Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017

Project Title:	**Drive Like Me - Personalisation of Autonomous Driving Behaviour**
Student:	**Alina Walch**
CID:	**00869949**
Course:	**EEE4**
Project Supervisor:	**Prof Yiannis Demiris**
Second Marker:	**Dr Krystian Mikolajczyk**

Acknowledgements

I would like to thank my academic supervisor Professor Yiannis Demiris for his helpful advice, constructive feedback and invaluable mentoring throughout the course of my project.

I further want to thank the entire research team in the Personal Robotics Lab, in particular Dr Theodosis Georgiou, for the many practical tips, for the incredible support and especially for the welcoming environment in the lab.

Last but not least, I want to thank my friends and family. Thank you for supporting me throughout the 4 years of my studies at Imperial College, and particularly for the many inspiring discussions that helped me bring this project forward.

Abstract

The aim of this project is the design and implementation of a controller for an autonomous vehicle that imitates human driving behaviour. For this, human driving data is collected from a virtual simulator and used to train a model which is capable of producing commands that resemble the user's driving style. In a two-stage approach, the generation of personalized high-level targets was separated from the computation of low-level driving commands. The high-level targets, namely target speed and target position of the vehicle on the track, are generated using a machine learning model. The low-level controls for throttle and brake pressure as well as the steering angle are implemented as a PID controller. The PID controller assures a robust performance on all test tracks and a reliable completion of all laps. An evaluation of k-nearest neighbor search and feed-forward neural networks showed the potential of these approaches to learn and imitate human driving behaviour. The implementation of the two-stage controller was capable of accurately imitating the human driving style. This was evaluated using an innovative analysis approach that combines metrics related to the speed and the lateral deviation of the controller from the human demonstration.

Table of contents

List of figures

List of tables

1 Introduction

1.1 Motivation

Autonomous driving is a quickly developing field with high aspirations to bring self-driving vehicles to the market within the next 5 years [1]. Despite the fast technological developments however, it is still unclear how well autonomous cars will be accepted by the general public. There are various factors that will determine the adoption of driverless cars. Safety and reliability are obvious factors, however driving comfort has also been identified as a key aspect to the user experience [2].

A feeling of comfort in an autonomously driven vehicle is very subjective. Hence, an autonomous vehicle must interact with its conductor in order to improve his sense of comfort. Personalising the driving behaviour of an autonomous vehicle has been investigated for automatic cruise control systems [3] with the goal of increasing the driver's comfort. Learning the preferences of a driver can make a significant contribution to a safer road traffic, as it can increase the acceptance of autonomous vehicles.

While interaction between the human and the vehicle during human driving has been an important topic of research for many years [4–6], not much has been done in the field of learning and replicating how a human drives. Even though it might not be desirable to imitate exactly what a human would do in every situation, it is certainly very beneficial to understand which preferences he has. An autonomous vehicle could then drive according to the user's preferences, but without dismissing safety standards.

Autonomous road driving however is not the only environment where an understanding of the user can make a difference. To people who are forced to sit in a wheelchair, a personalisation of the wheelchair can not only lead to a better acceptance of autonomous wheelchairs, but also result in a singificant improvement of the quality of living of the patients [7, 8].

1.2 Aims and Objectives

The aim of this project was to explore methods of personalising the behaviour of an autonomous vehicle, and to develop a controller for such a vehicle that is capable of driving just like a specific human driver would. There are multiple applications for which this personalised controller could be developed. Within the area of road vehicles, possible applications could include highway driving, driving in urban environments, or race driving.

For the implementation of this project, a single-vehicle car racing simulation was selected for testing of the developed controller. This assured the necessary simplicity in order to develop an

end-to-end system within the scope of the project, but provides realistic simulation results in order to draw general conclusions on the performance of the approach.

Within the scope of a single-vehicle racing simulation, human driving style was characterised by the vehicle's trajectory. The trajectory is defined in space and time as the speed of the vehicle and the deviation from the center of the track it is driving on. This characterisation is targeted in particular to identifying the turning behaviour of the human, i.e. how he approaches sharp turns or long smooth curves.

The controller was expected to learn the behaviour of the human driver based on the driving data obtained from the human, who has completed a set of tracks. Based on this data, a personalised model representing the human driving style was constructed. This model was then used to compute low-level driving commands in order to autonomously control a vehicle in the simulation. The low-level commands that were controlled include the throttle and brake pressure, as well as the steering angle.

In addition to the development of the controller software, the secondary objective of the project was to build a simulation platform that includes a physical driving simulator, a virtual racing game and a backend that processes the driving data and can send commands to control an autonomous vehicle.

1.3 Contributions

The deliverables of this project can be divided in three different categories: The development of the simulation environment, the implementation of different controller methods and the analysis of the performance of the developed controllers. The development of the simulation environment included contributions to the implementation of the virtual racing game itself, as well as the development of a platform to handle the data from the simulator and send commands to the vehicle. The delivered system provides a wide range of driving data, contains versatile structures to handle the data in a practical way and can seamlessly control an autonomous racing car. The platform was designed to be easily adaptable for future related research.

For the modelling of the human driver, three different methods were implemented and tested. All methods were based on an approach that forecasts a high-level driving objective using a machine-learning method and uses a direct control method in order to obtain the desired controls for throttle, brake and steering. The machine learning methods that were used are k-nearest neighbor regression, feed-forward neural networks and recurrent neural networks. Most controllers that were developed from the models showed excellent results in known environments, and good results in unknown environments. Further improvements to the models were suggested.

Finally, a comprehensive framework for analysing and comparing driving styles was developed, which goes beyond the frequently-used method of quantifying driving performance as the total progress along the track in a given time. In addition to this metric, the statistics for the speed

along the track and the distance from the center of the track were used to compare the models. These were identified as the most expressive metrics to measure driving style in the single-vehicle context. Based on this framework, an extensive analysis was carried out to assess the performance of the models developed for imitating human driving behaviour.

1.4 Report Structure

This report is structured as follows: Chapter 2 provides a background analysis of applications of the project, possible design approaches and common methods of evaluation based on the existing literature. Secondly, details of the requirements for this controller software and the simulation platform are given in chapter 3.

The analysis, design and implementation of three different controller methods are explained in chapters 4 and 5. The results and comparison of the testing of the different approaches are presented in chapter 6. Finally, the project outcomes, successes and challenges are discussed in chapter 7, and further work is suggested in chapter 8.

2 Background

To give a solid understanding of the context of the project and the work that has been carried out, this chapter is split into three sections. First, different applications of a personalised vehicle controller are explained and areas of previous research are highlighted. Secondly, different controller design approaches are analysed and their advantages and disadvantages are compared. Finally, methods to evaluate the performance of the controllers are assessed in terms of their usefulness in evaluating the similarity of the model to a human driver.

2.1 Applications

The development of controllers to imitate human driving behaviour in autonomously driven vehicles has attracted most attention in the area of virtual car racing games [9–12]. Virtual games are often pioneering the development of cutting-edge technologies and test new algorithmic approaches long before they become commercially available in the real world [13].

Personalisation of game content has formed an important part of game development for a long time, as it can be used to create content which is individually challenging to each user and thus makes playing a game more fun [11]. Developing new ways of imitating a player has hence attracted attention from researchers and industry. Microsoft, for example, developed a drivatar that represents a virtual model of the driver, which can be used to exploit new tracks or sent to other players to compare playing styles [10].

While the prediction of human driving behaviour in real vehicle driving has been a focus in the development of driving assistance systems, the reproduction of individual driving styles has not gained much attention yet [5]. The recent developments towards autonomous driving, however, has moved the personalisation of autonomous driving behaviour into the focus of academic and industrial research. Modeling individual users and adapting their driving style can improve the comfort of the user and thus reduce the barriers to adopting this new technology [2].

Experienced drivers may have a good intuition as to why such a personalisation can have a key impact: Sitting in someone else's car can cause a feeling of discomfort, for example when the driver approaches other cars too closely or is used to strong acceleration or deceleration. Other people become impatient if someone drives particularly slowly or carefully.

The same phenomenon can be experienced with current adaptive cruise control (ACC) systems. They often accelerate quickly, for example when a slow vehicle clears the lane, and break strongly. De Gelder *et al.* have identified this issue and proposed a personalised ACC which takes the user driving style into account [3]. This approach can lead to a feeling of more safety and comfort for experienced drivers, and hence a personalisation of the vehicle could be one solution to attracting more drivers to buying fully autonomous vehicles.

Adapting to human behaviour is significant in many areas where an autonomous system facilitates the human to carry out important tasks. Another type of vehicle which is very key to humans are wheelchairs for physically impaired people. For many users the wheelchair is a "part of their body" [7], so it should integrate seamlessly with the user's behaviour and habits. Padir [7] further identified co-designing the wheelchair assistance together with the user as a key feature to achieve acceptance and usability.

Shared control systems that give the user a partial control over the wheelchair, but interfere when needed, have been investigated by Carlson *et al.* [8]. Continuously learning the user's driving behaviour and preferences can be used to help patients with a quickly deteriorating condition and improve their long-term comfort. While still capable, the users can train the wheelchair on their own driving behaviour, which can be reproduced even if the user cannot control it any more.

2.2 Human Driver Modelling

The key objective of this project is the development of a controller that takes the current state of an autonomous vehicle as input, and outputs the low-level driving commands for steering, throttle and brake. This differs from the objectives of most real-world personalised driving controllers, as autonomous vehicles are typically directly controlled through a target trajectory [2, 3, 14].

Consequently, the analysis of related system designs was focused on literature in the field of car racing games. With the key objective of creating more entertaining game content, several attempts have been made to imitate human driving behaviour. In these projects, different methods of driver modelling were investigated, which can be divided into direct and indirect approaches [10–12].

Direct methods are classified as all methods which extract a model directly from the collected test data to compute the control signals, in particular the throttle, brake and steering commands. These methods use supervised learning approaches such as k-nearest neighbour search or neural networks which are trained using backpropagation [10, 11]. Section 2.2.1 provides some details on these supervised learning algorithms. Direct approaches were found to accurately model the driving behaviour until a small perturbation or an unknown situation caused the vehicle to crash [11]. Hence, they are not a preferred implementation for the desired purpose.

