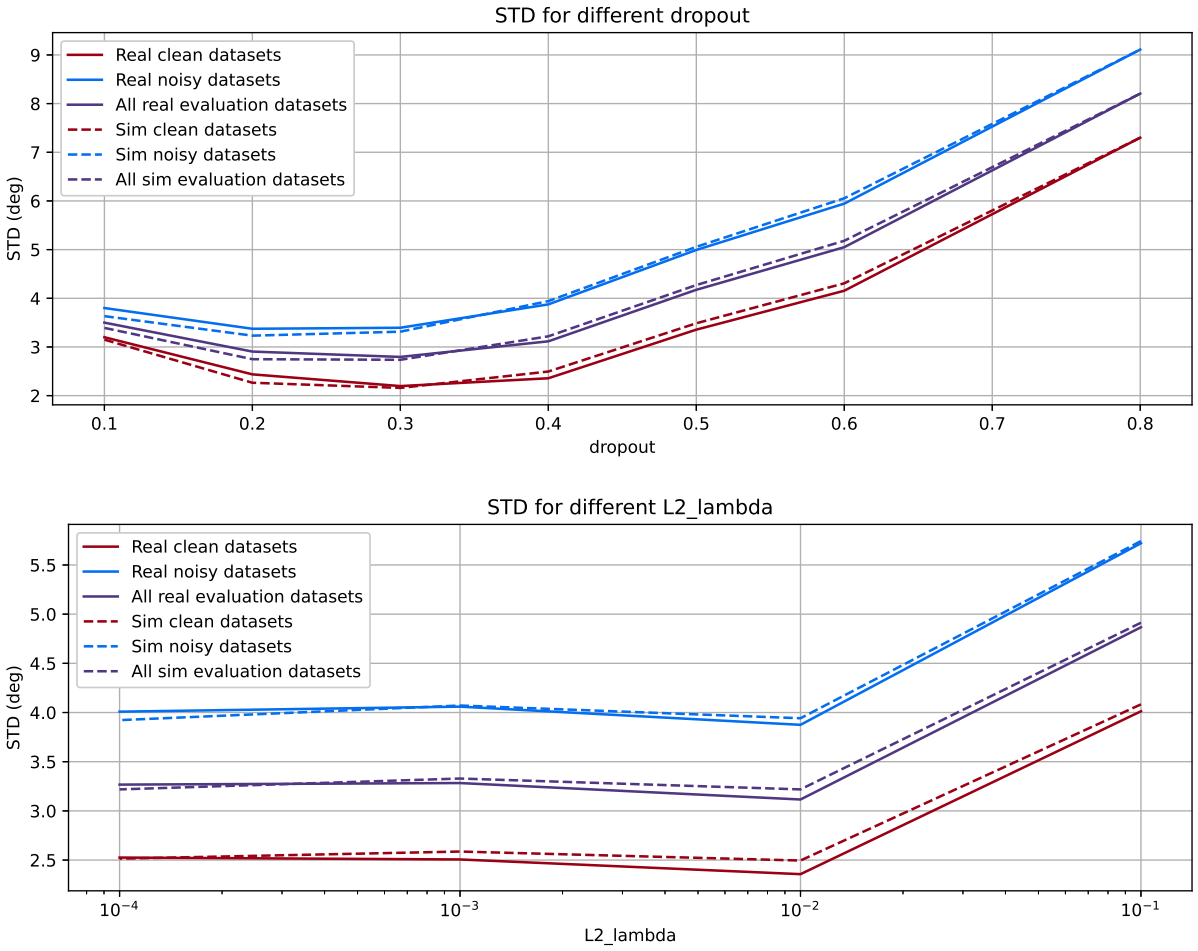

Figure 21: STD for different learning parameters (1)

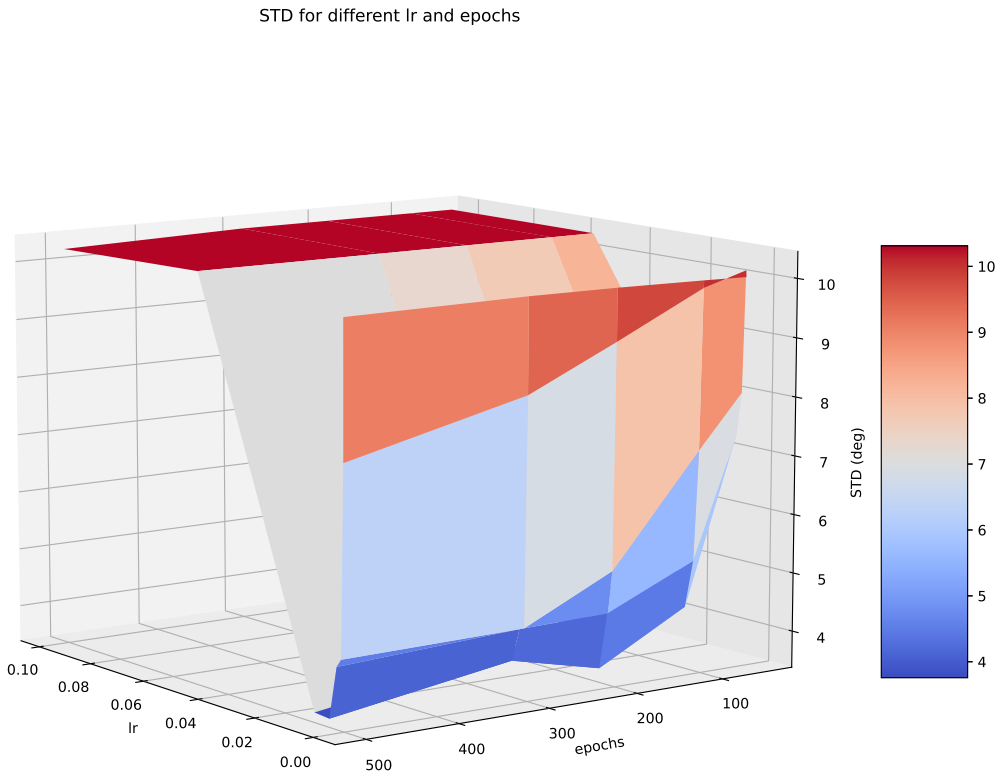

Figure 22: STD for different learning rates and number of epochs

Dataset parameters

The 2 most important dataset-related parameter are the amount of steering noise and the distance ahead of the heading error. Figure 23 shows the STD for different values of these parameters. The datasets without steering noise have the worse performance, since the training lacks variability. The best value is around $\sigma_s = 12$ deg. The minimum is actually slightly above $\sigma_s = 12$ deg for the noisy datasets, meaning that, very intuitively, the noisy training datasets are better to estimate noisy evaluation datasets. Looking at the distance ahead of the heading error, we can see that the STD is quiet low for all the datapoints, but has a minimum for the distance of 0.4 m. The network is predictably less accurate for further distances, since the heading error is less visible in the image. But, it's very interesting to notice that also very close distance, like 0.2 m, are harder to estimate, probably because the camera is mounted too ahead w.r.t. the rear axis, from which the distance is calculated. Figure 24 shows the STD for different values of the 2 parameters. The STD follows a nice convex shape, with a minimum for the aforementioned values.

From the generalization point of view, we can see that also in this case the STDs in simulation and real world are very similar. Confirming the generalization capabilities of the network. The network performs slightly better in the simulated environment for high steer noise, and the minimum STD for the heading error distance is actually slightly lower in simulation, a possible explanation for this behavior is that there might be a slight misalignment of the camera on the real car, that becomes relevant for very close distances.

