HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Autonomous Racing with Deep Learning

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Philipp Badenhoop
geboren am: 7.7.1996
geboren in: Berlin Neukölln

Gutachter/innen: Prof. Dr. Holger Schlingloff
Prof. Dr. Verena Hafner

eingereicht am: .... verteidigt am: ....

Abstract

This thesis covers the implementation of an autonomous racing car in scale 1:8 using deep learning. Based on the current camera frame, our method employs a neural network which predicts the vehicle's future path. The network is trained in a supervised fashion using data which is recorded while a human drives the car manually around the racing track. We propose a simple model to control the steering angle and speed using the network's predictions. Furthermore, we show that our method is both easier to train and to scale up to higher velocities than the well known end-to-end approach which uses a neural network to output the steering commands directly. With our autonomous racing car, we were able to win the third place in the Deep Berlin Robocars Challenge.

Contents

1. Introduction

The development of fully self-driving cars is currently one of the most exciting and demanding challenges in modern vehicle industry. Traveling on the road should become safer and more comfortable in the absence of the human driver. However, teaching a computer how to navigate the vehicle in complex areas such as cities appears to be a difficult problem. Therefore, it makes sense to tackle autonomous racing first since one can focus on fewer and more simplified aspects.

The goal of this thesis is to implement an autonomous racing system in scale 1:8 which is able to compete at the Deep Berlin Robocars Challenge on 12 September 2018 [14]. During the competition, each team let their car drive 10 rounds around the track. Only the fastest lap time counted and was compared across the teams. There weren't any obstacles or other racing cars on the track during driving. When all four tires left the track which was delimited by white lane markings, a two seconds penalty was added to the lap. Obviously, this gave some tolerance which could be exploited in terms of driving behavior. Over the last couple of months before the challenge, the teams were able to visit the final racing track once a week to collect data and tweak their algorithms.

Our car has been originally designed for the semester project "Hochautomatisiertes Fahren" [4] to develop an autonomous platooning system. However, that system was neither designed for racing nor for driving on the final track at all. In terms of sensoring, the car was equipped with a single monocular camera, a wheel encoder on each rear tire as well as an ultrasonic sensor which was not actually used for racing.

Based on the camera's current frame, our autonomous racing algorithm uses a neural network that predicts a trajectory which the car should follow. Using these trajectories, we can employ a driving model to control the steering angle and speed. This method differs from the traditional deep learning end-to-end approach which uses a neural network to predict the steering angle from the image directly. Our neural network tries to solve the problem of *where* the car should drive to instead of *how* it should drive. Therefore, we gain more control over the system. For instance, our car is able to recognize straight lines and turns to accelerate and brake respectively. With this approach we were able to win the third place out of 12 teams in the Deep Berlin Robocars Challenge.

1.1. Related Work

Autonomous racing is quite a young sports and research topic. Popular international competitions such as *Roborace* [32] [9] and *Self Racing Cars* [33] [26] exist only since 2016. The participating teams build full-sized autonomous vehicles which drive on a real racing track. In [13], de la Iglesia Valls et al. introduce the design of a driverless racing car which won the *Formula Student Driverless* competition in 2017. The vehicle was equipped with a variety of sensors including LIDAR, IMU, GPS and a stereo camera to perform *simultaneous localization and mapping* (SLAM). They employed the LIDAR to detect cones which were delimiting the track. Therefore, they didn't rely on visual information making it inapplicable to our use case where there only exists lane markings to define the track. Unfortunately, besides a few occasional blog posts, there are hardly any further scientific publications about other competitions yet.

Actually, most scientific work in the field of autonomous racing is primarily done either on miniature cars or in simulations. In [25], fairly complex control techniques are presented to show how to drive a 1:43 scale car at its physical limits. Rosolia et al. employ *model predictive control* to reach the same goal but in a simulated environment [30]. However, both approaches assume that the vehicle is able to localize itself on the track and do not focus on the visual aspects.

The rise of deep learning and convolutional neural networks revolutionized computer vision [15] and is excessively driving the development of autonomous vehicles (not specifically racing). In [7], *Nvidia* introduced a neural network architecture called *PilotNet* which acts as an end-to-end lane keeping assistent. The network takes an image as input and outputs a steering angle which is directly controlling the car. Ever since, several other advanced architectures have evolved using recurrent neural networks and residual connections for example [36].

There also exist some research on how to apply reinforcement learning to autonomous driving in simulations [31]. However, this method hasn't been successfully applied to the real world yet.

Very exiting work in autonomous racing including deep learning for visual perception has been done in [30] which uses a convolutional neural network to create a cost function for a model predictive control algorithm. It also employs a core idea of our trajectory prediction approach that is to use deep learning as a vision system and to split perception and control at the same time.

1.2. Structure of the Thesis

This thesis is structured as follows. Section 2 provides an overview on the car's hardware, chassis and sensors as well as the system's software architecture. In Section 3 we present an introduction to deep learning. Section 4 is devoted to the implementation of the autonomous racing algorithm. In Section 5 we describe our results and experiments with the system. Finally, Section 6 concludes the thesis by summarizing what has been done.

2. System Architecture

In this Section we give an overview of our racing car by presenting its chassis, sensors and hardware components. Furthermore, we provide the software architecture which outlines how to get from sensor data through steering commands to the final actuations on the mechanics.

2.1. Chassis

Since we adopted the car used in the semester project “Hochautomatisiertes Fahren”, there haven’t been many decisions regarding the chassis and mechanics of the system. The Losi Horizon Hobby 8IGHT-E 1/8-scale 4WD RTR Buggy serves as the car’s foundation. It includes a differential gear connecting the powerful 2500 kV motor to all four tires. The wheels are covered with rubber textures providing plenty of grip on the racing track. Since the track’s material was rubber too, there haven’t been any issues regarding slippage even under competitive speeds. Furthermore, a suspension system is build into the car, yielding a stable driving behavior in tight turns.

Figure 1: Photo of the racing car.

The acceleration of our motor turned out to be quite advantageous. In fact it’s specified to drive up to 50 mph. Although the top speed is far from reachable on such a small and curvy racing track, we definitely tried to exploit the car’s capabilities.

A servo is used to turn the front wheels, providing a maximum steering angle of

30° to each side. However, our initial servo appeared to be seriously slow. It took approximately half a second to turn the wheels from full left to full right and vice versa. This turned out to be a crucial problem and lead to certain design decisions in the racing algorithm which are covered in more detail later on. Most of the competitors used the Donkey Car [3] which is roughly half the size of our model. They come with smaller wheels and weigh less so their servos act almost instantly. Fortunately, by replacing the servos, we could improve the steering delay a bit.

In terms of powering the car, we make use of two 7.4 V lipo batteries, each having a capacity of 4500 mA h. One battery is powering the motor exclusively whereas the other one powers the rest of the hardware.

Altogther, one can say that our car provided the most potential in terms of motor power and stability. However, compared to the majority of the competition, the vehicle was rather big and heavy, resulting in less tolerance of staying in the track during the turns.

2.2. Hardware

To compute the high level logic which outputs the steering commands, an Odroid XU4 is used running Ubuntu 16.04. The Odroid is preferred over the popular Raspberry Pi due to its superior processor. Its recommended operation voltage is 5 V and under full load its power consumption rises up to 5 A. Since our batteries output 7.4 V we installed a DC to DC step down converter. However, the board seems to be really power demanding, so we had to increase the output voltage of the converter to 5.7 V. Anything below that would lead the system to reboot when running the racing algorithm.

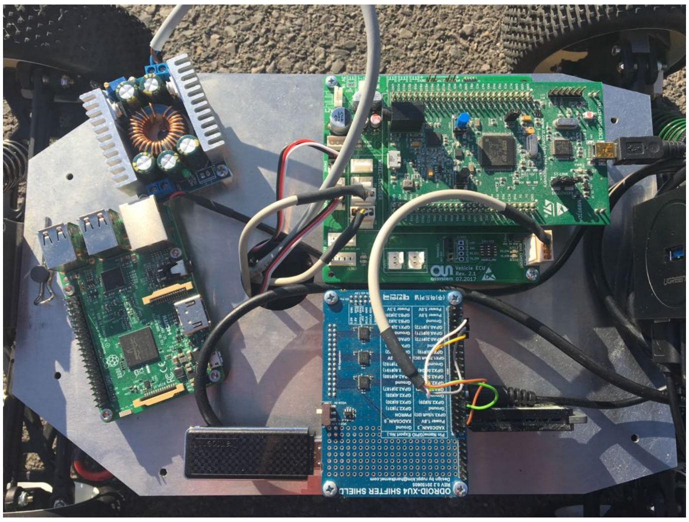


Figure 2: Photo of the car's hardware.

The STM32 microcontroller is responsible for running the low level realtime logic. This includes handling the servo's and motor's PWM, getting data from the wheel encoders as well as implementing a PID controller which controls the desired speed. A UART connection is employed to communicate with the Odroid using a Mavlink protocol. To be able to connect each component with the microcontroller, the STM32 is plugged into a breakout board made by *Assystem GmbH*.

2.3. Camera

Our car is equipped with a DFM 22BUC03-ML camera [2] from *The Imaging Source*. It is capable of producing frames at a resolution of 744×480 pixel at 76 frames per second which is a much higher resolution than actually required in our case since we cropped and rescaled the image as described in more detail later on.

An important aspect proved to be the camera's mounting position. Initially, it was mounted at the front of the car, a couple centimeters above the wheels. We ended up positioning the camera at the rear, roughly 40 cm above the ground. Consequently, it gained a much better overview of the track making the predictions of the neural network more reliable.

2.4. Wheel Encoders

To be able to measure the car's motion regarding its speed and position, there are two wheel encoders, one on each rear tire. These wheel encoders consist of two components:

- A dice containing equally spaced wholes which is connected to its corresponding shaft.
- An optical sensor which emits an electrical impuls whenever a whole of the dice is passing through it. These impulses are referred to as *ticks* and trigger an interrupt in software which increments a simple counter variable.

In Section 4.2.2 we explain how this data is used to compute a path, showing where the vehicle drives along in metric coordinates which is called *odometry*.

2.5. Software Architecture

In this Section we explain how the aforementioned parts connect through software.

Our system is composed of three software components which are shown in Figure 4. Since we use a deep learning approach in our racing algorithm, our system has to perform two use cases:

1. Recording training data while being controlled remotely.
2. Driving autonomously.

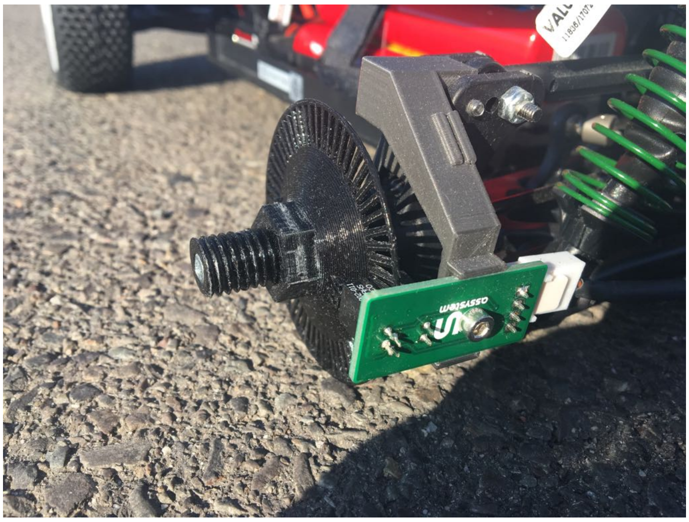


Figure 3: Photo of the rear left wheel encoder.

Therefore, we consider the car being either in *recording mode* or in *autonomous mode*. In recording mode, a human is able to drive the car remotely. During this mode, camera images and data from the wheel encoders are written to an external storage device. A game controller is plugged into a Laptop which runs a program sending UDP packets to the odroid containing the steering commands. This way, we are able to use a wired controller which we did initially. One may also choose to connect the sensor of a wireless controller directly to the odroid. However, the range of those devices may be quite weak, so one should definitely follow the car while driving remotely on larger tracks.

2.5.1. Odroid

To create the odroid's software component, our design is based on the *robot operating system* (ROS) [29] which is a popular collection of libraries and tools to build robot applications. One of its main concepts constitutes the messaging system. In ROS, messages are transported between *nodes* either over *topics* or *services*. Nodes are processes running in the OS just like any other application. This way, developers are able to write nodes in different programming languages. However, this implies that some overhead is involved in sharing data between nodes due to inter-process communication. Fortunately, ROS provides the *nodelet plugin* for C++. Instead of creating a process for each individual C++ node, only a single process is launched which is called the *nodelet manager*. The nodelet manager handles the execution and message transportation of each node it is responsible for in its own process.

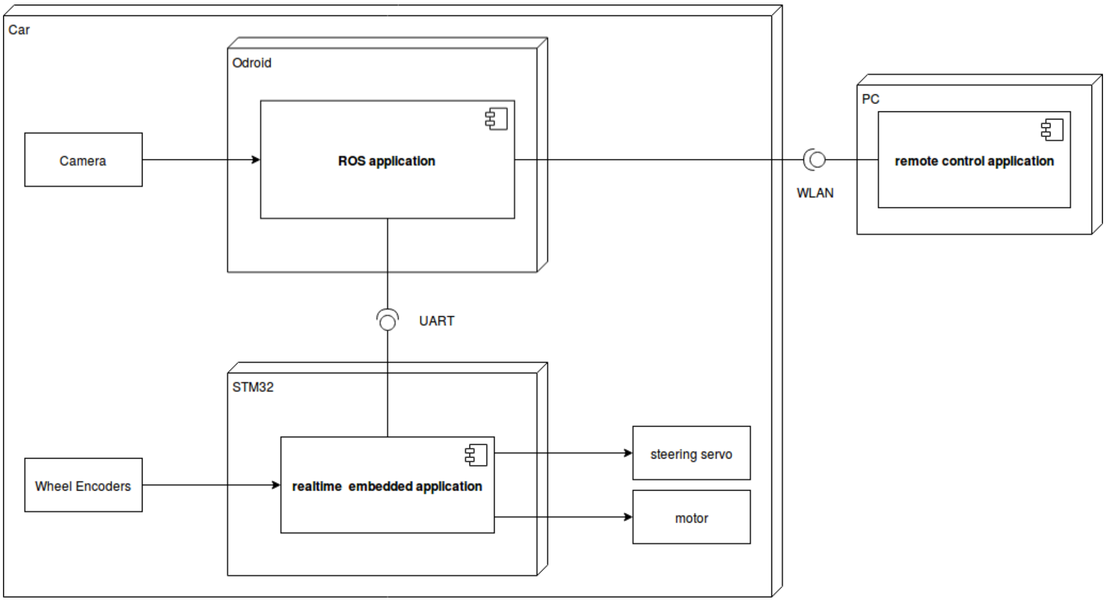


Figure 4: Component diagram of the system.

Messages and Topics

A node can publish messages to topics which other nodes can subscribe to. To avoid confusion, we point out that for each message in our particular system, there exists an equally named topic which the message is published to. For example, if there's a message *SetAngle*, then this message is transmitted over the topic */SetAngle*. These are the messages used in the system:

- *RemoteAngle* contains the steering angle transmitted over the network.
- *RemoteThrottle* contains the throttle value transmitted over the network.
- *RemoteState* encodes button presses so we can use the game controller to stop and continue recording.
- *SetAngle* contains the steering angle transmitted to the STM32 which should be applied to the servo.
- *SetThrottle* is interpreted as the speed the car should currently maintain and is transmitted to the STM32.
- *WheelTicks* provides an update from the STM32 containing the ticks for the left and right wheel encoders.

Note that there's no message containing camera information. We choose to read the camera images directly in the node where they are used. Thus, we avoid the overhead of serializing and deserializing the image data for transmission over a specific topic.

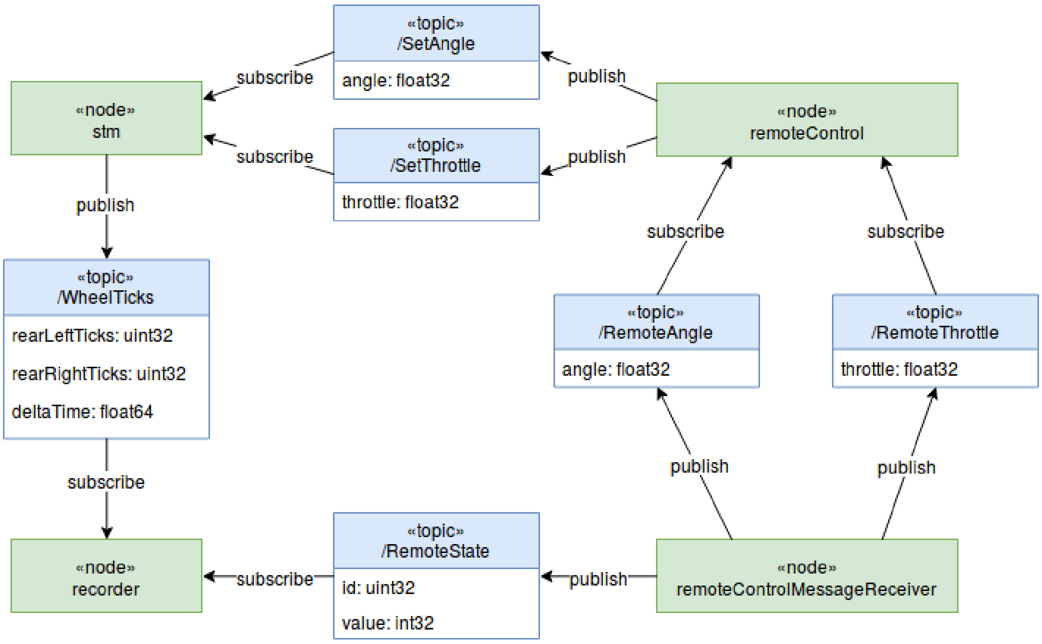
Nodes

In the following, we provide an overview of the nodes employed in the system.

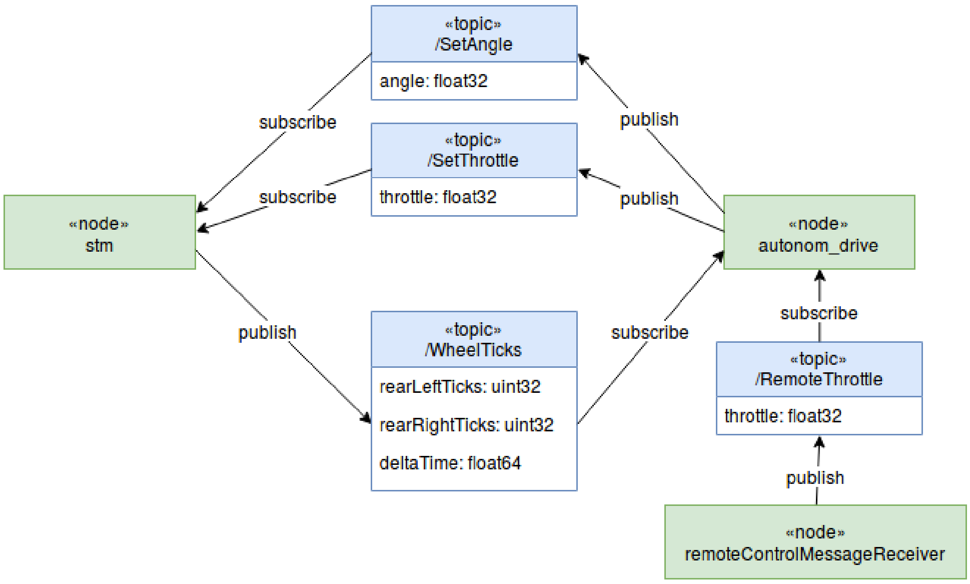
- *remoteControlMessageReceiver* receives UDP packets sent from the PC and publishes the contained information to the /RemoteAngle, /RemoteThrottle and /RemoteState topics respectively.
- *remoteControl* subscribes to /RemoteAngle and /RemoteThrottle and forwards that information by publishing it to the /SetAngle and /SetThrottle topics.
- *stm* subscribes to /SetAngle and /SetThrottle and transmits the data to the STM32 by handling the Mavlink communication over the UART. It also publishes the incoming wheel encoder updates to /WheelTicks.
- *recorder* is responsible for storing the training data which incorporates reading images from the camera as well as subscribing the /WheelTicks topic.
- *autonom_drive* runs the racing algorithm which publishes to /SetAngle and /SetThrottle. It subscribes the /WheelTicks and the /RemoteThrottle topic in order to still be able to get control over the vehicle's speed despite being in autonomous mode.

The ROS architectures for recording and autonomous modes are depicted in Figure 5(a) and Figure 5(b) respectively [1].

[1] Note that we don't provide any further lower level UML diagrams since we don't want to focus on the specific implementation details in this work.



(a) Active nodes and topics in recording mode.



(b) Active nodes and topics in autonomous mode.

Figure 5: ROS architectures.

2.5.2. STM32

Since we adopted the car from the semester project and there was already everything implemented that we needed, we left most parts of the STM32 unchanged. Its embedded architecture is based on a scheduler which periodically runs a set of tasks. A task is nothing more than a piece of C code. Global variables are used to share data between different tasks. In the following, we provide a list of the most important ones [2]:

- *mavlink* is responsible to handle the Mavlink protocol using the UART.
- *vehspdctrl* implements a PID controller to control the target speed and updates the motor's PWM module accordingly.
- *stangproc* updates the servo's PWM module based on the target steering angle.
- *eict* configures and reads input capture timers to receive the ticks from the wheel encoders.

[2] Note that the poor naming stems from the fact that most of the STM software was left unchanged from the semester project.

3. Deep Learning

This Section provides an introduction into deep learning which is the core technology used in this work to solve the autonomous racing problem.

3.1. Motivation

Machine learning has gained rapid success in recent years. The era of deep learning began with *AlexNet* [21], a Deep Convolutional Neural Network which considerably outperformed its competition in the standard image classification challenge ImageNet. In the following six years, this technology has been applied to a wide variety of other problems such as natural language processing [12], handwriting recognition [40] and cancer detection [5]. However, these networks already existed in the 1980s [16] and were only able to succeed due to huge advances in high throughput computing.

History showed that problems which humans perform well at are typically hard to solve algorithmically by writing traditional, imperative programs. The classification of images is known for being such a phenomenon. Obviously, humans are able to instantly recognize objects like trees, cats and cars on an image. However there doesn't seem to exist any explicit rules how we actually perform this task. Therefore, new systems have been developed that are able to learn how to solve these problem by themselves, given some example data.

In machine learning, the most well researched learning paradigm is known as *supervised learning*. On an abstract level, a supervised learning model learns a function by fitting it with examples of input-output mappings. Taking the image classification as an example, the model gets a vector of pixels as input and outputs a binary vector specifying which classes it detected. The model is trained with large sets of image-to-class-label pairs. If trained properly, it is able to learn the specific features of each class in order to *generalize* to new data making it perform well on images it has never seen before.

The most sophisticated deep learning models for supervised learning are *artificial neural networks* (ANNs).

3.2. Artificial Neural Networks

Despite their name, modern ANNs don't really aim to model their biological counterpart which they were inspired from [15]. An ANN can be described as a graph consisting of nodes which are connected through edges. A node gets one or multiple signals from its input edges, computes a weighted sum of its inputs, applies a threshold function and outputs a new signal through its output edges. ANNs are structured in consecutive *layers*, consisting of one input layer followed by at least one hidden layer and finally the output layer. This structure is also known as the *multilayer perceptron* [11]. An example ANN is depicted in Figure 6. Each layer computes a function which can be

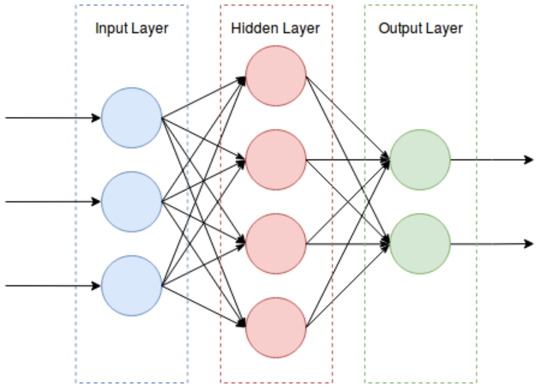


Figure 6: Visual representation of a basic multilayer perceptron consisting of a single hidden layer. Each node passes its output as weightened input to every node in the next layer.

represented as

$$f(x) = \sigma(Wx + b) \tag{1}$$

where

- x is the input vector of dimension m,
- W is the $n \times m$ *weight matrix* where n is the number of nodes in the layer,
- b is the *bias* vector of dimension n and
- σ is the *activation function*.

The output of f is a n-dimensional vector which expresses the *activation* of each node in the layer. Each activation is given as input to every node in the next layer.

As one can see, the input edges are weighted. Therefore, an edge can be of high significance if its weight is large enough. On the other hand, setting it to zero prevents information to pass through the next layer. The original intuition behind the activation function was to define for which summed input the neuron actually *fires*, which means that its output is greater than zero. For example, using a neuron as a binary classifier, one may take the binary activation function

$$\sigma_i(x) = \begin{cases} 1 & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases} \quad \forall i \in \{1, ..., n\} \tag{2}$$

so the i-th neuron becomes active as long as the term $(Wx + b)_i$ is greater than zero [3]. The bias can be viewed as a threshold of the weighted sum of inputs Wx which has to be reached so that the activation function lets the neuron fire.

[3] In fact, activation functions act always elementwise with the same operation for all i so one often omits the indexed notation.

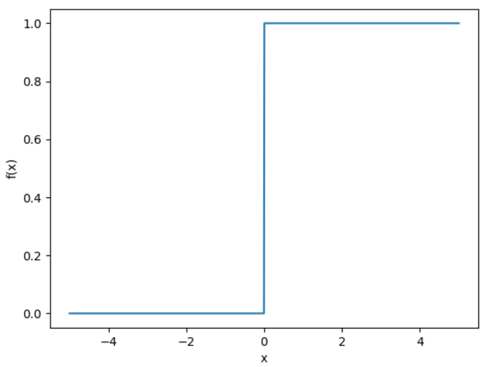
Another purpose of the activation function is to introduce nonlinearity into the system. Let $\sigma(x) = x$ be the identity function. Now, consider a three layer model

$$\begin{aligned} y &= W_1 x + b_1 \\ z &= W_2 y + b_2 \end{aligned} \tag{3}$$

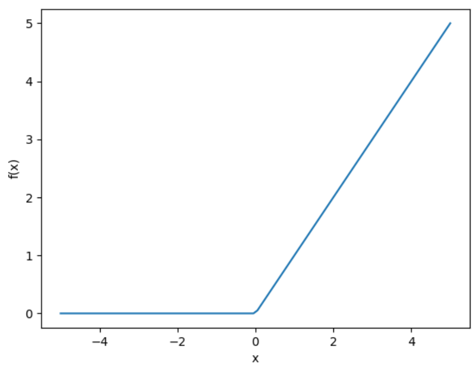
where x denotes the input layer, y the activations of the hidden layer and z the activations of the output layer. Thus, the input and hidden layer are connected through weights W_1 and the hidden and output layer are connected through weights W_2. We can observe that the third layer is actually redundant since two composed affine transformations can be expressed as a single affine transformation:

$$W_2(W_1 x + b_1) + b_2 = W_2 W_1 x + (W_2 b_1 + b_2). \tag{4}$$

In practice, we want our models to approximate highly non-linear functions and therefore, a non-linear activation function should be applied to our weighted sum. A very common choice is the ReLU function [34] depicted in Figure 7(b). There are a couple of desired properties the activation function may fullfill however, those can only be motivated by understanding how an ANN learns.



(a) The binary function.



(b) The ReLU function: $\sigma(x) = max(0, x)$.

Figure 7: Different activation functions.

3.3. Learning

Solving a supervised learning problem involves the following steps [22].

1. Collect example data and partition it into training, validation and test data. There is no general way to measure the quality of the data set so this always depends on the specific problem and has to be figured out by exploration.
2. Create (or refine) a model.
3. *Training step.* Optimize the model's parameters to perform well on the training data.
4. *Validation step.* Check how the model behaves on the validation data. If the model doesn't perform well, go back to step 2. Otherwise, continue with step 5.
5. *Testing step.* Once the model works reasonably on both training and validation data, we run it on the final test data right before deployment to see whether it is able to perform well in real life applications. It is crucial that this data is kept isolated during the previous steps since any further adaption to the model prevents us from objectively telling how the model generalizes to data it has never seen before.

When building neural network models, one has to deal with a wide variety of parameters. For structure, we partition them into two types:

a) *Learnable parameters.* Parameters which are set during the training step. In neural networks, these are the weights and biases.

b) *Non-learnable parameters* or *hyperparameters.* One may group these ones even further into:

   – *Topological parameters.* Regard the number and types of layers, the number of nodes on each layer as well as the choice of activation functions, to name a few.

   – *Training parameters.* Correspond to specific parameters of the training algorithm, the loss function and the method of weight initialization for example.