Indirect methods on the other hand are defined in the literature as algorithms which train a controller to exploit the best trade-offs between multiple (higher-level) objectives (e.g. progress along the track and speed). Indirect approaches are typically based on a feed-forward or recurrent neural network, which is evolved using multi-objective optimisation [10–12]. The literature presents different fitness functions used for the optimisation, which are discussed in section 2.3.

The evaluation of the different implementations of indirect methods showed an inherent trade-off between driver modelling and robustness of the controller. This means that in most cases, the resulting controller was either found to poorly imitate the human behaviour, or to imitate the

human behaviour well but to be sensitive to disturbances. Furthermore, the determination of appropriate objectives and suitable weights proves to be highly challenging [10].

Cardamone *et al.* propose a method that uses a two-stage design approach [9]. In this approach, a direct method is used to predict high-level actions of the car, namely the target speed and the target position of the vehicle with respect to the track center. A low-level linear controller then computes the driving signals based on the predicted high-level actions. Splitting the imitation learning task into modelling of strategic goals and low-level actions is an approach that has also been applied to the development of non-player characters for other computer games [15].

The two-stage approach showed robust performance of the controller, which was at the same time capable of following the trajectory of the modelled driver accurately [9]. Limitations of the performance of this approach are a poor performance in unknown environments and the introduction of a significant lag from the simple low-level controller [9].

Table 2.1 summarises the advantages and disadvantages of the different approaches considered in this analysis. The direct modelling approach was not further considered, as it showed poor performance in all tested applications. The indirect modelling approach has achieved some good results, but the inherent trade-off between robustness and imitation performance was found to be a serious limitation in all previous attempts [10–12].

The two-stage modelling approach has not gained much attention yet, but the initial results presented in [9] are promising as the controller is both robust and capable of imitating human driving behaviour well. Furthermore, this approach is similar to the control which is performed by autonomous road vehicles. It effectively introduces a layer of abstraction between the personalised high-level objectives and the translation of these into low-level driving commands.

For these reasons, the two-stage driver modelling approach was chosen as the starting point for the project. It appears to offer the best trade-off between reliable performance and personalised behaviour and is most closely related to real-world applications.

Table 2.1 Comparison of control designs based on the methods proposed in [9–12]. Two-stage direct modelling was chosen as the starting point for the project as it exhibited the best potential to achieve a robust implementation that represents a human driving style.

Approach	**Advantage**	**Disadvantage**
Direct Modelling	+ Straight-forward implementation	- Poor overall performance
Indirect Modelling	+ Can achieve good overall performance	- Trade-off between robustness and driver imitation - Difficult to choose appropriate fitness function and weights
Two-stage Direct Modelling	+ Robust performance + Good personalisation	- Difficulty to generalise to unseen environments - Low-level controller causes lag

2.2.1 Machine Learning Methods

This subsection aims to categorise the numerous machine learning methods that have been used or considered in the literature for modelling human driving behaviour in a comprehensive way. This categorisation is incomplete, as a vast number of machine learning approaches exist and are constantly being developed further. However, it should give an overview of the tools which are available for developing the controller.

K-nearest Neighbor Regression

The k-nearest neighbour (KNN) regression method is a straight-forward approach which directly uses the test data as the model. The set of features (inputs) and labels (outputs) for the entire data is computed. For every sample, the pool of test data is searched for the closest neighbours, i.e. the data points with the most similar features. The mean of the labels of these features then represent the new output [16]. This straight-forward approach showed surprisingly good results in both cases in which it was used for imitation of driving behaviour, but has limited performance in unseen situations [9, 11].

Neural Networks

Most methods considered in the literature are based on neural networks. Neural networks are structures that map inputs to outputs via a sequence of interconnected (hidden) layers of neurons or nodes. At each layer, the outputs are calculated as a weighted average of all connected inputs and a bias term, and multiplied by an activation function. The outputs of one layer then serve as the inputs of the next layer until the desired outputs of the system are reached.

Equation 2.1 shows the calculation of the outputs at each node $n_{i,j}$, where i indicates the layer and j determines the neuron within the layer. The input vector $\mathbf{x}$ is a vector of values from the previous layer, which is multiplied by the weights $\mathbf{w}$ and added to a bias term b. ϕ denotes the activation function that adds non-linearity to the network. Where specified, a *tanh* activation function was chosen in the analysed literature [10]. More details of the principles of neural networks can be found in [17].

$$n_{i,j} = \phi(\mathbf{w_i} * \mathbf{x_i} + b_{i,j}) \tag{2.1}$$

In order to design a neural network, a number of choices must be made: The designer must decide on the structure of the network, i.e. the number of layers and the size of each layer, as well as the cost function used to train the weights and biases of the input. These are highly dependent on the type of problem to be solved as well as the amount of training data available, as discussed in detail in [18]. Other design choices include the type of optimiser, where a gradient descent otimisation is most frequently used.

In contrast to feed-forward neural networks as described above, recurrent neural networks do not only take the current inputs into account, but also consider the inputs from the previous

time steps. This is implemented by adding memory to the nodes, making the output of every node a combination of the inputs and the previous output (or state) as shown in equation 2.2. The weights u determine how much of the inputs is to be retained, i.e. how strong the temporal relationship is [19].

$$n_{i,j,t} = \phi(\mathbf{w_i} * \mathbf{x_i} + u_{i,j} * n_{i,j,t-1}) \tag{2.2}$$

This design is particularly useful for characterising time-dependencies, which makes it an interesting approach within the scope of this project. Driving commands of a human driver most likely depend not only on the current situation but also the past, and hence the computational model of the human driver should also account for this time-dependency, as seen in [10]. One interesting method of implementing recurrent neural networks is using long short term memory (LSTM) units [19].

Evolutionary Algorithms

Evolutionary algorithms, which were applied in the indirect approaches to predict human behaviour, describe high-level optimisation methods and are frequently based on neural networks structures. The aim of these algorithms is to find the hyper-parameters of a model that are a best fit to a given objective. The objective may be a strategic goal, a Pareto front of optimal solutions to a multi-objective function or a maximum reward.

One evolutionary learning method that has proved useful for learning driving behaviour is reinforcement learning (RL). This approach is based on assigning numerical rewards from physical actions in a trial-and-error testing process [20]. Loiacono *et al.* use this apporach to learn optimal overtaking behaviour using a look-up table of previously exploited actions and their rewards [21]. Gorman *et al.* [15] combine clustering and reinforcement learning methods to develop a virtual video game player that imitates human behaviour.

Kuderer *et al.* perform inverse reinforcement learning (IRL) in order to determine the weights of a reward function from test data. The reward function is made up of 8 features that are calculated from the test data. This learnt cost function is then used to optimise the trajectory of an autonomous road vehicle [2].

Another approach to imitation learning, which is based on reinforcement learning methods, is Generative Adversarial Imitation Learning (GAIL) [22]. It is an approach aimed to learn a behavioural policy from expert behaviour. The idea is to design a model-free algorithm that induces the required action directly from a learned cost function. The method has been found to reproduce several machine learning tasks better than other methods of comparison, but comes at a high computational cost [22].

Kuefler *et al.* apply GAIL to imitating driving behaviour in a realistic highway-driving scenario. For this, they adapt GAIL to regression tasks and produce a policy which provides reliable results over long-term prediction horizons. This achieves a successful imitation of highway driving characteristics, measured for example through lane change rate, collision rate or turn rate [23].

In a different approach, Priesterjahn *et al.* [24] outline a rule-based approach to imitation learning. They define a set of rules with a well-defined input and output which defines the imitated player (agent). The authors claim that this method yielded very satisfactory results in modeling players for the *Quake3* computer game while requiring much less time and computational power than neural network approaches. However, the maximum complexity of this approach is limited as the rule base must be made of a set which can be searched in a realistic time [24].

Gaussian Process Regression

Gaussian process (GP) regression describes methods which are used to predict a behaviour in the form of a probabilistic distribution. A set of observations is used to adapt a *prior*, i.e. an initial estimate of the distribution. The resulting *posterior* distribution contains information about the likelihood of different outcomes, so that the confidence interval of each prediction is known.

GPs are collections of random variables, which have a joint Gaussian distribution when combined [25]. A Gaussian process is specified entirely by its mean and covariance function. The covariance function, or kernel, describes the joint variation of the random variables of the process and is thus a function of the model inputs. The choice of the covariance kernel thus has an important impact on the final performance of the model. A detailed explanation of Gaussian process models can be found in [25].

Georgiou *et al.* [26] used Gaussian Processes to construct a user behaviour model, consisting of 33 independent Gaussian Processes predicting the future state of the vehicle. The driving data was split into segments of similar curvatures. The information on the course of the track was omitted for reasons of computational efficiency [26].

Comparison

Table 2.2 shows a comparison of the different learning methods outlined previously. It highlights the underlying approach, the model type and the computational efficiency. While data-driven models are derived directly from the test data, rule-based approaches require the formulation of a higher-level cost or reward function. Such a reward function is used to characterise, for example, lane changing or collisions [23].

Many of the methods outlined in this section were tested and evaluated on their performance of predicting the next state of the user. The task for this project, however, is to infer the required control action for an autonomous virtual vehicle to drive in a human-like manner. Thus, the vehicle targets must be computed during simulation time at every time step. Hence, computational efficiency of the model is of high importance. As multiobjective evolution has not provided very robust results in the past (as explained previously), KNN and Neural Nets were chosen for the implementation of this project.

Table 2.2 Comparison of learning methods that were applied for similar tasks. In data-driven approaches, the model is directly derived from the test data, while rule-based methods predict a higher-order objective. Computational efficiency for online application is of high importance for the given task.