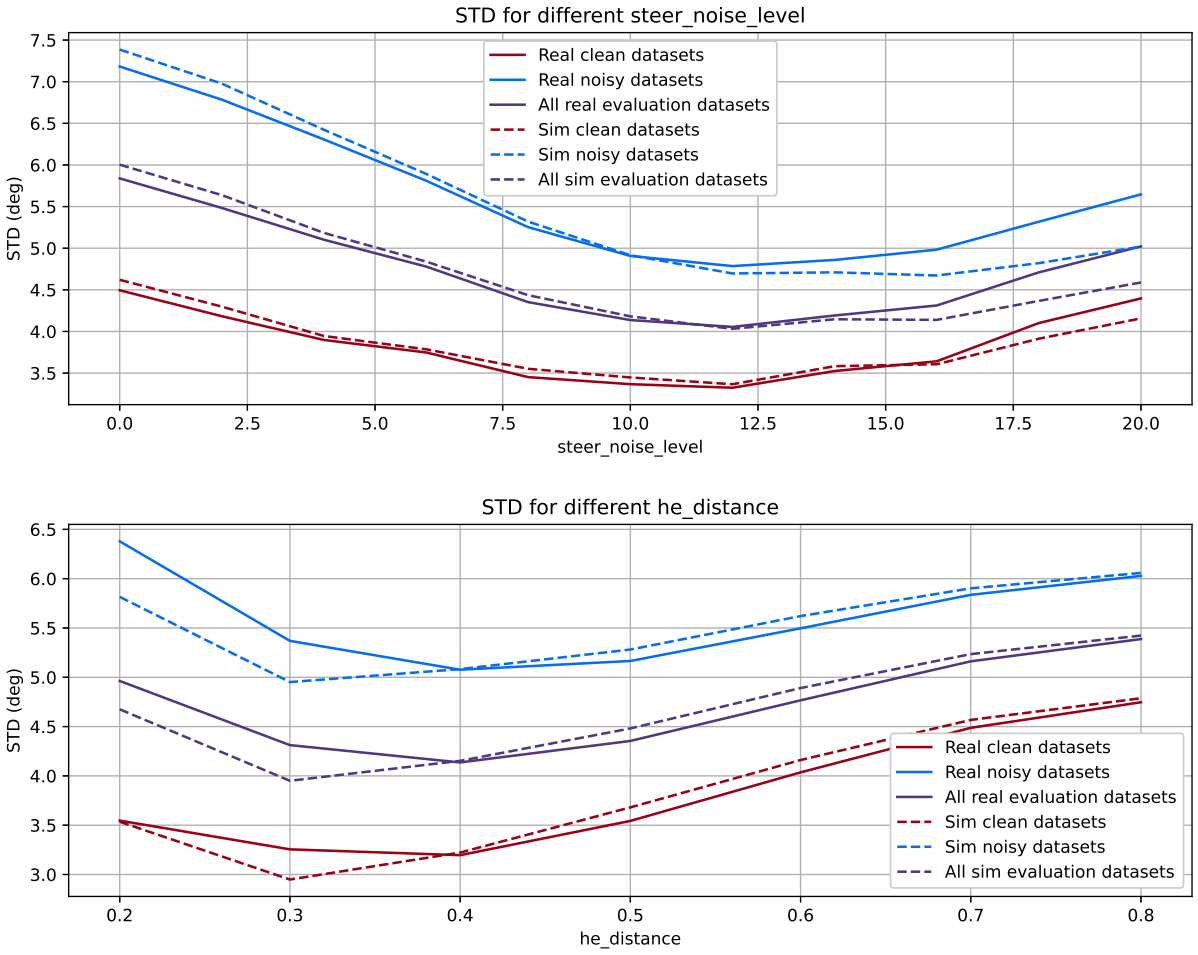

Figure 23: STD for different dataset parameters

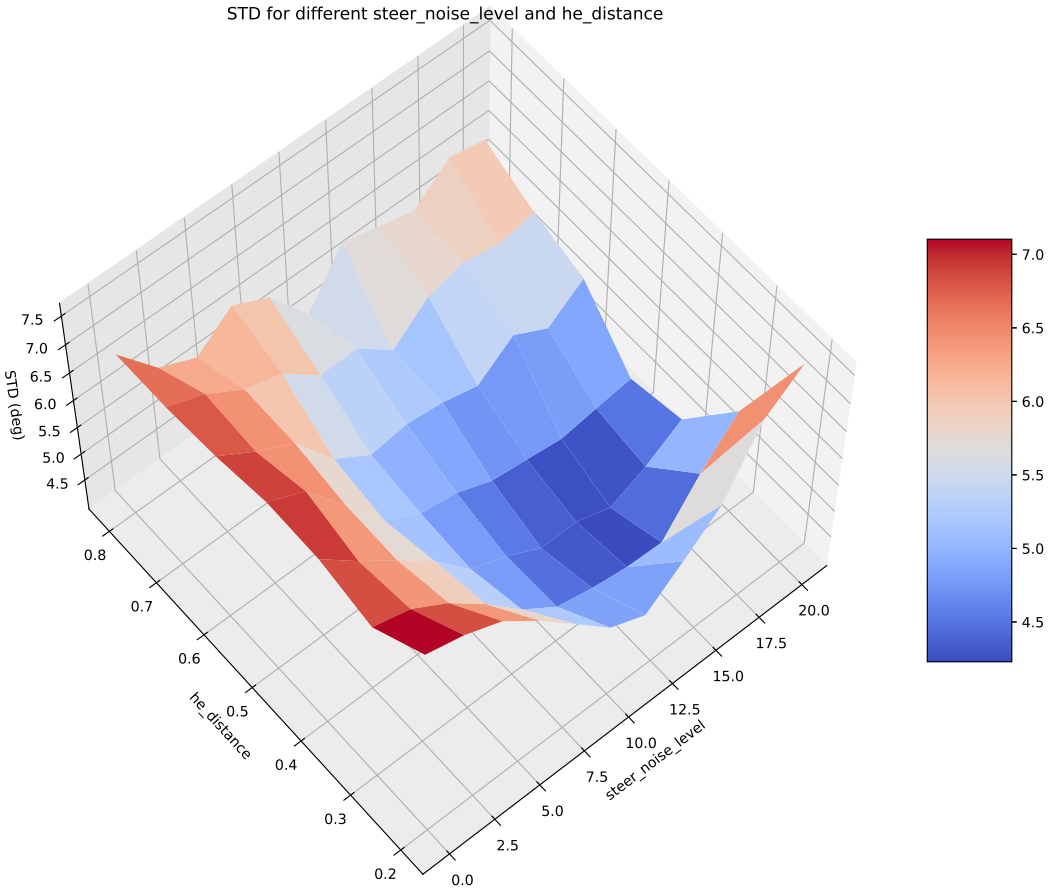

Figure 24: STD for different steering noise and distance ahead

Preprocessing parameters

Figures 25 and 26 shows the STDs for the preprocessing parameters. These parameters are the ones that have the least impact on the performance between the one analyzed, but that play a major role for the generalization capabilities of the strategy. In fact, if we look at the difference between the application and the non-application of Canny edge detection we see that in simulation we have better performance when the image is left intact, while in the real world the opposite is true. A possible explanation for this is the fact that Canny removes all the brightness information from the image, leaving only the strong edges, and this appears to make images from simulation and real life very similar, avoiding overfitting on training data. The effect could already be predicted by looking at Figure 9.

The blur kernel size has a minimum at around 3. The network performs better with low image noise, with a minimum at around 40/255. The percentage of the image cut has very little influence on the performance, however, looking at Figure 27, we can see that for estimating heading errors at further distances it is better to use a larger percentage of the image.