The difference between these types is that non-learnable parameters are selected by the human developer (through exploration and repetition of step 2) whereas learnable parameters are discovered by a training algorithm. For that, we need a metric which represents how well a model predicts on a given input and a *ground truth* output (or simply *label*). In literature, this metric is known as the *loss function.*

3.3.1. Loss Function

One may think of training or leanring as an optimization problem. Given n samples of training data as inputs X_i and their corresponding labels Y_i for $1 \leq i \leq n$ as well as a model $\Phi(x, \Theta)$ where x is the model's input and Θ are the learnable parameters, our goal is to find Θ_{opt} to minimize the error $\mathcal{L}$ between the model's predicted output and the labels

$$\Theta_{opt} = \arg\min_{\Theta} \mathcal{L}_{\Phi,X,Y}(\Theta) \tag{5}$$

with

$$\mathcal{L}_{\Phi,X,Y}(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{E}(\Phi(X_i, \Theta), Y_i). \tag{6}$$

Here, $\mathcal{E}(\cdot, \cdot)$ measures the error between the model's prediction $\Phi(X_i, \Theta)$ and a label Y_i so we call it the *output error measure*. $\mathcal{L}$ is the average error over n training examples X and Y and is referred to as the *loss function* or simply the *loss* [22].

To provide a more concrete understanding, we show how this could be applied to autonomous driving using basic ANN models. A simple approach would be to create an artificial neural network model which takes the current camera image as input and directly outputs the target steering angle. Since there's no other processing involved between the sensory data and the steering commands, this is known as the *end-to-end* approach [7]. Data is collected by a human driving the car while recording the current steering angle for each new frame. The objective is that the network should mimic the human behavior. Therefore, this technique is referred to as *behavioral cloning*.

First, we have to define the neural network model. Its input layer takes a vector representation of an image which is done by stacking all the pixel values. The output layer is simply a single node which gives the steering angle. Determining the network architecture and the non-learnable parameters is done by many iterations of the training and validation steps.

In order to train the model, we have to define an appropriate output error measure. Optimally, our network always predicts the exact same steering angle that the human driver had entered based on the collected data. Given a model's predicted steering angle y and a ground truth steering angle $\hat{y}$, the error measure could be as simple as the squared distance

$$\mathcal{E}(y, \hat{y}) = (y - \hat{y})^2. \tag{7}$$

By squaring the difference between y and $\hat{y}$, we remove the sign and increase the penalty on larger errors. A widely used generalization to n-dimensional outputs is known as the *mean squared error* (MSE) defined as

$$MSE = \mathcal{E}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{8}$$

There exist several other output error measures which are mainly divided into those that are suitable for regression problems (e.g. MSE or *mean absolute error* (MAE)) and the ones that are intended to be used for classification problems (e.g. *cross entropy* or *hinge*) [22].

Since we want to minimize the model's loss, we have to determine the appropriate learnable parameters Θ_{opt} as depicted in Equation 5. However, neural networks are extremely complex functions due to their possibly huge numbers of layers, neurons, weights and other parameters. Therefore, it's not feasable to get Θ_{opt} by trying to find an analytical solution. Thus, we need an optimization algorithm to compute the learnable parameters which minimize the loss function.

3.3.2. Optimization

To train our neural network, the first-order iterative optimization algorithm *gradient descent* [42] is used to find the appropriate learnable parameters Θ_{opt}. The *gradient* of a multi-variable function $f(x_1, ..., x_n)$ is defined as

$$grad(f) = \nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{9}$$

where $\frac{\partial f}{\partial x_i}$ is the partial derivative of f with respect to x_i. The important property is that for any given point x, the gradient $\nabla f(x)$ points to the direction of the greatest rate of increase of f. Likewise, $-\nabla f(x)$ points to the direction of the greatest rate of decrease of f.

Gradient descent uses this observation to find a local minimum. Starting at some initial point x_0, we iteratively perform the following updates

$$x_{n+1} = x_n - \gamma \nabla f(x_n) \tag{10}$$

until the sequence (x_n) converges at a local minimum. However, to fullfill the desired condition $f(x_0) \geq f(x_1) \geq f(x_2) \geq ...$ the update rate $\gamma \in R_+$ has to be chosen small enough.

In the following, we want to show how gradient descent is used to minimize the loss function $\mathcal{L}(\Theta)$. In practice, Θ is going to be high dimensional. For simplicity, let's assume a model consisting of only two learnable parameters $\Theta = (\theta_0, \theta_1)$ which we further assume to be both weights. Now, we can visualize the loss function as a

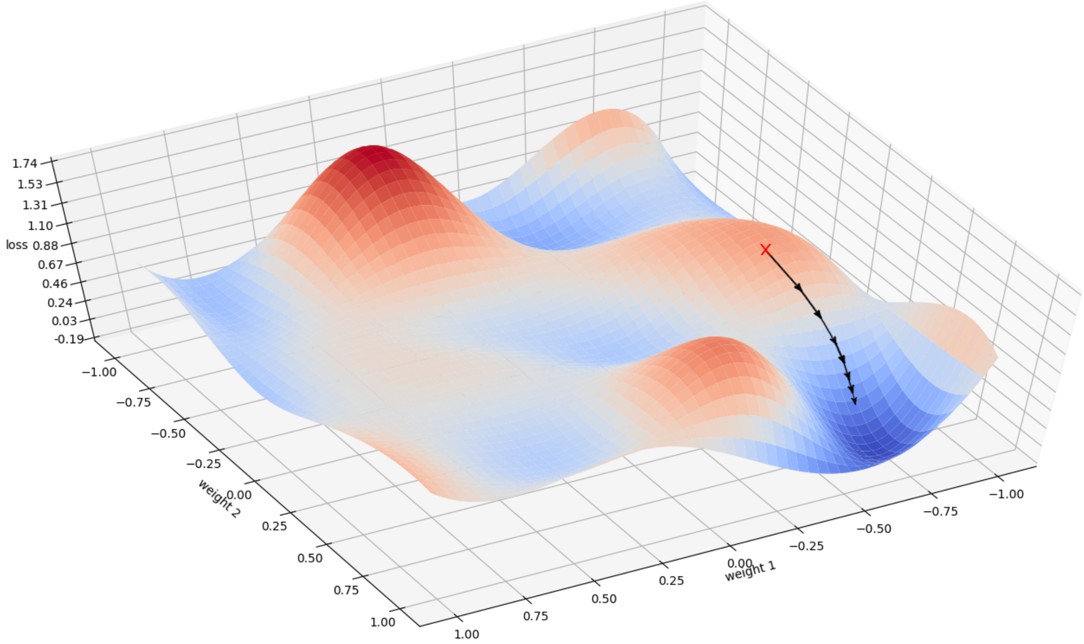


Figure 8: Visualization of a fictional error surface. The model contains only two weights as learnable parameters. Given some inital values for weight 1 and 2 (red cross), gradient descent is used to walk down to the nearest valley representing a local minimum in loss.

landscape where each configuration of θ_0 and θ_1 results in a position with height $\mathcal{L}(\Theta)$ as depicted in Figure 8. By initializing these weights randomly, we could potentially land on a peak of the error surface (red cross in Figure 8). To get the minimal local error, we use gradient descent to iteratively update the weights to walk down to a surrounding valley.

It should be noted that this method does not guarantee to find the *global* minimum. Hence, different weight initializations may lead to different results.

What's left is to determine the partial derivatives of $\mathcal{L}$ with respect to each θ_i to compute the gradient $\nabla\mathcal{L}$. We can then simply update the learnable parameters using Equation 11

$$\Theta_{n+1} = \Theta_n - \gamma \nabla \mathcal{L}_{\Phi,X,Y}(\Theta_n) \tag{11}$$

until (Θ_n) hopefully converges at a local minimum. γ is referred to as the *learning rate* which is an important hyperparameter. The partial derivatives with respect to each learnable parameter can be computed using the backpropagation algortihm [41] (see Section 3.3.3).

When computing the loss, it matters to choose an appropriate quantity of training samples, given as X and Y. Let's assume to select the entire training set to calculate the loss over. Consequently, we have to evaluate the model's prediction for each sample in the possibly huge training set in order to perform just a single update to the weights and biases. However, in practice we need to perform multiple updates since we do not know how far to walk into the direction of the gradient to arrive at the local minimim (see Figure 8 again). Since computing the loss function over the entire training set is usually very expensive, instead we can randomly subsample the data into smaller chunks called *mini-batches*. These mini-batches serve as an approximation of the full training set when used to compute the model's loss. The corresponding method is then known as *mini-batch stochastic gradient descent*.

To summarize the optimization procedure, Figure 9 gives high-level pseudocode of a mini-batch stochastic gradient descent algorithm used for training a neural network. The three most important steps are as follows:

1. *Forward pass* (line 4). We compute the model's predictions by processing the input X^{batch}. Then we can calculate the error between the network's output and the labels Y^{batch}. During the forward pass, we keep certain data in memory that is needed in order to perform the backward pass. For illustration, this data is returned as the function's result.
2. *Backward pass* (line 5). Based on the result of the forward pass, this function computes the gradient using backpropagation.
3. *Update* (lines 6 and 7). Finally updates the weights and biases according to Equation 11.

Algorithmus 1 : Mini-batch stochastic gradient descent algorithm

Input : $num\_samples$ training inputs X.
Input : $num\_samples$ training labels Y.
Input : $batch\_size$.
Input : Neural network model Φ with weights $\mathcal{W}$ and biases $\mathcal{B}$.
Input : Output error measure $\mathcal{E}$.
Input : Learning rate γ.
Output : Optimized weights $\mathcal{W}$ and biases $\mathcal{B}$.

$num\_batches \leftarrow \lfloor num\_samples/batch\_size \rfloor$
for $i \leftarrow 1, ..., num\_batches$ **do**
    `//` Retreive batches X^{batch} and Y^{batch} by randomly sampling $batch\_size$ samples from X and Y respectively.
    $X^{batch}, Y^{batch} \leftarrow sample\_batch(X, Y, batch\_size)$
    `//` Forward pass: Compute the model's predictions and the resulting error across the current batch.
    $result = forward\_pass(\Phi, \mathcal{W}, \mathcal{B}, \mathcal{E}, X^{batch}, Y^{batch})$
    `//` Backward pass: Use the result from the forward pass to compute the gradient vectors for $\mathcal{W}$ and $\mathcal{B}$.
    $\Delta\mathcal{W}, \Delta\mathcal{B} \leftarrow backward\_pass(result)$
    `//` Update: Use gradient descent to update the weights and biases.
    $\mathcal{W} \leftarrow \mathcal{W} - \gamma\Delta\mathcal{W}$
    $\mathcal{B} \leftarrow \mathcal{B} - \gamma\Delta\mathcal{B}$
    `//` Remove the samples used in the current batch.
    $X \leftarrow X - X^{batch}$
    $Y \leftarrow Y - Y^{batch}$
end
return ($\mathcal{W}$, $\mathcal{B}$)

Figure 9: High-level pseudocode showing the mini-batch stochastic gradient descent algorithm.

3.3.3. Backpropagation

In order to optimize the learnable parameters in a neural network, we still need a way to compute the gradient of the loss function. In other words, we have to get the partial derivatives of $\mathcal{L}$ with respect to each learnable parameter in $\Theta = (\theta_1, ..., \theta_m)$

$$\nabla \mathcal{L}_{\Phi,X,Y}(\Theta) = \begin{pmatrix} \frac{\partial \mathcal{L}_{\Phi,X,Y}}{\partial \theta_1} \\ \vdots \\ \frac{\partial \mathcal{L}_{\Phi,X,Y}}{\partial \theta_m} \end{pmatrix}. \tag{12}$$

The *backpropagation* algorithm allows us to evaluate these derivatives efficiently on arbitrarily complex neural networks.

First, we need to transform our model into a *computational graph*. Computational graphs represent functions where nodes are either input parameters or operations. Edges indicate the data flow. An example is depicted in Figure 10(a) which computes the function $f(x, w, b) = (wx + b)^2$. We could imagine that f represents a node in a neural network with a single input x and its corresponding weight w, bias b as well as the activation $\sigma(x) = x^2$. Note that we label each intermediate result of the operations, that is

$$p = wx \qquad\qquad q = p + b \qquad\qquad r = q^2.$$

Given some concrete input values to f, backpropagation computes the partial derivatives of the final output r with respect to each input parameter w, x and b which are $\frac{\partial r}{\partial w}$, $\frac{\partial r}{\partial x}$ and $\frac{\partial r}{\partial b}$. To refer to the neural network analogy again, the concrete inputs would be training input data x and randomly initialized weight w and bias b. In reality, one would actually need a fourth parameter y representing the label and some extra nodes in the graph in order to compute the error between the network's output and the label. However, we left that out for simplicity.

The algorithm consists of two steps, a *forward pass* (see Figure 10(b)) and a *backward pass* (see Figure 10(c)). During the forward pass, we evaluate each node's operation by forwarding the results through the graph in topological order. By doing this, we are able to compute the *local gradients* which are the partial derivatives of each node's output with respect to its own inputs. In our example, we start by calculating the local gradient for node p. It's inputs are $w = 2$ and $x = 4$ so we want to get $\frac{\partial p}{\partial w}$ and $\frac{\partial p}{\partial x}$

$$\frac{\partial p}{\partial w} = \frac{\partial wx}{\partial w} = x = 4 \qquad\qquad \frac{\partial p}{\partial x} = \frac{\partial wx}{\partial x} = w = 2.$$

Next, we propagate p's output to the input of q. The partial derivatives of node q with respect to its inputs $p = 8$ and $b = 3$ are given as

$$\frac{\partial q}{\partial p} = \frac{\partial (p+b)}{\partial p} = 1 \qquad \frac{\partial q}{\partial b} = \frac{\partial (p+b)}{\partial b} = 1.$$

Finally, we pass $q = 11$ to r and get $r = 121$ as well as

$$\frac{\partial r}{\partial q} = \frac{\partial q^2}{\partial q} = 2q = 22.$$

Using the results from the forward pass, the backward pass traverses the graph in reverse topological order and consecutively computes the partial derivatives of the graph's output with respect to each node's input. In other words, it measures the influence each node has on the final output (including the input nodes). Therefore, backpropagation heavily exploits the chain rule

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}. \tag{13}$$

For example, if we want to compute $\frac{\partial r}{\partial p}$, we can also write

$$\frac{\partial r}{\partial p} = \frac{\partial r}{\partial q} \frac{\partial q}{\partial p} = 22 \cdot 1 = 22.$$

Note that in this equation, $\frac{\partial q}{\partial p}$ is part of the local gradient of q which we already evaluated in the forward pass. On the other side, the first factor $\frac{\partial r}{\partial q}$ is the *backpropagated result* from q's output node which is node r. Having determined $\frac{\partial r}{\partial p}$, we backpropagate this result to node p to compute $\frac{\partial r}{\partial w}$ and $\frac{\partial r}{\partial x}$

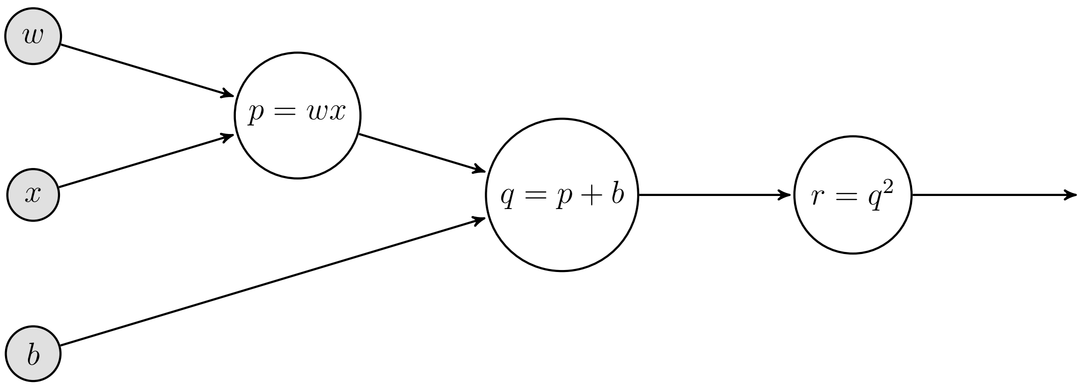
$$\frac{\partial r}{\partial w} = \frac{\partial r}{\partial p} \frac{\partial p}{\partial w} = 22 \cdot 4 = 88 \qquad \frac{\partial r}{\partial x} = \frac{\partial r}{\partial p} \frac{\partial p}{\partial x} = 22 \cdot 2 = 44.$$

Again, the second factor stems from the local gradient of p. By getting $\frac{\partial r}{\partial b}$ with

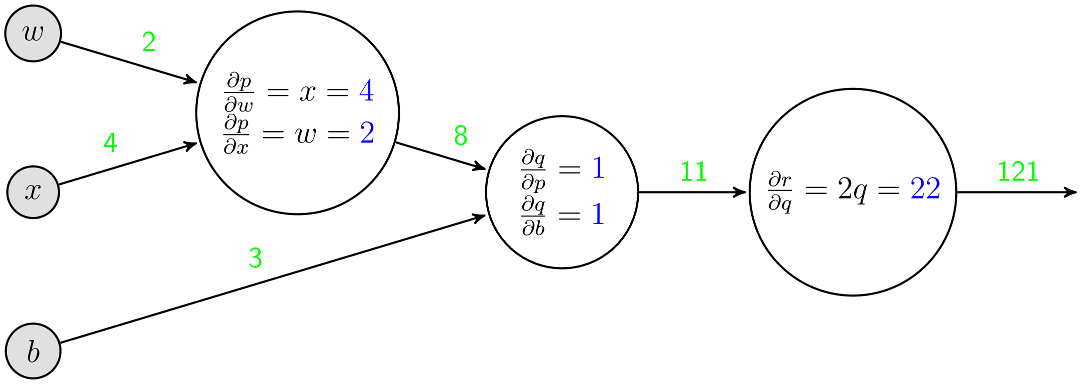
$$\frac{\partial r}{\partial b} = \frac{\partial r}{\partial q} \frac{\partial q}{\partial b} = 22 \cdot 1 = 22$$

the backpropagation algorithm finishes and outputs the gradient vector

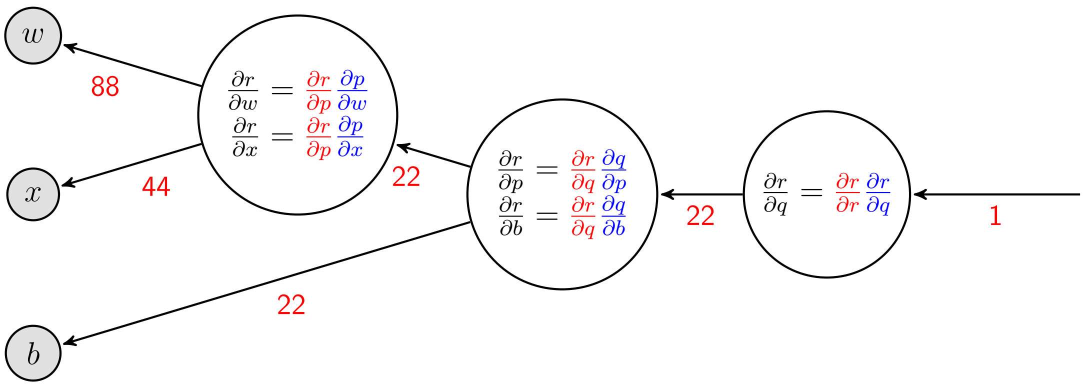
$$\nabla f(2, 4, 3) = \begin{pmatrix} 88 \\ 44 \\ 22 \end{pmatrix}.$$



(a) Computational graph representing the function $f(w, x, b) = (wx + b)^2$.



(b) Forward pass. We compute each node's output (green numbers) as well as their local gradients in topological order using a single pass. (blue numbers).



(c) Backward pass. By using the chain rule, we evaluate the partial derivative of the output r with respect to each node's input (red numbers) in reversed topological order.

Figure 10: Application of the Backpropagation algorithm on a computational graph.

Although the provided example is quite small, the described method remains the same even for very large computational graphs which represent complex neural network architectures with millions of weights. The algorithm is extremely efficient since it goes over each node exactly twice and performs rather simple calculations on them.

We finally presented all the theoretical tools required to train neural networks. However, we don't want our models to only perform well on the training data. Referring to the autonomous racing problem, we could build an end-to-end neural network as described in Section 3.3.1, train it with lots of images and steering angles and very likely, it still won't be able to drive as expected in the real world. In order to get desired results on the validation and test process, some more concepts are involved.

3.3.4. Overfitting

It often occurs that the model performs very well during training but miserably during the validation and testing steps. We say that the model *overfits* to the training data. Overfitting appears when the training loss decreases while the validation loss increases. *L2 regularization*, *dropout* and *data augumentation* are common strategies to mitigate this problem.

Regularization

The idea behind regularization stems from the observation that overly complex models don't generalize well [18] [23]. Therefore, we can add another term $\mathcal{R}$ to the loss function to penalize the model's complexity

$$\mathcal{L}_{\Phi,X,Y}(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{E}(\Phi(X_i,\Theta),Y_i) + \mathcal{R}(\Theta). \tag{14}$$

L2 regularization is a commonly used technique to penalize large weights which is defined as

$$\mathcal{R}(\Theta) = \lambda \sum_{i=1}^{m} \Theta_i^2. \tag{15}$$

Here, λ is another hyperparameter which represents the regularization strength. Intuitively, this technique could result in weights which are set so close to zero that they cancel out. As a consequence, large parts of the network may not be really in use anymore which virtually [4] shrinks the size of the model and potentially prevents overfitting [23].

[4] We say "virtually" since the actual computational costs are still the same as without regularization.

Dropout

Dropout is another regularization technique which randomly sets the activation of some neurons to zero (see Figure 11). The probability p of keeping neurons is another hyperparameter. We only drop activations during training. In order to maintain the same expected value across both phases, we multiply the output of each neuron by p during testing.

The intuition behind dropout is that it forces the neural network to have distributed representations since it can not rely on any single neuron to be reliably active for the representation of a state [24].

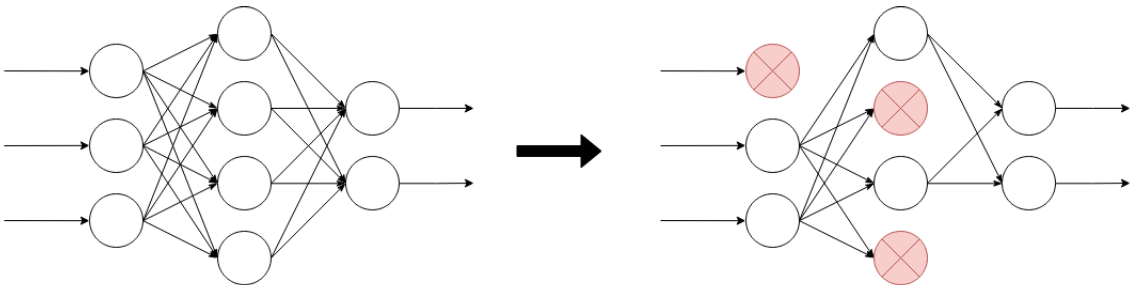


Figure 11: Visualization of dropout.

Data Augumentation

To further improve the model's generalization ability, we may expand the training set by applying some transformations. For example, we can mirror or rotate images or change the brightness and contrast. Thus we incorporate variations into the dataset which potentially makes predictions more invariant to different lighting conditions or perspectives not covered by the training data.

3.4. Convolutional Neural Networks

Neural networks are widely applied in image processing. However, traditional networks (as presented in Section 3.2) do not perform well on images. A 640×480 pixel image in RGB-colorspace results in an input layer with about 9.2×10^5 nodes, thus already 9.2×10^5 weights per node in the second layer. The second problem is that the network learns position-dependent features. Suppose we're training a neural network to recognize cats. On our training images, cats only appear in the upper left corner, so only neurons connected to this part get to know the specific features. Given an image where the cat sits in the bottom right corner, our network wouldn't be able to reliably detect it.

Convolutional layers are designed to alleviate these downsides. In a convolutional layer, each neuron has its own small region of the image it is looking at which is called the *receptive field*. Hence, neurons are not connected to every single pixel anymore. The

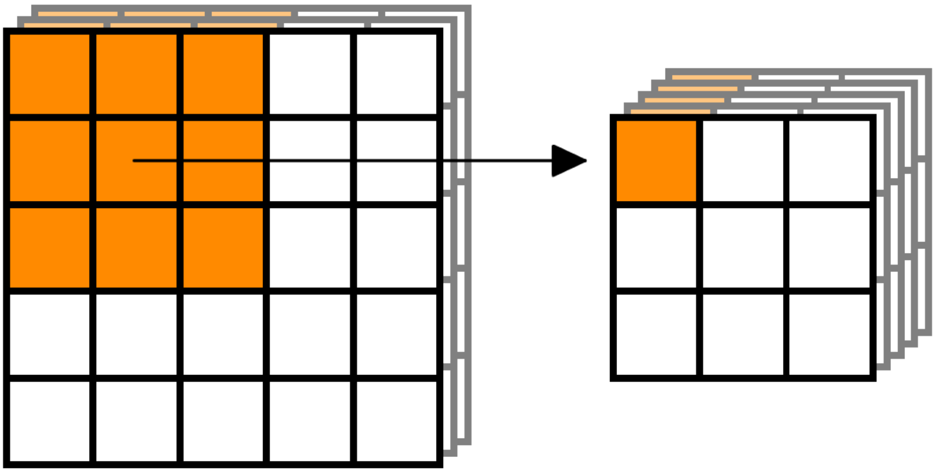


Figure 12: A convolutional layer, taking a $5 \times 5 \times 3$ image (left) as input. The third dimension represents the number of color channels. This convolutional layer has five 3×3 filters, resulting in five seperate layers of neurons (right). Note that filters always operate on the full depth of the input.

receptive field is a rectangular filter containing weights which learn to recognize a specific feature of the input. These weights are shared across each neuron and therefore, they're able to recognize features position-independently. One filter learns just one feature but a convolutional layer has multiple filters which results in multiple layers of neurons. Thus, convolutional layers can be thought of as three-dimensional volumes, visualized in Figure 12.

Filters can be represented as blocks, sliding across the image to compute the activations of its corresponding neurons. Thereby, they take the dot product of the filter values w and the input volume of their current receptive field x and add a bias b to it. Suppose a convolutional layer whose input layer has a depth of 3, the filter size is 5×5 and the number of filters is 10. Consequently, each filter has $5 \times 5 \times 3 + 1 = 76$ parameters where the $+1$ represents the bias. Since there are 10 filters, the layer has exactly 760 learnable parameters, regardless of the input's width and height.

While sliding the filter across the input, we may also choose an interval to skip certain pixels. This interval is referred to as the *stride*. For example, a stride of 3x2 means each time we move the filter horizontally, we skip two pixels, and each time we move the filter vertically, we skip one pixel. In addition, a *pad* can be used to add additional zero valued pixel to the border of the input in order to preserve the spatial size.

Powerful modern convolutional neural networks typically consists of lots of stacked convolutional layers. The idea is that the features in each layer become increasingly more abstract as we go deeper into the network. Filters in the first convolutional layer typically detect small characteristics such as edges. Then, the second layer may combine the features from the first layer into curves or other shapes. Eventually in some deeper layer, the composition of higher level features starts to represent meaningful objects.

4. Autonomous Driving

In this Section we present two implementations of autonomous racing using deep learning: the established end-to-end approach and our new trajectory prediction approach.

4.1. End-to-End Approach

In the autonomous driving terms, end-to-end means that a neural network model directly outputs steering commands based on sensory inputs such as camera images. *Nvidia* developed a neural network called *PilotNet* [7] that performs lane keeping on real cars. PilotNet receives a single image of a front-mounted camera as input and outputs a steering angle. The model is trained based on collected data from human driver sessions. Since the network seemed to be very promising, our idea was to adapt it for our application and to see whether it can be trained to drive the car fast, reliably and competitive on the racing track.

4.1.1. Preprocessing

Before the image is passed to the network, we involve some preprocessing. The original image resolution produced by the camera is 744×480. We select the region of interest by cropping the upper 744×200 pixels since these do not reveal any useful information about the track. Then, we resize the cropped image to have a resolution of 200×100. Note that our network implementation differs in this aspect from PilotNet which resizes its image to 200×66 pixels. However, the lane markings are quite thin so by further reducing the resolution, some markings in curves would simply disappear.

As proposed by *Nvidia*, we also convert the RGB image into the YUV color space. Since the environment is mostly black and white, we would expect the RGB channels to be the same anyway. The Y channel may yield more variation across the channels, since it represents luminance and not color value.

We apply batch normalization to the input layer which makes each individual pixel value's distribution zero mean and unit variance. Input normalization is a commonly used technique as it helps the gradient descent algorithm to improve convergence time. Ioffe and Szegedy provide an in-depth discussion about the underlying theory which goes beyond the scope of this work [17].

During training, each sample has a 60% chance to be augumented. When being augumented, we apply a random change of brightness to the image. Furthermore, there's another 50% chance to flip the image horizontally. Of course, in this case we also change the sign of the steering angle. The idea behind this augumentation is that the model may learn how to drive in both directions based on the same image.

4.1.2. Neural Network Model

Figure 13 shows our model's architecture which is an adaption of PilotNet. There are five convolutional layers which are intended to be feature extractors and their parameters were chosen empirically by *Nvidia*.