Method	**Approach**	**Model type**	**Efficiency**
KNN	Search data set for similar situations	Data driven	High
Neural Nets	Train network weights to minimise a cost function	Data driven	High
Multiobjective evolution	Enhance network weights towards a multiobjective goal	Rule-based	High
RL/IRL	Assign numerical awards to action/ Learn weights of a reward function	Rule-based	Low
GAIL	Combine RL and IRL to obtain a model-free behaviour policy	Rule-based	Low
GPR	Find probability distribution of likely next action	Data driven	Low

Kuefler *et al.* [23] provide an informative comparison of some of the learning methods outlined above. The analysis distinguishes between *behavioural cloning*, which is equivalent to the data-driven models in table 2.2, and the GAIL method. While the benefits of the GAIL implementation can be clearly observed for the imitation of high-level objectives such as lane changing or off-road driving, the behavioural cloning actually behaved better in terms of acceleration and speed performance, as well as the turn rate [23]. As this is the level of abstraction which is of interest for the analysis carried out in this project, a data-driven algorithm appears to be the most promising approach.

2.2.2 Driving Control Approaches

In the chosen system design, the prediction of a high-level target is separated from the computation of low-level driving commands. The high-level target trajectory is defined as the target speed and the target distance of the vehicle to the center of the track, while the low-level driving commands are the throttle, brake and steering commands.

Using this approach, it is possible to apply common trajectory-following methods for autonomous vehicles, including both real cars (see [27–29]) and simulated vehicles (see [30]). For this analysis, a number of approaches have been considered for the low-level control of the vehicle, including a simple linear controller, a proportional-integral-derivative (PID) controller and a model predictive control (MPC) approach.

Furthermore, advanced methods for controlling the steering based on the generation of sub-goals for the vehicle motion are compared. The design of the selected approach is explained in section 4.2.2.

Linear Controller

In their research on a two-stage controller to imitate human driving behaviour, Cardamone *et al.* [9] suggest to obtain the target values for the low-level driving commands directly from the errors in speed and trajectory of the vehicle. They propose the following linear equations:

$$throttle_{t+\Delta t} = 0.05 * (speed_{t+\Delta t} - speed_t), \quad throttle_{t+\Delta t} \in [0;1] \tag{2.3}$$

$$brake_{t+\Delta t} = -0.05 * (speed_{t+\Delta t} - speed_t), \quad brake_{t+\Delta t} \in [0;1] \tag{2.4}$$

$$steering_{t+\Delta t} = \varepsilon * (distance_{t+\Delta t} - distance_t), \quad steering_{t+\Delta t} \in [-1;1] \tag{2.5}$$

The quality of the results obtained with this approach is analysed in section 4.2. As this approach is simple, it may not yield very satisfactory results. Hence, more advanced control methods were evaluated as well.

PID Controller

PID controllers are the most common method of controlling autonomous vehicles, both in the real and in the virtual environment [29–32]. They are feedback controllers, which means that the outputs of the system to be controlled are fed back to the input of the system. The current output of the system is subtracted from the desired output, giving an error $e(t)$. This error is to be minimised by the controller.

PID controllers are beneficial as they are straight-forward to implement and offer flexibility in the design through three controller gains. These gains are defined as the proportional gain (K_p), which multiplies the current error by a constant term, the integral gain (K_i), which removes steady-state errors, and the derivative gain (K_d), which controls the rate of change of the error. The given control signal ($u(t)$) is thus computed from the error as shown in equation 2.6:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)dt + K_d \cdot \frac{d}{dt}e(t) \tag{2.6}$$

For the implementation of a vehicle controller, typically the steering control is separated from the speed control ([29, 30]). Speed control entails both the control for the throttle and for the brake, which is calculated from the same error. It thus seems logical to have one speed controller only, and not two controllers as suggested in equations 2.3 & 2.4.

Mange *et al.* [29] explain the significance of the gain terms for a speed and steering controller. In their implementation, the error of the steering controller is defined as the difference in the current position and the target position. Thus, the proportional gain compensates directly for position errors, while the integral gain counteracts a lateral movement of the vehicle, e.g. a slope of the track. The derivative component removes direction errors.

For the speed control, the error is given by the difference between the target speed and the current speed, which is controlled by the proportional gain. The integral term compensates for over- and under-steering, and the derivative term removes acceleration errors [29].

Melder *et al.* [30] suggest different amendments for a vehicle PID controller. One of these amendments is to use *split channels*, i.e. using two seperate controllers for brake and throttle, depending on whether the error is positve or negative. The two separate PID controllers can thus react dynamically to the different responses of throttle and brake.

Another suggestion made by Melder *et al.* [30] is to use different controllers, which are appropriate for different types of situations, and implement a switch to select the most appropriate controller for a given situation. Thus, for example, a different steering control might be appropriate for different surfaces, or one might wish to alter the braking behaviour in the case of emergency braking.

Several different methods for tuning a PID controller have been used in the related literature. In [30], a systematical trial-and-error method is detailed. In [29], three different optimisation methods, namely a genetic algorithm, a "simulated annealing" approach and the Nelder-Mead Simplex algorithm, were compared. The analysis highlights the Nelder-Mead optimisation as giving the best performance of the resulting controller, when tested for speed control.

In [33], a multiobjective optimisation approach for a PI controller is explained, which gives a Pareto front of optimal controller gains. This allows to exploit the different trade-offs that are inherent to the controller gain choices. It must be noted that all algorithms for tuning PID controllers are based on a simulation of the vehicle dynamics, and run $50 - 100$ times in order to provide convergence.

For practical reasons, a trial-and-error method for tuning the PID controller was chosen. This choice is further explained and justified in section 6.2.

Advanced Steering Control

Methods of improving the steering behaviour of an autonomously driven vehicle are typically known as path following algorithms. Path following algorithms are used to control the direction of the vehicle's motion in an intelligent way by predicting the trajectory of the vehicle over a given amount of time.

In [31], Sujit *et al.* provide a comprehensive overview of commonly used path following approaches and compare them with respect to performance and control effort. The aim of the approaches is to determine an appropriate heading angle of the vehicle by generating a sub-goal for the vehicle on its trajectory. The target heading angle can then be passed to the PID controller as part of the error term, so that the difference between the orientation of the vehicle and the target heading angle is minimised.

Based on the analysis carried out in [31], three algorithms can be used for reliable steering controllers: *Non-linear guidance*, *pure pursuit* and *carrot following*.

For the non-linear guidance law, the sub-goal that is generated is the intersection of the trajectory to be followed with a circle of a given radius. The inverse tangent function used to obtain the sub-goal introduces a non-linearity to the algorithm. Both pure pursuit and carrot following approaches are based on calculating the sub-goal as a point on the track, which is a specified *lookahead*-distance away from the vehicle. This logic is visualised in figure 4.4.

In the pure pursuit algorithm, a circle is fitted such that both the current position and the sub-goal lie on the circumference of the circle. The steering angle is chosen such that the vehicle follows the curvature of the circle. The carrot following algorithm follows a more direct approach, which calculates the steering angle directly as the angle required to reach the sub-goal [31, 32].

The pure pursuit and carrot following approaches were also used in [32] to implement a controller for an autonomous land vehicle. The testing showed that the pure pursuit approach is good when errors are small, but tends to oscillate far off the desired track. The carrot following approach, on the other hand, is very robust to disturbances but may lead to oscillation at high speeds.

Comparing the three approaches considered here, the *carrot following* approach can be classified as the algorithm which is most robust to disturbances, and requires little control effort for delivering a good performance. The downside of the carrot following approach is that, if not tuned correctly, it can lead to cutting of corners and oscillations at high speeds.

This approach was also recommended by Tomlinson *et al* [34] for steering controls in artificial intelligence applications for racing games. Here the look-ahead is called a "runner" and is suggested to be implemented as a separate AI object that is updated at every time step.

Model Predictive Control

Model predictive control (MPC) is a control strategy which predicts the inputs to a system for a given prediction horizon, which are required to reach a specified goal. MPC is based on a model of the system and updates the optimal system inputs at every time step based on the current error.

In the case of autonomous driving, the required model contains the underlying physical model that relates the applied throttle and brake pressure, as well as the steering angle, to the resulting motion of the vehicle. It is assumed that the trajectory which the vehicle is meant to follow is known. This model is usually simplified to decrease the computational cost of the prediction [27, 28].

This approach is particularly used for real vehicle applications, as for these an accurate prediction is crucial for reasons of safety. Furthermore, MPC allows to track complex trajectories accurately to achieve high-level targets such as lane keeping or overtaking [27].

Despite the theoretical advantages of MPC, they are rarely used for virtual driving simulations and racing games. This is due to the computational cost related to re-computing the targets at every time step, as well as due to the complexity of the physical models that would need to be implemented. This goes beyond the scope of this project, and thus a PID controller was chosen to be most appropriate for the implementation.

2.3 Performance Evaluation

Determining "human-likeness" of an autonomous vehicle is not trivial, and can be defined in multiple ways. In this section, the methods for quantifying similarity to human driving in the considered literature are compared in terms of their ability to quantify an individual human driving style.

The methods of evaluation can be roughly divided into three levels of abstraction: Evaluation of the driver's *objectives*, evaluation of the driven *trajectory* and evaluation of the low-level *driving commands*.

High-level driver objectives are mainly used for rule-based learning methods, as these methods are specifically designed to learn the reward for completing some high-level task. Examples of such metrics are the lane changing behaviour, overtaking behaviour, collision rate or following distance to other vehicles [2, 21, 23]. For the scope of this project, these metrics are not appropriate, as no high-level objectives are evaluated within the simulation.

An evaluation of the trajectory of the vehicle has been considered in most of the related literature that implement an autonomous personalised controller in a virtual racing environment [9–12]. All implementations are compared based on the total distance covered in a given number of time steps or game ticks (typically $1000 - 3000$ steps). While the total distance covered in a given time is a good metric to distinguish between good and bad drivers, it is not sufficient in order to describe the driving style in an informative way, as highlighted in [11].

For the models that are based on multi-objective evolution, a further analysis of the performance of the model training is used for evaluation. For example, the controller explained in [12] is evaluated based on the convergence of the weights of the objective function, namely the speed and lateral deviation from the track center. While these may be a good indicator for the theoretical model performance, this method does not provide proof that the controller based on these models actually exhibits human-like driving behaviour.