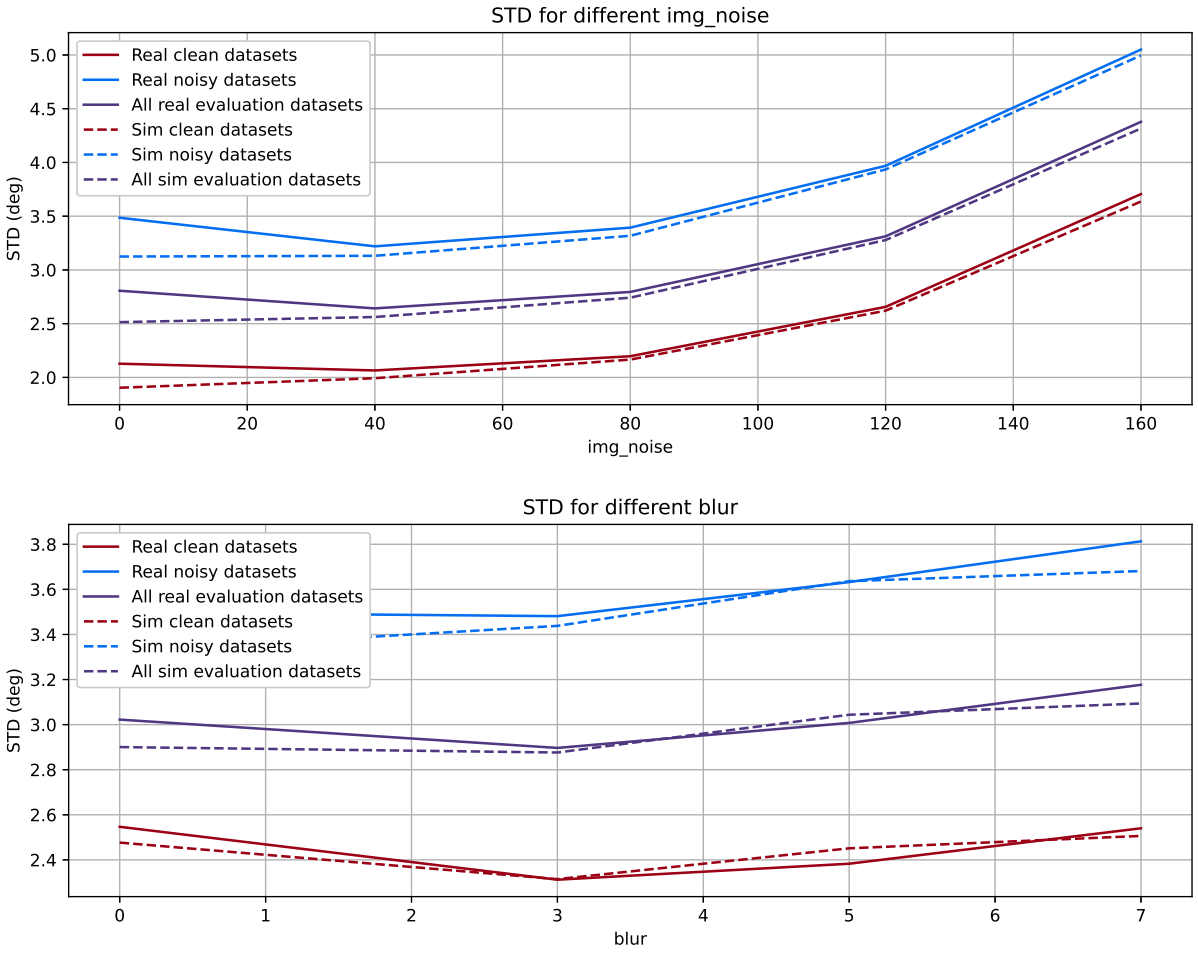

Figure 25: STD for different preprocessing parameters(1)

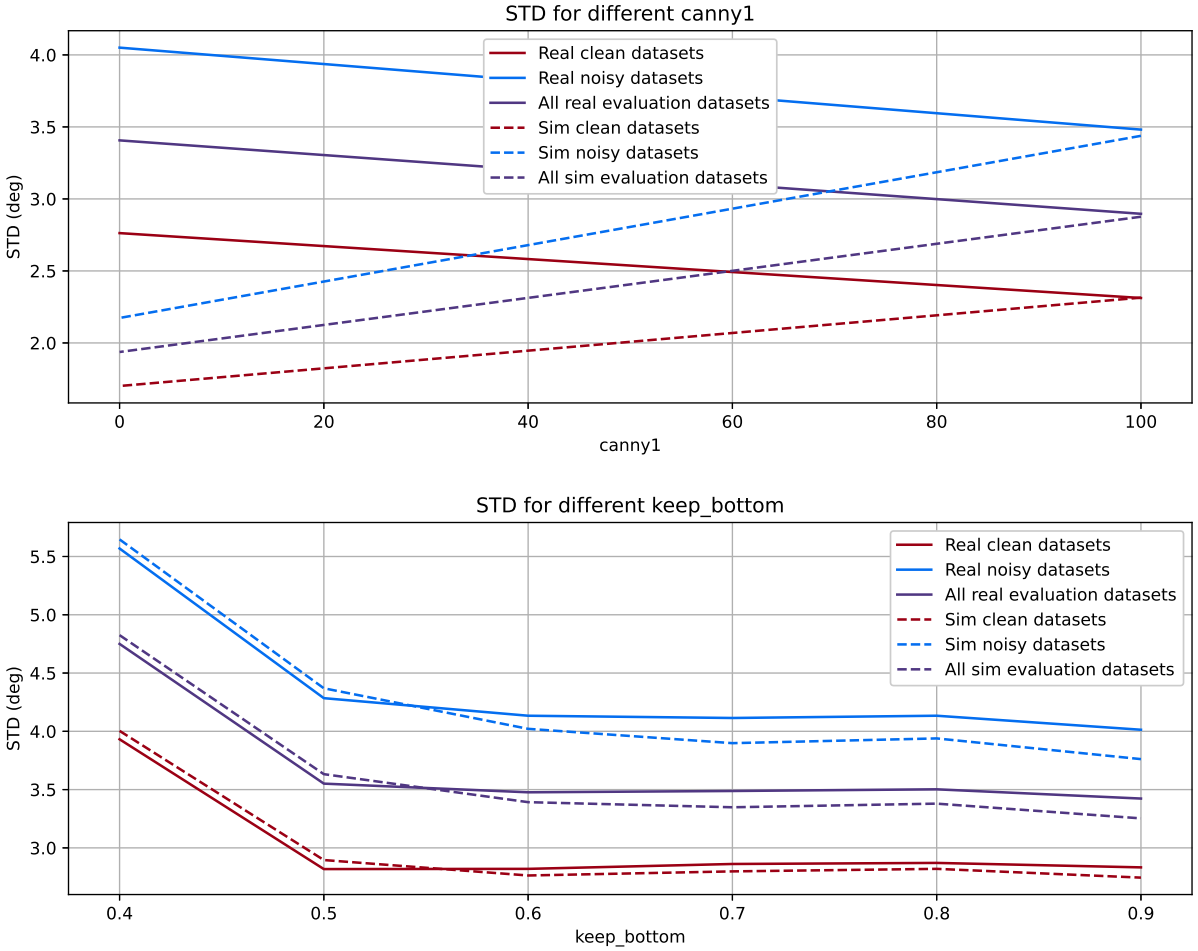

Figure 26: STD for different preprocessing parameters(2), note that Canny has only 2 values: 0 meaning it is not applied and 100 meaning it is applied