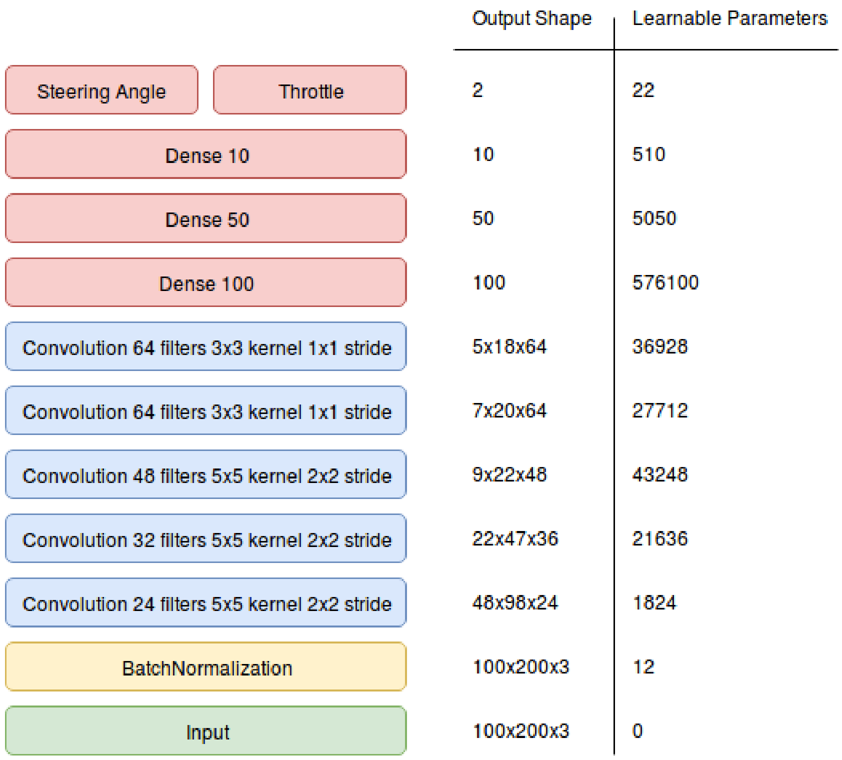


Figure 13: End-to-end model architecture based on PilotNet.

The first three convolutional layers feature a kernel size of 5×5 and a 2×2 stride whereas the two last convolutional layers only have a 3×3 kernel and a 1×1 stride. This is due to the fact that the output surface becomes increasingly smaller throughout the network. Furthermore, the filter size increases from 24 in the first layer up to 64 in the fourth and fifth layer which compensates the decrease in output volume size and adds more higher level features.

The three-dimensional output volume of the last convolutional layer is flattened into a one-dimensional vector of activations. This means that there are $5 \times 18 \times 64 = 5760$ neurons which are connected to the following 100 neurons of the first dense layer [5]. There follow another two dense layers of 50 and 10 neurons respectively.

The final output layer consists of two neurons which represent the steering angle and the throttle. PilotNet only outputs the steering angle so this is another modification

[5] A dense (or fully connected) layer is just another name for a layer of stacked neurons as presented in Section 3.2.

as we wanted our model not only to be a lane keeping system but also to control the speed.

We use ReLU activation functions throughout the network except for the two output nodes which have linear activation functions. Unfortunately, *Nvidia* doesn't state explicitly which activation functions they use in PilotNet. However, it's quite likely that ReLU functions are employed due to their enormous popularity [39]. For computing the loss, we take the mean squared error measure and trained the network using the ADAM optimization algorithm [20]. ADAM is similar to the gradient descent algorithm but adds some improvements to get to a local minimum more quickly. Our learning rate is set to $\gamma = 5 \times 10^{-4}$ which was discovered by experimenting.

We wrote the model in Python using the high-level neural network library *Keras* which uses *TensorFlow* for the backend. This makes it easy to implement the network by simply using the Keras layers API. The full code for implementing the end-to-end model is given in Appendix A.1.

4.1.3. Training

We split the recorded data into a training set (80% of all samples) and a validation set (20% of all samples). An extra test set is not required since testing happens directly on the racing track. The two sets are randomly sampled into batches of size 40. Furthermore, the model is trained in epochs. In each epoch, the model processes the entire training set.

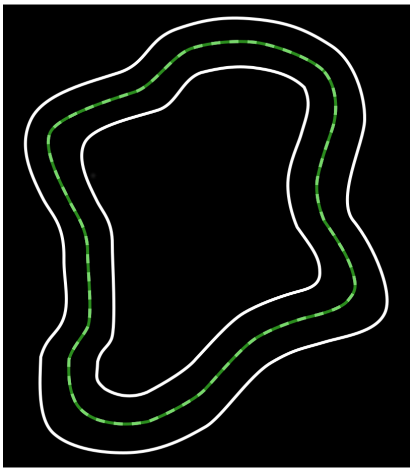
Training was very slow on the laptop we used initially which was a *Lenovo* Thinkpad L450 with an *Intel* Core i5 5200U and 8 GB memory. It took hours to train the neural network reasonably well on our datasets which made the process of model tuning very time consuming. Therefore, we wanted to train our model on a powerful GPU which speeds up learning due to its parallelization capabilities. The new training system contains an *Intel* Core i7 2600K with 4 cores, 16 GB memory and an *Nvidia* GTX 1080. However, to take advantage of the GPU, we had to adapt and optimize the data loading and preprocessing procedure.

To provide the training algorithm with data, we implemented batch generators which are callbacks to generate the next training or validation samples. Whenever the batch generator was called, it was reading 40 images from the hard drive into memory which was then sent over to the GPU. Since reading from hard drive is slow, most of the time wasn't spent on training but on loading data. The solution was to simply cache as many images as possible into the 16 GB memory. As a result, we could reduce the training time by a factor of 12 in our particular case.

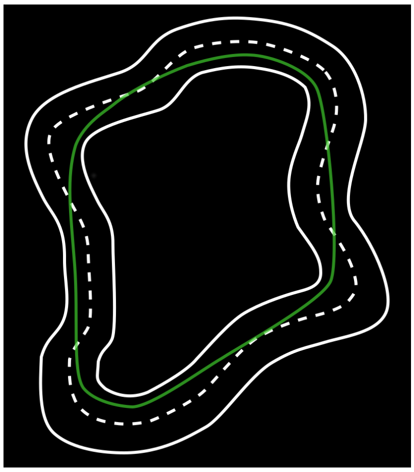
Of course, when increasing the size of the dataset, we would also have to increase the memory size on the machine in order to maintain the same level of speedups. However, since our largest dataset only contained about 40000 images, this wasn't a big issue.

4.1.4. Dataset

To make the system move autonomously, we had to collect data by driving the car ourselves on the racing track. Therefore, we created two datasets. In the first dataset, we were trying to drive the car as close as possible on the track's center line which is depicted in Figure 14(a). The second one was produced by driving the car on an ideal line which is illustrated in Figure 14(b). By ideal line, we refer to the strategy of keeping the wheels mostly straight in order to accelerate as long as possible. The idea is to exploit our excessive motor power to outperform the competition.



(a) Center line.



(b) Ideal line.

Figure 14: Illustration of the driving strategies performed on the racing track to record two datasets.

Both datasets contained about 10000 images of driving. For the center line session, we were driving 12 laps with relatively slow speed. During the ideal line session, we recorded 28 laps but at the maximum speed which we could still drive without leaving the track. It should be noted here, that even though we recorded more laps in the second run, the datasets contain roughly the same number of images. This is due to the fact that we were simply driving faster while recording the ideal line.

We like to mention here that we actually recorded more data than 20000 images. In addition, we built our own test tracks in our office and performed several tests on them but we think that the two described datasets are the most representative and decision leading ones. Furthermore, since we changed the camera mounting position a couple of weeks before the final racing, we couldn't use the data we had previously recorded anymore. Nevertheless, this was the correct decision as Figure 15 shows the difference between the two setups. The car gained a much cleaner, wider and undistorted overview of the track. In fact, this becomes even more crucial for the trajectory prediction approach.

(a) Low-mounted camera position.

(b) High-mounted camera position.

Figure 15: Different camera positions.

4.1.5. Test Results

We trained our neural network model with the center line and ideal line datasets over 50 epochs using the same parameters.

Loss Analysis

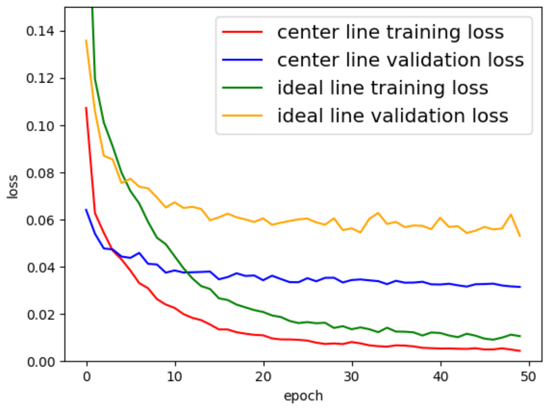
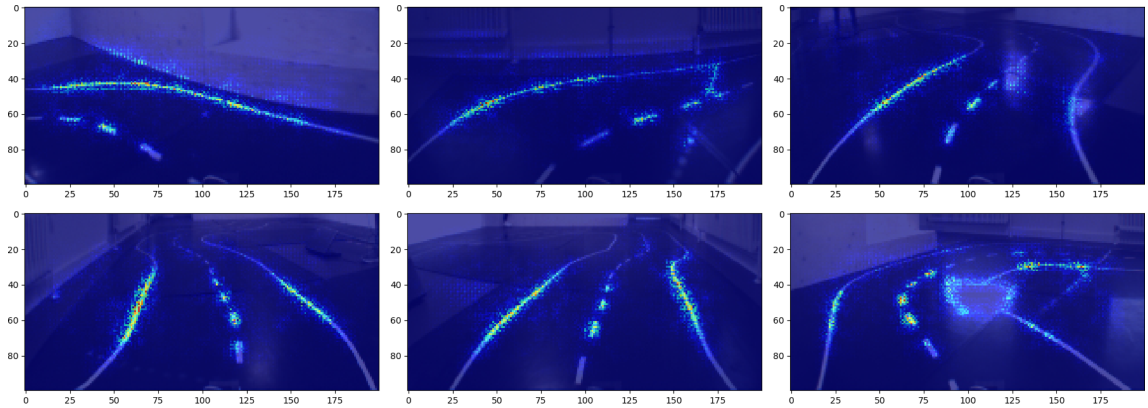


Figure 16: Training and validation loss of the model when trained with the center line and ideal line datasets over 50 epochs.

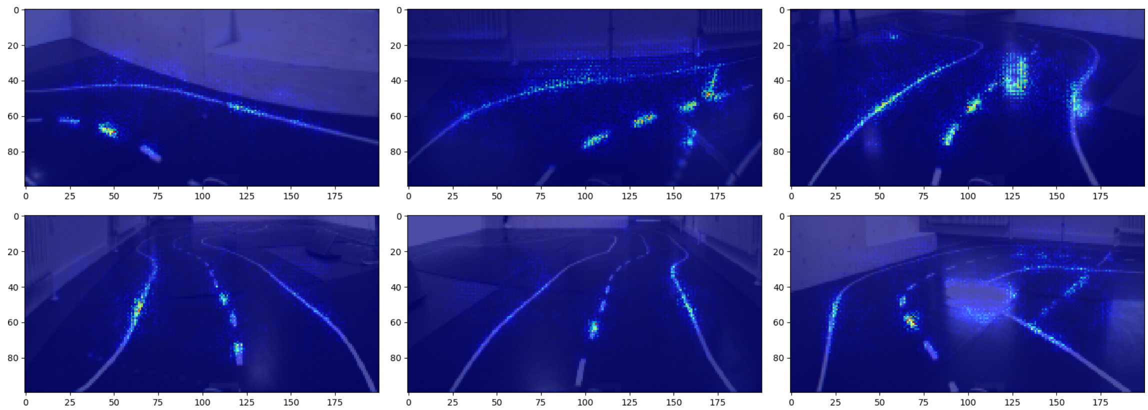
Figure 16 shows the loss diagram during training. We can observe that during both runs, the model didn't overfit to the training data since the validation loss does not tend to increase. The model seems to have more problems learning the ideal line dataset since the validation loss of the center line dataset is about 33% lower. Since we recorded 28 laps in the ideal line set, it contains much more variation compared to the 12 laps of the center line run. Furthermore, in the center line dataset, the steering angle heavily corresponds to the curvyness of the center lane markings directly in front of the car. This is not the case when driving the ideal line since there are cases when the car drives straight even though the track makes a slight curve.

Visualization

To have some clue about which features of the image the neural network takes into account, Figure 17 illustrates *saliency maps* [35] based on some samples of the track. Saliency maps visualize the gradients from the network's output with respect to the input pixels. The intuition is that if the gradient of a certain pixel input has a high value, it has much influence on the output and is important to the network. In Figure 17, we computed the gradients from the output of the fifth convolutional layer using *guided backpropagation* [37] and layered them over the original image.



(a) Saliency maps of the model trained with the center line dataset.



(b) Saliency maps of the model trained with the ideal line dataset.

Figure 17: Saliency maps.

On the center line saliency maps in Figure 17(a), we can clearly see that the model primarily looks at the lane markings and even ignores the reflections on the image at the top right. However, on the ideal line saliency maps in Figure 17(b), the markings are not nearly as much highlighted. It also seems to be the case that the model is triggered by noise and reflections seen at the top middle and top right images.

Driving Behavior

To test our car, we were driving it in the same environmental conditions that prevailed during the recordings. Then we performed another test in the evening to see how it reacts to different lighting conditions.

In the first test, the car was indeed behaving quite similar to the way we were driving it manually. It could both stay on the center line and also adapt the driving behavior of the ideal line. We don't actually have a robustness metric but the car would surely perform valid rounds in most of the ten available attempts under the given conditions.

However, this appeared to be quite different during the second test. Our car had severe problems to reliably stay on the track. The drastic change in behavior was quite unexpected since we augumented the dataset with different brightness settings which should actually lead to a better generalization ability. Maybe this problem can be fixed by extending the datasets with recordings in various other lighting conditions.

A much bigger problem turned out to be that we were not able to scale up the speed. By doing so with any dataset, the car was always leaving the track very soon. We believe that the main cause of this problem was produced by the slow steering of the servo. While the human driver adapts to this problem by intuitively steering earlier, the neural network is unable to take these dynamics into account. Replacing the servo improved the situation a little bit but we still didn't nearly reach superhuman level of racing.

We discovered another unpleasant property of the end-to-end approach: the car's ability to learn a certain driving behavior depended heavily on the way we performed steering inputs. Therefore, we created two recordings while driving the car on the center line. During the first run, we maxed out the steering input for a split of a second whenever we had to perform any course corrections. This means that there exists basically only three steering inputs: full left, full right and neutral. During the second run, we pushed very gently on the controls to perform smooth transitions to the input. While the car was driving well when fed with the data from the second recording, by training it with the first recording, it was barely able to drive a single lap.

This result might be inferred as follows. Imagine that we are creating a dataset by trying to keep the car as closely as possible on the center line but our inputs are only full left, full right and neutral. Next, consider driving a wide left curve. In order to keep the car on the center line, it's not possible to consistently maintain full left as input since the car would sheer off too far. Therefore, one has to quickly alternate between full left and neutral at a certain frequency to follow the line safely. Let's consider a sequence of consecutive images in this dataset. These images look quite similar to each other due to high recording frame rates. However, the sequence possesses a distribution of extremely dissimilar steering angles. If similar training inputs have strongly dissimilar labels, the network may either learn to approximate the mean of the label distribution or overfits by learning the subtle and irrelevant differences in these inputs. This is due to the nature of the learning process being just an optimization algorithm which tries to minimize the error between the network's output and the labels. In both cases, the car performs extremely poorly as the neural network is unable to reproduce the actual driving behavior. This emphasizes the fact that the success of the employed deep learning techniques depend heavily on the choice of training data.

Our conclusion is that the end-to-end approach might seem to be an elegant way to implement a system. However, it has some serious downsides when being applied to autonomous racing. Training the car to obtain a desired driving behavior requires profound manual driving skills. Therefore, the vehicle will not perform much better than the human steering the car during the recordings.

4.2. Trajectory Prediction Approach

Based on the experiences we gained using the end-to-end approach, we asked ourselves two questions:

- Can we design a system that uses deep learning to process images into steering informations but being easy to train at the same time?
- Can we scale the speed such that the car is able to outperform any manually driven lap times?

In the following, we present the new design we came up with by considering these two requirements.

4.2.1. Overview

Our concept exploits the information retreived by the wheel encoders to construct an odometry which gives a path of positions we were driving along. Based on some position p_0 in the odometry we can easily compute the position the car will be located at in a given distance. This allows us to sample the coordinates (p_1, p_2, p_3) which represent the car's future locations at 0.6, 1.2 and 1.8 meters ahead of p_0. We then transform these coordinates into a local coordinate system where we assume that the car is currently located at $(0, 0)$ and pointing at direction $(0, 1)$. After applying the transformation, the new points (p'_1, p'_2, p'_3) represent a *local trajectory* which the car follows along. Figure 18 further illustrates this coordinate transformation which will be explained in more detail later on.

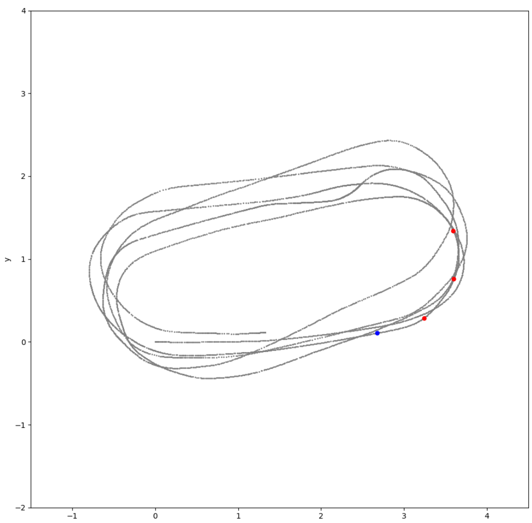
We modified the output layer of our end-to-end model to predict these trajectory coordinates instead of the steering angle and throttle values. Thus, we refer to this approach as *trajectory prediction*. For each image in our dataset, we compute the corresponding local trajectories the car was driving during the recording sessions. These image-to-trajectory mappings become the new training data which the model is fed with. Finally, during autonomous mode, we compute the steering angle and target speed based on the predicted trajectories using an appropriate driving model.

The core idea behind this approach is that we no longer train the neural network on *how* to drive the car but instead it only learns *where* it should be located at in the future. This has several advantages. First, the trajectories the network is trained with are computed independently of time. This means, during recording, we can drive the car around the track as slowly as required. Therefore, the car's behavior doesn't depend on manual driving skills anymore.

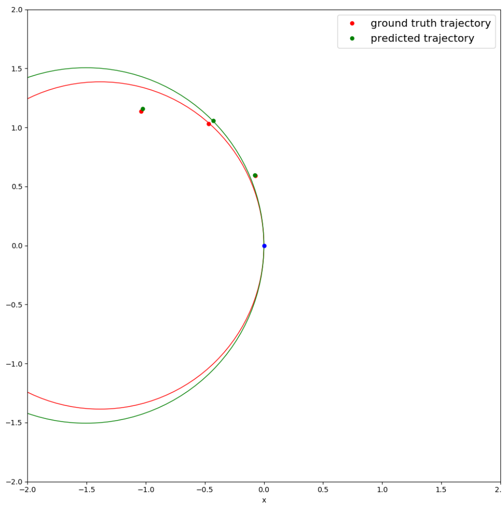
Secondly, we gain control over the driving dynamics. Assume the neural network would always predict the desired trajectories. This allows us to compute the speed and steering angle based on the car's physical limits. Obviously, a complex physical driving model would be required which was not possible to implement during the given time scope. However, we did make use of the fact that we we're able to predict straight lines and curves to adapt the speed accordingly.

In the following sections we give an in-depth explanation about the implementation of our trajectory prediction approach.

(a) Current camera image of the car, driving along our test track.



(b) Recorded odometry data. The blue dot shows where the car is currently located. The red dots reveal the future locations in 0.6, 1.2 and 1.8 meters.



(c) The trajectory (red dots) computed by projecting the odometry coordinates into a local coordinate system were the center is the car's current position. The green dots represent the network's predicted trajectory given the current camera image as input. The circles give the turning radius which is then processed into a steering angle.

Figure 18: Illustration of the coordinate projection and the trajectory prediction.

4.2.2. Odometry

An odometry is a sequence of positions which are computed by estimating the car's movement based on the wheel encoder ticks. It's a relative positioning model which means that position p_t is evaluated based on p_{t-1} and the estimated change of position obtained by the current sensor output. Such models are sensitive to errors since any slight deviation from the actual position is accumulated over the sequence.

Positioning Model

Given the vehicle's current position $p_{t-1} = (x_{t-1}, y_{t-1})$ and orientation ω_{t-1}, we compute its new position p_t and orientation ω_t using the following *dead-reckoning* equations [19]. Initially, we choose $p_0 = (0, 0)$ and $\omega_0 = 0$.

Let d_l and d_r be the diameters of the rear left and right wheels respectively and let η be the number of encoder ticks per wheel revolution. The travelled distance per tick on each side is given by

$$c_l = \frac{\pi d_l}{\eta} \qquad c_r = \frac{\pi d_r}{\eta}. \tag{16}$$

Suppose that N_l and N_r are the number of ticks counted by the wheel encoders during the current time interval. Then, the travelled distance for each wheels is

$$\Delta s_l = N_l c_l \qquad \Delta s_r = N_r c_r. \tag{17}$$

We take the average to model the actual travelled distance of the car (or the imaginary center wheel):

$$\Delta s = \frac{\Delta s_l + \Delta s_r}{2}. \tag{18}$$

The change in orientation is approximated by

$$\Delta\omega = \frac{\Delta s_r - \Delta s_l}{T} \tag{19}$$

where T is the distance between the points where the rear wheels are touching the floor. We refer to this distance as the *track width*. Now, we can get the new orientation

$$\omega_t = \omega_{t-1} + \Delta\omega. \tag{20}$$

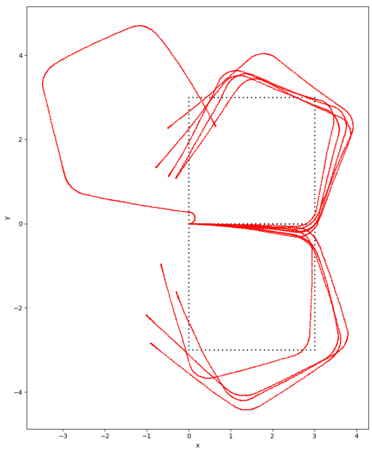
Finally, we update the position using simple trigonometric equations

$$\begin{aligned} x_t &= x_{t-1} + cos(\omega_t)\Delta s \\ y_t &= y_{t-1} + sin(\omega_t)\Delta s. \end{aligned} \tag{21}$$

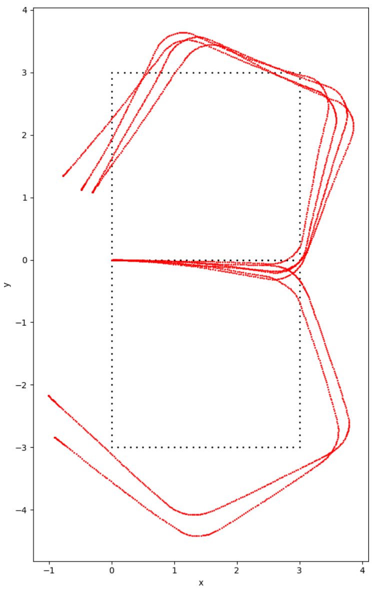
Calibration

As mentioned already, odometry is very prone to errors. These errors can be classified into *systematic* and *non-systematic* errors [19]. Non-systematic errors are caused by the fact that the vehicle mostly doesn't drive in a perfect environment. For example, the wheels may be affected by slippage resulting in the registration of wheel revolutions that didn't actually move the car forward. Even small irregularities of the floor such as bumps lead to errors since Equation 17 doesn't model horizontal displacements.

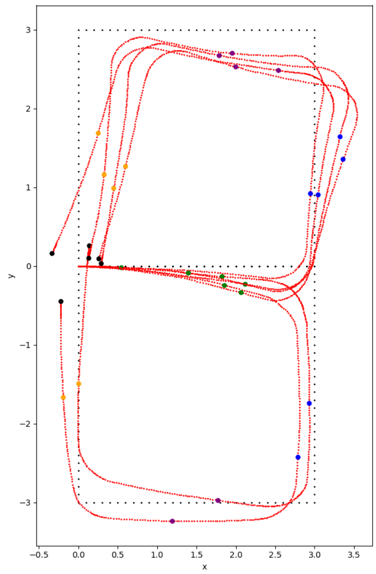
Systematic errors are caused by measurement inaccuracies of the wheel diameters and the track width. For instance, by manufactoring rubber wheels, it's hard to avoid at least some slight deviations in size. Uncertainty is involved in measuring the track width since there doesn't exist a true single point where the wheels are in contact with the floor. In addition, by driving in a curve, the car might tilt to the side which shifts these points slightly along the wheels.



(a) Odometries using the original parameters.



(b) Odometries used for calibration (outliers excluded).



(c) Odometries using the optimized parameters. The colored dots represent points where the error is measured.

Figure 19: Reconstructed odometries using the ticks obtained by driving around a 3×3 meter square (black dots) in clockwise and counter-clockwise direction respectively.

Fortunately, compared to the inevitableness of the non-systematic error, we can minimize the systematic error by performing a calibration procedure. Therefore, the objective is to minimizes the error between a computed odometry (using the equations above) and a corresponding ground-truth odometry by optimizing the parameters d_l, d_r and b.

In our case, we were driving the car along a 3×3 meter square by recording the accumulated wheel encoder ticks N_l and N_r in a 20 millisecond time interval. We created five runs in clockwise and counter-clockwise direction respectively. The initial parameters were measured using a ruler which gave us the following dimensions:

$$\eta = 120 \qquad d_l = 11cm \qquad d_r = 11cm \qquad T = 27.5cm.$$

The resulting odometry is shown in Figure 19(a). Using these parameters, the car's position seems to drift heavily apart from the square. In addition, we observed that even though we were keeping the vehicle precisely on the line, there exist some substantial outliers which couldn't possibly be caused either by inaccurate parameterization or by incorrect driving. Therefore, during optimization, we only took those runs into account which seemed to be the most reasonable ones (see Figure 19(b)).

We measure the error between a reconstructed odometry and the ground truth odometry by meaning the squared distances between the final position at t_{final} and between positions at manually selected indices where we know the car was driving straight. At those selected indices, we only take either the distance in x- or y-coordinate into account, depending on whether the car was moving horizontally or vertically in the particular section.

For example, the blue dots in Figure 19(c) represent indices τ_i where we know for sure that the vehicle was heading vertically. Therefore, the x-coordinates of the reconstructed odometries at positions τ_i should be about 3.

In order to define the error function for the optimization problem, we need some formalism. An odometry can be described as a function $o(p_0, \Theta, N, t)$ [8] where

- p_0 is the start position (in our case $p_0 = (0, 0)$),
- $\Theta = (c_l, c_r, T)$ is the parameter vector,
- N represents the wheel encoder ticks and
- t is the current time or index.

Let N_i^{cw} and N_i^{ccw} be the number of ticks of the i-th run along the 3×3 meter square in clockwise (cw) and counter-clockwise (ccw) direction respectively. Furthermore, let $\tau_{j,i}^{cw}$ and $\tau_{j,i}^{ccw}$ be the manually selected indices for each individual run where

- $\tau_{1,i}^{cw}$ and $\tau_{1,i}^{ccw}$ is an index in the first horizontal section of run i,
- $\tau_{2,i}^{cw}$ and $\tau_{2,i}^{ccw}$ is an index in the first vertical section of run i,
- $\tau_{3,i}^{cw}$ and $\tau_{3,i}^{ccw}$ is an index in the second horizontal section of run i,
- $\tau_{4,i}^{cw}$ and $\tau_{4,i}^{ccw}$ is an index in the second vertical section of run i and
- $\tau_{5,i}^{cw}$ and $\tau_{5,i}^{ccw}$ is the final index of the odometry of run i.

Given ticks data N with n runs in clockwise and m runs in counter-clockwise direction and indices τ, our error function E is defined as:

$$
\begin{aligned}
E_{p_0,N,\tau}(\Theta) = \frac{1}{2}\Bigg(\quad & \frac{1}{5n}\sum_{i=1}^{n} \Bigg[&& o(p_0,\Theta,N_i^{cw},\tau_{1,i}^{cw})_y^2 \\
&&& +(3 - o(p_0,\Theta,N_i^{cw},\tau_{2,i}^{cw})_x)^2 \\
&&& +(-3 - o(p_0,\Theta,N_i^{cw},\tau_{3,i}^{cw})_y)^2 \\
&&& +o(p_0,\Theta,N_i^{cw},\tau_{4,i}^{cw})_x^2 \\
&&& +\|o(p_0,\Theta,N_i^{cw},\tau_{5,i}^{cw})\|^2 \Bigg] \\
& +\frac{1}{5m}\sum_{i=1}^{m} \Bigg[&& o(p_0,\Theta,N_i^{ccw},\tau_{1,i}^{ccw})_y^2 \\
&&& +(3 - o(p_0,\Theta,N_i^{ccw},\tau_{2,i}^{ccw})_x)^2 \\
&&& +(3 - o(p_0,\Theta,N_i^{ccw},\tau_{3,i}^{ccw})_y)^2 \\
&&& +o(p_0,\Theta,N_i^{ccw},\tau_{4,i}^{ccw})_x^2 \\
&&& +\|o(p_0,\Theta,N_i^{ccw},\tau_{5,i}^{ccw})\|^2 \Bigg] \Bigg)
\end{aligned}
\tag{22}
$$

where $\|\cdot\|$ is the euclidean distance.