Finally, in some approaches to imitation of human drivers, the commands for throttle, brake and steering are compared to the user's commands [10]. The mean-squared error of the model training is then used for evaluation. As outlined by van Hoorn *et al.*, the steering, throttle and brake commands exhibit a poor imitation of human driving styles.

With the purpose of modelling drivers for personalised track evolution, Georgiou *et al.* [35] developed a performance score which is formed of a weighted sum of the mean of a number of performance metrics. This measure is used to estimate a driver's skill level. The metrics considered for this analysis are a combination of simulator outputs, user inputs and external sensors including eye tracking and head pose estimation [35]. For this project, a comparison consisting of multiple metrics is also desirable, however it may be more informative to investigate individual metrics separately rather than forming one overall score.

3 Requirements Capture

In this chapter, the objectives and deliverables of the project are detailed. This includes the specifications for the controller software and the requirements that must be fulfilled by a method which evaluates the performance of the controller. Additionally, the requirements for the simulation, which has been developed as part of the project, are specified.

3.1 Project Specification

The aim of the project is to develop a vehicle controller that is capable of learning from and imitating human driving behaviour. Such controllers are of interest for virtual car racing games, where they are being used to generate more challenging game content [11]. They can further be applied to improve the comfort of users of autonomous vehicles such as cars or possibly autonomous wheelchairs, and increase the adoption rate of these vehicles.

The deliverables of the project include the design and implementation of a reliable and personalised controller software (see chapter 4 and 5), a simulation test bench to demonstrate the behaviour of the controller (specified in section 3.2) and a thorough theoretical and practical analysis of the benefits and shortcomings of different implementation approaches that have been tested in the project (see chapter 6 & 7).

3.1.1 Controller development

As mentioned above, the primary deliverable of the project is a personalised controller algorithm design. The controller must be capable of inferring low-level control signals (i.e. throttle, brake and steering commands) from the current state of the vehicle (such as position and speed) and the information about the track it follows. For the scope of this project, the personalised controller performance is defined by the following objectives:

- Follow a similar trajectory and speed profile as the user on the training track
- Exhibit clearly personalised behaviour in an unseen environment
- Be robust to disturbances and reliably complete the track

In addition to the software design, a thorough analysis of the individual system components and the different implementations that have been considered is necessary in order to assess the overall performance and ability to imitate human driving. This analysis is provided in chapter 6.

With future development in mind, the design was further aimed to be adaptive, such that new learning methods or additional constraints can be easily implemented. Such constraints could include safety considerations in the form of adding a secondary objective of keeping the vehicle on the track, avoiding collisions with obstacles or adhere to a speed limitation.

3.1.2 Quantification of imitation performance

Measuring and comparing human driving style is not a trivial task and can be defined in a multitude of ways, some examples of which were detailed in section 2.3. However, much of the literature focuses on assessing the fitness of a learning algorithm rather than comparing the results of the objective function for different users or different tracks. In cases where the methods were tested on different tracks and for different users, the methods of comparison were found to not express the driving style of a particular human well.

As mentioned in 2.3, the imitation of low-level driving commands carries little meaning about how well a vehicle imitates the driving style of a human. In a situation of autonomous driving, the conductor of the vehicle does not know what these low-level commands would be, nor is it of interest. Instead, driving comfort is much more effectively described by high-level metrics. Within the scope of this project and with a one-vehicle racing track in mind, the following metrics were thus defined to convey most information about the user's driving style:

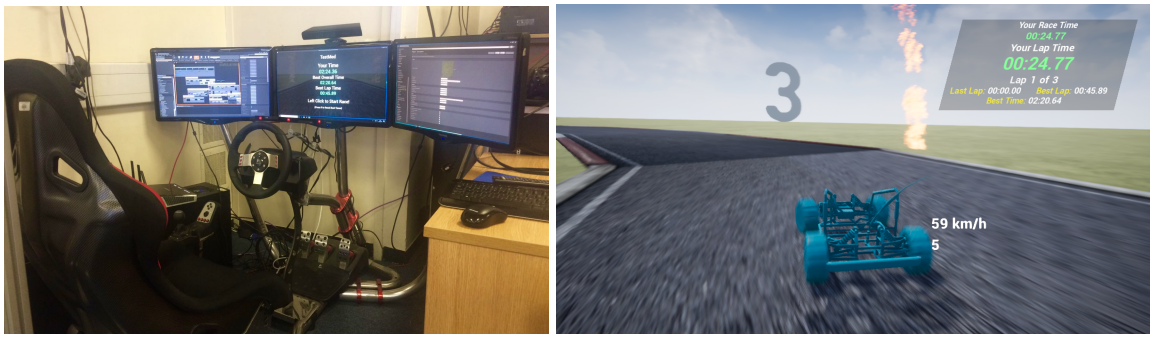
- Speed profile
- Deviation from the track center
- Total progress on the track

While the first three metrics indicate how close the vehicle's behaviour is to a given example, namely the user's driving logs, the total progress is chosen as a measure of robustness and validation of the first three metrics. This means that it is expected that a robust controller, if it imitates the human driving well, should complete the laps in a similar time as the user. This is fundamentally different to the methods considered in the literature, where the total progress is frequently considered the most important performance measure.

3.2 Testing environment

In order to test and validate the controller implementations, a driving simulator consisting of a physical racing setup and a virtual racing game was used. The physical setup of the simulation consists of the Vision Racer Motion Driving Simulator with a Logitech G27 force feedback steering wheel connected to a setup with 3 screens, as shown in figure 3.1a. Furthermore, eye tracking sensors have been installed and provide an extra source of data that may be used.

Two virtual simulators were considered for this project: rFactor 2 and the Unreal game engine. rFactor 2 is a simulation which is specifically designed for car racing games, while Unreal Engine is a game development engine that can be flexibly designed for a multitude of applications. The simulation is required to transmit all necessary information from the vehicle and the driven track to the controller and allow the controller to set steering, throttle and brake commands.



(a) Physical setup *(b) Unreal engine game screenshot*

Fig. 3.1 Left: The physical driving simulator including Vision Racer hardware and driving screens. Right: Screenshot of the game at its current state including vehicle, track and race timers.

3.2.1 rFactor 2 Simulation

The rFactor 2 Simulation, which has been used for previous projects in the Personal Robotics Lab [26, 35], was used in the initial phase of the project for the controller design analysis outlined in section 4.2.1. The existing simulation contained over 100 different output variables. Only a small number of these were relevant for the analysis, namely the throttle, steering and brake as well as vehicle state information including speed, position and acceleration of the vehicle. A full overview of all used variables can be found in Appendix B.

While the rFactor simulation offers the advantage of being designed specifically for car driving tasks and has a large amount of information available, its capacities to interfere with the vehicle control are limited. Hence, in order to allow more freedom to design the simulation in any desired direction, Unreal Engine was chosen as a platform to implement a new racing game.

3.2.2 Unreal Engine Simulation

The simulation based on the Unreal Engine 4 game engine was started in January 2017 and has been developed in cooperation with other researchers in the Personal Robotics Lab. A summary of the contributions that were made as part of this project are given in section 5.1.1. As the simulation is in constant development, some of the implementations and structures explained in this section might change in the future. Figure 3.1b shows a screenshot of the racing game at its current state.

The simulation consists of a "Time Attacker"-style racing game and uses ROS (Robot Operating System) to send and receive instructions and data. It was started from a video tutorial series and was enhanced in order to fulfill the additional requirements for the ROS communication in order to provide and receive all required data. The data can be grouped into simulator outputs, driving inputs and track information, which are key to enabling autonomous vehicle control.

Simulator Outputs

Table 3.1 details the simulator outputs, which can be roughly divided into 3 categories: The previous driving commands (which are set by the controller in the case of autonomous driving), the vehicle state used to determine the appropriate driving commands and information on the vehicle environment. This information is not relevant for the controller but is used for evaluation purposes.

Table 3.1 Relevant Unreal simulation outputs. They contain low-level driving commands, vehicle state information to compute new driving commands and environment variables used for evaluation.

Driving Commands	Vehicle State	Environment
Steering	Global position (x,y,z-direction)	Active Segment
Throttle	Global rotation (roll, pitch, yaw)	Time
Brake	Global velocity (x,y,z; in m/s)	Game counter
Gear	Total speed (in km/h)	
	Rotations per minute (rpm)	

Driving Inputs

The inputs necessary for autonomous driving which are being sent by the controller can be classified into axis control signals, action control signals and enabling signals. Enabling signals allow the controller to set the driving commands. While autonomous driving replaces overall human driving by driving through the controller, the automatic gear change indicates that the controller overrules the otherwise automatic gear setting. Axis controls are variables that vary continuously and are re-set at every game tick. Action control signals, on the other hand, trigger events which then set the corresponding game inputs, so they only act "on demand".

Table 3.2 Relevant driving inputs, consisting of continuously queried axis control signals, event-triggering action control signals and enabling signals which allow the controller to set the driving inputs.

Axis Control	Action Control	Enabling Signals
Steering ($\in [-1,1]$)	Gear (-1 for reverse, max. 5)	Automatic Gear Change (boolean)
Throttle ($\in [0,1]$)	Handbrake (boolean)	Autonomous Driving (boolean)
Brake ($\in [0,1]$)		

Track Data

Finally, data on the center of the race track is required in order to relate the global position and orientation of the vehicle to a position and orientation relative to the race track and thus make the driving behaviour generalisable to new tracks.

The track is constructed using a spline array to represent the center of the track, which is illustrated in figure 3.2. Each spline point has two properties: A position and a tangent, which

are both defined as vectors. The position is given by its x, y and z coordinates, while the tangent is a 3-dimensional vector which has the spline point position as its center. This representation can be used to reconstruct the spline mathematically, using cubic Bezier curves.