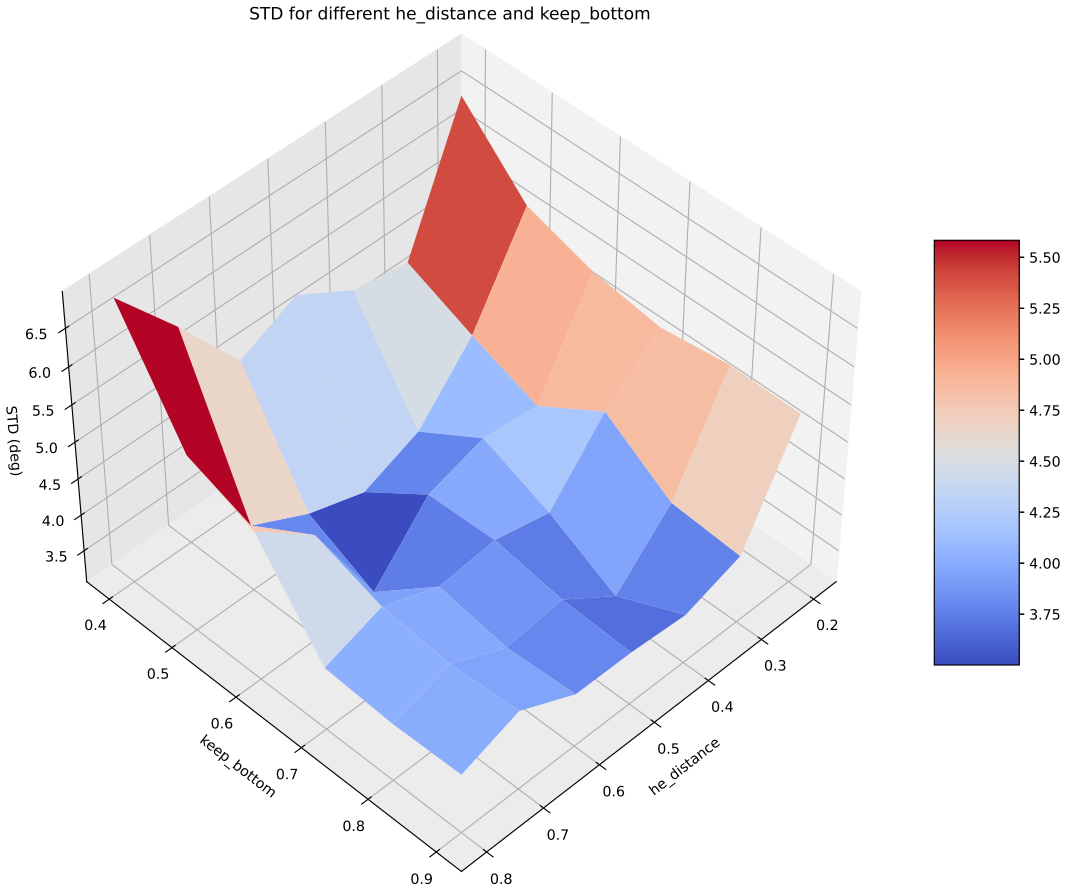

Figure 27: STD for different percentage of the image cut and distance ahead

7.5 Control loop application

In this section we will briefly discuss how the network performs in a real control loop scenario. Two tests have been performed:

- **Test 1**: The best network with a heading error distance of 0.5m has been used. The network is fed with images from the camera mounted on the car. The heading error is used as input of a PP controller that operates at a frequency of 30 Hz and acts on the steering. The whole control scheme is implemented on a Raspberry Pi 4 mounted on the car. The vehicle is kept at a constant velocity of $0.3\mathrm{m\,s^{-1}}$.

- **Test 2**: The best network with a heading error distance of 0.8m has been used. The speed is increased to $1.0\mathrm{m\,s^{-1}}$. The controller used in this test is more advanced and considers also a derivative term and a speed profile that slightly decrease the speed in sharp turns.

The results of the two tests are shown in Figures 28 and 29. The first test shows how the network, together with the PP controller, is able to keep the vehicle on the road and perfectly tracking the centerline.

In the second test the vehicle is able to remain inside the lane, even at high speed. The tracking is not perfect since the whole control scheme is limited by the maximum steering rate allowed by the steering actuator. An interesting point is how the network performs better than a controller based on the true position of the vehicle obtained from the Vicon system. This is because, due to the delay in the Vicon system, the controller is not able to react fast enough and cannot keep the lane at high speeds.