We approximate the gradient ∇E using finite differences and apply the *Broyden-Fletcher-Goldfarb-Shanno* algorithm (BFGS) [28] to compute the optimal parameters Θ. The resulted odometries are shown in Figure 19(c).

We can see that the final positions and the overall shape of the odometries heavily improved. However, given the fact that the runs were performed on a reasonably small 3×3 meter square, the error is still very large.

We tried out various other optimization techniques, including the popular *UMBmark* [19] method as well as other error functions [8] and even tried to manually calibrate the parameters. Nevertheless, the presented method provided the odometries which seemed to be closest to the actual square.

Trajectory Extraction

Let $f(p_t, s, t)$ be a function computing the position on the linearly interpolated path p_t which is reached by following this path for distance s while starting at index t. Given odometry (p_t, ω_t), to compute the trajectory $Q' = (q'_1, ..., q'_n)$ at index τ, we sample positions $Q = (q_1, ..., q_n)$ at distances $\mathcal{D} = (D_1, ..., D_n)$ apart from p_τ by computing

$$q_i = f(p_t, D_i, \tau) \tag{23}$$

To get Q', we apply the transformation

$$q'_i = \mathcal{R}\left(\frac{\pi}{2} - \omega_\tau\right)(q_i - p_\tau) \tag{24}$$

where $\mathcal{R}$ is the rotation matrix

$$\mathcal{R}(\alpha) = \begin{pmatrix} cos\ \alpha & -sin\ \alpha \\ sin\ \alpha & cos\ \alpha \end{pmatrix}.$$

In our experiments, choosing $n = 3$ positions with distances $d_1 = 0.6\,\mathrm{m}$, $d_2 = 1.2\,\mathrm{m}$ and $d_3 = 1.8\,\mathrm{m}$ away from the car's current position p_τ turned out to work well. Larger distances may lead to bad model predictions since when entering a sharp turn, the camera cannot see much of the track. In this situation, the neural network may not be able to use the visual information of the track to predict a position which is far in the future and might learn different featurers instead during training.

We think that choosing more predicted positions to increase the resolution of the trajectory is unnecessary as three or even only two positions are enough to infer the steering angle in our case.

4.2.3. Neural Network Model

The only aspect in the neural network architecture that differs from the one presented in Section 4.1.2 is the output layer. We replaced the nodes for the steering angle and throttle with two nodes for each trajectory position which output their x- and y-coordinates respectively. Just as before, MSE is used as the error measure. The Keras implementation of the network is shown in Appendix A.2.

There are only slight changes in the preprocessing and training procedures. For example, during augumentation, instead of flipping the steering angle's sign, we now flip the sign of the x-coordinate of the trajectory positions.

4.2.4. Driving Model

In the following, we describe how to get from a predicted trajectory to a final steering angle and throttle command.

Controlling the Steering Angle

We model the car's motion using a bicycle model as shown in Figure 20. Assuming no slippage, the car follows along a circle with radius r when setting steering angle α. Given α the circle's radius is modeled by r_α and given r the required steering angle is represented by α_r using the two equations

$$r_\alpha = \frac{b}{tan(\alpha)} \qquad\qquad \alpha_r = arctan\left(\frac{b}{r}\right) \tag{25}$$

where b is the distance between the front and the rear axle which we refer to as the *wheel base*.

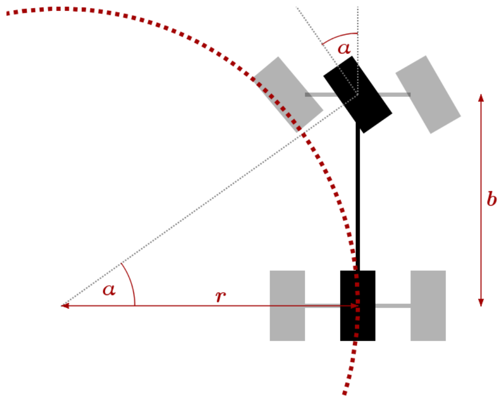


Figure 20: Illustration of the bicycle model.

Let $Q = (q_1, ..., q_n)$ be a trajectory, our goal is to find steering angle α resulting in a turning circle of radius $|r_\alpha|$ and center $(r_\alpha, 0)$ which fits the trajectory best (see Figure 18(c) again). We can formulate this objective as another optimization problem where we want to find the scalar α which minimizes the error function

$$E_Q(\alpha) = \begin{cases} \frac{1}{n}\sum_{i=1}^{n}(q_{i,x})^2 & \text{if } \alpha = 0 \\ \frac{1}{n}\sum_{i=1}^{n}(\|q_i - (r_\alpha, 0)\| - |r_\alpha|)^2 & \text{otherwise} \end{cases} \tag{26}$$

given trajectory Q and the constraint $\alpha_{min} \leq \alpha \leq \alpha_{max}$. This formula computes the mean squared distance between point q_i in the trajectory and the point lying on the border of the circle constructed by r which is closest to q_i. Note that the special treatment of $\alpha = 0$ is required due to the fact that r_0 is not defined. In this case, we just take the distance between q_i and the x-axis.

Instead of applying a numerical approximation method to find the optimal α_{opt}, it turned out to be most efficient to simply compute

$$\alpha_{opt} = \arg\min_{\alpha}\{E_Q(\alpha) \mid \alpha \in \mathbb{Z},\ \alpha_{min} \leq \alpha \leq \alpha_{max}\}. \tag{27}$$

So we just iterate over the integer set of steering angles[6] and keep α where $E_Q(\alpha)$ is lowest. The loss in precision is negligible due to the low accuracy of the car's servo which exhibits a tolerance of about $\pm 2°$.

It should be mentioned that we only considered the trajectory points $Q = (q_1, q_2)$ to fit the turning circle so we left out q_3 which is 1.8 m apart from our current position. By leaving it in, the car starts to turn too early in front of curves and leaves the track as a consequence. In addition, small course corrections in q_1 and q_2 would be very likely to average out. Therefore, choosing Q carefully is crucial to fine tune the driving behavior.

[6]In our case, the maximum steering angle turning the wheels to the left is $\alpha_{min} = -30°$ whereas $\alpha_{max} = 30°$ turns the wheels most to the right.

Controlling the Speed

A desired driving behavior is to accelerate on straight lines and maintain a lower speed in curves to provide stability and avoid slippage. Due to the lack of time, we employed a very simple model to implement this behavior. Given trajectory $Q = (q_1, q_2, q_3)$, we just use the x-coordinate of q_3 as a curve detector and compute the target speed $w(Q)$ (in meter per second) with the formula

$$w(Q) = \begin{cases} \nu_1 & \text{if } |q_{3,x}| \leq \chi \\ \nu_2 & \text{otherwise} \end{cases} \tag{28}$$

where

- ν_1 is the target speed when driving straight,
- ν_2 is the target speed when detecting a curve and
- χ is the threshold in the x-coordinate of q_3 which decides when to use v_1 or v_2.

Thus, there are only two possible target speeds ν_1 and ν_2. The choice of χ determines how early the car should start to slow down to ν_2 in front of a curve. Obviously, these three constants have to be discovered by experimenting which is a downside of this approach.

After determining the target speed, we have to set the motor torque accordingly to maintain it. In each time step t we compute the error between the target speed w_t and the measured speed v_t

$$e_t = w_t - v_t \tag{29}$$

where v_t is computed by

$$v_t = \frac{\Delta s}{\Delta t} \tag{30}$$

and where Δs is taken from Equation 18. Our objective is to apply corrections u_t to the motor torque value $z_t \in [-1, 1]$ which minimizes the error e_t over time. Therefore, we utilize the popular *proportional–integral–derivative controller* (PID-controller) [6] [44] which calculates an appropriate correction with

$$u_t = K_p e_t + K_i \sum_{t'=0}^{t} (e_{t'} \Delta t) + K_d \frac{e_t - e_{t-1}}{\Delta t} \tag{31}$$

where $K_p, K_i, K_d \in \mathbb{R}_{\geq 0}$ are the coefficients for the proportional, integral and derivative terms respectively. We ended up choosing $K_p = 5 \cdot 10^{-3}$, $K_i = 0$ and $K_d = 1 \cdot 10^{-3}$ after tuning these coefficients manually. Using this configuration, we compromised a bit of error overshooting for a more rapid acceleration and deceleration.

5. Results

In this Section we present the results of our trajectory prediction approach. We do an offline analysis again and then show how it performed during the final race. In addition, we point out some issues of the system and propose possible improvements.

5.1. Wheel Encoder Anomaly

During our tests using the trajectory prediction approach, we noticed that each time the car performed a tight right turn, it was strongly accelerating. However, when driving left turns or straight, the controller was keeping the target speed just fine. It turned out that there exists an anomaly in the measurement of the wheel encoder ticks. We did three experiments where we were driving the car manually for a fixed distance s and recorded the wheel encoder ticks for each run. In the first experiment, we were driving a circle in clockwise direction while keeping steering angle $\alpha = 30°$. We measured the radius r of that circle and determined s as the circumference $s = 2\pi r$. During the second experiment, the exact same circle was followed but in counter-clockwise direction with steering angle $\alpha = -30°$. Finally, in the third experiment, we were keeping the vehicle on a straight line for distance s.

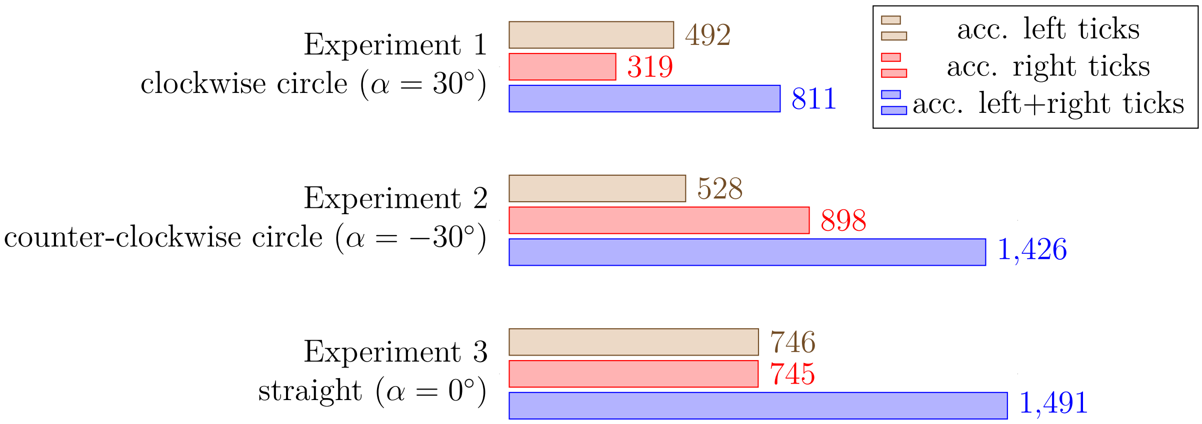


Figure 21: Results of the three experiments driving the car manually for a fixed distance. The diagrams show that the accumulated number of emitted encoder ticks are about 45% less when driving a clockwise circle as compared to when driving a counter-clockwise circle or a straight line with the same travelled distance.

Since the car covered the same distance across these runs, we would expect the accumulated number of recorded wheel ticks to be about the same too. For each experiment, Figure 21 shows the accumulated number of ticks recorded by the left and right wheel encoder. To make sure that our measurements are reliable, we repeated each experiment three times and averaged the results. We can see that our expectations hold for the second and third experiments. However, when driving the clockwise circle, the total number of accumulated ticks is about 45% less compared to the other runs.

These results explain the misbehavior of the speed controller. When driving a clockwise turn, the number of emitted ticks reduces and consequently the measured distance over time decreases too even though the car maintains the same actual speed. Therefore, the PID controller thinks that the car slows down and misleadingly increases the motor torque which causes the aforementioned acceleration. In addition, the experiments may also explain the bad results of the odometry calibration. Unfortunately, we were not able to fix this problem since there wasn't enough time. In order to mitigate this problem, we decided to only drive counter-clockwise runs during the final race and during the recordings for our dataset.

5.2. Dataset

Our final dataset includes about 17000 samples of recordings while driving the car on the ideal line in the noon and in the evening. However, when only following the ideal line, the neural network doesn't learn how to behave in different situations such as when leaving the track accidentally. Therefore, we added the functionality to pause and resume the recording session using the buttons on our game controller. This allows us to hit pause and drive the car into a certain position that hasn't been covered by the dataset yet. Then we resume the recording session and lead the car back to the ideal line. We repeated this procedure to teach the car how to behave for various deviations from the ideal line.

Since we had no indication of whether we were currently pausing or resuming, after the final race it unfortunately turned out that we didn't include another 7000 samples by forgetting to hit the resume button.

5.3. Final Race

We can list the important parameters for the trajector approach as

$$\Omega = (c_l, c_r, T, \mathcal{D}, b, \nu_1, \nu_2, \chi, K_p, K_i, K_d)$$

which were all introduced previously. The concrete values used in the final race are given in Appendix B.

We trained the nerual network over 50 epochs with learning rate $\gamma = 1 \times 10^{-4}$. The training and validation loss were both converging so there didn't seem to be an indication for overfitting.

Figure 22 shows saliency maps using the trained trajectory prediction model. Although there seems to be much interest on the lanes (especially on the center line), especially the walls and reflections became salient objects which is not really desired.

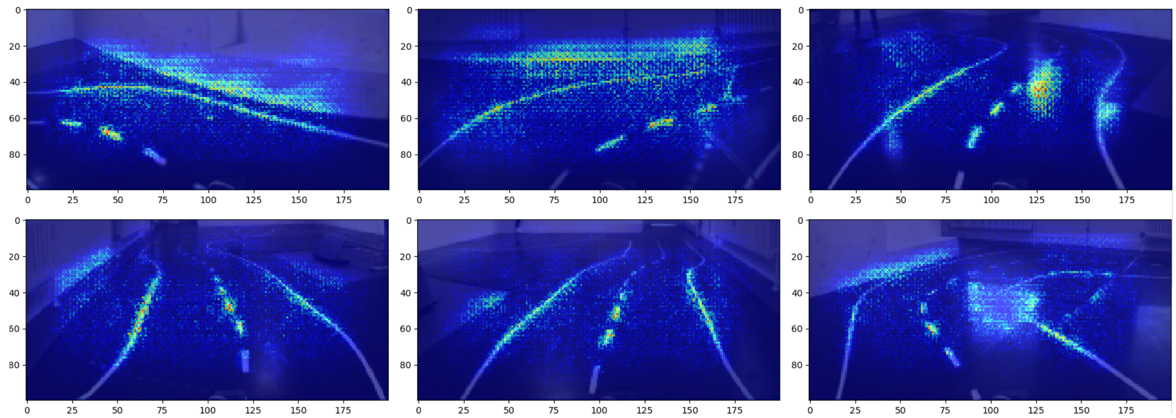


Figure 22: Saliency maps of the trajectory prediction model.

We decided to use the trajectory prediction approach for the final race since we managed to make the car much faster on both the office and the final track compared to the end-to-end approach. Two days before the race, our best lap time was about 8.5 seconds. To get a sense of that speed, there was no competitor who was able to drive a lap under 7 seconds. The best lap-time performed by a human driver was about 6.9 seconds.

Despite being reasonably fast, the car did leave the track quite frequently. However, it was capable of returning back to the track surprisingly often. We think that the camera's large overview and the specially recorded situations to return to the ideal line definitely helped in this regard. The overall driving behavior was much as we expected. It was following the ideal line most of the times and did manage to accelerate on straight lines and was braking in front of curves.

Our strategy was to sacrifice stability for speed. Even though the end-to-end model which was trained on the center line dataset would have probably been making fewer mistakes, there was no chance of winning the competition had it been used. The rules gave room to drive riskfully since we only had to perform a single valid round. Our best valid lap-time during the final race was 10.0 seconds which we achieved the third place with. The winner's lap-time on this day was 8.93 seconds.

It turned out that all cars including ours had severe problems with different lighting conditions. There was a short preparation time at 4 p.m. where the teams were allowed to perform the last runs on the track. Nobody ever trained their car at that time of day and lighting. The result was that the vehicles were barely able to drive a single lap without leaving the track. By asking around, we figured out that every competitor was using deep learning approaches to process the camera frames. This raises questions about the overall generalization ability of neural networks.

5.4. Limits

There are several reasons why we think that our trajectory approach hasn't reached its full potential. The decision to lowering the speed in curves didn't stem from the fact that the wheels started to slip. The vehicle rather seemed to not react aggressively enough. However, we do not think that there is an issue regarding the computational speed since we measured that the system processes 20 frames per second on average. In our opinion, this should be more than enough visual information between two consecutive frames to react even to sharp turns just in time. Also, we belief that after replacing the servo, the steering speed wasn't an issue as well.

What we do belief is that the actual problem source was rooted in the non-systematic error of the odometry which was produced by the weird anomaly of the wheel encoder output. Even after calibration, the odometry comprised too large deviations from the actual driven path. Since the odometry constructed the trajectories which were used as the labels in our training data, the non-systematic error propagated into the predictions of the neural network. Furthermore, we noticed that the steering servo produced a rather non-linear translation between the input angle and the actual angle of the wheels which is another source of inaccuracy.

5.5. Possible Improvements

There has been lots of research on how to implement localization methods for robots. More advanced techniques such as *simultaneous localization and mapping* (SLAM) may be employed in order to establish a more accurate position estimate. In order to use SLAM, it is necessary to provide odometry data as well as periodic scans of the environment which are often acquired by *LIDAR* sensors. Besides dead-reckoning with wheel encoders, there exist several other methods on how to retrieve odometry. For example, *visual odometry* [27] determines the travelled distance using camera images. Zhang and Singh propose a method to extract odometry data from LIDAR [43]. Furthermore, one can combine information from different sources using *bayes filters* such as the *Kalman filter* or the *particle filter* in order to further improve the state estimate [38]. For instance, we could fuse the wheel encoder odometry with data obtained from a relatively cheap *inertial measurement unit* (IMU) which measures linear acceleration and rotational velocity [10].

There could be more work into finding a network architecture that may generalize better. For example, one may use *recurrent neural networks* which incorporate information about the past. This could especially help in tight curves where the camera cannot see much of the upcoming section of the track so it's not possible to make reliable trajectory predictions. Using some type of memory, the network would eventually memorize the shape of the track before entering the turn.

Since it's quite demanding to compute the output of the neural network on the odroid, it may be advantageous to add additional hardware to increase the frame rate. For instance, the *Intel Movidius Neural Compute Stick* [1] is a very compact computing unit which is optimized for deep learning applications. It's connected via USB and specified to consume only 1 ampere of current. Therefore, it would be suitable for our use case.

We found out that splitting the dataset into training and validation loss by random sampling seemed to pose problems in our case. Because of large similarities between the images while recording with high frame rates, the validation data becomes very similar to the training data. Consequently, this destroys the meaning behind the validation loss since it should actually reflect how well the model performs on new data. To provide evidence to this hypothesis, instead of random sampling, we used the first 20% of our samples as validation data and the remaining 80% as training data. Figure 23 shows a comparison between both methods by training the same network with the same hyperparameters over 50 epochs. We can see that using random sampling, the training and validation loss is almost exactly the same. However, with non-random sampling, the validation loss is significantly lower than the training loss. Thus the neural network performs considerably worse on non-training data. This example demonstrates that we should carefully choose the training and validation data in order to retain a metric of how well the car will perform on the racing track under untrained conditions. If we then still observe an increase in validation loss, we'll clearly know that the model overfits so we may apply techniques discussed in Section 3.3.4.

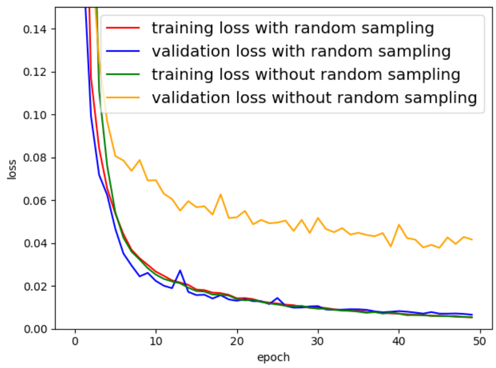


Figure 23: Comparison between training and validation loss using random sampling and non-random sampling to create the training and validation datasets.

In order to unleash the full potential of the trajectory prediction approach, the driving model has to be adapted to take the vehicle's physical properties such as tire friction into account.

6. Conclusion

In this work, we presented two methods for autonomous racing with deep learning. Of course, this problem could have also been solved using traditional image processing techniques including contour and edge detection. However, this requires lots of hand-written rules and tweaking to figure out where the lanes are, how to filter noise such as reflections and how to steer accordingly. It's far more elegant to create a system that learns this processing by itself.

Unfortunately, the end-to-end approach turned out to be suboptimal when applied to racing. Since the neural network tries to mimic the manual driving behavior, the system is limited by human capabilities and does not scale well to higher velocities. Our trajectory prediction approach overcomes this problem by using deep learning only as a subcomponent to predict the vehicle's optimal future path. This provides the flexibility to incorporate an arbitrarily complex driving model to transform a trajectory into steering commands. In addition, collecting training data becomes much easier as it's just a process of guiding the vehicle along the desired path on the track, regardless of the speed.

Due to strange problems with our wheel encoders, we were not able to show the full potential of this approach. Even the best neural network architecture is worthless without good training data. Better sensors and more advanced algorithms are required in order to create quality localization data to train the model with. We belief that if this had been the case, we would have performed considerably better at the Deep Berlin Robocars challenge.

Still there remain some open questions regarding deep learning aspects. Despite the fact that our adaption of *Nvidia*'s PilotNet architecture worked in principle, it is unclear whether there exist better architectures and hyperparameter configurations for our particular application. For example, we don't know what would have happened if we had used fewer convolutional layers. Overly complex models are known to generalize worse.

Further investigations will be needed to answer the question why almost every car performed poorly on lighting conditions it hasn't been trained on. It's not clear whether this was caused by problems in the network architecture or in the training data. Therefore, future work has to concentrate on the generalization capabilities of neural networks. Nevertheless, visualization techniques such as saliency maps already help to detect the existence of such problems.

The results achieved by applying deep learning to image classification and object detection make it a promising technique to design future autonomous systems. However, this thesis showed that neural networks are not able to provide complete end-to-end solutions for all applications yet. Until now, we should rather consider them a powerful tool to help solving complex problems as part of a processing pipeline.

References

[1] Intel neural compute stick webpage. URL https://software.intel.com/en-us/neural-compute-stick.

[2] The imaging source dfm 22buc03-ml webpage. URL https://www.theimagingsource.com/products/board-cameras/usb-2.0-color/dfm22buc03ml/.

[3] Donkey car homepage. URL http://donkeycar.com.

[4] *Semesterprojekt Hochautomatisiertes Fahren Dokumentation*, 2018. URL https://www2.informatik.hu-berlin.de/~hs/Lehre/2017-WS_SP-HAF/20180524_SPHAF-Dokumentation.pdf.

[5] S. Albarqouni, C. Baur, F. Achilles, V. Belagiannis, S. Demirci, and N. Navab. Aggnet: Deep learning from crowds for mitosis detection in breast cancer histology images. *IEEE Transactions on Medical Imaging*, 35(5):1313–1321, May 2016. ISSN 0278-0062. doi: 10.1109/TMI.2016.2528120.

[6] K. H. Ang, G. Chong, and Y. Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13:559–576, 2005.

[7] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. D. Jackel, and U. Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *CoRR*, abs/1704.07911, 2017. URL http://arxiv.org/abs/1704.07911.

[8] D. M. Bradley. Odometry : Calibration and error modeling. 2005.

[9] M. Burgess. Roborace is building a 300kph ai supercar – no driver required, March 2018. URL https://www.wired.co.uk/article/roborace-car-formyla-e-robocar-uk-autonomous-race-denis-sverdlov.

[10] B.-S. Cho, W.-s. Moon, W.-J. Seo, and K. Baek. A dead reckoning localization system for mobile robots using inertial sensors and wheel revolution encoding. *Journal of Mechanical Science and Technology*, 25, 11 2011. doi: 10.1007/s12206-011-0805-1.

[11] R. Collobert and S. Bengio. Links between perceptrons, mlps and svms. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 23–, New York, NY, USA, 2004. ACM. ISBN 1-58113-838-5. doi: 10.1145/1015330.1015415. URL http://doi.acm.org/10.1145/1015330.1015415.

[12] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York,

NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390177. URL http://doi.acm.org/10.1145/1390156.1390177.

[13] M. de la Iglesia Valls, H. F. C. Hendrikx, V. Reijgwart, F. V. Meier, I. Sa, R. Dubé, A. R. Gawel, M. Bürki, and R. Siegwart. Design of an autonomous racecar: Perception, state estimation and system integration. *CoRR*, abs/1804.03252, 2018. URL http://arxiv.org/abs/1804.03252.

[14] emo berlin.de. Abschlussrennen der deep berlin robocars challenge, September 2018. URL https://www.emo-berlin.de/de/newsarchiv/news/abschlussrennen-der-deep-berlin-robocars-challenge/.

[15] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] J. B. H. II and A. H. Waibel. A novel objective function for improved phoneme recognition using time-delay neural networks. *IEEE Trans. Neural Networks*, 1 (2):216–228, 1990. doi: 10.1109/72.80233. URL https://doi.org/10.1109/72.80233.

[17] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[18] S. Jain. An overview of regularization techniques in deep learning (with python code), April 2018. URL https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/.

[19] L. F. Johann Borenstein. Umbmark: a benchmark test for measuring odometry errors in mobile robots, 1995. URL https://doi.org/10.1117/12.228968.

[20] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

[22] F.-F. Li, J. Johnson, and S. Young. Lecture 3: Loss functions and optimization. April 2017. URL http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture3.pdf.

[23] F.-F. Li, J. Johnson, and S. Young. Lecture 6: Training neural networks, part 1. April 2017. URL http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf.

[24] F.-F. Li, J. Johnson, and S. Young. Lecture 7: Training neural networks, part 2. April 2017. URL `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture7.pdf`.

[25] A. Liniger, A. Domahidi, and M. Morari. Optimization-based autonomous racing of 1:43 scale rc cars. *Optimal Control Applications and Methods*, 36:628–647, 09 2015. doi: 10.1002/oca.2123.

[26] K. Majek. Self-racing cars 2017 – udacity team soulless, 2017. URL `https://karolmajek.pl/self-racing-cars-2017-udacity-team-soulless/`.

[27] D. Nistér, O. Naroditsky, and J. R. Bergen. Visual odometry. *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, 1:I–I, 2004.

[28] B. Póczos and R. Tibshirani. Convex optimization cmu-10725 quasi newton methods. URL `http://www.stat.cmu.edu/~ryantibs/convexopt-F13/lectures/11-QuasiNewton.pdf`.

[29] M. Quigley, K. Conley, B. P Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y Ng. Ros: an open-source robot operating system, 01 2009.

[30] U. Rosolia, A. Carvalho, and F. Borrelli. Autonomous racing using learning model predictive control. In *2017 American Control Conference (ACC)*, pages 5115–5120, May 2017. doi: 10.23919/ACC.2017.7963748.

[31] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *CoRR*, abs/1704.02532, 2017.

[32] D. Shapiro. Go, autonomous speed racer, go! nvidia drive px 2 to power world's first robotic motorsports competition, April 2016. URL `https://blogs.nvidia.com/blog/2016/04/05/roborace/`.

[33] D. Shapiro. Self-racing cars kick off first autonomous vehicle track day, June 2016. URL `https://blogs.nvidia.com/blog/2016/06/03/autonomous-vehicles/`.

[34] S. Sharma. Activation functions: Neural networks, Sep 2017. URL `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

[35] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013. URL `http://arxiv.org/abs/1312.6034`.

[36] A. Simpson. Self-driving car steering angle prediction based on image recognition. 2017.

[37] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. URL `http://arxiv.org/abs/1412.6806`.

[38] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*.

[39] A. S. Walia. Activation functions and it's types-which is better?, May 2017. URL `https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f`.

[40] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng. End-to-end text recognition with convolutional neural networks. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 3304–3308, Nov 2012.

[41] P. Werbos and P. J. (Paul John. Beyond regression : new tools for prediction and analysis in the behavioral sciences /. 01 1974.

[42] Y.-x. Yuan. Step-sizes for the gradient method. 1999.

[43] J. Zhang and S. Singh. Loam: Lidar odometry and mapping in real-time. In *Robotics: Science and Systems Conference*, July 2014.

[44] J. Zhong. Pid controller tuning: A short tutorial. 2006.