For all implementations developed within this project, a densely interpolated representation of the spline is required. This was initially implemented using a 3D Bezier interpolation developed previously in the lab. However, Unreal Engine offers a functionality to perform this computation within the virtual simulation. The output from the simulation showed to deliver a more accurate representation of the actual track center and was hence used for the track representations. Details on the implementation of methods to prepare, send and receive the track are explained in section 5.1.

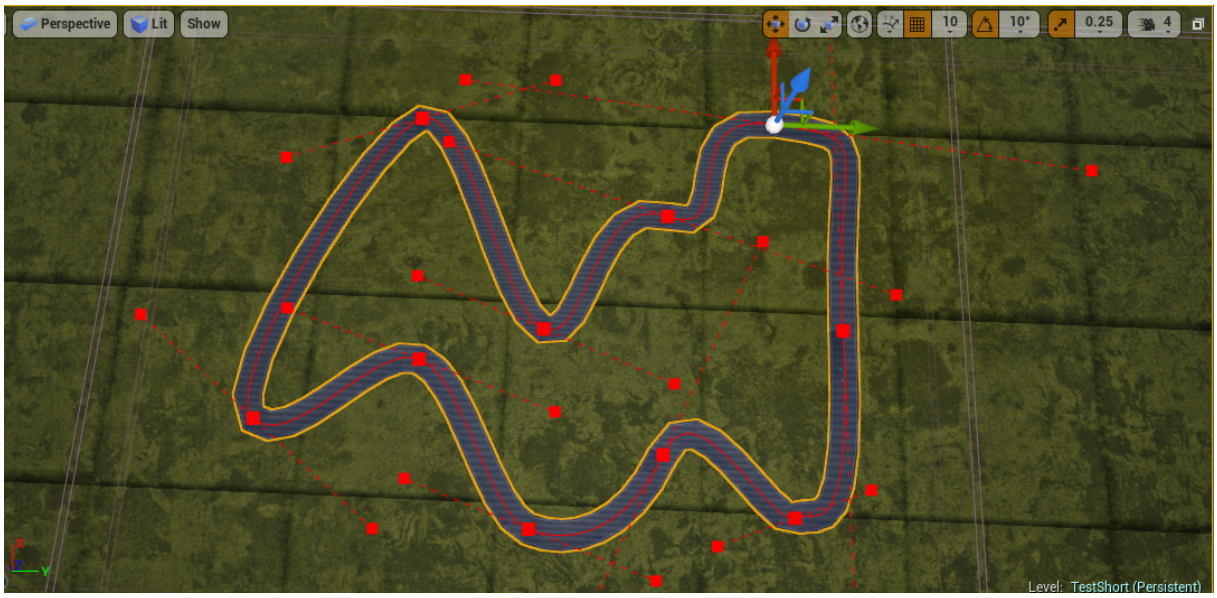


Fig. 3.2 Illustration of the spline used for track representation. It is defined by a set of points and their tangents. The length and direction of the tangents gives all necessary information for reconstruction of the track.

4 Analysis and Design

The following chapter gives an overview of the controller system and explains the design choices which were made during the development process. Specifically, the two-stage design approach and the different designs for high-level and low-level controllers are outlined. An analysis of a sample data set was used to give an understanding of the intuition behind the controller and to justify the design choices.

4.1 Design Overview

Based on the findings from the literature review presented in section 2.2, a two-stage design approach was chosen as the most promising structure to provide a both robust and personalised controller. It is based on a direct method used to determine the target trajectory of the vehicle through a machine-learning model. From this high-level information, a controller computes the low-level driving commands as suggested by Cardamone *et al.* [9].

Throughout the rest of this report, the following terminology is used: The target trajectory describes the forecasted speed of the vehicle and its distance from the track center. This information is expressive as it translates the global position of the car to a relative position, which can be used to compare data from different human-driven and autonomous data sets. The low-level driving commands are defined as throttle, brake and steering and, if specified, the gear. The values for these variables lie within the range specified in table 3.2.

Figure 4.1 visualises the approach: The vehicle outputs specified in table 3.1, in particular the speed, position and rotation of the vehicle, are fed to the high-level model together with the information about the track. The model, which has been trained previously on human driving data, is then used to compute the target speed and the target distance. This information is used by the low-level controller to compute the vehicle control commands.

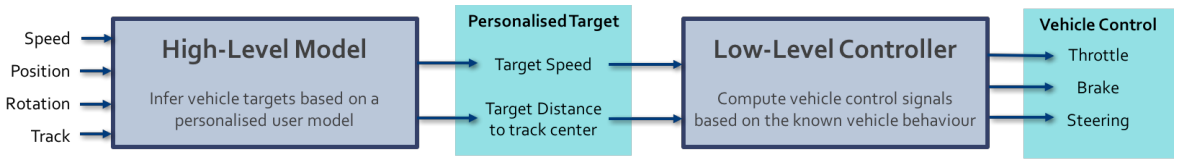


Fig. 4.1 High-level controller design using a two-stage approach which combines a personalised driver model with a low-level controller for driving commands.

This approach leaves much freedom to design the two individual components, which can now be analysed and designed (mostly) independently. The high-level model should be designed to accurately produce targets that mimic the human behaviour, whereas the low-level controller should accurately follow a given trajectory at a given speed. For both designs, the various methods detailed in sections 2.2.1 and 2.2.2 were analysed in order to make the design choices.

The low-level controller was designed and implemented first, for the practical reason that the low-level controller is needed to make the vehicle drive and is hence a prerequisite for verifying the performance of the high-level model experimentally.

4.2 Low-Level Driving Controller

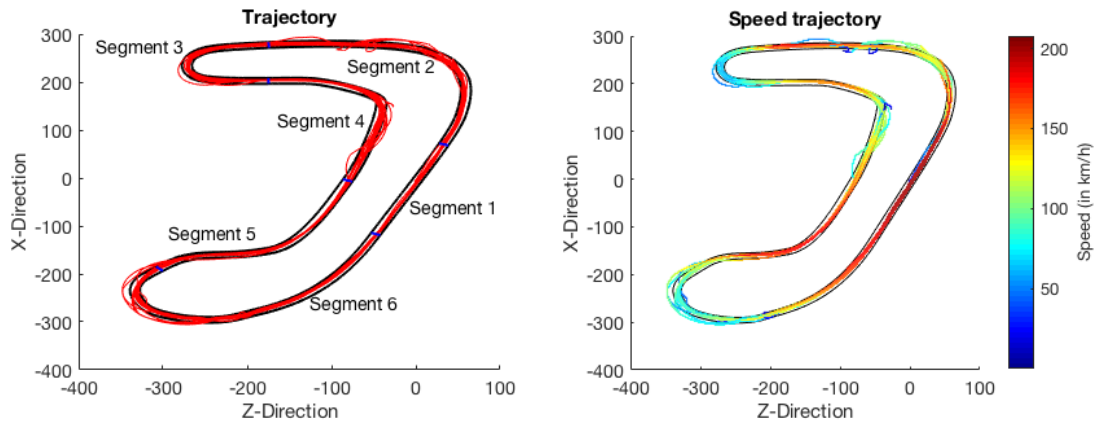
Based on the analysis performed in 2.2.2, two feasible designs of a low-level controller that calculates the commands for throttle, brake and steering from a target trajectory were selected. These are a simple linear controller and a PID controller.

The testing of the linear controller was carried out as the first stage of the controller design and was based on data from the rFactor 2 simulation. Based on this analysis, the viability of the low-level controller design proposed in [9] was tested. This analysis showed the poor performance of the controller, and hence a PID design was chosen for the final design. The PID controller has two elements, one of which is a speed controller that controls throttle and brake, and the other one is the steering controller using an advanced path following design.

4.2.1 Linear Controller

For the analysis of a linear controller, a data set from the rFactor 2 simulation was tested. This data set contains 19 laps of a single user along the track shown in figure 4.2. The relevant outputs for the analysis performed here is shown in Appendix B. The data was split into the segments as shown in figure 4.2a in order to allow a situation-based assessment of the data.

Figure 4.2b shows the speed profile of this driver: On long, straight parts of the track the speed is very high, while it is reduced significantly in sharp turns. One can also observe that the driver crashed multiple times along the track, indicated by the low-speed (dark blue) parts.



(a) Track segmentation *(b) Driving speed*

Fig. 4.2 Test data used for analysis of the linear controller, showing the segmentation of the track (left) and the speed of the driver along the track (right).

The outcomes of the analysis of this data set was used to assess how well the proposed controller design represents the actual commands for throttle and speed, and to highlight the need for a more advanced controller such as the PID controller explained in section 4.2.2.

Speed Control

In the linear controller approach, it is assumed that the necessary throttle and brake commands can be entirely deduced from the current speed and the target speed of the car. Figure 4.3 shows the throttle and braking data obtained of the sample driver for one full lap without crashes (red and blue data points), together with the linear controller implementation suggested in chapter 2.2.2 (indicated by $throttleController$ and $brakeController$ legend entries). While the braking controller suggested in equation 2.4 nearly fits the observed trend, the throttle control law is very different from the suggestion made in [9].

The testing data suggests that there must be more inputs than just the change of speed (i.e. the acceleration) that determine the amount of throttle/brake which is applied, as the data points show a large deviation. However, one can identify a somewhat linear relationship between acceleration and brake/throttle, which is used as basis for the new model. It is particularly noticeable that often the throttle is pressed even though the acceleration is actually negative, which may be an effect of engine braking.

Based on these observations, a suggestion for a linear controller suitable to the rFactor 2 game was made by amending equations 2.3 & 2.4 to equations 4.1 & 4.2. These amended equations are shown in figure 4.3 by the yellow and purple lines ($trottleLin$ and $brakeLin$).

$$throttle_{t+\Delta t} = 0.3 + 0.09 * (speed_{t+\Delta t} - speed_t), \quad throttle_{t+\Delta t} \in [0;1] \tag{4.1}$$

$$brake_{t+\Delta t} = -0.06 * (speed_{t+\Delta t} - speed_t), \quad brake_{t+\Delta t} \in [0;1] \tag{4.2}$$

Figure 4.3 shows purely experimental results. It is likely that equations 4.1 & 4.2 are very different for different racing games or even different cars or surfaces. More than anything, this result shows that it is difficult to find a general expression for a linear controller, which makes this approach infeasible in a wider context.