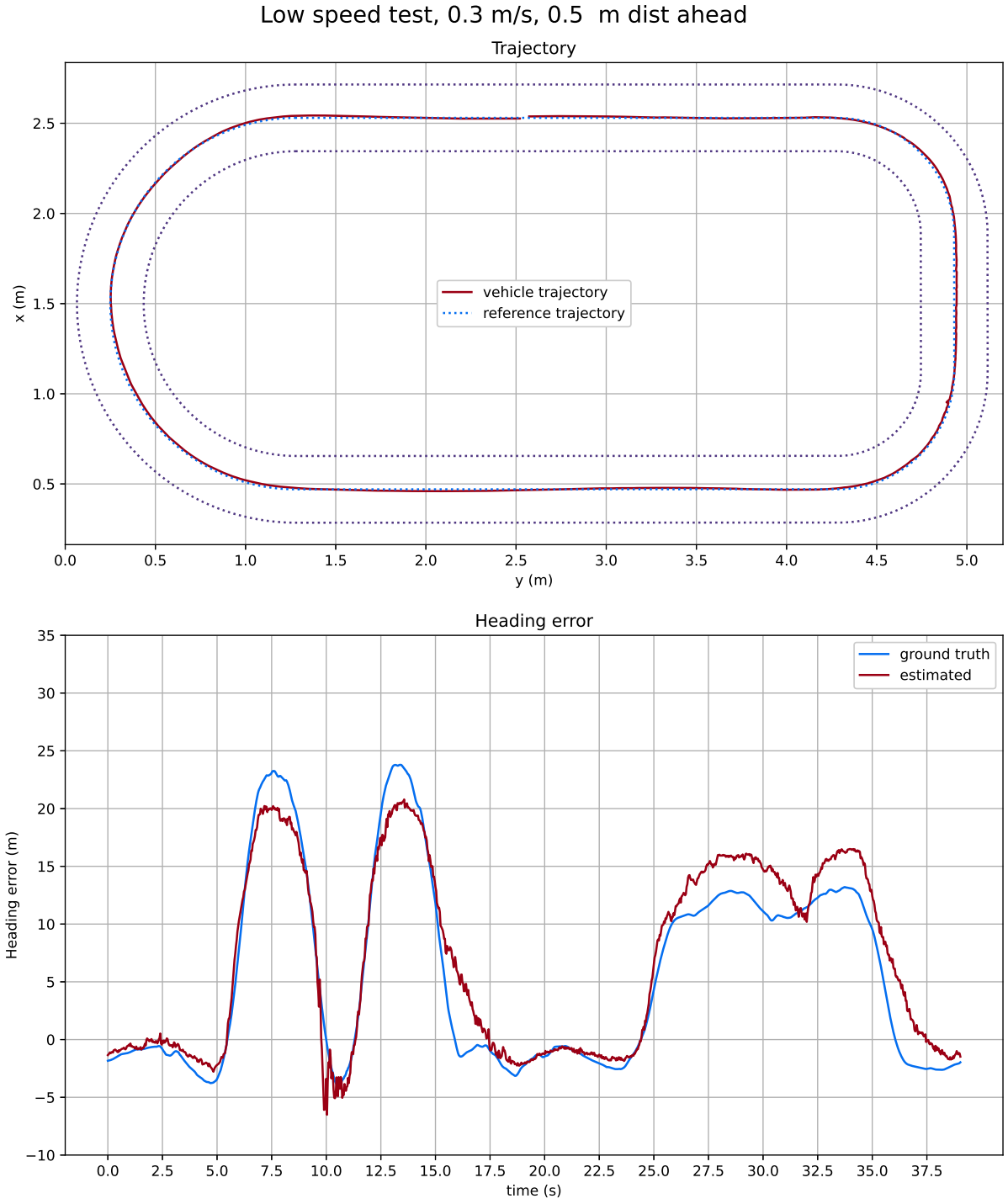

Figure 28: Control loop results: low speed test

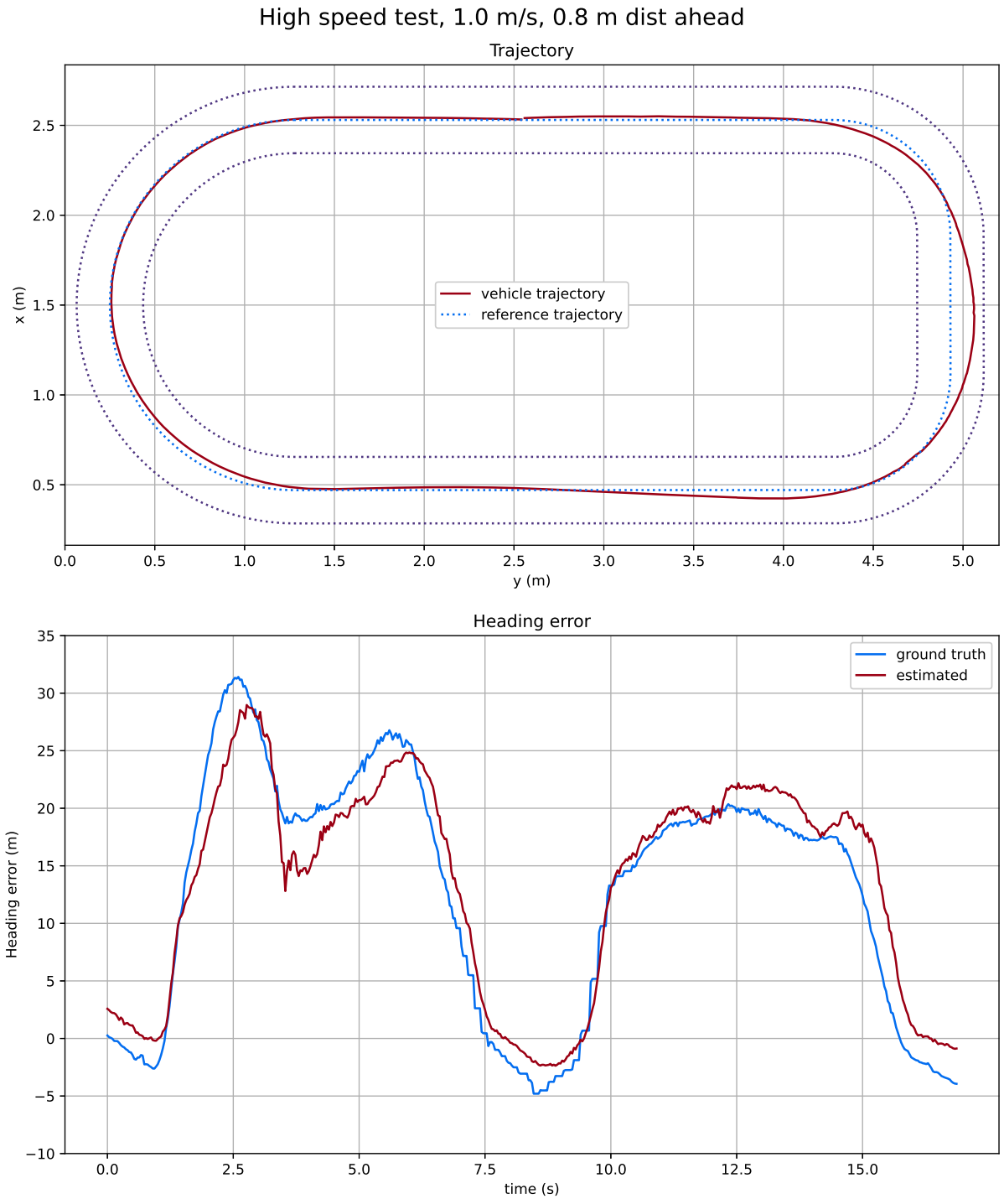

Figure 29: Control loop results: high speed test

8 Conclusion

The goal of this master thesis is to design and implement an efficient and reliable camera-based and gps-denied estimation strategy for the heading error of a scaled vehicle. Keeping in mind the strategy has been developed for the BFMC, to be applied in an embedded device on a track we didn't have access to, in this chapter we will briefly summarize the main takeaways and present an outlook for future work.

8.1 Summary

The main concepts we dealt with in this thesis and their respective most important takeaways are the following:

- **Heading error estimation**: the estimation strategy is centered around the heading error, which is the key input for the PP controller. The concept of separating the perception of the road ahead from the vehicle control showed to be very effective and allowed us to fine tune the performance of the whole control scheme on the days of the competition without worrying about the underlying estimation process.

- **Neural network estimation**: due to the fact that the vehicle is operating without having access to a global positioning systems, and due to the non-existence of sensors capable of locally measuring the heading error of the vehicle, the choice of using a vision-based estimation strategy came very naturally. The idea of using a convolutional neural network followed because of the high rate of success in similar problems of this architecture. During the development we discovered that a very small and simple convolutional neural network is capable of solving the estimation problem, and outperforms more complex architectures; with the added benefit of being very computationally

efficient. From the hyperparameters point of view we confirmed the very well established importance of the learning rate and the number of epochs as determinant factors for the final estimation performance.

- **Training in simulation**: the idea of using the simulation environment to generate the training datasets was driven by the fact that we could not access the real competition track in the development phase. The rationale behind it was that if the strategy works in simulation and in our recreated real life scenario, then there is a high probability for it to be working in the real track. Generating the datasets in simulation has several advantages, from being able to create unlimited amount of samples, to very precise control the dataset generation process. From the analysis we discovered the importance of both the preprocessing applied to the images and the amount of variability in the training examples. The preprocessing step is what allows to almost seamlessly translate from the simulation environment to reality. The amount of variability in the training set is also a delicate parameter, and we discovered how networks trained on datasets with many examples of the vehicle in unusual positions and orientations perform better on average.

- **Application in closed loop control scheme**: When testing the vehicle capabilities in the test track in the university laboratory and at the competition track, the whole estimation and control strategy worked very well and was able to keep the lane even at high speed.

8.2 Future Work

The proposed strategy presents several opportunities for future work. The estimation could be extended from the single camera to a full 360 degrees view of the surroundings to improve the localization capabilities, or the estimation target

could be extended from the heading error to a section of the centerline ahead, or some more road parameters could be estimated like, for example, the curvature ahead. The network architecture could be improved by considering sequences of frames rather than single frames, in order to add redundancy and time coherence.

Finally, the general concept of using machine learning to estimate control inputs could be applied in other control schemes where the input cannot be retrieved using standard methods.