A. Code Snippets

A.1. End-to-End Model Implementation using Keras

```
def build_end_to_end_model():
    img_in = Input(shape=(100, 200, 3)), name="img_in")
    x = img_in
    x = BatchNormalization(axis=3)(x)
    x = Convolution2D(filters=24, kernel_size=(5, 5),
                      strides=(2, 2), activation="relu")(x)
    x = Convolution2D(filters=36, kernel_size=(5, 5),
                      strides=(2, 2), activation="relu")(x)
    x = Convolution2D(filters=48, kernel_size=(5, 5),
                      strides=(2, 2), activation="relu")(x)
    x = Convolution2D(filters=64, kernel_size=(3, 3),
                      strides=(1, 1), activation="relu")(x)
    x = Convolution2D(filters=64, kernel_size=(3, 3),
                      strides=(1, 1), activation="relu")(x)
    x = Flatten()(x)
    x = Dense(units=100, activation="relu")(x)
    x = Dense(units=50, activation="relu")(x)
    x = Dense(units=10, activation="relu")(x)

    angle_out = Dense(units=1, name="angle_out",
                      activation="linear")(x)
    throttle_out = Dense(units=1, name="throttle_out",
                         activation="linear")(x)

    model = Model(inputs=[img_in], outputs=[angle_out, throttle_out])
    model.compile(optimizer="adam",
                  loss={"angle_out": "mean_squared_error",
                        "throttle_out": "mean_squared_error"})
    return model
```

A.2. Trajectory Prediction Model Implementation using Keras

```python
def build_trajectory_prediction_model(args):
    img_in = Input(shape=(100, 200, 3)), name="img_in")
    x = img_in
    x = BatchNormalization(axis=3)(x)
    x = Convolution2D(filters=24, kernel_size=(5, 5),
                      strides=(2, 2), activation="relu")(x)
    x = Convolution2D(filters=36, kernel_size=(5, 5),
                      strides=(2, 2), activation="relu")(x)
    x = Convolution2D(filters=48, kernel_size=(5, 5),
                      strides=(2, 2), activation="relu")(x)
    x = Convolution2D(filters=64, kernel_size=(3, 3),
                      strides=(1, 1), activation="relu")(x)
    x = Convolution2D(filters=64, kernel_size=(3, 3),
                      strides=(1, 1), activation="relu")(x)
    x = Flatten()(x)
    x = Dense(units=100, activation="relu")(x)
    x = Dense(units=50, activation="relu")(x)
    x = Dense(units=10, activation="relu")(x)

    loss = {}
    outputs = []
    for i in range(3):
        name_x = "coord_x_out_" + str(i)
        name_y = "coord_y_out_" + str(i)
        coord_x_out = Dense(1, name=name_x, activation="linear")(x)
        coord_y_out = Dense(1, name=name_y, activation="linear")(x)
        loss[name_x] = "mean_squared_error"
        loss[name_y] = "mean_squared_error"
        outputs.append(coord_x_out)
        outputs.append(coord_y_out)

    model = Model(inputs=[img_in], outputs=outputs)
    model.compile(optimizer="adam", loss=loss)
    return model
```

B. Trajectory Prediction Approach Parameters

parameter	value
c_l	3.1697932657906436
c_r	3.189793265790643
T	305
$\mathcal{D}$	$(0.6, 1.2, 1.8)$
b	0.32
ν_1	2.5
ν_2	1.5
χ	0.3
K_p	$5 \cdot 10^{-3}$
K_i	0
K_d	$1 \cdot 10^{-3}$

C. Source Code

C.1. Odroid

C.1.1. C++ Source Code

AsyncCapture.h

```
//
// Created by philipp on 17.08.18.
//

#ifndef CAR_ASYNCCAPTURE_H
#define CAR_ASYNCCAPTURE_H

#include <opencv2/opencv.hpp>
#include <opencv2/core/mat.hpp>

#include <gst/gst.h>
#include <gst/app/gstappsink.h>

namespace car
{

class AsyncCapture
{
public:
  using OnFrameCapturedCallback = std::function<void(cv::Mat & frame)>;

  explicit AsyncCapture(int frameWidth = 744, int frameHeight = 480, int frameRate
      ↪ = 60);

  ~AsyncCapture();

  AsyncCapture(const AsyncCapture &) = delete;

  AsyncCapture(AsyncCapture && other) noexcept;

  AsyncCapture & operator=(const AsyncCapture &) = delete;

  AsyncCapture & operator=(AsyncCapture && other) noexcept;

  void start(const OnFrameCapturedCallback & callback);

  void stop();

private:
  int frameWidth, frameHeight, frameRate;

  OnFrameCapturedCallback onFrameCapturedCallback;

  GstElement * pipeline = nullptr;
  GstElement * source = nullptr;
  GstElement * inputCaps = nullptr;
  GstElement * queue = nullptr;
  GstElement * bayer2rgb = nullptr;
  GstElement * convert = nullptr;
  GstElement * appsink = nullptr;

  static GstFlowReturn appsinkFrameCallback(GstAppSink * appsink, gpointer data);

  void createPipeline();

```

```
  void ensureReadyState();
};

}

#endif //CAR_ASYNCCAPTURE_H
```

AsyncCapture.cpp

```
//
// Created by philipp on 17.08.18.
//

#include "camera/AsyncCapture.h"

namespace car
{

AsyncCapture::AsyncCapture(int frameWidth, int frameHeight, int frameRate)
  : frameWidth(frameWidth), frameHeight(frameHeight), frameRate(frameRate)
{
  if (!gst_is_initialized())
    gst_init(nullptr, nullptr);

  createPipeline();
  ensureReadyState();
}

AsyncCapture::~AsyncCapture()
{
  stop();
  gst_object_unref(pipeline);
}

AsyncCapture::AsyncCapture(AsyncCapture && other) noexcept
  : pipeline(other.pipeline)
{
  other.pipeline = nullptr;
}

AsyncCapture & AsyncCapture::operator=(AsyncCapture && other) noexcept
{
  gst_object_unref(pipeline);
  pipeline = other.pipeline;
  other.pipeline = nullptr;
  return *this;
}

void AsyncCapture::start(const AsyncCapture::OnFrameCapturedCallback & callback)
{
  onFrameCapturedCallback = callback;
  GstAppSinkCallbacks callbacks = {nullptr, nullptr, appsinkFrameCallback};
  gst_app_sink_set_callbacks(GST_APP_SINK(appsink), &callbacks, this, nullptr);

  gst_element_set_state(pipeline, GST_STATE_PLAYING);
  gst_element_get_state(pipeline, nullptr, nullptr, GST_CLOCK_TIME_NONE);
}

void AsyncCapture::stop()
{
  gst_element_set_state(pipeline, GST_STATE_NULL);
  gst_element_get_state(pipeline, nullptr, nullptr, GST_CLOCK_TIME_NONE);
}

GstFlowReturn AsyncCapture::appsinkFrameCallback(GstAppSink * appsink, gpointer data)
```

```cpp
{
  auto this_ = static_cast<AsyncCapture *>(data);
  auto sample = gst_app_sink_pull_sample(appsink);
  bool success{false};
  if (sample)
  {
    auto buffer = gst_sample_get_buffer(sample);
    auto caps = gst_sample_get_caps(sample);
    auto structure = gst_caps_get_structure(caps, 0);
    int width, height;
    gst_structure_get_int(structure, "width", &width);
    gst_structure_get_int(structure, "height", &height);
    GstMapInfo info;
    if (gst_buffer_map(buffer, &info, GST_MAP_READ))
    {
      const auto frameData = info.data;
      // 'data' now contains a pointer to readable image data
      cv::Mat frame{height, width, CV_8UC3, frameData};
      this_->onFrameCapturedCallback(frame);
      // unmap after use
      gst_buffer_unmap(buffer, &info);
      success = true;
    }
  }
  gst_sample_unref(sample);
  if (!success)
  {
    cv::Mat emptyFrame{};
    this_->onFrameCapturedCallback(emptyFrame);
  }
  return GST_FLOW_OK;
}

void AsyncCapture::createPipeline()
{
  pipeline = gst_pipeline_new("pipeline");

  source = gst_element_factory_make("tcambin", nullptr);
  if (!source)
    throw std::runtime_error("'tcambin' could not be initialized! Check tiscamera installation");

  inputCaps = gst_element_factory_make("capsfilter", nullptr);
  std::ostringstream oss;
  oss << "video/x-bayer, width=" << frameWidth
      << ", height=" << frameHeight
      << ", framerate=" << frameRate << "/1";
  g_object_set(inputCaps, "caps", gst_caps_from_string(oss.str().c_str()), nullptr);

  queue = gst_element_factory_make("queue", nullptr);
  g_object_set(queue, "leaky", true, "max-size-buffers", 2, nullptr);

  bayer2rgb = gst_element_factory_make("bayer2rgb", nullptr);

  convert = gst_element_factory_make("videoconvert", nullptr);

  appsink = gst_element_factory_make("appsink", nullptr);
  g_object_set(appsink, "caps", gst_caps_from_string("video/x-raw, format=BGR"), nullptr);

  assert(pipeline && source && inputCaps && queue && bayer2rgb && convert && appsink);

  gst_bin_add_many(
    GST_BIN(pipeline),
    source, inputCaps, queue, bayer2rgb, convert, appsink, nullptr);
```

```
  auto result = gst_element_link_many(
    source, inputCaps, queue, bayer2rgb, convert, appsink, nullptr);
  assert(result);
}

void AsyncCapture::ensureReadyState()
{
  GstState state;
  if ((gst_element_get_state(source, &state, nullptr, GST_CLOCK_TIME_NONE) ==
      ↪ GST_STATE_CHANGE_SUCCESS) &&
      state == GST_STATE_NULL)
  {
    gst_element_set_state(source, GST_STATE_READY);
    gst_element_get_state(source, nullptr, nullptr, GST_CLOCK_TIME_NONE);
  }
}

}
```

Config.h

```
//
// Created by philipp on 09.07.18.
//

#ifndef CAR_CONFIG_H
#define CAR_CONFIG_H

#include "Json.h"

namespace car
{
namespace config
{

std::string directory();

nlohmann::json open();

}
}

#endif //CAR_CONFIG_H
```

Config.cpp

```
//
// Created by philipp on 09.07.18.
//

#include "utils/Config.h"
#include <fstream>

namespace car
{
namespace config
{

std::string directory()
{
  auto userHome = std::string{std::getenv("HOME")};
  return userHome + "/.car";
}
```

```

nlohmann::json open()
{
  auto configPath = directory() + "/config.json";
  std::ifstream file{configPath};
  nlohmann::json j;
  file >> j;
  return j;
}

}
}
```

Recorder.h

```
//
// Created by philipp on 20.06.18.
//

#ifndef CAR_RECORDER_H
#define CAR_RECORDER_H

#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <logging/MessageOStream.h>
#include <atomic>
#include <opencv2/opencv.hpp>
#include <opencv2/core/mat.hpp>
#include <NetworkingLib/Time.h>
#include <camera/AsyncCapture.h>
#include <mutex>

#include "car/WheelTicks.h"
#include "car/Odometry.h"
#include "car/SetThrottle.h"
#include "car/SetAngle.h"
#include "car/UssDistance.h"
#include "car/StopRecording.h"
#include "car/RemoteState.h"

namespace car
{

class Recorder : public nodelet::Nodelet
{
public:
  void onInit() override;

  Recorder();

private:
  ros::NodeHandle nh;

  ros::Subscriber wheelTicksSubscriber;
  ros::Subscriber odometrySubscriber;
  ros::Subscriber setThrottleSubscriber;
  ros::Subscriber setAngleSubscriber;
  ros::Subscriber ussDistanceSubscriber;
  ros::Subscriber stopRecordingSubscriber;
  ros::Subscriber remoteStateSubscriber;

  MessageOStream messageOStream;

  float ticksRL{0};
  float ticksRR{0};
```

```
  std::mutex ticksMutex;

  std::atomic<float> x{0};
  std::atomic<float> y{0};

  std::atomic<float> setThrottle{0.0f};
  std::atomic<float> setAngle{0.0f};
  std::atomic<float> distance{0.0f};

  std::atomic<bool> recording{true};
  std::atomic<bool> recordingSequence{true};

  AsyncCapture capture;
  cv::Mat currFrame;
  std::mutex frameCopyMutex;
  std::atomic<bool> receivedFrame{false};

  std::string recordingsDirectory;
  networking::time::Duration frameTime;

  void wheelTicksCallback(const WheelTicks::ConstPtr msg);

  void odometryCallback(const Odometry::ConstPtr msg);

  void setThrottleCallback(const SetThrottle::ConstPtr msg);

  void setAngleCallback(const SetAngle::ConstPtr msg);

  void ussDistanceCallback(const UssDistance::ConstPtr msg);

  void stopRecordingCallback(const StopRecording::ConstPtr msg);

  void remoteStateCallback(const RemoteState::ConstPtr msg);

  void record();

  std::string createRecordingDirectory();

  void writeCsvRecord(std::ofstream & dataFile, const std::string & imgFilename);

  void readConfig();
};

}

#endif //CAR_RECORDER_H
```

Recorder.cpp

```
//
// Created by philipp on 20.06.18.
//

#include <pluginlib/class_list_macros.h>
#include <thread>
#include "recorder/Recorder.h"
#include <boost/filesystem.hpp>
#include <utils/Config.h>

PLUGINLIB_EXPORT_CLASS(car::Recorder, nodelet::Nodelet);

namespace car
{

Recorder::Recorder()
```

```
  : messageOStream(nh, "recorder")
    , capture(744, 480, 60)
{}

void Recorder::onInit()
{
  messageOStream.write("onInit", "START");

  wheelTicksSubscriber = nh.subscribe("WheelTicks", 1000, &Recorder::wheelTicksCallback, this);
  odometrySubscriber = nh.subscribe("Odometry", 1, &Recorder::odometryCallback, this);
  setThrottleSubscriber = nh.subscribe("SetThrottle", 1, &Recorder::setThrottleCallback, this);
  setAngleSubscriber = nh.subscribe("SetAngle", 1, &Recorder::setAngleCallback, this);
  ussDistanceSubscriber = nh.subscribe("UssDistance", 1, &Recorder::ussDistanceCallback, this);
  stopRecordingSubscriber = nh.subscribe("StopRecording", 1, &Recorder::stopRecordingCallback, this);
  remoteStateSubscriber = nh.subscribe("RemoteState", 1, &Recorder::remoteStateCallback, this);

  readConfig();

  capture.start(
    [&](auto & frame)
    {
      std::lock_guard<std::mutex> lock{frameCopyMutex};
      currFrame = frame.clone();
      receivedFrame = true;
    });

  std::thread recorderThread{[&] { this->record(); }};
  recorderThread.detach();

  messageOStream.write("onInit", "END");
}

void Recorder::wheelTicksCallback(const WheelTicks::ConstPtr msg)
{
  std::lock_guard<std::mutex> lock{ticksMutex};
  ticksRL += msg->rearLeftTicks;
  ticksRR += msg->rearRightTicks;
}

void Recorder::odometryCallback(const Odometry::ConstPtr msg)
{
  x = msg->xDistance;
  y = msg->yDistance;
}

void Recorder::setThrottleCallback(const SetThrottle::ConstPtr msg)
{
  setThrottle = msg->throttle;
}

void Recorder::setAngleCallback(const SetAngle::ConstPtr msg)
{
  setAngle = msg->angle;
}

void Recorder::ussDistanceCallback(const UssDistance::ConstPtr msg)
{
  distance = msg->distance;
}
```

```
void Recorder::stopRecordingCallback(const StopRecording::ConstPtr msg)
{
  recording = false;
}

void Recorder::remoteStateCallback(const RemoteState::ConstPtr msg)
{
  if (msg->id == 0)
    recordingSequence = msg->value != 0 ? true : false;
}

void Recorder::record()
{
  auto dataDirName = createRecordingDirectory();

  std::ostringstream oss;
  oss << dataDirName << "/data.csv";
  std::ofstream dataFile{oss.str()};

  dataFile << "ticksRL,ticksRR,x,y,setThrottle,setAngle,distance,recording,img\n";

  auto last = networking::time::now();

  for (std::size_t counter = 0; recording; counter++)
  {
    // keep constant frame rate
    auto now = networking::time::now();
    auto lastFrameTime = now - last;
    if (lastFrameTime < frameTime)
      std::this_thread::sleep_for(frameTime - lastFrameTime);
    last = networking::time::now();

    while (!receivedFrame);
    cv::Mat frame{};
    {
      std::lock_guard<std::mutex> lock{frameCopyMutex};
      frame = currFrame;
    }
    if (frame.empty())
      continue;

    // save image
    oss = std::ostringstream{};
    oss << "img-" << counter << ".jpg";
    auto imgFilename = oss.str();
    oss = std::ostringstream{};
    oss << dataDirName << "/" << imgFilename;
    auto imgPath = oss.str();
    cv::imwrite(imgPath, frame);

    writeCsvRecord(dataFile, imgFilename);
  }

  messageOStream.write("MSG", "stopped recording");
}

std::string Recorder::createRecordingDirectory()
{
  std::ostringstream oss;
  auto time = std::time(nullptr);
  oss << recordingsDirectory << "/data-" << std::put_time(std::localtime(&time), "%Y
    ↪ -%m-%d-%H-%M-%S");
  auto dirName = oss.str();
  boost::filesystem::path dir{dirName};
  boost::filesystem::create_directory(dir);
```

```
  return dirName;
}

void Recorder::writeCsvRecord(std::ofstream & dataFile, const std::string &
    ↪ imgFilename)
{
  float ticksRLCopy{0}, ticksRRCopy{0};
  {
    std::lock_guard<std::mutex> lock{ticksMutex};
    ticksRLCopy = ticksRL;
    ticksRRCopy = ticksRR;
    ticksRL = 0;
    ticksRR = 0;
  }

  dataFile << ticksRLCopy << ","
       << ticksRRCopy << ","
       << x.load() << ","
         << y.load() << ","
           << setThrottle.load() << ","
           << setAngle.load() << ","
           << distance.load() << ","
       << recordingSequence.load() << ","
           << imgFilename << "\n";
}

void Recorder::readConfig()
{
  auto j = config::open();

  if (j.count("recordingsDirectory") > 0)
    recordingsDirectory = j.at("recordingsDirectory").get<std::string>();
  else
    recordingsDirectory = config::directory() + "/recordings";

  int fps = 60;
  if (j.count("recordingFPS") > 0)
    fps = j.at("recordingFPS").get<int>();
  frameTime = std::chrono::nanoseconds(1000000000 / fps);
}

}
```

RemoteControl.h

```
//
// Created by philipp on 20.06.18.
//

#ifndef CAR_REMOTECONTROL_H
#define CAR_REMOTECONTROL_H

#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <logging/MessageOStream.h>

#include "car/RemoteAngle.h"
#include "car/RemoteThrottle.h"

namespace car
{

class RemoteControl : public nodelet::Nodelet
{
public:
```

```
  void onInit() override;

  RemoteControl();

private:
  ros::NodeHandle nh;

  ros::Subscriber remoteThrottleSubscriber;
  ros::Subscriber remoteAngleSubscriber;

  ros::Publisher setThrottlePublisher;
  ros::Publisher setAnglePublisher;

  MessageOStream messageOStream;

  bool controlThrottle{false};
  bool controlSteeringAngle{false};

  void remoteThrottleCallback(const RemoteThrottle::ConstPtr & msg);

  void remoteAngleCallback(const RemoteAngle::ConstPtr & msg);

  void readConfig();
};

}

#endif //CAR_REMOTECONTROL_H
```

RemoteControl.cpp

```
//
// Created by philipp on 20.06.18.
//

#include <pluginlib/class_list_macros.h>
#include <utils/Config.h>
#include "remoteControl/RemoteControl.h"
#include "car/SetThrottle.h"
#include "car/SetAngle.h"

PLUGINLIB_EXPORT_CLASS(car::RemoteControl, nodelet::Nodelet);

namespace car
{

RemoteControl::RemoteControl()
  : messageOStream(nh, "remoteControl")
{}

void RemoteControl::onInit()
{
  messageOStream.write("onInit", "START");

  readConfig();

  if (controlThrottle)
  {
    remoteThrottleSubscriber = nh.subscribe("RemoteThrottle", 1, &RemoteControl::
        ↪ remoteThrottleCallback, this);
    setThrottlePublisher = nh.advertise<SetThrottle>("SetThrottle", 1);
  }

  if (controlSteeringAngle)
  {
```

```cpp
    remoteAngleSubscriber = nh.subscribe("RemoteAngle", 1, &RemoteControl::
        ↪ remoteAngleCallback, this);
    setAnglePublisher = nh.advertise<SetAngle>("SetAngle", 1);
  }

  messageOStream.write("onInit", "END");
}

void RemoteControl::remoteThrottleCallback(const RemoteThrottle::ConstPtr & msg)
{
  SetThrottle throttleMsg;
  throttleMsg.throttle = msg->throttle;
  setThrottlePublisher.publish(throttleMsg);
}

void RemoteControl::remoteAngleCallback(const RemoteAngle::ConstPtr & msg)
{
  SetAngle angleMsg;
  angleMsg.angle = msg->angle;
  setAnglePublisher.publish(angleMsg);
}

void RemoteControl::readConfig()
{
  auto j = config::open();
  if (j.count("enableThrottleRemoteControl") > 0)
    controlThrottle = j.at("enableThrottleRemoteControl").get<bool>();
  if (j.count("enableSteeringAngleRemoteControl") > 0)
    controlSteeringAngle = j.at("enableSteeringAngleRemoteControl").get<bool>();
}

}
```

RemoteControlMessage.h

```cpp
//
// Created by philipp on 21.06.18.
//

#ifndef CAR_REMOTECONTROLMESSAGE_H
#define CAR_REMOTECONTROLMESSAGE_H

#include "NetworkingLib/Message.h"
#include <boost/algorithm/string.hpp>
#include <utility>

namespace car
{

struct RemoteControlMessage
{
  RemoteControlMessage() = default;

  RemoteControlMessage(std::string key, std::string value)
    : key(std::move(key)), value(std::move(value))
  {}

  std::string key;
  std::string value;
};

}

namespace networking
{
```

```cpp
namespace message
{

template<>
struct Decoder<car::RemoteControlMessage>
{
  void operator()(car::RemoteControlMessage & message, const std::string & data)
      const
  {
    std::vector<std::string> split;
    boost::split(split, data, [](char c) {return c == '=';});
    message.key = split.at(0);
    message.value = split.at(1);
  };
};

}
}

#endif //CAR_REMOTECONTROLMESSAGE_H
```

RemoteControlMessageReceiver.h

```cpp
//
// Created by philipp on 23.08.18.
//

#ifndef CAR_REMOTECONTROLMESSAGERECEIVER_H
#define CAR_REMOTECONTROLMESSAGERECEIVER_H

#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <logging/MessageOStream.h>

#include "NetworkingLib/DatagramReceiver.h"

#include "RemoteControlMessage.h"

namespace car
{

class RemoteControlMessageReceiver : public nodelet::Nodelet
{
public:
  void onInit() override;

  RemoteControlMessageReceiver();

private:
  ros::NodeHandle nh;

  ros::Publisher remoteThrottlePublisher;
  ros::Publisher remoteAnglePublisher;
  ros::Publisher remoteStatePublisher;

  MessageOStream messageOStream;

  networking::Networking net;

  networking::message::DatagramReceiver<RemoteControlMessage>::Ptr messageReceiver;

  void receiveMessages();

  void handleMessage(const RemoteControlMessage & message);
};
```

```
}

#endif //CAR_REMOTECONTROLMESSAGERECEIVER_H
```

RemoteControlMessageReceiver.cpp

```
//
// Created by philipp on 23.08.18.
//

#include <pluginlib/class_list_macros.h>
#include "remoteControl/RemoteControlMessageReceiver.h"
#include <boost/algorithm/string.hpp>

#include "car/RemoteThrottle.h"
#include "car/RemoteAngle.h"
#include "car/RemoteState.h"

PLUGINLIB_EXPORT_CLASS(car::RemoteControlMessageReceiver, nodelet::Nodelet);

namespace car
{

RemoteControlMessageReceiver::RemoteControlMessageReceiver()
  : messageOStream(nh, "remoteControlMessageReceiver")
{}

void RemoteControlMessageReceiver::onInit()
{
  messageOStream.write("onInit", "START");

  remoteThrottlePublisher = nh.advertise<RemoteThrottle>("RemoteThrottle", 1);
  remoteAnglePublisher = nh.advertise<RemoteAngle>("RemoteAngle", 1);
  remoteStatePublisher = nh.advertise<RemoteState>("RemoteState", 1);

  messageReceiver = networking::message::DatagramReceiver<RemoteControlMessage>::
      ↪ create(net, 10288);
  receiveMessages();

  messageOStream.write("onInit", "START");
}

void RemoteControlMessageReceiver::receiveMessages()
{
  messageReceiver->asyncReceive(
    networking::time::Duration::max(),
    [&](const auto & error, const auto & message, const auto & host, auto port)
    {
      if (error)
        return;
      this->handleMessage(message);
      this->receiveMessages();
    });
}

void RemoteControlMessageReceiver::handleMessage(const RemoteControlMessage & message
    ↪ )
{
  if (message.key == "angle")
  {
    RemoteAngle angleMsg;
    angleMsg.angle = std::stof(message.value);
    remoteAnglePublisher.publish(angleMsg);
  }
```

```
  else if (message.key == "speed")
  {
    RemoteThrottle throttleMsg;
    throttleMsg.throttle = std::stof(message.value);
    remoteThrottlePublisher.publish(throttleMsg);
  }
  else if (message.key == "state")
  {
    RemoteState remoteState;
    std::vector<std::string> split;
    boost::split(split, message.value, [](char c) { return c == ','; });
    remoteState.id = std::stoi(split[0]);
    remoteState.value = std::stoi(split[1]);
    remoteStatePublisher.publish(remoteState);
  }
}

}
```

Stm.h

```
#ifndef ENVIRONMENT_H
#define ENVIRONMENT_H

#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <logging/MessageOStream.h>

#include "NetworkingLib/Networking.h"
#include "VeloxProtocolLib/Connection.h"

#include "car/SetAngle.h"
#include "car/SetThrottle.h"

namespace car
{

class Stm : public nodelet::Nodelet
{
public:
    void onInit() override;

    Stm();

private:
    ros::NodeHandle nh;

    ros::Publisher odometryPublisher;
  ros::Publisher wheelTicksPublisher;

  ros::Subscriber setThrottleSubscriber;
  ros::Subscriber setAngleSubscriber;

  MessageOStream messageOStream;

  networking::Networking net;
  veloxProtocol::Connection::Ptr veloxConnection;

  float throttleGain{0.0};

  void readConfig();

    void onStmDataReceived();

  void setThrottleCallback(const SetThrottle::ConstPtr & msg);
```

```

  void setAngleCallback(const SetAngle::ConstPtr & msg);
};

}
#endif
```

Stm.cpp

```
#include <pluginlib/class_list_macros.h>
#include <ros/ros.h>
#include <utils/Config.h>

#include "stm/Stm.h"
#include "car/Odometry.h"
#include "car/WheelTicks.h"

PLUGINLIB_EXPORT_CLASS(car::Stm, nodelet::Nodelet);

namespace car
{

Stm::Stm()
    : messageOStream(nh, "Stm")
{}

void Stm::onInit()
{
    using namespace std::chrono_literals;

    messageOStream.write("onInit", "START");

  readConfig();

    odometryPublisher = nh.advertise<Odometry>("Odometry", 1);
  wheelTicksPublisher = nh.advertise<WheelTicks>("WheelTicks", 1000);

    setThrottleSubscriber = nh.subscribe("SetThrottle", 1, &Stm::setThrottleCallback
        , this);
    setAngleSubscriber = nh.subscribe("SetAngle", 1, &Stm::setAngleCallback, this);

    veloxConnection = veloxProtocol::Connection::create(net);
    veloxConnection->open(
        "/dev/ttySAC0",
        [this]
        { onStmDataReceived(); },
        [this]
        { messageOStream.write("ERROR", "UART was closed"); });

    messageOStream.write("onInit", "END");
}

void Stm::onStmDataReceived()
{
    auto stmOdometry = veloxConnection->getOdometry().get();
    Odometry odometryMsg;
    odometryMsg.speed = stmOdometry.speed;
    odometryMsg.xDistance = stmOdometry.xDistance;
    odometryMsg.yDistance = stmOdometry.yDistance;
    odometryMsg.yawAngle = stmOdometry.yawAngle;
  odometryMsg.steeringAngle = stmOdometry.steeringAngle;
    odometryPublisher.publish(odometryMsg);

  WheelTicks wheelTicksMsg;
  wheelTicksMsg.rearLeftTicks = (uint32_t) stmOdometry.yawAngle;
```

```
  wheelTicksMsg.rearRightTicks = (uint32_t) stmOdometry.steeringAngle;
  // we treat the 'steeringAngleTimestamp' value as the delta time in us between the last ticks update
  wheelTicksMsg.deltaTime = (double) stmOdometry.steeringAngleTimestamp * 1e-6;
  wheelTicksPublisher.publish(wheelTicksMsg);
}

void Stm::setThrottleCallback(const SetThrottle::ConstPtr & msg)
{
  // float throttle = std::max(std::min(msg->throttle * throttleGain, 1.0f), -1.0f);
    float throttle = msg->throttle * throttleGain;
    veloxConnection->setSpeed(throttle);
}

void Stm::setAngleCallback(const SetAngle::ConstPtr & msg)
{
    veloxConnection->setSteeringAngle(msg->angle);
}

void Stm::readConfig()
{
  auto j = config::open();
  throttleGain = j.at("throttleGain");
}

}
```