Steering Control

For the steering control, a "carrot following" approach explained in [32] was used. This was chosen over other path following methods given in the literature because of its robust and overshoot-free characteristic and the good trade-off between performance and control effort (see section 2.2.2 for details).

In this approach, the target distance from the track center is converted into a target orientation of the vehicle. The difference between the vehicle's orientation and the target orientation is hence

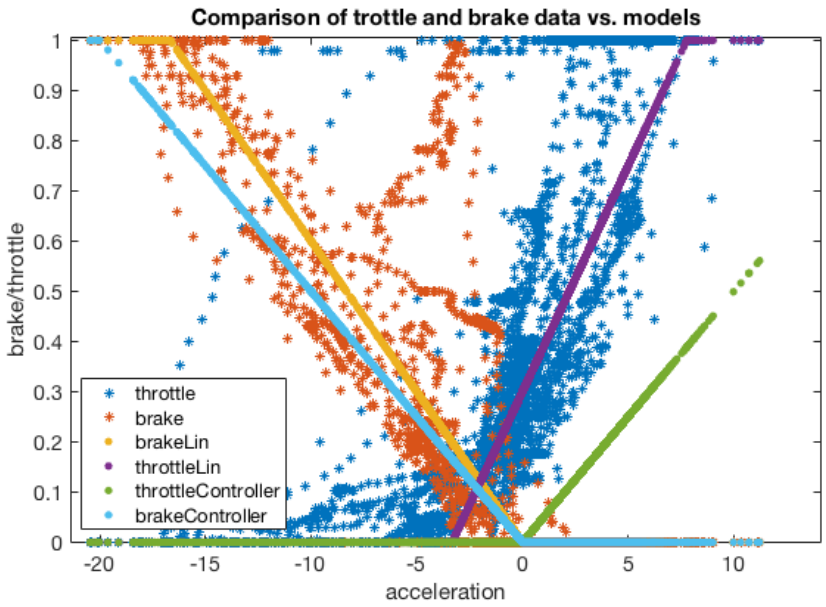


Fig. 4.3 Visualisation of the speed control, showing throttle and brake commands of the test data (throttle & brake), the linear model suggested in the literature (-Controller, see eq. 2.3 & 2.4) and a proposed better fit (-Lin, eq. 4.1 & 4.2).

the error to be minimised by the controller. The target position represents the *carrot*, which is chased by the vehicle. This means that the target position is always a certain distance away from the vehicle (as the carrot is re-computed at every time step), leading to smooth steering.

Figure 4.4 visualises the carrot following approach. At every time step, the target position of the vehicle (indicated by P) is determined from the target distance (red) and the "look ahead"-distance (green). P is thus found by moving a "look ahead"-distance along the track and moving a target distance perpendicular to the track. While the target distance is given by the high-level model output, the "look ahead"-distance is a design parameter. The target orientation of the vehicle is determined by the orientation of the vector spanning from the current position of the vehicle to its target position, and hence α indicates the angle to be minimised by steering.

In order to obtain a steering command in the range of $[-1,1]$, the target steering angle α must be truncated to this range. A steering value of ± 1 corresponds to a maximum steering angle of the vehicle of $\pm 45°$, hence all angles are divided by $45°$ and all angles $|\alpha| > 45°$ are cut-off, giving a steering command calculated as follows:

$$steering_{t+\Delta t} = \frac{(orientation_{t+\Delta t} - orientation_t)}{45°}, \quad steering_{t+\Delta t} \in [-1;1] \tag{4.3}$$

Again, the appropriateness of this approach was analysed using test data from the rFactor 2 simulation. The steering controller code for this analysis is shown in Appendix B. For the scope of this analysis, the target position is set to the position of the vehicle at time $t + \Delta t$ in the future, i.e. a fixed number of time-steps away. The target orientation is then given by the direction of the vector from the current location to the target location.

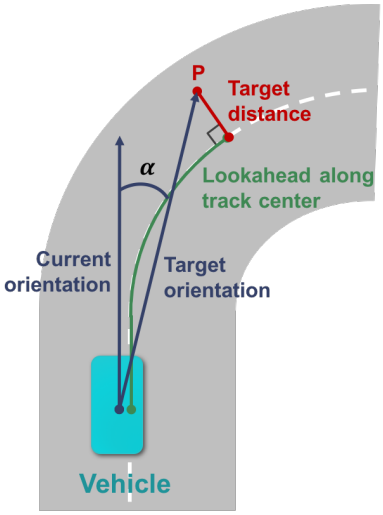
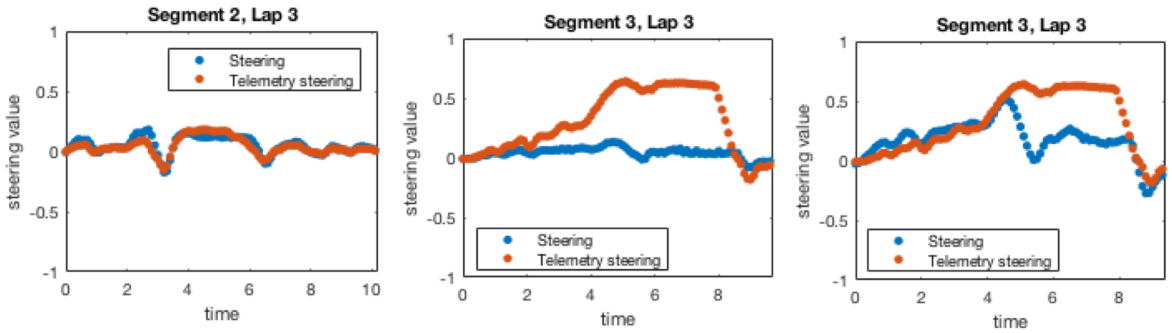


Fig. 4.4 Diagram of the "carrot following" approach. The target orientation is calculated as the vector pointing from the vehicle to the target position, which is found by moving a "look ahead"-distance along the track and a target distance perpendicular to the track.

Figure 4.5 shows the performance and effect of the "look ahead" approach on different segments of the track. The graphs in 4.5a and 4.5b are based on a look ahead of 1, i.e. based on a prediction of the immediately next vehicle orientation. While this shows good results for easier track segments such as segment 2, it leads to large deviations for more difficult segments such as segment 3. The steering angle is constantly low as the orientation of the car only changes marginally from one time step (0.1s) to the next time step.

Looking further ahead into the future can therefore allow a better estimation of the steering behavior, as shown in figure 4.5c. This graph uses a look ahead horizon of 3 time-steps. The error in figure 4.5c is clearly reduced compared to figure 4.5b, but it still shows a considerable error. There are two possible explanations for this: Either, an even larger lookahead is required for steep curves, or it is not necessary to steer as strongly in this situation as the driver did.



(a) Good steering match *(b) Bad steering match* *(c) Lookahead improvement*

Fig. 4.5 Illustration of the effect of increasing the "look ahead" for a smooth track segment (a) and a segment with a steep curve (b) and (c). In the smooth segment, a small look-ahead is sufficient to reliably imitate the driving behaviour, while for a steep segment, a larger "look-ahead" (here 3 time steps) can achieve a significantly better matching of the steering commands. The data was obtained from the rFactor 2 game engine.

Figure 4.6 shows the fitness of the equations 4.1, 4.2 and 4.3 above to recreate the vehicle's low-level driving commands for different prediction horizons or "look aheads" (in multiples of 0.1s). For throttle and brake commands, a look ahead of 0.1s results in the smallest error. The order of magnitude of the error however is much smaller for the brake. Error here is defined as the difference between the calculated and measured throttle and brake values.

For the steering command, the difference is not so clear. While a small look ahead yields low errors for flat track segments (e.g. segments 1 and 5), it gives a larger error for the tight curve in segment 3. The best results can be achieved in this case by a look ahead of 0.2s. The analysis also shows that large look-aheads are not desirable either, as the error increases significantly for large lookaheads. This may be explained through effects of oversteering.

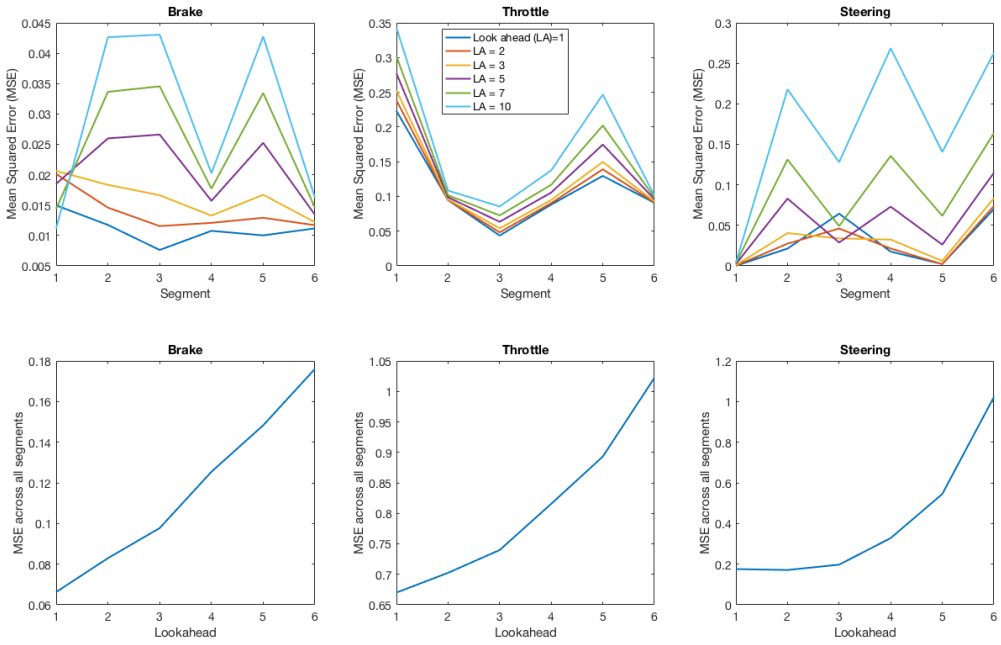


Fig. 4.6 Comparison of the mean squared error of the brake (left), throttle (center) and steering (left) commands for different "look-ahead" values. While a prediction horizon of 1 best represents throttle and brake, a larger "look-ahead" can be appropriate for steering.