9 Appendix

9.1 Additional convolutional layer activations

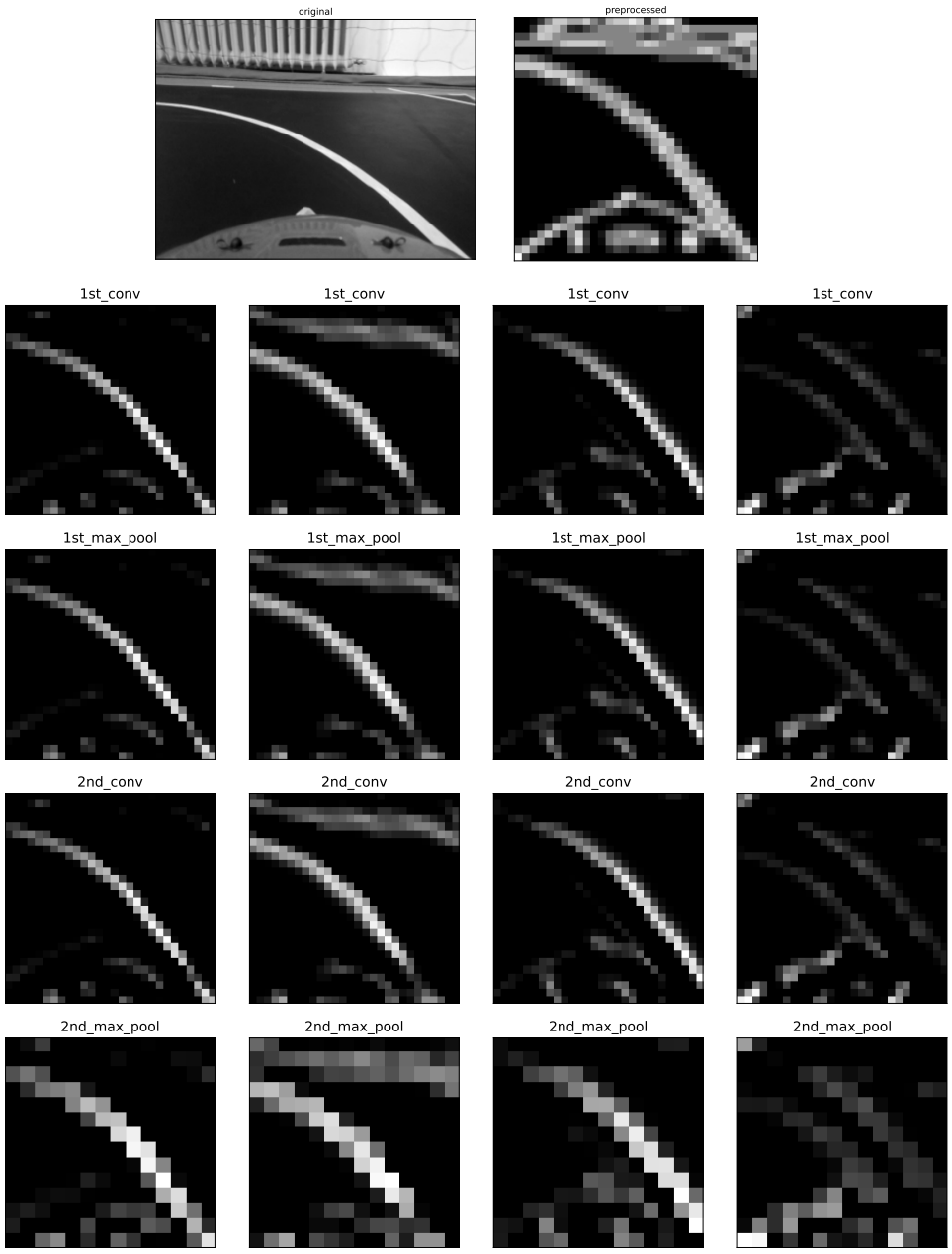

Figure 30: Activations for strong left turn

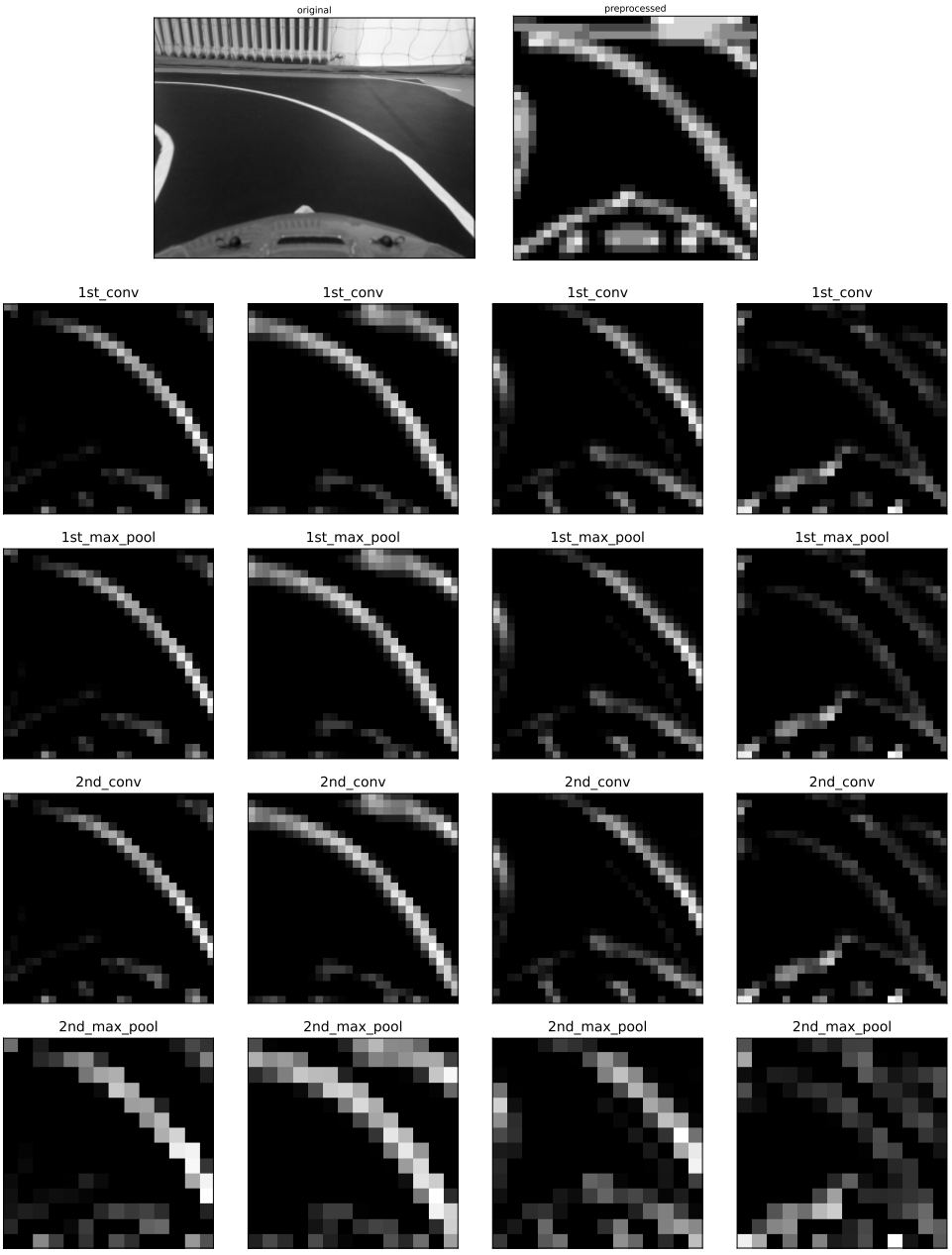

Figure 31: Activations for weak left turn

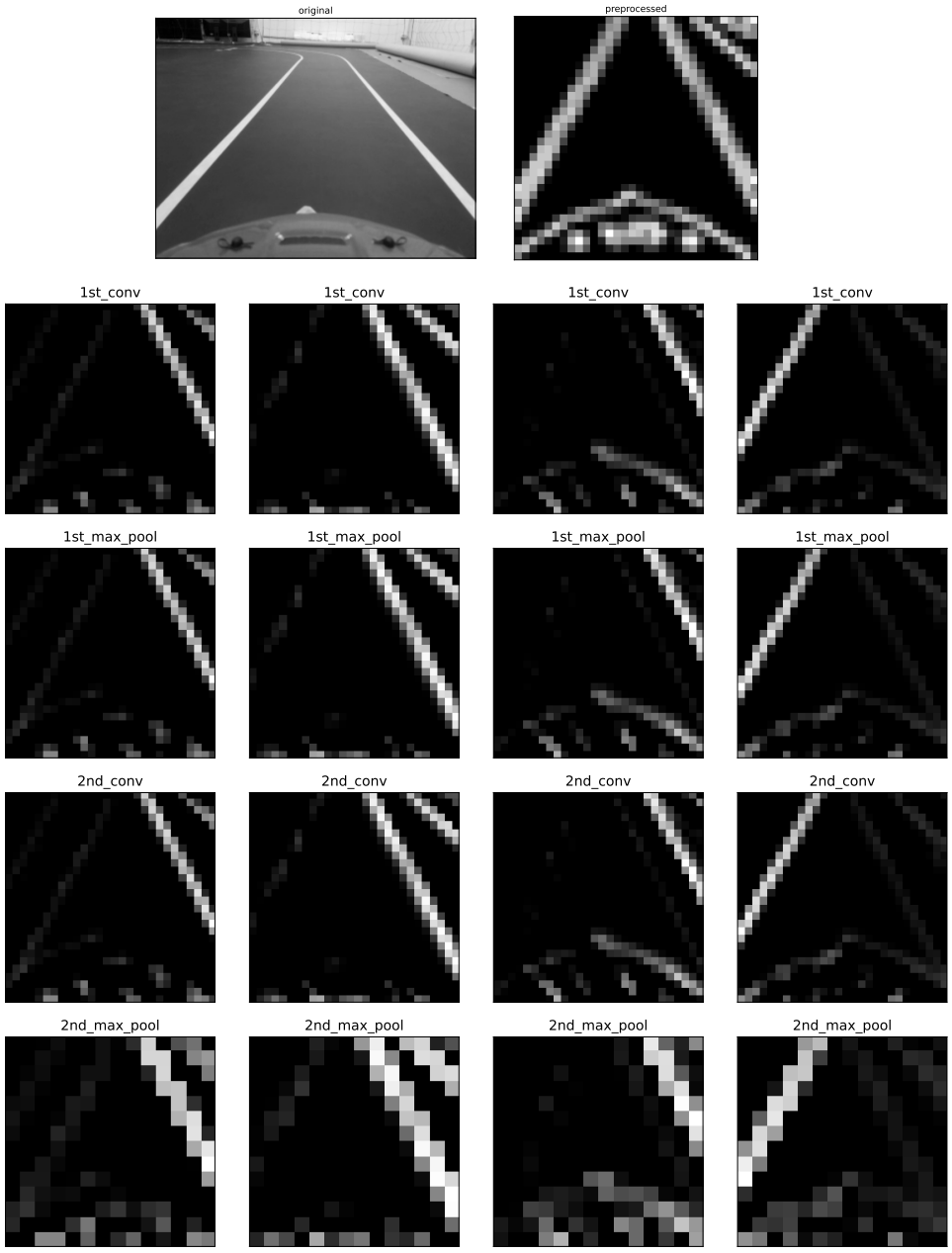

Figure 32: Activations for straight

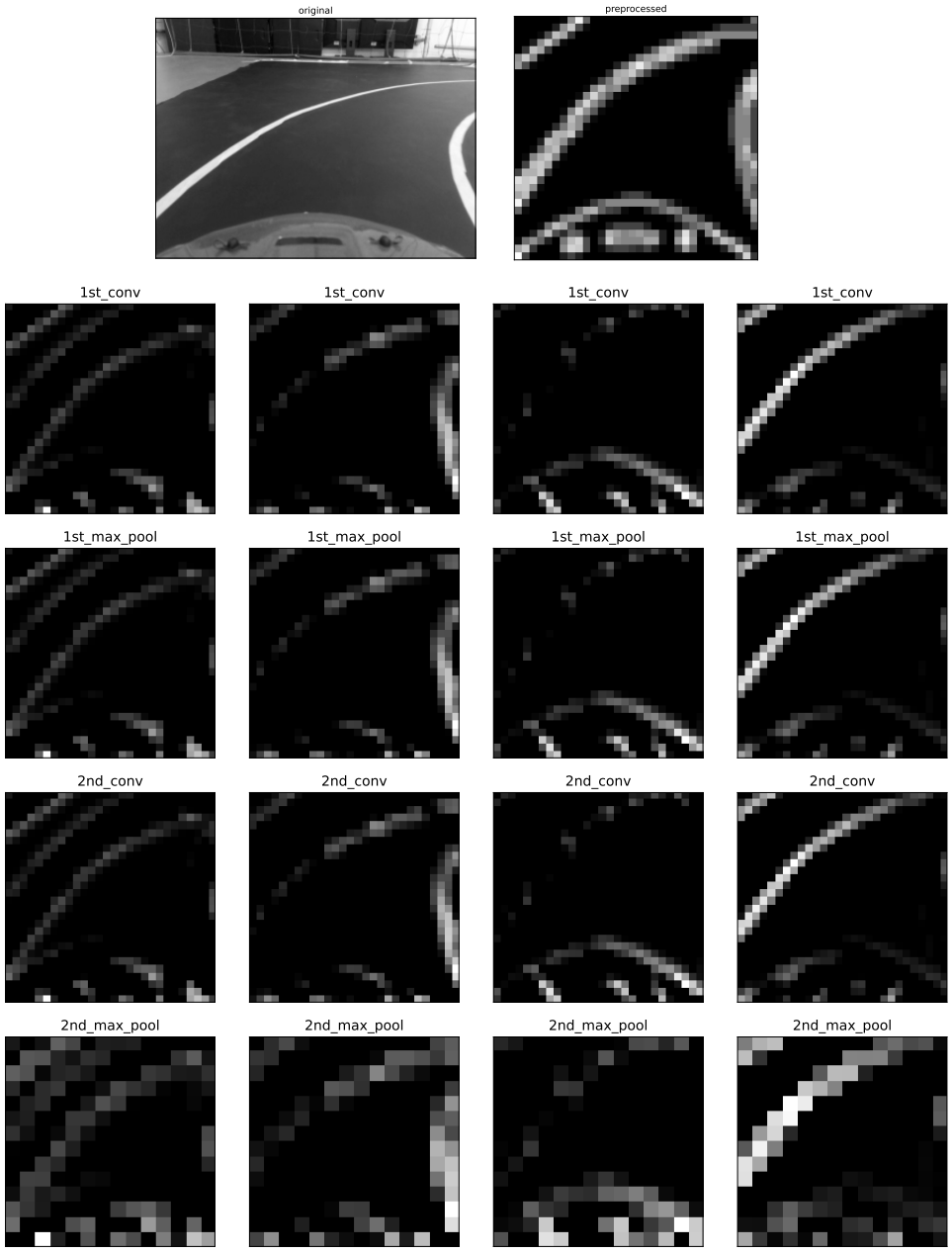

Figure 33: Activations for weak right turn

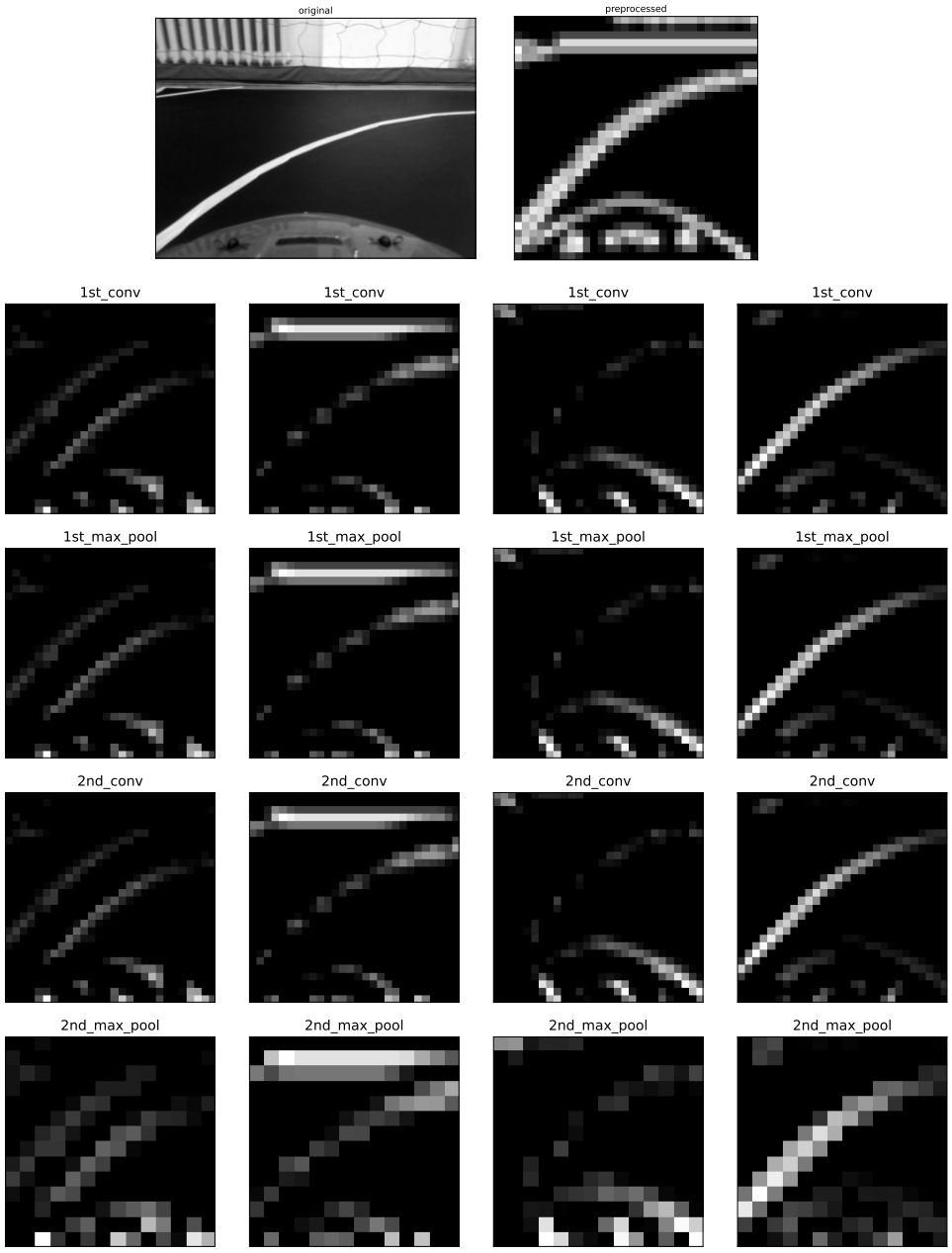

Figure 34: Activations for strong right turn

9.2 Python implementation of heading error calculation

```
import numpy as np
def get_heading_error(x,y,yaw,path,dist_ahead, tolerance=0.01):
    p = np.array([x,y]).T #current position of the car
    min_index = np.argmin(np.linalg.norm( path-p,axis=1)) #index of
        clostest point on path
    #roll path
    path = np.roll(path, -min_index, axis=0)
    p_min = path[0]
    #rotate p_min to car frame
    p_min_car = np.array([p_min[0]-x, p_min[1]-y])
    p_min_car = p_min_car @ np.array([[np.cos(yaw), -np.sin(yaw)], [
        np.sin(yaw), np.cos(yaw)]])
    dist = p_min_car[0] #signed distance to closest point on path,
        approx
    #calculate point ahead
    k = dist_ahead / 10.0 + 0.01
    path_ahead = path[0:int(path.shape[0]*k)]
    path_behind = path[int(path.shape[0]*(1-k)):]
    dists_a = np.abs(np.linalg.norm(path_ahead-p,axis=1) -
        dist_ahead) #distances to all points on path
    dists_b = np.abs(np.linalg.norm(path_behind-p,axis=1) -
        dist_ahead) #distances to all points on path
    closest_a = np.argmin(dists_a) #index of closest point on path
    closest_b = np.argmin(dists_b) #index of closest point on path
    min_dist_a = dists_a[closest_a] #distance to closest point on
        path
```

```
    min_dist_b = dists_b[closest_b] #distance to closest point on
        path
    if min_dist_a > tolerance or min_dist_b > tolerance: # too far
        from the path
        pHE = path_ahead[closest_a] #point ahead = closest point on
            path
    else:
        pa = path_ahead[np.max(np.where(dists_a < tolerance))]
        pb = path_behind[np.max(np.where(dists_b < tolerance))]
        #calculate the point going dist ahead from the car in the
            car yaw direction
        pp = p + dist_ahead*np.array([np.cos(yaw), np.sin(yaw)])
        #choose between pa and pb
        if np.linalg.norm(pp-pa) < np.linalg.norm(pp-pb):
            pHE = pa
        else:
            pHE = pb
    #calculate heading error
    yaw_ref = np.arctan2(pHE[1]-p[1],pHE[0]-p[0]) #yaw reference in
        world frame
    he = diff_angle(yaw_ref, yaw) #heading error
    return he
```

References

[1] J.C. McCall and M.M. Trivedi. Video-based lane estimation and tracking for driver assistance: survey, system, and evaluation. *IEEE Transactions on Intelligent Transportation Systems*, 7(1):20–37, 2006.