C.1.2. Python Source Code

autonom_drive.py

```
#!/usr/bin/env python
import rospy
from utils import *
from keras.models import load_model
from keras import backend as K
from car.msg import RemoteThrottle, WheelTicks, SetAngle, SetThrottle
from threading import Thread, Lock
import cv2
import numpy as np
import signal
import datetime
from image_stream import ImageStream
import motion
import odometry
from parameters import *


class AutonomDrive:
    def __init__(self):
        self.running = True
        self.contrast = 1.0
        self.remote_throttle = 0.0
        self.speed = 0.0
        self.curr_angle_estimator = motion.CurrentAngleEstimator(ANGLE_CHANGE_RATE, confidence_duration=ANGLE_ESTIMATOR_CONFIDENCE_DURATION)
        self.remote_throttle_sub = rospy.Subscriber("RemoteThrottle", RemoteThrottle, self.remote_throttle_callback, queue_size=1)
        self.wheel_ticks_sub = rospy.Subscriber("WheelTicks", WheelTicks, self.wheel_ticks_callback, queue_size=1)
        self.set_angle_pub = rospy.Publisher("SetAngle", SetAngle, queue_size=1)
        self.set_throttle_pub = rospy.Publisher("SetThrottle", SetThrottle, queue_size=1)
```

```
        self.model_path = None
        self.read_config()
        self.image_stream = ImageStream(self.contrast)
        self.model = load_model(self.model_path)
        rospy.init_node("autonom_drive", log_level=rospy.INFO)
        signal.signal(signal.SIGINT, self.signal_handler)

    def read_config(self):
        config = open_config()

        if "model" not in config:
            raise AttributeError("attribute 'model' is missing in config.json")
        model_filename = str(config["model"])
        models_dir = config["modelsDirectory"] if "modelsDirectory" in config else os
            ↪ .path.join(get_config_dir(), "models")
        self.model_path = os.path.join(models_dir, model_filename)

        if "numTensorflowThreads" in config:
            num_threads = int(config["numTensorflowThreads"])
            K.set_session(K.tf.Session(config=K.tf.ConfigProto(
                ↪ intra_op_parallelism_threads=num_threads,
                ↪ inter_op_parallelism_threads=num_threads)))

        if "contrast" in config:
            self.contrast = float(config["contrast"])

    def signal_handler(self, sig, frame):
        self.running = False
        self.image_stream.stop()

    def remote_throttle_callback(self, msg):
        self.remote_throttle = msg.throttle

    def wheel_ticks_callback(self, msg):
        self.speed = odometry.speed(msg.rearLeftTicks, msg.rearRightTicks, msg.
            ↪ deltaTime, DEFAULT_ODOMETRY_PARAMS)

    def publish(self, angle, throttle):
        set_angle_msg = SetAngle()
        set_angle_msg.angle = angle
        self.set_angle_pub.publish(set_angle_msg)

        set_throttle_msg = SetThrottle()
        set_throttle_msg.throttle = throttle
        self.set_throttle_pub.publish(set_throttle_msg)

    def run(self):
        self.image_stream.start()
        while self.running:
            if not self.image_stream.running:
                break

            image = self.image_stream.read()
            image = np.expand_dims(image, axis=0)

            prediction_start_time = datetime.datetime.now()

            prediction = self.model.predict(image, batch_size=1)
            coords = np.array([(float(prediction[2 * i]), float(prediction[2 * i
                ↪  + 1])) for i in range(int(len(prediction)/ 2))])

            angle = motion.circular_regression(coords[0:2], ANGLES, WHEEL_BASE)
            speed = 1.5 if abs(coords[-1][0]) > 0.3 else 2.5

            throttle = speed * self.remote_throttle
            self.publish(np.rad2deg(angle), throttle)
```

```

            now = datetime.datetime.now()
            prediction_duration = now - prediction_start_time
            rospy.loginfo("\nprediction duration: %f\nframe processing duration: %f\n\n", \
                    prediction_duration.total_seconds(), self.image_stream.last_frametime)


if __name__ == "__main__":
    try:
        drive = AutonomDrive()
        drive.run()
    except rospy.ROSInterruptException:
        pass
```

image_stream.py

```
import cv2
from threading import Thread
import gi
import numpy as np
import numexpr as ne
import datetime
from utils import *
import sys

gi.require_version("Tcam", "0.1")
gi.require_version("Gst", "1.0")

from gi.repository import Tcam, Gst, GLib


class ImageStream:
    def __init__(self, contrast, crop_top=CROP_TOP, scale_width=IMAGE_WIDTH, scale_height=IMAGE_HEIGHT, frame_rate=60):
        Gst.init(sys.argv)  # init gstreamer
        self.contrast = contrast
        self.crop_top = crop_top
        self.scale_width = scale_width
        self.scale_height = scale_height
        self.frame_rate = frame_rate
        self.frame = None
        self.running = False
        self.init_pipeline()
        self.last_timestamp = datetime.datetime.now()
        self.last_frametime = 0

    def init_pipeline(self):
        source = Gst.ElementFactory.make("tcambin")
        input_caps = Gst.ElementFactory.make("capsfilter")
        input_caps.set_property("caps", Gst.Caps.from_string("video/x-bayer,framerate=" + str(self.frame_rate) + "/1"))
        queue = Gst.ElementFactory.make("queue")
        queue.set_property("leaky", True)
        queue.set_property("max-size-buffers", 2)
        bayer2rgb = Gst.ElementFactory.make("bayer2rgb")
        convert = Gst.ElementFactory.make("videoconvert")
        crop = Gst.ElementFactory.make("videocrop")
        crop.set_property("top", self.crop_top)
        balance = Gst.ElementFactory.make("videobalance")
        balance.set_property("contrast", self.contrast)
        scale = Gst.ElementFactory.make("videoscale")
        output = Gst.ElementFactory.make("appsink")
```

```python
        output.set_property("caps", Gst.Caps.from_string("video/x-raw, format=RGB, width=" + str(self.scale_width) + ", height=" + str(self.scale_height)))
        output.set_property("emit-signals", True)
        output.connect("new-sample", self.on_new_sample)

        self.pipeline = Gst.Pipeline.new()
        self.pipeline.add(source)
        self.pipeline.add(input_caps)
        self.pipeline.add(queue)
        self.pipeline.add(bayer2rgb)
        self.pipeline.add(convert)
        self.pipeline.add(crop)
        self.pipeline.add(balance)
        self.pipeline.add(scale)
        self.pipeline.add(output)

        source.link(input_caps)
        input_caps.link(queue)
        queue.link(bayer2rgb)
        bayer2rgb.link(convert)
        convert.link(crop)
        crop.link(balance)
        balance.link(scale)
        scale.link(output)

        self.pipeline.set_state(Gst.State.READY)
        if self.pipeline.get_state(10 * Gst.SECOND)[0] != Gst.StateChangeReturn.SUCCESS:
            raise RuntimeError("Failed to start video stream.")

    def on_new_sample(self, sink):
        sample = sink.emit("pull-sample")
        if sample:
            buf = sample.get_buffer()

            caps = sample.get_caps()
            width = caps.get_structure(0).get_value("width")
            height = caps.get_structure(0).get_value("height")

            try:
                res, mapinfo = buf.map(Gst.MapFlags.READ)
                img_array = np.asarray(bytearray(mapinfo.data), dtype=np.uint8)

                # Performance-critical section here!
                # Keep this in mind if there's any change in the preprocessing!
                frame = img_array.reshape((height, width, 3))
                frame = cv2.cvtColor(frame, cv2.COLOR_RGB2YUV)
                self.frame = ne.evaluate("frame / 127.5 - 1.0")

                now = datetime.datetime.now()
                self.last_frametime = (now - self.last_timestamp).total_seconds()
                self.last_timestamp = now

            finally:
                buf.unmap(mapinfo)

        return Gst.FlowReturn.OK

    def start(self):
        self.pipeline.set_state(Gst.State.PLAYING)
        while self.frame is None:
            pass
        self.running = True

    def stop(self):
```

```
        self.pipeline.set_state(Gst.State.PAUSED)
        self.pipeline.set_state(Gst.State.READY)
        self.pipeline.set_state(Gst.State.NULL)
        self.frame = None
        self.running = False

    def read(self):
        return self.frame
```

motion.py

```python
# -*- coding: utf-8 -*-
import numpy as np
import odometry
from scipy.spatial.distance import euclidean
from scipy.optimize import minimize
import datetime
from parameters import *


def angle_to_turn_radius(angle, wheelbase):
    return wheelbase / np.tan(angle) if angle != 0.0 else float("inf")


def turn_radius_to_angle(turn_radius, wheelbase):
    return np.arctan(wheelbase / turn_radius) if turn_radius != 0.0 else 0.0


def circular_regression(coords, angles, wheelbase):
    min_err = float("inf")
    final_angle = 0
    for angle in angles:
        err = 0
        if angle == 0.0:
            for c in coords:
                err += c[0] ** 2
        else:
            r = angle_to_turn_radius(angle, wheelbase)
            for c in coords:
                err += (np.sqrt((c[0] - r) ** 2 + c[1] ** 2) - abs(r)) ** 2
        if err < min_err:
            min_err = err
            final_angle = angle
    return final_angle


def circular_regression_numerically(coords, angles, wheelbase):
    def err(angle, coords):
        if angle == 0:
            return np.mean([c[0]**2 for c in coords])
        r = angle_to_turn_radius(angle, WHEEL_BASE)
        return np.mean([(np.sqrt((c[0] - r) ** 2 + c[1] ** 2) - abs(r)) ** 2 for c in coords])

    min_angle = np.deg2rad(-30)
    max_angle = np.deg2rad(30)

    result = minimize(err, 0, (coords), bounds=[(min_angle, max_angle)], method="L-BFGS-B", options={"maxiter": 10})
    return result["x"]


def compute_turns(coords, wheel_base):
    coords1 = coords[0:3]
    angle1 = circular_regression(coords1, ANGLES, wheel_base)
```

```python

    direction2 = odometry.direction(coords, 1)
    odom2 = coords[1:4]
    distance2 = odometry.distance(odom2)
    coords2 = odometry.trajectory_coords(odom2, 0, direction2, 2, distance2 / 2 - 0.00001)
    angle2 = circular_regression(coords2, ANGLES, wheel_base)

    return angle1, angle2


def compute_future_angle(curr_angle, set_angle, angle_change_rate, time):
    angle_direction = 1 if set_angle >= curr_angle else -1
    future_angle = curr_angle + (angle_direction * angle_change_rate * time)
    return min(future_angle, set_angle) if angle_direction == 1 else \
           max(future_angle, set_angle)


class CurrentAngleEstimator:
    def __init__(self, turn_speed, initial_curr_angle=0.0, initial_set_angle=0.0, confidence_duration=0.5):
        """
        confidence_duration: If the angle hasn't changed much for 'confidence_duration' seconds,
                             we can assume that we're driving the set angle.
        """
        self.turn_speed = turn_speed
        self.curr_angle = initial_curr_angle
        self.last_set_angle = initial_set_angle
        self.last_set_timestamp = datetime.datetime.now()
        self.last_hard_turn_timestamp = datetime.datetime.now()
        self.confidence_duration = confidence_duration # s

    def update(self, set_angle):
        now = datetime.datetime.now()
        last_set_angle = self.last_set_angle
        last_set_timestamp = self.last_set_timestamp
        self.last_set_angle = set_angle
        self.last_set_timestamp = now

        if abs(set_angle - self.curr_angle) > DELTA_ANGLE_TOLERANCE:
            self.last_hard_turn_timestamp = now

        if (now - self.last_hard_turn_timestamp).total_seconds() >= self.confidence_duration:
            self.curr_angle = set_angle
            return

        self.curr_angle = compute_future_angle(self.curr_angle, last_set_angle, self.turn_speed, (now - last_set_timestamp).total_seconds())
```

odometry.py

```python
# -*- coding: utf-8 -*-
import numpy as np
from scipy.spatial.distance import euclidean
from scipy.optimize import minimize
import pandas as pd
from matplotlib import pyplot as plt
from matplotlib import cm
from keras.models import load_model
import cv2
import utils
import motion
import datetime
```

```python
from parameters import *


def normalize_vector(x):
    n = np.linalg.norm(x)
    return 0 if n == 0 else x / n


def follow(odom, idx, dist, backwards=False):
    inc = -1 if backwards else 1
    i0 = idx
    i1 = i0 + inc
    if i1 < 0 or i1 >= len(odom):
        raise ValueError("Not enough coordinates!")
    p0 = odom[i0]
    p1 = odom[i1]
    acc_dist = 0
    next_dist = euclidean(p0, p1)

    while acc_dist + next_dist <= dist + FLOAT_TOLERANCE:
        acc_dist += euclidean(p0, p1)
        i0 = i1
        i1 = i1 + inc
        if i1 < 0 or i1 >= len(odom):
            raise ValueError("Not enough coordinates!")
        p0 = odom[i0]
        p1 = odom[i1]
        next_dist = euclidean(p0, p1)

    if acc_dist == dist:
        return p0, i0
    direction = normalize_vector(p1 - p0)
    rest_dist = dist - acc_dist
    v = rest_dist * direction
    return v + p0, i0


def distance(odom):
    if len(odom) < 2:
        raise ValueError("Cannot measure the distance of odom of size < 2.")
    dist = 0
    for i in range(len(odom) - 1):
        dist += euclidean(odom[i], odom[i + 1])
    return dist


def plot_odometry(odom, ax=None, start_idx=0, end_idx=-1, color="gray", strength=1, elements=None):
    x = [coord[0] for coord in odom]
    y = [coord[1] for coord in odom]
    if elements is None:
        return ax.plot(x[start_idx:end_idx], y[start_idx:end_idx], "o", markersize=strength, color=color)[0]
    elements.set_data(x[start_idx:end_idx], y[start_idx:end_idx])


def plot_future(odom, idx, odom_ax=None, future_ax=None, elements=None):
    out_elements = {}
    p = odom[idx]

    if elements is None:
        out_elements["current_pos"] = odom_ax.plot(p[0], p[1], "o", markersize=5, color="b")[0]
        out_elements["future"] = []
    else:
        elements["current_pos"].set_data(p[0], p[1])
```

```python

    for i in range(utils.NUM_COORDS_TO_PREDICT):
        q, _ = follow(odom, idx, (i + 1) * COORD_SPACING)
        if elements is None:
            out_elements["future"].append(odom_ax.plot(q[0], q[1], "o", markersize=5, color="r")[0])
        else:
            elements["future"][i].set_data(q[0], q[1])

    # find the index where we start to move
    start_of_motion = idx
    while np.array_equal(direction(odom, start_of_motion), (0, 0)): start_of_motion += 1

    coords = trajectory_coords(odom, idx, direction(odom, start_of_motion), NUM_COORDS_TO_PREDICT, COORD_SPACING)
    x = [coord[0] for coord in coords]
    y = [coord[1] for coord in coords]
    angle = motion.circular_regression(coords[0:2], ANGLES, WHEEL_BASE)
    r = motion.angle_to_turn_radius(angle, WHEEL_BASE)
    label = "ground truth angle: {}".format(np.rad2deg(angle))
    if elements is None:
        out_elements["coords"] = future_ax.plot(x, y, "o", markersize=5, color="r")[0]
        out_elements["zero"] = future_ax.plot(0, 0, "o", markersize=5, color="b")[0]
        out_elements["circle"] = plot_circle(r, label, future_ax, color="r", pos=(0.05, 0.15))
        return out_elements
    else:
        elements["coords"].set_data(x, y)
        elements["zero"].set_data(0, 0)
        plot_circle(r, label, elements=elements["circle"])


def plot_prediction(idx, model, data_dir, odom_ax=None, future_ax=None, image_ax=None, elements=None):
    out_elements = {}

    image = utils.load_image(data_dir, "img-" + str(idx) + ".jpg")
    if elements is None:
        out_elements["image"] = image_ax.imshow(mark_image(image)) if image_ax is not None else None
    else:
        if elements["image"] is not None:
            elements["image"].set_data(mark_image(image))
    image = utils.preprocess_image(image)
    image = np.expand_dims(image, axis=0)
    prediction = model.predict(image, batch_size=1)
    coords = np.array([(float(prediction[2 * i]), float(prediction[2 * i + 1])) for i in range(int(len(prediction)/ 2))])
    x = [coord[0] for coord in coords]
    y = [coord[1] for coord in coords]
    angle = motion.circular_regression(coords[0:2], ANGLES, WHEEL_BASE)
    r = motion.angle_to_turn_radius(angle, WHEEL_BASE)
    label = "predicted angle: {}".format(np.rad2deg(angle))

    if elements is None:
        out_elements["future"] = future_ax.plot(x, y, "o", markersize=5, color="g")[0]
        out_elements["circle"] = plot_circle(r, label, future_ax, color="g")
        return out_elements
    else:
        elements["future"].set_data(x, y)
        plot_circle(r, label, elements=elements["circle"])


```

```
def plot_circle(r, label, ax=None, color="b", pos=(0.05, 0.05), elements=None):
    out_elements = {}

    if elements is None:
        circle = plt.Circle((r, 0), r, color=color, fill=False)
        ax.add_artist(circle)
        out_elements["circle"] = circle
        text_props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
        out_elements["text"] = ax.text(pos[0], pos[1], label,
            transform=ax.transAxes, fontsize=10, verticalalignment='bottom', bbox=
                ↪ text_props)
    else:
        elements["circle"].set_radius(r)
        elements["circle"].center = (r, 0)
        elements["text"].set_text(label)

    if elements is None:
        return out_elements


def mark_image(image):
    image = image.copy()
    image[279 + 1][:] = [255, 0, 0]
    image[279 + 0][:] = [255, 0, 0]
    image[279 + -1][:] = [255, 0, 0]

    image[306 + 1][:] = [255, 0, 0]
    image[306 + 0][:] = [255, 0, 0]
    image[306 + -1][:] = [255, 0, 0]

    image[364 + 1][:] = [255, 0, 0]
    image[364 + 0][:] = [255, 0, 0]
    image[364 + -1][:] = [255, 0, 0]

    # 0.5m
    cv2.line(image, (493, 240), (744, 444), (0, 255, 0), 2)
    cv2.line(image, (251, 240), (0, 444), (0, 255, 0), 2)
    # 1.0m
    cv2.line(image, (744, 326), (532, 240), (0, 255, 0), 2)
    cv2.line(image, (0, 326), (212, 240), (0, 255, 0), 2)

    cv2.line(image, (372, 240), (372, 480), (0, 255, 0), 2)
    return image


def rotation_matrix(angle):
    """ angle in radians """
    c, s = np.cos(angle), np.sin(angle)
    return np.array(((c, -s), (s, c)))


def direction(odom, idx):
    if idx == 0:
        return np.array((0, 0))
    c = odom[idx]
    i = idx - 1
    d = c - odom[i]
    while i > 0 and np.array_equal(d, (0, 0)):
        i -= 1
        d = c - odom[i]
    return d


def trajectory_coords(odom, idx, direction, num_coords, coord_spacing):
    if np.array_equal(direction, (0, 0)):
        ValueError("'direction' must not be (0, 0).")
```

```
    # handle divide by zero case
    if direction[0] == 0.0:
        angle = 0.0 if direction[1] >= 0 else np.pi
    else:
        angle = np.arctan(direction[1] / direction[0])
        angle += 0.5 * np.pi if direction[0] < 0 else 1.5 * np.pi
    rotation = rotation_matrix(-angle)
    coords = np.empty([num_coords, 2])
    for i in range(num_coords):
        coord, _ = follow(odom, idx, coord_spacing * (i + 1))
        coord -= odom[idx]
        coord = np.dot(rotation, coord)
        coords[i] = coord
    return coords


class OdometryBuilder:
    def __init__(self, resolution):
        self.resolution = resolution
        self.odom = np.array([(0, 0)])
        self.p = np.array((0, 0))

    def left(self, s):
        coords = np.empty([int(s / self.resolution), 2])
        for i in range(len(coords)):
            coords[i][0] = self.p[0] - (i + 1) * self.resolution
            coords[i][1] = self.p[1]
        self.p = self.p - (s, 0)
        if len(coords) == 0 or coords[-1][0] != self.p[0]:
            coords = np.append(coords, [self.p], axis=0)
        self.odom = np.append(self.odom, coords, axis=0)
        return self

    def right(self, s):
        coords = np.empty([int(s / self.resolution), 2])
        for i in range(len(coords)):
            coords[i][0] = self.p[0] + (i + 1) * self.resolution
            coords[i][1] = self.p[1]
        self.p = self.p + (s, 0)
        if len(coords) == 0 or coords[-1][0] != self.p[0]:
            coords = np.append(coords, [self.p], axis=0)
        self.odom = np.append(self.odom, coords, axis=0)
        return self

    def up(self, s):
        coords = np.empty([int(s / self.resolution), 2])
        for i in range(len(coords)):
            coords[i][0] = self.p[0]
            coords[i][1] = self.p[1] + (i + 1) * self.resolution
        self.p = self.p + (0, s)
        if len(coords) == 0 or coords[-1][1] != self.p[1]:
            coords = np.append(coords, [self.p], axis=0)
        self.odom = np.append(self.odom, coords, axis=0)
        return self

    def down(self, s):
        coords = np.empty([int(s / self.resolution), 2])
        for i in range(len(coords)):
            coords[i][0] = self.p[0]
            coords[i][1] = self.p[1] - (i + 1) * self.resolution
        self.p = self.p - (0, s)
        if len(coords) == 0 or coords[-1][1] != self.p[1]:
            coords = np.append(coords, [self.p], axis=0)
        self.odom = np.append(self.odom, coords, axis=0)
        return self

```

```python

def reference_odometry_cw(resolution=0.1):
    builder = OdometryBuilder(resolution)
    builder.right(3).down(3).left(3).up(3)
    return builder.odom


def reference_odometry_ccw(resolution=0.1):
    builder = OdometryBuilder(resolution)
    builder.right(3).up(3).left(3).down(3)
    return builder.odom


def load_data(csv):
    data_df = pd.read_csv(csv, usecols=["x", "y", "ticksRL", "ticksRR"])
    odom = data_df[["x", "y"]].values.astype(np.float64)
    ticks = data_df[["ticksRL", "ticksRR"]].values.astype(np.float64)
    return odom, ticks


def compress_ticks(ticks, steps):
    n = int(len(ticks) / steps)
    compressed_ticks = np.empty([n, 2])
    for i in range(n):
        dts_left = 0
        dts_right = 0
        for j in range(steps):
            dts_left += ticks[i * steps + j][0]
            dts_right += ticks[i * steps + j][1]
        compressed_ticks[i][0] = dts_left
        compressed_ticks[i][1] = dts_right
    return compressed_ticks


def ticks_to_odometry(ticks, params):
    odom = np.empty([len(ticks), 2])
    x = 0
    y = 0
    yaw = 0
    for i, item in enumerate(ticks):
        ticks_rl, ticks_rr = item
        dts_left = ticks_rl * params.dts_tick_left
        dts_right = ticks_rr * params.dts_tick_right
        dts = (dts_left + dts_right) / 2.0
        dt_yaw = (dts_right - dts_left) / params.track_width
        x += np.cos(yaw) * dts * 0.001
        y += np.sin(yaw) * dts * 0.001
        yaw += dt_yaw
        odom[i][0] = x
        odom[i][1] = y
    return odom


def speed(ticks_left, ticks_right, delta_time, params):
    dts_left = ticks_left * params.dts_tick_left
    dts_right = ticks_right * params.dts_tick_right
    dts = (dts_left + dts_right) / 2.0
    return (dts * 0.001) / delta_time


def mean_squared_distance_error(odom1, odom2, resolution, ax=None):
    dist1 = distance(odom1)
    dist2 = distance(odom2)
    dist = min(dist1, dist2)
    n = int(dist / resolution) - 1
    #err = (dist1 - dist2) ** 2
```

```python
    err = 0
    for i in range(1, n):
        p1, _ = follow(odom1, 0, i * resolution)
        p2, _ = follow(odom2, 0, i * resolution)
        if ax is not None:
            ax.plot([p1[0], p2[0]], [p1[1], p2[1]], color="black")
        err += euclidean(p1, p2) ** 2
    return err / n


def reference_error(x, reference_odom_to_ticks_list, resolution):
    """
    reference_odom_to_ticks_list:
    [(reference_odom_1, [ticks_1, ticks_2, ticks_3]),
     (reference_odom_2, [ticks_1, ticks_2, ticks_3]),
     ....
     (reference_odom_n, [ticks_1, ticks_2, ticks_3])]
    """
    params = OdometryParams(x[0], x[1], x[2])
    err = 0
    n = 0
    for item in reference_odom_to_ticks_list:
        reference_odom = item[0]
        ticks_list = item[1]
        n += len(ticks_list)
        for ticks in ticks_list:
            odom_reconstructed = ticks_to_odometry(ticks, params)
            err += mean_squared_distance_error(reference_odom, odom_reconstructed, resolution)
    return err / n


def mean_center_error(x, cw, ccw):
    params = OdometryParams(x[0], x[1], x[2])
    odoms_cw = [ticks_to_odometry(t, params) for t in cw]
    odoms_ccw = [ticks_to_odometry(t, params) for t in ccw]
    center_err_cw = np.linalg.norm(mean_center_of_gravity(odoms_cw))
    center_err_ccw = np.linalg.norm(mean_center_of_gravity(odoms_ccw))
    distance_err = np.mean([abs(12 - distance(odom)) for odom in np.append(odoms_cw, odoms_ccw, axis=0)])
    print("center_err: " + str(max(center_err_cw, center_err_ccw)))
    print("distance_err: " + str(distance_err))
    return max(center_err_cw, center_err_ccw) + distance_err


def mean_offset_error(x, cw, ccw):
    err = 0
    params = OdometryParams(x[0], x[1], x[2])

    odoms_cw = [ticks_to_odometry(t, params) for t in cw]
    odoms_ccw = [ticks_to_odometry(t, params) for t in ccw]

    err += np.mean([(odom[int(0.175 * len(odom))][1] - 0.0)**2 for odom in odoms_cw])
    err += np.mean([(odom[int(0.175 * len(odom))][1] - 0.0)**2 for odom in odoms_ccw])

    err += np.mean([(odom[int(0.375 * len(odom))][0] - 3.0)**2 for odom in odoms_cw])
    err += np.mean([(odom[int(0.375 * len(odom))][0] - 3.0)**2 for odom in odoms_ccw])

    err += np.mean([(odom[int(0.55 * len(odom))][1] - (-3.0))**2 for odom in odoms_cw])
    err += np.mean([(odom[int(0.55 * len(odom))][1] - 3.0)**2 for odom in odoms_ccw])

    err += np.mean([(odom[int(0.75 * len(odom))][0] - 0.0)**2 for odom in odoms_cw])
    err += np.mean([(odom[int(0.75 * len(odom))][0] - 0.0)**2 for odom in odoms_ccw])
```

```python

    err += np.mean([np.linalg.norm(odom[-1])**2 for odom in odoms_cw])
    err += np.mean([np.linalg.norm(odom[-1])**2 for odom in odoms_ccw])

    return err / 10.0


def mean_center_of_gravity(odoms):
    cog = np.array([odom[-1] for odom in odoms])
    x = np.mean(cog[:,0])
    y = np.mean(cog[:,1])
    return np.array((x, y))


def manual_calibration(fig, ax, ticks_list, params):
    isarray = isinstance(ax, (list, tuple, np.ndarray))
    lines_list = []
    mean_distance = 0
    for i, ticks in enumerate(ticks_list):
        odom = ticks_to_odometry(ticks, params)
        mean_distance += distance(odom)
        axis = ax[i] if isarray else ax
        lines, = axis.plot([o[0] for o in odom], [o[1] for o in odom], "ro", markersize=1)
        lines_list.append(lines)
    mean_distance /= len(ticks_list)

    text_props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
    label = "dts left: {}\ndts right: {}\ntrack width: {}\nmean distance: {}".format(
        params.dts_tick_left, params.dts_tick_right, params.track_width, mean_distance)
    axis = ax[0] if isarray else ax
    text = axis.text(0.05, 0.95, label, transform=axis.transAxes, fontsize=14,
            verticalalignment='top', bbox=text_props)

    def event_handler(event):
        dts_step = 0.005
        track_step = 0.5
        if event.key == "t":
            params.dts_tick_left += dts_step
        elif event.key == "g":
            params.dts_tick_left -= dts_step
        elif event.key == "i":
            params.dts_tick_right += dts_step
        elif event.key == "k":
            params.dts_tick_right -= dts_step
        elif event.key == "ü":
            params.track_width += track_step
        elif event.key == "ä":
            params.track_width -= track_step
        mean_distance = 0
        for i, ticks in enumerate(ticks_list):
            odom = ticks_to_odometry(ticks, params)
            mean_distance += distance(odom)
            lines_list[i].set_data([o[0] for o in odom], [o[1] for o in odom])
        mean_distance /= len(ticks_list)
        text.set_text("dts left: {}\ndts right: {}\ntrack width: {}\nmean distance: {}".format(
                params.dts_tick_left, params.dts_tick_right, params.track_width, mean_distance))

    fig.canvas.mpl_connect("key_press_event", event_handler)
    plt.ion()
    while True:
        plt.pause(0.1)