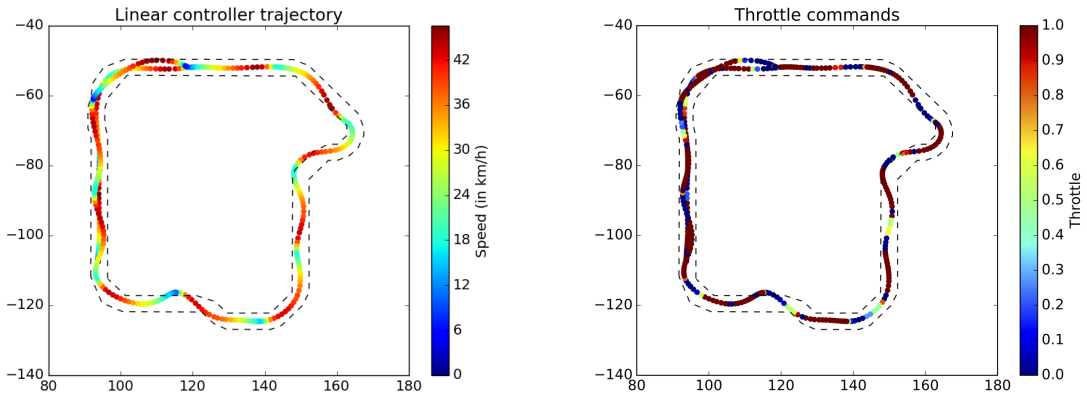
A controller based on the linear equations in 4.1, 4.2 and 4.3 was the first controller to be tested online in the new Unreal Engine simulation. An analysis similar to the one shown in 4.3 was carried out for the new simulation, giving again a different optimal throttle control law, as shown in equation 4.4. The brake control was unchanged from equation 4.1.

$$throttle_{t+\Delta t} = 0.7 + 0.15 * (speed_{t+\Delta t} - speed_t), \quad throttle_{t+\Delta t} \in [0;1] \tag{4.4}$$

Figure 4.7b shows the performance of the controller using the linear equations for throttle, brake and steering specified in 4.4, 4.2 and 4.3, respectively. The vehicle was set to follow the center of the track, which is a default Unreal Engine track, at a constant speed of $30 km/h$. This track was used throughout the implementation and validation stages of the project.

The trajectory is represented in terms of speed and position (fig. 4.7a). The corresponding throttle commands of the vehicle is shown in figure 4.7b. The trajectory clearly shows over-steering as well as oscillation in the speed domain. The throttle commands vary between the extremes of 0 and 1 and fail to find the appropriate equilibrium for the required speed, which leads to the observed oscillation.

This demonstrates the limitation of the linear speed controller: While it is capable of reproducing the acceleration behaviour of the vehicle, it is incapable of stabilising the throttle at a given speed. The steering shows a similar behaviour. While the "carrot following" implementation leads to some smoothing of the path, the steering commands are too large which leads to oversteering.



(a) Autonomous vehicle trajectory *(b) Controller throttle commands*

Fig. 4.7 Trajectory and throttle commands of a vehicle with linear controller on the Unreal Engine test track. The trajectory is shown by x-y-position of the vehicle and the speed (with red as high speed in a). (b) shows the corresponding throttle commands. The trajectory is not smooth as expected due to over-steering and too extreme use of the throttle.

The analysis performed above gives two key insights into important concepts that must be considered for the design of a PID controller. Firstly, the behaviour of throttle and brake vary largely between different environments, which may make it necessary to re-tune the PID implementation when deployed in a new environment or with a new car.

Secondly, it is necessary to consider some look-ahead for the steering control in order to react to upcoming curves appropriately. The amount of lookahead that is appropriate depends on the characteristics of the track.

4.2.2 PID Controller

The above analysis showed that the linear controller does not perform the given control task well, despite being capable of reproducing the driving commands. Hence, the design must be improved in order to implement a robust and responsive controller. Of the different controller approaches analysed in section 2.2.2, the PID controller is best suitable for this task. It does not require knowledge of the underlying physics model as an MPC approach would, but can be

designed to achieve a stable design, which is widely used for similar applications of controlling virtual vehicles [30].

Effectively, the linear controllers for speed and steering are proportional controllers, with the error to be minimised defined as follows:

$$e_{throttle/brake}(t) = speed(t+\Delta t) - speed(t) \tag{4.5}$$

$$e_{steering}(t) = orientation(t+\Delta t) - orientation(t) \tag{4.6}$$

Using this definition of the error, it is possible to construct a PID controller as defined in equation 2.6. Thus, instead of computing the steering and throttle commands solely based on the current error, an integral and a derivative term are added, which minimise the steady-state error and damp oscillations. Instead of just a single design parameter for each controller, a PID controller requires 3 controller gains to be determined, namely the proportional gain (K_p), the integral gain (K_i) and the derivative gain (K_d).

Determining these parameters appropriately is key in order to design a robust controller. For the speed controller, it is particularly important to have a quickly responding controller, and some oscillation may be acceptable. For a steering controller however, overshoot should be avoided, as humans are very susceptible to oscillations in steering. These considerations must be taken into account when tuning the controller.

The tuning of the PID controller is discussed in section 6.2. For this project, a trial-and-error approach to finding the controller gains was chosen, as other automatic tuning methods were not feasible within the project scope. With appropriate parameter choices, it is theoretically possible to obtain a controller that follows any trajectory, at any given speed.

In practice, the controller follows a given trajectory with some (small) error, given by the lag of the controller (determined by its time constant), the limitation in maximum throttle, brake and steering commands and the dynamics of high-speed driving such as drifting and strong centrifugal forces in steep curves. These limitations are discussed in more details in section 6.2.3. The controller design forms the basis for testing of human driver models.

The choice of the controller gains is a trade-off between the rise time of the controller (i.e. the lag), the maximum overshoot, and the controller's disturbance rejection properties. Reasonable targets for these characteristics were found experimentally.

The design choices were defined to not exceed a rise time of $0.5s$, and to limit the overshoot to $< 10\%$ Avoiding overshoot is more crucial for the steering than it is for the throttle control. For more details on the PID parameter choices, see section 6.2.

Steering Backwards

The previous analysis focused exclusively on the forward motion of the vehicle. However, in cases where the vehicle crashes or leaves the track, it might be necessary to drive the car

backwards. Thus, the controller should also be able to handle backwards driving. This might seem very straight forward, however the fact that the vehicle's orientation is no longer aligned with the direction of motion requires some extra steps in the controller design.

Firstly, the acceleration behaviour is inverted when moving backwards. Decreasing the speed now means that the car accelerates, which requires the throttle to be pressed. Inversely, braking is equivalent to a positive change in the error. Thus, the sign of the speed error must be inverted in order to achieve backwards motion.

Secondly, some adjustments to the steering are required. Figure 4.8 illustrates these amendments. Based on the "carrot following" approach, the target orientation will be calculated as the vector from the vehicle to the target position highlighted in red, as the target position lies behind the vehicle (in the direction of motion, shown in green). However, if this angle is given to the controller, it will aim to turn the vehicle so that it faces the target position, which is the opposite of what is desired (and results in the vehicle spinning wildly).

Instead, the back of the vehicle should face the target position, so that the vehicle can reach this position in the direction of motion. Hence, for backwards driving, the target orientation must be shifted by 180°, giving the angle α as the angle to be minimised, just as in figure 4.4.

Finally, one must consider which steering commands are required in order to miminise the angle α. When moving backwards, the steering commands are in fact reversed: Turning the vehicle facing right, one must steer left (as in figure 4.8 and vice versa). Or, put differently, one must steer to the opposite direction (here left of the track center) as the desired orientation (right of the track center).

Hence, three actions are required for driving backwards:

1. Invert the sign of the speed error
2. Shift the target orientation by 180° to face forward
3. Negate the resulting steering command to move in the correct direction

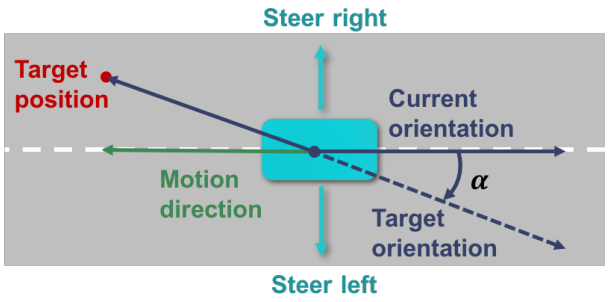


Fig. 4.8 Diagram of the geometry of driving backwards. For a vehicle with the current orientation shown, the target orientation is now opposite the desired direction of motion and the steering commands must be inverted.

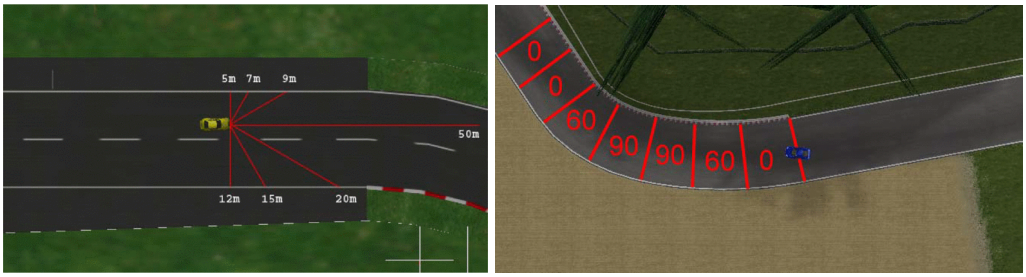
4.3 High-Level Behavioural Model

The high-level behavioural model must be capable of capturing the user's driving style and deduce typical behavioural patterns from the test data. This has two key aspects: Choosing a suitable model and, possibly even more importantly, selecting appropriate inputs to feed the model.