[2] Jun Li, Xue Mei, Danil Prokhorov, and Dacheng Tao. Deep neural network for structural prediction and lane detection in traffic scene. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):690–703, 2017.

[3] Yaofu Huang, Zengshan Tian, Qing Jiang, and Junxing Xu. Path tracking based on improved pure pursuit model and pid. In *2020 IEEE 2nd International Conference on Civil Aviation Safety and Information Technology (ICCASIT*, pages 359–364, 2020.

[4] Dirk E. Smith and John M. Starkey. Effects of model complexity on the performance of automated vehicle steering controllers: Model development, validation and comparison. *Vehicle System Dynamics*, 24(2):163–181, 1995.

[5] Philip Polack, Florent Altché, Brigitte d'Andréa Novel, and Arnaud de La Fortelle. The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles? In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 812–818, 2017.

[6] R Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, 1992.

[7] Hiroki Ohta, Naoki Akai, Eijiro Takeuchi, Shinpei Kato, and Masato Edahiro. Pure pursuit revisited: Field testing of autonomous vehicles in urban areas. In *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 7–12, 2016.

[8] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[9] Y Kanayama and B Hartman. Smooth local path planning for autonomous vehicles (part ii. cubic spirals). In *Proc. Annual Conference of Robotics Society of Japan*, pages 93–96, 1988.

[10] Hervé Delingette, Martial Hebert, and Katsushi Ikeuchi. Trajectory generation with curvature constraint based on energy minimization. 1991.

[11] A Segovia, M Rombaut, A Preciado, and D Meizel. Comparative study of the different methods of path generation for a mobile robot in a free environment. In *Fifth International Conference on Advanced Robotics' Robots in Unstructured Environments*, pages 1667–1670. IEEE, 1991.

[12] Boris Lau, Christoph Sprunk, and Wolfram Burgard. Kinodynamic motion planning for mobile robots using splines. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2427–2433. IEEE, 2009.

[13] Madhavan Shanmugavel, Antonios Tsourdos, Brian White, and Rafał Żbikowski. Co-operative path planning of multiple uavs using dubins paths with clothoid arcs. *Control engineering practice*, 18(9):1084–1092, 2010.

[14] S. Fleury, P. Soueres, J.-P. Laumond, and R. Chatila. Primitives for smoothing mobile robot trajectories. *IEEE Transactions on Robotics and Automation*, 11(3):441–448, 1995.

[15] Jean-Daniel Boissonnat, André Cérézo, Elena V. Degtiariova-Kostova, Vladimir P. Kostov, and Juliette Leblond. Shortest plane paths with bounded

derivative of the curvature. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 329(7):613–618, 1999.

[16] T. Fraichard and A. Scheuer. From reeds and shepp's to continuous-curvature paths. *IEEE Transactions on Robotics*, 20(6):1025–1035, 2004.