```

```
def UMBmark_optimize(b, D_L, D_R, dts_L, dts_R, cog_cw, cog_ccw, L):
    """
    b: wheel base: distance between the two wheels
    D_L: diameter of left wheel
    D_R: diameter of right wheel
    dts_L: left wheel travel per encoder pulse
    dts_R: right wheel travel per encoder pulse
    cog_cw: center of gravity in clockwise direction
    cog_ccw: center of gravity in counter-clockwise direction
    L: length of the square the robot has driven
    """
    alpha = (cog_cw[0] + cog_ccw[0]) / (-4 * L)
    beta = (cog_cw[0] - cog_ccw[0]) / (-4 * L)

    R = (L / 2) / (np.sin(beta / 2))
    Ed = (R + (b / 2)) / (R - (b / 2))
    D_a = (D_R + D_L) / 2
    D_L = (2 / (Ed + 1)) * D_a
    D_R = (2 / (1 / Ed) + 1) * D_a
    c_L = 2 / (Ed + 1)
    c_R = 2 / ((1 / Ed) + 1)
    dts_L *= c_L
    dts_R *= c_R

    E_b = (np.pi / 2) / ((np.pi / 2) - alpha)
    b *= E_b

    return b, D_L, D_R, dts_L, dts_R


def load_square_ticks(compression_steps=5):
    cw = [compress_ticks(load_data("/media/philipp/Transcend/odometryCalibration/cw/"
        ↪ + str(i) + ".csv")[1], compression_steps) for i in range(5)]
    ccw = [compress_ticks(load_data("/media/philipp/Transcend/odometryCalibration/ccw
        ↪ /" + str(i) + ".csv")[1], compression_steps) for i in range(5)]

    cw = [cw[i] for i in [2, 3]]
    ccw = [ccw[i] for i in [0, 1, 2, 4]]

    return cw, ccw
```

parameters.py

```
# -*- coding: utf-8 -*-
import numpy as np


class OdometryParams:
    def __init__(self, dts_tick_left, dts_tick_right, track_width):
        """ all units are given in millimeters """
        self.dts_tick_left = dts_tick_left #
        self.dts_tick_right = dts_tick_right
        self.track_width = track_width

    def __str__(self):
        return "dts_tick_left: {} mm\ndts_tick_right: {} mm\ntrack_width: {} mm".
            ↪ format(
            self.dts_tick_left, self.dts_tick_right, self.track_width)


ANGLES = [np.deg2rad(deg) for deg in list(range(-30, 31))]
LOOK_AHEAD_DISTANCE = 2 # m
GRAVITY_ACC = 9.81 # m / s^2
WHEEL_BASE = 0.32 # m
```

```
#WHEEL_BASE = 0.2304363
# the wheels turn from full left (-30) to full right (30) in approx. 0.5 seconds
ANGLE_CHANGE_RATE = np.deg2rad(20) # rad / s
MAX_ANGLE_DELAY_DISTANCE = 0.2 # m
# defines a delta angle tolerance for comparing two angles:
# angle_1 ~ angle_2 if abs(angle_1 - angle_2) <= DELTA_ANGLE_TOLERANCE
DELTA_ANGLE_TOLERANCE = np.deg2rad(1) # rad
STRAIGHT_ANGLE_TOLERANCE = np.deg2rad(10)
ANGLE_ESTIMATOR_CONFIDENCE_DURATION = 0.5 # s
STATIC_FRICTION_COEF = 1.0

IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 100, 200, 3
INPUT_SHAPE = (IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS)
COORD_SPACING = 0.6 # m
NUM_COORDS_TO_PREDICT = int(LOOK_AHEAD_DISTANCE / COORD_SPACING)
CROP_TOP = 200

FLOAT_TOLERANCE = 10e-6
DISC_RESOLUTION = 120
WHEEL_DIAMETER = 0.11 # m
# original
# DTS_TICK_LEFT   = (np.pi * WHEEL_DIAMETER / DISC_RESOLUTION) * 1000.0 # mm
# DTS_TICK_RIGHT   = (np.pi * WHEEL_DIAMETER / DISC_RESOLUTION) * 1000.0 # mm
# TRACK_WIDTH = 275 # mm
# end up using
DTS_TICK_LEFT = (np.pi * WHEEL_DIAMETER / DISC_RESOLUTION) * 1000.0 - 0.01 + 0.3 # mm
DTS_TICK_RIGHT = (np.pi * WHEEL_DIAMETER / DISC_RESOLUTION) * 1000.0 + 0.01 + 0.3 #
    ↪ mm
TRACK_WIDTH = 305 # mm

ERROR_RESOLUTION = 1 # m
SHOULD_OPTIMIZE = True
OPTIMIZATION_METHOD = "BFGS"
MAX_ITERATIONS = 100
SQUARE_LENGTH = 3 # m
SQUARE_DISTANCE = 4 * SQUARE_LENGTH # m
SQUARE_DISTANCE_TOLERANCE = 1
DEFAULT_ODOMETRY_PARAMS = OdometryParams(DTS_TICK_LEFT, DTS_TICK_RIGHT, TRACK_WIDTH)
```

saliency.py

```
from vis.visualization import visualize_saliency, visualize_cam, overlay
from keras import activations
from utils import *
from vis.utils import utils
from keras.models import load_model
from matplotlib import pyplot as plt
import numpy as np
import cv2
from keras.models import Model
from matplotlib import cm

# ----
# hella images
# ---
images = [
    load_image("I:/recordings/data-2018-08-31-12-12-14", "img-300.jpg"), # left,
        ↪ ideal line
    load_image("I:/recordings/data-2018-08-31-12-12-14", "img-6175.jpg"), # right,
        ↪ ideal line
    load_image("I:/recordings/data-2018-08-31-12-12-14", "img-351.jpg"), # right,
        ↪ ideal line
    load_image("I:/recordings/data-2018-08-31-10-42-27", "img-10937.jpg"), # right,
        ↪ center line

```

```python
    load_image("I:/recordings/data-2018-08-31-10-42-27", "img-625.jpg"), # straight, center line
    load_image("I:/recordings/data-2018-08-31-10-42-27", "img-9880.jpg"), # right, center line

    # afternoon
    load_image("I:/recordings/data-2018-09-07-15-59-32", "img-180.jpg"),
    load_image("I:/recordings/data-2018-09-07-15-59-32", "img-820.jpg"),
    load_image("I:/recordings/data-2018-09-07-15-59-32", "img-1110.jpg"),

    load_image("I:/recordings/data-2018-09-07-15-59-32", "img-10450.jpg"),
    load_image("I:/recordings/data-2018-09-07-15-59-32", "img-11300.jpg"),
    load_image("I:/recordings/data-2018-09-07-15-42-32", "img-390.jpg"),
]

# ----
# flur images
# ---
# images = [
#     load_image("I:/recordings/data-2018-09-08-17-19-41", "img-540.jpg"),
#     load_image("I:/recordings/data-2018-09-08-17-19-41", "img-610.jpg"),
#     load_image("I:/recordings/data-2018-09-08-17-19-41", "img-660.jpg"),
#     load_image("I:/recordings/data-2018-09-08-17-19-41", "img-1460.jpg"),
#     load_image("I:/recordings/data-2018-09-08-17-19-41", "img-1515.jpg"),
#     load_image("I:/recordings/data-2018-09-08-17-19-41", "img-2090.jpg"),
# ]

model = load_model("G:/car/models/model-040.h5")

fig, axes = plt.subplots(int(len(images) / 3), 3)

layer_idx = 6
model.layers[layer_idx].activation = activations.linear
model = utils.apply_modifications(model)
for i, image in enumerate(images):
    image = preprocess_image(image)
    grads = visualize_saliency(model, layer_idx, filter_indices=None, seed_input=image, backprop_modifier="guided")
    jet_grads = (np.delete(cm.jet(grads), 3, 2) * 255.0).astype(np.uint8)

    image = (image + 1.0) * 127.5
    image = image.astype(np.uint8)
    image = cv2.cvtColor(image, cv2.COLOR_YUV2RGB)

    axes[int(i/3), int(i%3)].imshow(overlay(image, jet_grads, alpha=0.3))

plt.show()
```

train.py

```python
import numpy as np
np.random.seed(0)
import pandas as pd
from sklearn.cross_validation import train_test_split
from keras.models import Sequential, Model
import keras.regularizers
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, TensorBoard, CSVLogger
from keras.layers import Lambda, Convolution2D, MaxPooling2D, Dropout, Dense, Flatten, Input, BatchNormalization
from keras import regularizers
from utils import *
import argparse
import os
from time import time
```

```python
import odometry
import json
import merge


def load_data(args):
    X = y = None

    with open(args.training_config, "r") as f:
        config = json.load(f)
        recordings = config["recordings"]
        for recording in recordings:
            directory = recording["directory"]
            csv = os.path.join(directory, "data.csv")
            data_df = pd.read_csv(csv)
            contains_subsequences = "recording" in data_df

            image_paths_recording = [os.path.join(directory, img.strip()) for img in data_df["img"].values]

            odom_params = read_odometry_params(
                recording["odometryParameters"]) if "odometryParameters" in recording \
                                                 else DEFAULT_ODOMETRY_PARAMS

            ticks = data_df[["ticksRL", "ticksRR"]].values.astype(np.float64)
            odom_recording = odometry.ticks_to_odometry(ticks, odom_params)

            for sequence in merge.sequences_from_json(recording["sequences"]):
                sequence_start = sequence[0]
                sequence_end = sequence[1]

                subsequences = get_subsequences(data_df, sequence_start, sequence_end) \
                               if contains_subsequences \
                               else [(sequence_start, sequence_end)]

                for subsequence in subsequences:
                    subsequence_start = subsequence[0]
                    subsequence_end = subsequence[1]
                    subsequence_odom = np.array([odom_recording[i] for i in range(subsequence_start, subsequence_end + 1)])
                    try: subsequence_coords_list, first_idx, last_idx = get_coords_list(subsequence_odom)
                    except ValueError: continue
                    subsequence_image_paths = np.array([image_paths_recording[i]
                                                         for i in range(subsequence_start + first_idx, subsequence_start + last_idx + 1)])

                    X = np.append(X, subsequence_image_paths, axis=0) if X is not None else np.array(subsequence_image_paths)
                    y = np.append(y, subsequence_coords_list, axis=0) if y is not None else np.array(subsequence_coords_list)

    X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=args.test_size, random_state=1)
    return X_train, X_valid, y_train, y_valid


def get_coords_list(odom):
    # from every coordinate in the training set we must be able
    # to travel a distance of (NUM_COORDS_TO_PREDICT * COORD_SPACING) into the future
    _, last_idx = odometry.follow(odom, len(odom) - 1, NUM_COORDS_TO_PREDICT * COORD_SPACING, backwards=True)
    last_idx -= 1
```

```python
    if last_idx < 0:
        raise ValueError("Not enough coordinates!")
    # find the index where we start to move + 1
    # which is where direction() does not return (0, 0)
    first_idx = 0
    while np.array_equal(odometry.direction(odom, first_idx), (0, 0)): first_idx += 1
    # get list of coordinates to predict
    coords_list = [
        odometry.trajectory_coords(odom, i, odometry.direction(odom, i),
            ↪ NUM_COORDS_TO_PREDICT, COORD_SPACING)
        for i in range(first_idx, last_idx + 1)]
    return coords_list, first_idx, last_idx


def get_subsequences(data_df, section_start, section_end):
    subsequences = []
    subsequence_start = None
    recording = False
    for i in range(section_start, section_end + 1):
        recording = bool(data_df["recording"][i])
        if recording:
            if subsequence_start is None:
                subsequence_start = i
        elif subsequence_start is not None:
            subsequences.append((subsequence_start, i - 1))
            subsequence_start = None
    if subsequence_start is not None and recording:
        subsequences.append((subsequence_start, i))
    return subsequences


def read_odometry_params(jparams):
    dts_tick_left_mm = float(jparams["dts_tick_left_mm"]) if "dts_tick_left_mm" in
        ↪ jparams else odometry.DEFAULT_ODOMETRY_PARAMS.dts_tick_left_mm
    dts_tick_right_mm = float(jparams["dts_tick_right_mm"]) if "dts_tick_right_mm" in
        ↪  jparams else odometry.DEFAULT_ODOMETRY_PARAMS.dts_tick_right_mm
    track_width_mm = float(jparams["track_width_mm"]) if "track_width_mm" in jparams
        ↪ else odometry.DEFAULT_ODOMETRY_PARAMS.track_width_mm
    return odometry.OdometryParams(dts_tick_left_mm, dts_tick_right_mm,
        ↪ track_width_mm)


def build_model_nvidia(args):
    img_in = Input(shape=INPUT_SHAPE, name='img_in')

    x = img_in
    x = BatchNormalization(axis=3)(x)
    x = Convolution2D(24, (5, 5), strides=(2, 2), activation='relu')(x)
    x = Convolution2D(36, (5, 5), strides=(2, 2), activation='relu')(x)
    x = Convolution2D(48, (5, 5), strides=(2, 2), activation='relu')(x)
    x = Convolution2D(64, (3, 3), strides=(1, 1), activation='relu')(x)
    x = Convolution2D(64, (3, 3), strides=(1, 1), activation='relu')(x)
    x = Flatten()(x)
    x = Dense(100, activation="relu")(x)
    x = Dense(50, activation="relu")(x)
    x = Dense(10, activation="relu")(x)

    loss = {}
    outputs = []
    for i in range(NUM_COORDS_TO_PREDICT):
        name_x = "coord_x_out_" + str(i)
        name_y = "coord_y_out_" + str(i)
        coord_x_out = Dense(1, name=name_x, activation="linear")(x)
        coord_y_out = Dense(1, name=name_y, activation="linear")(x)
        loss[name_x] = "mean_squared_error"
        loss[name_y] = "mean_squared_error"
```

```python
        outputs.append(coord_x_out)
        outputs.append(coord_y_out)

    model = Model(inputs=[img_in], outputs=outputs)
    model.compile(optimizer="adam", loss=loss)
    return model


def train_model(model, args, X_train, X_valid, y_train, y_valid):
    checkpoint = ModelCheckpoint(get_config_dir() + '/models/model-{epoch:03d}.h5',
                                 monitor='val_loss',
                                 verbose=0,
                                 save_best_only=args.save_best_only,
                                 mode='auto')

    tensorboard = TensorBoard(log_dir=get_config_dir() + "/logs/tensorboard/{}_{}".format(args.log_label, time()))
    csvLogger = CSVLogger(get_config_dir() + "/logs/csv/{}_{}.csv".format(args.log_label, time()), separator=',', append=False)

    model.compile(loss='mean_squared_error', optimizer=Adam(lr=args.learning_rate))

    cache_capacity = min(int(args.image_memory_size / (IMAGE_HEIGHT * IMAGE_WIDTH * IMAGE_CHANNELS * 8)), len(X_train) + len(X_valid))
    print("Cache capacity: {} images".format(cache_capacity))
    image_cache = ImageCache(cache_capacity)
    print("Buffering images...")
    image_cache.load_images(np.append(X_train, X_valid, axis=0))
    print("Completed!")

    train_sequence = BatchSequence(args.data_dir, X_train, y_train, args.batch_size, True, image_cache)
    valid_sequence = BatchSequence(args.data_dir, X_valid, y_valid, args.batch_size, False, image_cache)

    model.fit_generator(train_sequence,
                        None,
                        args.nb_epoch,
                        max_queue_size=args.max_queue_size,
                        validation_data=valid_sequence,
                        validation_steps=None,
                        callbacks=[checkpoint, tensorboard, csvLogger],
                        verbose=1,
                        workers=args.workers,
                        shuffle=False)

def s2b(s):
    s = s.lower()
    return s == 'true' or s == 'yes' or s == 'y' or s == '1'


def main():
    default_training_dir = get_config_dir() + "/trainingData"
    parser = argparse.ArgumentParser(description='Behavioral Cloning Training Program')
    parser.add_argument('training_config', help='training config file', type=str)
    parser.add_argument('-d', help='data directory',        dest='data_dir',        type=str,   default=default_training_dir)
    parser.add_argument('-t', help='test size fraction',    dest='test_size',       type=float, default=0.2)
    parser.add_argument('-k', help='drop out probability',  dest='keep_prob',       type=float, default=0.5)
    parser.add_argument('-n', help='number of epochs',      dest='nb_epoch',        type=int,   default=50)
    parser.add_argument('-b', help='batch size',            dest='batch_size',      type=int,   default=40)
```

```
    parser.add_argument('-o', help='save best models only', dest='save_best_only',    type=s2b,    default='false')
    parser.add_argument('-l', help='learning rate',         dest='learning_rate',     type=float, default=1.0e-4)
    parser.add_argument('-x', help='tensorboard log label', dest='log_label',         type=str,    default='log')
    parser.add_argument('-w', help='number of workers',     dest='workers',           type=int,    default=8)
    parser.add_argument('-q', help='max queue size ',       dest='max_queue_size',    type=int,    default=1)
    parser.add_argument('-m', help='image memory size',     dest='image_memory_size', type=int,    default=12e9)
    args = parser.parse_args()

    print('-' * 30)
    print('Parameters')
    print('-' * 30)
    for key, value in vars(args).items():
        print('{:<20} := {}'.format(key, value))
    print('-' * 30)

    print("Loading data...")
    data = load_data(args)
    print("Completed!")

    model = build_model_nvidia(args)
    train_model(model, args, *data)


if __name__ == '__main__':
    main()
```

utils.py

```
import cv2, os
import numpy as np
import numexpr as ne
import matplotlib.image as mpimg
from os.path import expanduser
import json
import keras
from threading import Thread, Condition, Lock
import sys
is_py2 = sys.version[0] == '2'
if is_py2:
    from Queue import Queue, Empty
else:
    from queue import Queue, Empty
from parameters import *


def get_config_dir():
    if os.name == "nt":
        return "G:/car"
    home = expanduser("~")
    return home + "/.car"


def open_config():
    f = open(get_config_dir() + "/config.json", "r")
    return json.load(f)


def load_image_from_path(path):
    return cv2.imread(path)
```

```python
def load_image(data_dir, image_file):
    return load_image_from_path(os.path.join(data_dir, image_file.strip()))


def crop(image):
    return image[CROP_TOP:, :, :]


def resize(image):
    return cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)


def normalize(image):
    return ne.evaluate("image / 127.5 - 1.0")


def convert(image):
    image = image.astype(np.uint8)
    return cv2.cvtColor(image, cv2.COLOR_BGR2YUV)


def bgr2yuv(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2YUV)


def random_flip(image, coords):
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 1)
        for i, coord in enumerate(coords):
            coords[i][0] = -coord[0]
    return image, coords


def apply_contrast(image, ratio):
    image = image.astype(np.float64)
    image = np.minimum(image * ratio, 255)
    return image.astype(np.uint8)


def apply_brightness(image, ratio):
    image = image.astype(np.uint8)
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    hsv = hsv.astype(np.float64)
    hsv[:,:,2] = np.minimum(hsv[:,:,2] * ratio, 255)
    hsv = hsv.astype(np.uint8)
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)


def random_brightness(image):
    # HSV (Hue, Saturation, Value) is also called HSB ('B' for Brightness).
    ratio = np.random.uniform(0.5, 2.0, 1)[0]
    return apply_brightness(image, ratio)


def augument(image, coords):
    image, coords = random_flip(image, coords)
    image = random_brightness(image)
    return image, coords


def preprocess_image(image, do_crop=True, do_resize=True, do_convert=True, do_normalize=True):
    if do_crop:
        image = crop(image)
```

```
    if do_resize:
        image = resize(image)
    if do_convert:
        image = convert(image)
    if do_normalize:
        image = normalize(image)
    return image


def preprocess_angle(angle):
    return angle / 30


def preprocess_throttle(throttle):
    return throttle


def postprocess_angle(angle):
    return angle * 30


def postprocess_throttle(throttle):
    return throttle


class BatchSequence(keras.utils.Sequence):
    def __init__(self, data_dir, X, y, batch_size, is_training, image_cache):
        self.data_dir = data_dir
        self.X = X
        self.y = y
        self.batch_size = batch_size
        self.is_training = is_training
        self.sequence = self.next_sequence()
        self.image_cache = image_cache

    def __len__(self):
        return int(np.floor(len(self.X) / float(self.batch_size)))

    def __getitem__(self, idx):
        inputs = np.empty([self.batch_size, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS])
        outputs = [np.empty(self.batch_size) for i in range(2 * NUM_COORDS_TO_PREDICT)]

        for batch_idx in range(self.batch_size):
            seq_idx = (idx * self.batch_size + batch_idx) % len(self.sequence)
            sample_idx = self.sequence[seq_idx]
            image_path = self.X[sample_idx]
            image = self.image_cache.get_image(image_path)
            coords = self.y[sample_idx].copy()

            # augumentation
            if self.is_training and np.random.rand() < 0.6:
                image, coords = augument(image, coords)

            inputs[batch_idx] = normalize(convert(image))
            for i, coord in enumerate(coords):
                outputs[2 * i][batch_idx] = coord[0]
                outputs[2 * i + 1][batch_idx] = coord[1]

        return inputs, outputs

    def on_epoch_end(self):
        self.sequence = self.next_sequence()

    def next_sequence(self):
```

```python
        return np.random.permutation(len(self.X))


class ImageCache:
    def __init__(self, capacity):
        self.images = np.empty([capacity, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS])
        self.path_to_idx = {}

    def load_images(self, image_paths):
        for i in range(min(len(self.images), len(image_paths))):
            image_path = image_paths[i]
            self.images[i] = self.load_image(image_path)
            self.path_to_idx[image_path] = i

    def load_image(self, image_path):
        image = load_image_from_path(image_path)
        return preprocess_image(image, do_convert=False, do_normalize=False)

    def get_image(self, image_path):
        if image_path in self.path_to_idx:
            return self.images[self.path_to_idx[image_path]]
        return self.load_image(image_path)
```

C.2. STM32

C.2.1. C Source Code

Note that the following code was originally delivered by Assystem GmbH and includes modifications from us.

mavlink.h

```c

#ifndef MAVLINK_H_
#define MAVLINK_H_

/************* Includes *******************************************************************/

#include <stdlib.h>
#include <stdbool.h>
#include "per/irq.h"
#include "sys/util/io.h"
#include "per/uart.h"
#include "sys/sigp/sigp.h"
#include "sys/systime/systime.h"
#include "protocol/velox/mavlink.h"
#include "sys/errorhandler/errorHandler.h"

/************* Public typedefs ************************************************************/

/************* Macros and constants *******************************************************/

#define MAVLINK_BAUDRATE 115200
#define MAX_BUFFER_LEN 1000
/************* Public function prototypes *************************************************/
/**
```

```
 *  @brief The init function needs to be called during the initialization of the system.
 *  It sets all local variables to their initial values.
 */
int8_t mavlink_init(void);

/**
 *  @brief read messages from uart buffer and interpret them
 */
void mavlink_getMessages(void);

/**
 *  @brief pack and send all messages that are marked as active
 */
void mavlink_sendMessages(void);

/**
 *  @brief activate outgoing messages that are linked to the specified state
 */
void mavlink_activateStateMessages(uint8_t state);

/**
 *  @brief deactivate outgoing messages that are linked to the specified state
 */
void mavlink_deactivateStateMessages(uint8_t state);

#endif /* MAVLINK_H_ */
```

mavlink.c

```
/******************************************************************************************
 * Unit in charge:
 * @file     odom.c
 * @author   mbrieske
 * $Date:: 2017-05-22 17:17:17 #$
 *
 * -----------------------------------------------------------
 * SVN information of last commit:
 * $Rev:: 1538                    $
 * $Author:: mbrieske      $
 *
 * -----------------------------------------------------------
 *
 * @brief MAVLink see mavlink.h
 *
 ***************************************************************************************** */

/************* Includes ******************************************************************/
#include "../mavlink.h"
#include "per/eict.h"
#include "app/leddebug.h"

/************* Private typedefs **********************************************************/

#define NUM_MSGS (sizeof mavlinkMessageInfo / sizeof mavlinkMessageInfo[0])

typedef struct msgmeta {
    uint8_t       msgid;      /* mavlink message id */
    bool         active;     /* will send message if set to TRUE */
    uint8_t       period;     /* signal will be send every [period] * 10 ms */
```

```c
    bool         customRequest;  /* true if message was requested via mavlink message */
} msgmeta_t;

/************ Private signals and variables **********************************************************/

static const mavlink_message_info_t mavlinkMessageInfo[] = MAVLINK_MESSAGE_INFO;
mavlink_system_t mavlinkSystem;
msgmeta_t msgMeta[NUM_MSGS];
static bool prevTimeout;

static uint32_t lastTimestamp;

/* UART */
uart_t uart;
static uint8_t byteBuffer;
static int8_t receiveBuffer[MAX_BUFFER_LEN], transmitBuffer[MAX_BUFFER_LEN];
static uint32_t receiveBufferSize = MAX_BUFFER_LEN, transmitBufferSize = MAX_BUFFER_LEN;
static uint32_t receiveBufferTimestamp[MAX_BUFFER_LEN], transmitBufferTimestamp[MAX_BUFFER_LEN];

static uint16_t transmitcounter;

/* Signals */
uint32_t *pSyncOffset_mavlink;
heartbeat_t *pHeartbeat_mavlink;
carcontrol_t *pCarControl_mavlink;
trajectory_t *pTrajectory_mavlink;
uint8_t *pState_mavlink;
uint8_t *pRequestedState_mavlink;
uint64_t *pErrorRegister_mavlink;

uint32_t *pOdomTimestamp_mavlink;
uint32_t *pStAngTimestamp_mavlink;
float32_t *pVehSpdOdom_mavlink;
float32_t *pVehXDistOdom_mavlink;
float32_t *pVehYDistOdom_mavlink;
float32_t *pVehYawAngOdom_mavlink;
float32_t *pStAngAct_mavlink;

/************ Private functions **********************************************************/

bool configureOutgoingMessage(uint8_t msgid, bool active, uint8_t period, bool customRequest)
{
  if (msgid == MAVLINK_MSG_ID_HEARTBEAT && customRequest) return false; // do not allow heartbeat to be reconfigured by custom requests
  for (uint8_t i = 0; i < NUM_MSGS; i++)
    {
      if (msgMeta[i].msgid == msgid)
      {
        if (msgMeta[i].customRequest == false || customRequest == true)
        {
          msgMeta[i].active = active;
          msgMeta[i].period = period;
          msgMeta[i].customRequest = customRequest;
        }
        return true;
      }
    }
    return false;
}