[17] Mišel Brezak and Ivan Petrović. Real-time approximation of clothoids with bounded error for path planning applications. *IEEE Transactions on Robotics*, 30(2):507–515, 2014.

[18] Yong Chen, Yiyu Cai, Jianmin Zheng, and Daniel Thalmann. Accurate and efficient approximation of clothoids using bézier curves for path planning. *IEEE Transactions on Robotics*, 33(5):1242–1247, 2017.

[19] AW Nutbourne, PM McLellan, and RML Kensit. Curvature profiles for plane curves. *Computer-aided design*, 4(4):176–184, 1972.

[20] B.A. Barsky and T.D. DeRose. Geometric continuity of parametric curves: three equivalent characterizations. *IEEE Computer Graphics and Applications*, 9(6):60–69, 1989.

[21] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[22] Nicolas Montes, Alvaro Herraez, Leopoldo Armesto, and Josep Tornero. Real-time clothoid approximation by rational bezier curves. In *2008 IEEE International Conference on Robotics and Automation*, pages 2246–2251, 2008.

[23] Johannes Schmitz and phillipd94. pyclothoids. `https://github.com/phillipd94/pyclothoids`, 2020. [Online; accessed 2022].

[24] Enrico Bertolazzi and Marco Frego. G1 fitting with clothoids. *Mathematical Methods in the Applied Sciences*, 38(5):881–897, 2015.

[25] Enrico Bertolazzi and Marco Frego. On the g2 hermite interpolation problem with clothoids. *Journal of Computational and Applied Mathematics*, 341:99–116, 2018.

[26] Enrico Bertolazzi and Marco Frego. Interpolating clothoid splines with curvature continuity. *Mathematical Methods in the Applied Sciences*, 41(4):1723–1737, 2018.

[27] Marco Frego and Enrico Bertolazzi. Point-clothoid distance and projection computation. *SIAM Journal on Scientific Computing*, 41(5):A3326–A3353, 2019.

[28] Ravi Kumar Satzoda and Mohan M. Trivedi. On performance evaluation metrics for lane estimation. In *2014 22nd International Conference on Pattern Recognition*, pages 2625–2630, 2014.

[29] Shinko Yuanhsien Cheng and Mohan Manubhai Trivedi. Lane tracking with omnidirectional cameras: Algorithms and evaluation. *EURASIP Journal on Embedded Systems*, 2007:1–8, 2007.

[30] R. K. Satzoda and Mohan M. Trivedi. Selective salient feature based lane analysis. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 1906–1911, 2013.

[31] Youcheng Zhang, Zongqing Lu, Xuechen Zhang, Jing-Hao Xue, and Qingmin Liao. Deep learning in lane marking detection: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(7):5976–5992, 2022.

[32] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8:679 – 698, 12 1986.

[33] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation, 2017.

[34] Yann Lecun, Patrick Haffner, and Y. Bengio. Object recognition with gradient-based learning. 08 2000.

[35] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[37] Utku Ozbulak. Pytorch cnn visualizations. `https://github.com/utkuozbulak/pytorch-cnn-visualizations`, 2019.