```

```c
void configureStateMessages(uint8_t state, bool active)
{
  switch (state)
  {
  case SYSTEM_STATE_INITIALIZING:
    configureOutgoingMessage(MAVLINK_MSG_ID_CMD_REQUEST_CLOCKSYNC, active, 50, false);
    break;
  case SYSTEM_STATE_IDLE:
    configureOutgoingMessage(MAVLINK_MSG_ID_CMD_REQUEST_CLOCKSYNC, active, 10, false);
    break;
  case SYSTEM_STATE_RUNNING_EXT:
    configureOutgoingMessage(MAVLINK_MSG_ID_CMD_REQUEST_CLOCKSYNC, active, 10, false);
    configureOutgoingMessage(MAVLINK_MSG_ID_ODOMETRY, active, 1, false);
    break;
  case SYSTEM_STATE_RUNNING_RC:
    configureOutgoingMessage(MAVLINK_MSG_ID_CMD_REQUEST_CLOCKSYNC, active, 10, false);
    break;
  case SYSTEM_STATE_EMERGENCY:
    configureOutgoingMessage(MAVLINK_MSG_ID_ERROR, active, 10, false);
    break;
  default:
    io_printf("MAVLINK: switched to unknown state");
  }
}

bool sendMessage(uint8_t msgid)
{
  mavlink_message_t msg;
  uint8_t buf[MAVLINK_MAX_PACKET_LEN];

  uint16_t ticksRL;
  uint16_t ticksRR;
  if (msgid == MAVLINK_MSG_ID_ODOMETRY)
    eict_getCounter2(&ticksRL, &ticksRR);

  uint32_t now;
  uint32_t deltaTime;

  irq_disable();
  uint32_t syncOffset = *pSyncOffset_mavlink;
  switch (msgid)
  {
  case MAVLINK_MSG_ID_HEARTBEAT:
    mavlink_msg_heartbeat_pack(mavlinkSystem.sysid, mavlinkSystem.compid, &msg, *pState_mavlink);
    break;
  case MAVLINK_MSG_ID_ERROR:
    mavlink_msg_error_pack(mavlinkSystem.sysid, mavlinkSystem.compid, &msg, *pErrorRegister_mavlink);
    break;
  case MAVLINK_MSG_ID_ODOMETRY:
    // use this for real driving:
    //mavlink_msg_odometry_pack(mavlinkSystem.sysid, mavlinkSystem.compid, &msg, *pOdomTimestamp_mavlink - syncOffset, *pStAngTimestamp_mavlink - syncOffset, *pVehSpdOdom_mavlink, *pVehXDistOdom_mavlink, *pVehYDistOdom_mavlink, *pVehYawAngOdom_mavlink, *pStAngAct_mavlink);

    // use this for calibration:
    now = systime_getSysTime();
    deltaTime = now - lastTimestamp;
    lastTimestamp = now;
```

```
    mavlink_msg_odometry_pack(mavlinkSystem.sysid, mavlinkSystem.compid, &msg, *pOdomTimestamp_mavlink - syncOffset, deltaTime, *pVehSpdOdom_mavlink, *pVehXDistOdom_mavlink, *pVehYDistOdom_mavlink, (float32_t) ticksRL, (float32_t) ticksRR);

    break;
  case MAVLINK_MSG_ID_CMD_REQUEST_CLOCKSYNC:
    mavlink_msg_cmd_request_clocksync_pack(mavlinkSystem.sysid, mavlinkSystem.compid, &msg, 0);
    break;
  default:
    io_printf("MAVLink: not implemented or unknown msgid\n");
    irq_enable();
    return false;
  }
  irq_enable();

  uint16_t len = mavlink_msg_to_send_buffer(buf, &msg);
  if (uart.sendring.size - uart.sendring.count > len) {
    uart_writeString(&uart, (int8_t*) buf, len);
  }
  else {
    io_printf("MAVLink: UART TX buffer full\n");
    return false;
  }

  return true;
}

void handleMessage(mavlink_message_t *msg)
{
  uint32_t recTimestamp = systime_getSysTime();
  irq_disable();
  uint32_t syncOffset = *pSyncOffset_mavlink;
  switch(msg->msgid)
  {
  case MAVLINK_MSG_ID_HEARTBEAT:
    pHeartbeat_mavlink->lastReceiveTimestamp = recTimestamp;
    pHeartbeat_mavlink->mavlink_version = mavlink_msg_heartbeat_get_mavlink_version(msg);
    pHeartbeat_mavlink->state = mavlink_msg_heartbeat_get_state(msg);
    break;
  case MAVLINK_MSG_ID_CARCONTROL:
    pCarControl_mavlink->lastReceiveTimestamp = recTimestamp;
    pCarControl_mavlink->vehSpdTgtExt = mavlink_msg_carcontrol_get_vehspd(msg);
    pCarControl_mavlink->stAngTgtExt = mavlink_msg_carcontrol_get_stang(msg) - 4;
    break;
  case MAVLINK_MSG_ID_TRAJECTORY:
    pTrajectory_mavlink->lastReceiveTimestamp = recTimestamp;
    pTrajectory_mavlink->timestamp = mavlink_msg_trajectory_get_timestamp(msg) + syncOffset;
    mavlink_msg_trajectory_get_x(msg, pTrajectory_mavlink->x);
    mavlink_msg_trajectory_get_y(msg, pTrajectory_mavlink->y);
    mavlink_msg_trajectory_get_theta(msg, pTrajectory_mavlink->theta);
    mavlink_msg_trajectory_get_kappa(msg, pTrajectory_mavlink->kappa);
    mavlink_msg_trajectory_get_v(msg, pTrajectory_mavlink->v);
    break;
  case MAVLINK_MSG_ID_CMD_REQUEST_MSG: ; /* semicolon is needed here ... */
    uint8_t msgid = mavlink_msg_cmd_request_msg_get_msgid(msg);
    bool_t active = (bool_t) mavlink_msg_cmd_request_msg_get_active(msg);
    uint8_t period = mavlink_msg_cmd_request_msg_get_period(msg);
    configureOutgoingMessage(msgid, active, period, true);
    break;
  case MAVLINK_MSG_ID_CMD_REQUEST_STATECHANGE:
    *pRequestedState_mavlink = mavlink_msg_cmd_request_statechange_get_state(msg);
    break;
```

```
  default:
    io_printf("MAVLINK: Don't know how to handle message id %u\n", msg->msgid);
  }
  irq_enable();
}

/************* Public functions ************************************************************/

int8_t mavlink_init(void)
{
  io_printf("MAVLINK: initializing\n");
  uart.baudrate = MAVLINK_BAUDRATE;
  uart.channel = UART_CH3;
  if (uart_init(&uart, receiveBuffer, receiveBufferSize, transmitBuffer, transmitBufferSize, transmitBufferTimestamp, receiveBufferTimestamp)) {
    return -1;
  }

  mavlinkSystem.sysid = 0;
  mavlinkSystem.compid = MAV_COMP_ID_STM;

  for (uint8_t i = 0; i < NUM_MSGS; i++)
  {
    msgMeta[i].msgid = mavlinkMessageInfo[i].msgid;
    msgMeta[i].active = false;
    msgMeta[i].customRequest = false;
  }

  /* heartbeat is always active, might as well activate it here (and never deactivate ) */
  configureOutgoingMessage(MAVLINK_MSG_ID_HEARTBEAT, true, 10, false);

  prevTimeout = true;

  return 0;
}

void mavlink_getMessages(void)
{
  mavlink_message_t msg;
  mavlink_status_t status;
  uint8_t chan = 0;

  while(uart_readByte(&uart, (int8_t*) &byteBuffer))
  {
    if (mavlink_parse_char(chan, byteBuffer, &msg, &status))
    {
      handleMessage(&msg);
    }
  }

  bool timeout = (systime_getSysTime() - pHeartbeat_mavlink->lastReceiveTimestamp) > 2.5e5; // heartbeat timeout after 250ms
  if (timeout && !prevTimeout)
  {
    errorHandler_setError(ERROR_MAVLINK_Timeout);
    prevTimeout = true;
  }
  else if (!timeout && prevTimeout) {
    errorHandler_clearError(ERROR_MAVLINK_Timeout);
    prevTimeout = false;
  }
}

void mavlink_sendMessages(void)
```

```
{
  transmitcounter++;
  for (uint8_t i = 0; i < NUM_MSGS; i++)
  {
    if (msgMeta[i].active && (transmitcounter % msgMeta[i].period) == 0)
    {
      sendMessage(msgMeta[i].msgid);
    }
  }
}

void mavlink_activateStateMessages(uint8_t state)
{
  configureStateMessages(state, true);
}

void mavlink_deactivateStateMessages(uint8_t state)
{
  configureStateMessages(state, false);
}
```

stangproc.h

```
/** *******************************************************************************************
 * Unit in charge:
 * @file  stangproc.h
 * @author  hinze
 * $Date:: 2017-09-06 12:43:27 #$
 *
 * ----------------------------------------------------------
 * SVN information of last commit:
 * $Rev:: 2158                  $
 * $Author:: mbrieske     $
 *
 * ----------------------------------------------------------
 *
 * @brief The torque processing transforms the valid torque target value into the expected format and range of the actuating module.
 *
 *******************************************************************************************/

#ifndef STANGPROC_H_
#define STANGPROC_H_

/************* Includes ******************************************************************/
#include "brd/startup/stm32f4xx.h"
#include "dev/servo/servo.h"
#include "per/irq.h"
#include "sys/systime/systime.h"

/************* Public typedefs ***********************************************************/

/************* Macros and constants ****************************************************/

/************* Public function prototypes **********************************************/
/**
 * @brief Initializes the steering angle processing
 *
 * @param [in,out] void
```

```
 *
 * @return 0
 */
int8_t stangproc_init(void);

/**
 * @brief This function process the steering angle provided by application
 * steering angle target selector in a angle that can be read by the servo.
 * The process consists in an linear transformation
 *
 * @param [in,out] void
 *
 * @return void
 */
void stangproc_step(void);

#endif /* STANGPROC_H_ */
```

stangproc.c

```
/**********************************************************************************************
    ↪
 * Unit in charge:
 * @file     stangproc.c
 * @author   hinze
 * $Date:: 2017-09-06 12:43:27 #$
 *
 * ----------------------------------------------------------
 * SVN information of last commit:
 * $Rev:: 2158                  $
 * $Author:: neumerkel       $
 *
 * ----------------------------------------------------------
 *
 * @brief The steering processing consist in the linear transformation of the
     ↪ steering angle
 * in an angle which can be read by the servo device.
 *
 *
 *******************************************************************************************
     ↪ */

/************* Includes
    ↪  *****************************************************************/
#include "../stangproc.h"
#include "sys/util/io.h"

/************* Private typedefs
    ↪  **********************************************************/

/************* Macros and constants
    ↪  ******************************************************/

/************* Global variables
    ↪  **********************************************************/
/* Signal pointers for input signals */
float32_t* pStAngTgtSel_stangproc;

/* Signal pointers for output signals */
uint32_t* pStAngTimestamp_stangproc;
float32_t* pStAngTgt_stangproc;
float32_t* pStAngAct_stangproc;
float32_t* pSrvAngAct_stangproc;      /**< signal from servo  */

/* Parameter pointers */
```

```c
float32_t* pStAngMax_stangproc;
float32_t* pStAngMin_stangproc;
float32_t* pStAngTgtFailSafe_stangproc;
float32_t* pMinAbsAngSrv1_stangproc;
float32_t* pMaxAbsAngSrv1_stangproc;

/************* Private static variables ***********************************************/
static float32_t stAngMax = 0;
static float32_t stAngMin = 0;
static float32_t stAngTgtFailSafe = 0;
static float32_t stAngTgt = 0;
static float32_t srvAngMax = 0;
static float32_t srvAngMin = 0;

/************* Private function prototypes *********************************************/
static void updateParameters(void);
static float32_t transformAng(float32_t inAng);
static float32_t transformSvrAngToStAng(float32_t servoAngAct);

/************* Public functions ***************************************************/
int8_t stangproc_init(void)
{
  stAngMax = 0;
  stAngMin = 0;
  stAngTgtFailSafe = 0;
  stAngTgt = 0;
  srvAngMax = 0;
  srvAngMin = 0;

  /* fetch updated parameters */
  updateParameters();
  /* set output signals to init values */
  *pStAngTimestamp_stangproc = 0;
  *pStAngTgt_stangproc = stAngTgtFailSafe;

  return 0;
}

void stangproc_step(void)
{
  float32_t sigSrvAngAct = 0.0;
  /* fetch updated parameters */
  irq_disable();
  updateParameters();
  sigSrvAngAct = *pSrvAngAct_stangproc;
  /* store target value as intermediate taret value */
  stAngTgt = *pStAngTgtSel_stangproc;
  /* transform steer angle target value into servo steer angle target value */
  *pStAngTgt_stangproc = transformAng(stAngTgt);
  /* transform the actual servo angle value into the actual steer angle  */
  *pStAngAct_stangproc = transformSvrAngToStAng(sigSrvAngAct);
  /* publish timestamp for steering angle */
  *pStAngTimestamp_stangproc = systime_getSysTime();
  irq_enable();

  /* call servo device driver to update to the new target value */
  #ifndef USE_RX24F
  servo_updateAngleSrv1();
  #endif
}

/************* Private functions *************************************************/
```

```
static void updateParameters(void)
{
  /* update all parameters to current values from parameter system */
  stAngMax = *pStAngMax_stangproc;
  stAngMin = *pStAngMin_stangproc;
  srvAngMax = *pMaxAbsAngSrv1_stangproc;
  srvAngMin = *pMinAbsAngSrv1_stangproc;
  stAngTgtFailSafe = transformAng(*pStAngTgtFailSafe_stangproc);
}

static float32_t transformAng(float32_t inAng)
{
  float32_t ret = 0;
  float32_t m = 0;

  /* linear transformation of steer angle input into servo steer angle */
  m = (srvAngMax - srvAngMin) / (stAngMax - stAngMin);
    ret = m * (inAng - stAngMin) + srvAngMin;

  return ret;
}

/* transform the actual servo angle value into the actual steer angle  */
static float32_t transformSvrAngToStAng(float32_t servoAngAct)
{
  float32_t retStAngAct = 0.0;
  float32_t res = 0.0;

  /* linear transformation of servo steer angle input into steer angle */
  res = (servoAngAct - srvAngMin) / (srvAngMax - srvAngMin);
  retStAngAct = res * (stAngMax - stAngMin) + stAngMin;

  return retStAngAct;
}
```

vehspdctrl.h

```

/** ***************************************************************************************
 * Unit in charge:
 * @file  vehspdctrl.h
 * @author  Brieske
 * $Date:: 2017-09-12 08:12:11 #$
 *
 * ----------------------------------------------------------
 * SVN information of last commit:
 * $Rev:: 2022                  $
 * $Author:: Brieske            $
 *
 * ----------------------------------------------------------
 *
 * @brief TODO
 *
 *****************************************************************************************
     */

#ifndef VEHSPDCTRL_H_
#define VEHSPDCTRL_H_

/************* Includes
     ****************************************************************/
#include "brd/startup/stm32f4xx.h"
#include "per/irq.h"
#include "sys/algo/pid.h"
```

```c
#include "sys/algo/pt1.h"
/************* Public typedefs ************************************************************/

/************* Macros and constants *******************************************************/

/************* Public function prototypes ************************************************/
int8_t vehspdctrl_init(void);
/**
 * @brief Initialize the parameters and signals used by the application vehspdctrl
 *
 * @return int8_t success
 **/
void vehspdctrl_step(void);
/**
 * @brief TODO
 */

#endif /* VEHSPDPROC_H_ */
```

vehspdctrl.c

```c
/** ****************************************************************************************
 * Unit in charge:
 * @file  vehspdctrl.c
 * @author  Brieske
 * $Date:: 2017-09-12 08:12:11 #$
 *
 * ------------------------------------------------------------
 * SVN information of last commit:
 * $Rev:: 2022                  $
 * $Author:: Brieske            $
 *
 * ------------------------------------------------------------
 *
 * @brief TODO
 *
 ******************************************************************************************** */

/************* Includes *******************************************************************/
#include "../vehspdctrl.h"
#include "app/stm/stm.h"
#include "dev/rcrec/rcrec.h"
#include "dev/servo/servo.h"
#include "app/leddebug.h"

/************* Global variables ***********************************************************/
/* input signals */
float *pVehSpdOdom_vehspdctrl;
uint8_t *pState_vehspdctrl;
carcontrol_t *pCarControl_vehspdctrl;
//float *pVehSpdTgtTraj_vehspdctrl;
uint16_t *pRcChannel1_vehspdctrl;

/* output signals */

/* parameters */

```

```c
/************* Private typedefs ***********************************************************/

/************* Macros and constants *******************************************************/
#define TDELTA 0.02

#define PID_K_P 0.005
// No I value required since we do not need to correct a constant error.
#define PID_K_I 0
// D value to avoid overshoot.
#define PID_K_D 0.001

#define PID_GAIN 1

#define PT1_T 0.08
#define PT1_T_OPT (1. / (PT1_T / TDELTA) + 1)

#define MOT_MAXVAL 10.
// #define U_DEADBAND_POSITIVE 0.0583
// #define U_DEADBAND_NEGATIVE -0.075
#define U_DEADBAND_POSITIVE 0.02
#define U_DEADBAND_NEGATIVE -U_DEADBAND_POSITIVE * 4

#define MOTOR_LIMIT 0.4f
#define MOTOR_FULLSTOP -10
#define MOTOR_AFTER_FULLSTOP 0

#define MAX_SPEED 3.0f
#define FLOAT_TOLERANCE 0.0001f

#define MIN(a, b) (a < b ? a : b)
#define MAX(a, b) (a > b ? a : b)
#define ABS(x)    (x < 0 ? -x : x)

/************* Private static variables ***************************************************/
static pidCtrl_t pid;
static pt1_t pt1;

static float vehSpdOdom;
static uint8_t state;

static float motTrqTgtSrv;

// tells whether there was a fullstop on the last step
static bool lastFullstop;

/************* Private function prototypes ************************************************/

static float spdCtrlAssystem(float w);
static void spdCtrl(float destSpd);
static void setThrottle(float throttle);

/************* Public functions ***********************************************************/
int8_t vehspdctrl_init(void)
{
  pCarControl_vehspdctrl->vehSpdTgtExt = 0.0f;

  pt1.K = 1;
  pt1.T_opt = PT1_T_OPT;
  pt1.y_old = 0;

  pid.kp = PID_K_P;
```

```
  pid.ki = PID_K_I;
  pid.kd = PID_K_D;

//  pid.ki = 0.F;
//  pid.kd = 0.F;

  pid.gain = PID_GAIN;
  pid.ta = TDELTA;
  pid.min = -MOT_MAXVAL;
  pid.max = MOT_MAXVAL;

  pid_initialize(&pid);

  return 0;
}

void vehspdctrl_step(void)
{
  irq_disable();
  state = *pState_vehspdctrl;
  irq_enable();

  switch (state) {
  case SYSTEM_STATE_INITIALIZING:
    // motor is disabled, nothing to do here
    return;
  case SYSTEM_STATE_IDLE:
    motTrqTgtSrv = 0;
    lastFullstop = false;
  case SYSTEM_STATE_EMERGENCY:
    spdCtrl(0.F);
    break;
  case SYSTEM_STATE_RUNNING_RC:
    irq_disable();
    float rcvalue = (float) *pRcChannel1_vehspdctrl;
    irq_enable();
//    if (rcvalue < 1400) {
//      motTrqTgtSrv = 0;
//    }
//    else {
      motTrqTgtSrv = (rcvalue - 1500.0F) / (float) (MAX_PERIOD_CHANNEL -
          ↪ MIN_PERIOD_CHANNEL);
      motTrqTgtSrv *= 0.8;
//      motTrqTgtSrv *= 3;
//      motTrqTgtSrv = spdCtrl(motTrqTgtSrv);
//    }
    break;
  case SYSTEM_STATE_RUNNING_EXT:
    irq_disable();
    float spdTgt = pCarControl_vehspdctrl->vehSpdTgtExt;
    irq_enable();

    // debugging
    if (spdTgt != 0.0f)
      debugLedOn();
    else
      debugLedOff();

    spdCtrl(spdTgt);
    break;
//  case SYSTEM_STATE_RUNNING_TRAJ:
//    TODO system state not implemented yet
//    irq_disable();
//    float spdTgt = *pVehSpdTgtTraj_vehspdctrl;
//    irq_enable();
//    motTrqTgtSrv = spdCtrl(spdTgt);
```

```c
//     break
  default:
    // cannot go here
    break;
  }

  servo_updateMotorTrq(motTrqTgtSrv);
}

/************* Private functions *********************************************************/

static float spdCtrlAssystem(float w)
{
  float u = 0;  // manipulated value
  float u_f = 0;  // feed forward value
  float y = 0;  // controlled value

  irq_disable();
  y = *pVehSpdOdom_vehspdctrl;
  irq_enable();

  y = pt1_calculate(&pt1, y);
  pid_controller(&pid, y, w, &u, -MOT_MAXVAL, MOT_MAXVAL);

  if (u > 0) {
    u_f = U_DEADBAND_POSITIVE;
  }
  else if (u < 0) {
    u_f = U_DEADBAND_NEGATIVE;
  }

  u += u_f;

  return u;
}

static void spdCtrl(float destSpeed)
{
  destSpeed = MIN(destSpeed, MAX_SPEED);

  // Here we first check if speed 0 was received.
  // In this case we perform a fullstop with maximum brake power.
  // This might have been implemented differently but there wasn't enough time at this point.
  // However this functionality is definitely required since when left out the PID tries to smoothly reduce speed to 0
  // which is not what we desire when trying to avoid a collision.
  bool fullstop;
  if (destSpeed <= FLOAT_TOLERANCE)
    fullstop = true;
  else
    fullstop = false;

  if (!fullstop && lastFullstop)
  {
    motTrqTgtSrv = MOTOR_AFTER_FULLSTOP;
    pid_initialize(&pid);
  }

  lastFullstop = fullstop;

  if (fullstop)
  {
    motTrqTgtSrv = MOTOR_FULLSTOP;
    return;
```

```
  }

  float change = 0;    // value calculated by PID
  float currSpeed = 0;  // controlled value

  irq_disable();
  currSpeed = ABS(*pVehSpdOdom_vehspdctrl);
  irq_enable();

  //currSpeed = pt1_calculate(&pt1, currSpeed);
  pid_controller(&pid, currSpeed, destSpeed, &change, -MOT_MAXVAL, MOT_MAXVAL);

  motTrqTgtSrv += change;

  if (motTrqTgtSrv > 0 && motTrqTgtSrv < U_DEADBAND_POSITIVE)
    motTrqTgtSrv = U_DEADBAND_POSITIVE;

  motTrqTgtSrv = MIN(motTrqTgtSrv, MOTOR_LIMIT);
}

static void setThrottle(float throttle)
{
  throttle = MIN(throttle, 1.0f);
  throttle = MAX(throttle, 0.0f);

  if (throttle == 0.0f)
  {
    motTrqTgtSrv = MOTOR_FULLSTOP;
    return;
  }

  motTrqTgtSrv = throttle;
}
```

eict.h

```

/** ******************************************************************************************
 * Unit in charge:
 * @file  eict.h
 * @author  Yoga
 * $Date:: 2015-05-05 13:31:20 #$
 *
 * -----------------------------------------------------------
 * SVN information of last commit:
 * $Rev::                    $
 * $Author::            $
 *
 * -----------------------------------------------------------
 *
 * @brief This unit configures an input capture timer triggering on the edges of the wheel encoder.
 *
 ****************************************************************************************** */


#ifndef EICT_H_
#define EICT_H_

/************* Includes *****************************************************************/
#include "per/hwallocation.h"

```

```c
/************* Public typedefs ***************************************************************/

/************* Privat typedefs ***************************************************************/

/************* Macros and constants ********************************************************/
/************* Public function prototypes *************************************************/
void eict_init(void);

void eict_getCounter(uint16_t *frontLeftEnc, uint16_t *frontRightEnc, uint16_t *rearLeftEnc, uint16_t *rearRightEnc);

void eict_getCounter2(uint16_t *rearLeftEnc, uint16_t *rearRightEnc);

#endif /* EICT_H_ */
```

eict.c

```c
/** ************************************************************************************************
 * Unit in charge:
 * @file  eict.h
 * @author  Yoga
 * $Date:: 2015-05-05 13:31:20 #$
 *
 * ----------------------------------------------------------
 * SVN information of last commit:
 * $Rev::                 $
 * $Author::            $
 *
 * ----------------------------------------------------------
 *
 * @brief This unit configures an input capture timer triggering on the both edges of the wheel encoder signal.
 *
 ************************************************************************************************ */

/************* Includes *********************************************************************/
#include "brd/startup/stm32f4xx.h"
#include "app/config.h"
#include "per/eict.h"
#include "per/irq.h"


/************* Private typedefs **************************************************************/
/************* Macros and constants ********************************************************/
#define SR_INC_MAX       100

/************* Globale Variable ************************************************************/

/************* Private static variables ***************************************************/
static uint16_t frontLeftCapture = 0;
static uint16_t frontRightCapture = 0;
static uint16_t rearLeftCapture = 0;
static uint16_t rearRightCapture = 0;

```

```c
static uint16_t rearLeftCapture2 = 0;
static uint16_t rearRightCapture2 = 0;

/************* Private function prototypes *************************************************/
static void gpio_config(void);

static void nvic_config(void);

/************* Public functions ********************************************************/
void eict_init(void)
{

#ifdef WENC_V3
  uint16_t polarity = TIM_ICPolarity_Falling;
#else
  uint16_t polarity = TIM_ICPolarity_Rising;
#endif

  TIM_ICInitTypeDef TIM_ICStructure;

  /* GPIO configuration */
  gpio_config();

  /* nvic configuration */
  nvic_config();

#ifdef FOUR_WHEEL_ODOM
  /* TIMER3 channel 1(PB4) Configuration:  Input Capture mode */
  TIM_ICStructure.TIM_Channel     = TIM_Channel_1;
  TIM_ICStructure.TIM_ICFilter    = 0x0;
  TIM_ICStructure.TIM_ICPolarity  = polarity;    /* rising and falling edges are used as active edge */
  TIM_ICStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;       /* triggering on each edge */
  TIM_ICStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
  TIM_ICInit(WENC_FRONT_TIMER, &TIM_ICStructure);

  /* TIMER3 channel 2(PB5) Configuration:  Input Capture mode */
  TIM_ICStructure.TIM_Channel     = TIM_Channel_2;
  TIM_ICStructure.TIM_ICFilter    = 0x0;
  TIM_ICStructure.TIM_ICPolarity  = polarity;    /* rising and falling edges are used as active edge */
  TIM_ICStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;       /* triggering on each edge */
  TIM_ICStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
  TIM_ICInit(WENC_FRONT_TIMER, &TIM_ICStructure);
#endif

  /* TIMER2 channel 4(PB11) Configuration:  Input Capture mode */
  TIM_ICStructure.TIM_Channel     = TIM_Channel_4;
  TIM_ICStructure.TIM_ICFilter    = 0x0;
  TIM_ICStructure.TIM_ICPolarity  = polarity;    /* rising and falling edges are used as active edge */
  TIM_ICStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;       /* triggering on each edge */
  TIM_ICStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
  TIM_ICInit(WENC_REAR_TIMER, &TIM_ICStructure);

  /* TIMER2 channel 2(PB3) Configuration:  Input Capture mode */
  TIM_ICStructure.TIM_Channel     = TIM_Channel_2;
  TIM_ICStructure.TIM_ICFilter    = 0x0;
  TIM_ICStructure.TIM_ICPolarity  = polarity;    /* rising and falling edges are used as active edge */
```

```c
  TIM_ICStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;       /* triggering on each edge */
  TIM_ICStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
  TIM_ICInit(WENC_REAR_TIMER, &TIM_ICStructure);
#ifdef FOUR_WHEEL_ODOM
    /* TIMER3 enable counter */
  TIM_Cmd (WENC_FRONT_TIMER, ENABLE);

  /* Timer 3 enable the CC Interrupt request */
  TIM_ITConfig(WENC_FRONT_TIMER, TIM_IT_CC1 | TIM_IT_CC2, ENABLE);

  /* Timer 3 clear CC Flag */
  TIM_ClearFlag(WENC_FRONT_TIMER, TIM_IT_CC1 | TIM_IT_CC2);
#endif

   /* TIMER2 enable counter */
  TIM_Cmd (WENC_REAR_TIMER, ENABLE);

  /* Timer 2 enable the CC Interrupt request */
  TIM_ITConfig(WENC_REAR_TIMER,TIM_IT_CC4 | TIM_IT_CC2, ENABLE);

  /* Timer 2 clear CC Flag */
  TIM_ClearFlag(WENC_REAR_TIMER, TIM_IT_CC4 | TIM_IT_CC2);
}

void eict_getCounter(uint16_t *frontLeftEnc, uint16_t *frontRightEnc, uint16_t *rearLeftEnc, uint16_t *rearRightEnc)
{
  irq_disable();

  *frontLeftEnc  = frontLeftCapture;
  *frontRightEnc = frontRightCapture;
  *rearLeftEnc   = rearLeftCapture;
  *rearRightEnc  = rearRightCapture;

  /* clear capture Counter */
  frontLeftCapture = 0;
  frontRightCapture = 0;
  rearLeftCapture = 0;
  rearRightCapture = 0;

  irq_enable();
}

void eict_getCounter2(uint16_t *rearLeftEnc, uint16_t *rearRightEnc)
{
  irq_disable();

  *rearLeftEnc   = rearLeftCapture2;
  *rearRightEnc  = rearRightCapture2;

  /* clear capture Counter */
  rearLeftCapture2 = 0;
  rearRightCapture2 = 0;

  irq_enable();
}

#ifdef FOUR_WHEEL_ODOM
void TIM3_IRQHandler()
{
  if(TIM_GetITStatus(WENC_FRONT_TIMER, TIM_IT_CC1)) {
    TIM_ClearITPendingBit(WENC_FRONT_TIMER, TIM_IT_CC1);
    frontLeftCapture++;

  }
```

```
  if(TIM_GetITStatus(WENC_FRONT_TIMER, TIM_IT_CC2)) {
    TIM_ClearITPendingBit(WENC_FRONT_TIMER, TIM_IT_CC2);
    frontRightCapture++;
  }
}
#endif

void TIM2_IRQHandler()
{
  if(TIM_GetITStatus(WENC_REAR_TIMER, TIM_IT_CC4)) {
    TIM_ClearITPendingBit(WENC_REAR_TIMER, TIM_IT_CC4);
    rearLeftCapture++;
    rearLeftCapture2++;
  }
  if(TIM_GetITStatus(WENC_REAR_TIMER, TIM_IT_CC2)) {
    TIM_ClearITPendingBit(WENC_REAR_TIMER, TIM_IT_CC2);
    rearRightCapture++;
    rearRightCapture2++;
  }
}

/************* Private function ************************************************/
static void gpio_config(void)
{

  GPIO_InitTypeDef GPIO_InitStructure;

  /* GPIOB clock enable */
  RCC_AHB1PeriphClockCmd(WENC_PIN_CLOCK, ENABLE);

  /* Pin configuration for Timer 2 and 3 */
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Pin = WENC_RR_CHAN_2_PIN | WENC_FL_CHAN_1_PIN | WENC_FR_CHAN_2_PIN | WENC_RL_CHAN_4_PIN;
  GPIO_Init(WENC_PORT, &GPIO_InitStructure);

  /* Configure the the GPIO pin as alternate function */
  GPIO_PinAFConfig(WENC_PORT, WENC_FL_CHAN_1_AF_PIN, WENC_FL_CHAN_1_AF);
  GPIO_PinAFConfig(WENC_PORT, WENC_FR_CHAN_2_AF_PIN, WENC_FR_CHAN_2_AF);
  GPIO_PinAFConfig(WENC_PORT, WENC_RL_CHAN_4_AF_PIN, WENC_RL_CHAN_4_AF);
  GPIO_PinAFConfig(WENC_PORT, WENC_RR_CHAN_2_AF_PIN, WENC_RR_CHAN_2_AF);
}

static void nvic_config(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;

  /* Timer 2 peripheral clock enable */
  RCC_APB1PeriphClockCmd (WENC_REAR_TIM_CLOCK, ENABLE);

  /* Timer 3 peripheral clock enable */
  RCC_APB1PeriphClockCmd (WENC_FRONT_TIM_CLOCK, ENABLE);

#ifdef FOUR_WHEEL_ODOM
  /* enable the TIM3 global Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x01;
  NVIC_Init(&NVIC_InitStructure);
#endif

  /* enable the TIM2 global Interrupt */
```

```
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x00;
  NVIC_Init(&NVIC_InitStructure);
}
```

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den December 18, 2018 ....