4.3.1 Model inputs and track representation

For the two-stage controller illustrated in figure 4.1, the outputs of the high-level model are defined as the target speed of the vehicle and its target distance to the track center. These outputs are then the inputs to the low-level PID controllers.

The inputs to the driver model are not as clearly defined, and here it is crucial to consider what impacts the driving behaviour. One necessary piece of information is the track development ahead of the car, and possibly the track development behind the vehicle. Two possible representations of the track have been considered: The frequently used rangefinder sensors used in [10–12] and a representation of the track based on its curvature, as proposed in [9]. The principle of both representations is shown in figure 4.9.



(a) Rangefinder Sensor *(b) Curvature Sensor*

Fig. 4.9 Sensor options for representation of the course of the track. The common rangefinder sensor (left) determines the the relative position of the vehicle on the track by finding the distances to the track boundaries along rays of different angles. The curvature sensor (right) splits the track up into segments and assigns a curvature value to each segment ahead (and possibly behind). [9]

The representation of the track based on its curvatures shown in figure 4.9b was chosen for the following reasons: Firstly, it is possible to compute the information from this "sensor" just based on the track center, which is available from the simulation. Secondly, this representation of the track is closer to the human perception of the track ahead: One is more likely to change the driving based on how steep a curve is that comes up ahead, instead of checking the distances to the track boundaries (which however may be important to assure that the vehicle stays on track). Thirdly, in the comparison conducted in [9] it was found that this implementation achieves better results than a rangefinder sensor.

The track representation is thus designed as follows: The track ahead (and behind) the vehicle is split into N segments, each of length L. Each of these segments is characterised by the mean

curvature, giving every segment a single scalar value that characterises it. The computation of the curvature is explained in section 5.2.1.

In addition to the list of N curvatures as explained above, the outputs of the simulator, i.e. the current speed, current distance to the center of the track, the current orientation and the acceleration can be used as model inputs. This forms a feedback loop, as the variables to be determined, i.e. the trajectory variables, are fed back as inputs to the system.

4.3.2 Model selection

In section 2.2.1, a number of methods that have been used in the literature for similar applications have been identified. Within the scope of this project, three of these methods were tested and compared: k-nearest neighbor regression (KNN), feed-forward neural networks (NN) and recurrent neural networks (RNN) using long short-term memory (LSTM).

These methods were identified as the most promising methods based on the comparison summarised in table 2.2. They were chosen due to their computational efficiency, which makes them suitable for online application, and their robust performance.

K-nearest neighbor regression

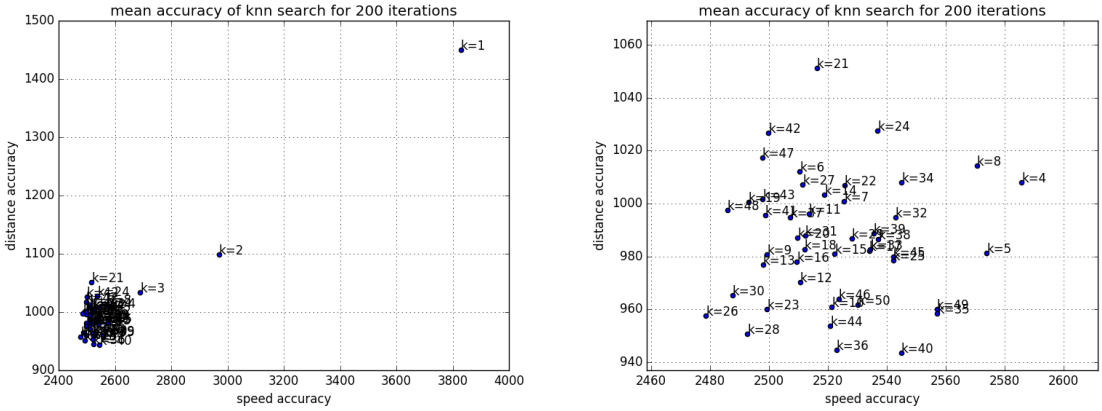
As the k-nearest neighbor model is based on the training data, its design is straight-forward. The design choices for this approach are thus limited to the number of neighbors which are considered for the regression, and the weighting function that is used to compute the target value.

The number of neighbors considered in the analysis is given by k, and it is the key parameter of any k-nearest neighbor search. If $k = 1$, the output of the model is equal to the most similar instance of the user data, while a larger k considers more test samples.

The weighting function used in this project is a weighting which is proportional to the inverse of the euclidean distance between the current features and the sample features. This is shown in equation 4.7. The training features are denoted by the vector $\mathbf{x_i}$, while the labels are denoted by $\mathbf{y_i}$. $\mathbf{x}$ and $\mathbf{y}$ are the features and outputs of the current data point, respectively. The benefits of using such a weighting function is that it penalises distant samples, and thus performs better in well-known situations. This makes it less sensitive to changes in k.

$$\mathbf{y} \propto \sum_{i=1}^{k} \frac{\mathbf{y_i}}{\|\mathbf{x} - \mathbf{x_i}\|} \tag{4.7}$$

Figure 4.10 shows how the choice of k impacts the performance of the algorithm. For the analysis, a test data set of ≈ 4000 human driving samples was chosen and randomly divided into separate training and testing sets. The squared sum of the error, averaged over 100 repetitions of the prediction, is shown in the figure. Values of k from 1 to 50 were chosen.



(a) KNN with $k \in [1, 50]$ *(b) Zoom-in on $k \in [4, 50]$*

Fig. 4.10 Accuracy of the predictions of the k-nearest neighbor model based on a test dataset. While there is a large difference in performance for $k < 5$, the performance for all $k \geq 5$ is similar due to distance weighting. The results were averaged over 200 trials.

From figure 4.10a, it is obvious that there is a large difference between very small k ($k < 5$) and all $k \geq 5$. Figure 4.10b shows a zoomed-in version of figure 4.10a, and shows that there is no real trend observable for varying k. This is due to the effect of the "distance weighting" given in equation 4.7, as little weighting is given to distant features. As a larger k also comes with a larger computational cost, k was chosen to be between $10 - 20$.

In addition to k and the weighting, the specific inputs of the model must be defined, as well as the output values. The model inputs, i.e. the features, were defined as the vector of curvatures as explained in the previous section. Every element of the vector describes the curvature of a segment of a specified length. Segments ahead and behind the vehicle are considered. The model outputs are the target speed and target distance to the track center.

Using a KNN model, it is certain that the output values are always within the range of the training data set, as they are always formed of a mean of recorded output values. This has the benefit that a large deviation of the outputs is avoided, but on the downside it might not adapt well to unseen situations, as it does not "know" any behaviour outside the seen spectrum.

Feed-forward neural network

For the first neural network implementation, a feed-forward network of fully connected layers was chosen, as shown in figure 4.11. In such a network, the value at each neuron is calculated as a weighted sum of all inputs, i.e. all neurons in the previous layer (see eqn. 2.1). The first layer of the net is always equal to the feature vector, and the last layer contains the model output. Feed-forward neural nets are widely used for a great variety of learning tasks and were defined as a promising approach to imitating driving behaviour as analysed in section 2.2.1.

There are a number of design choices that must be made when creating a neural net. These include the structure of the network, i.e. the number of fully-connected layers, and the size of

each layer (which is equivalent to the number of neurons in each layer). The more layers, and the larger each layer, the more complex does the model become. A large and deep network can be used to model very complex behaviours, but it also has a higher risk of over-fitting the data, so that it behaves in unexpected ways in unseen environments.

Additionally, an activation function for the network can be chosen. The activation function is a nonlinear function which multiplies the output layer of the system. Introducing some non-linearity to the system allows to model more complex relationships, and thus activation functions are frequently used. For regression problems, the most common activation function is the *tanh* function, which was used for example for the training of the network in [10]. The *tanh* function suppresses all outputs into the range $[-1,1]$. Hence, the inputs and outputs should be scaled down to unity if an activation function is to be used. The scaling applied to the features in this model is discussed in section 5.3.1.

In order to validate the network training and to detect over-fitting, the total data set is split into two separate sets during the training phase for cross-validation. One set is used for training (ususally the larger set) and one set is used for validation. The measure of fitness of the network weights is given by the loss function, which was defined as the mean squared error between the labels (i.e. the correct output values) and the actual system outputs as shown in equation 4.8.

$$MSE = \frac{1}{n}\sum(label_i - output_i)^2 \tag{4.8}$$

In addition to the training and validation data, which is drawn from the same original data set, a separate testing was also conducted. The testing uses the driving data from a track that was not used for training, and calculates the mean squared error from the model outputs. If the testing error is low, it can be concluded that the model generalises well to other environments.

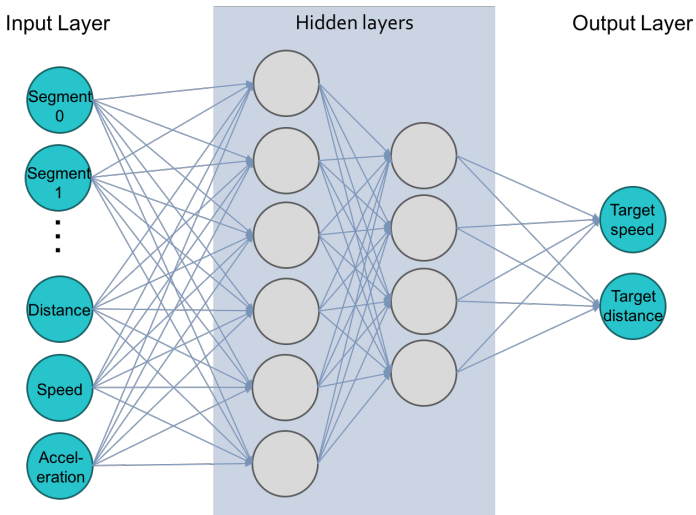


Fig. 4.11 Diagram of the fully connected neural net. The input layer contains all relevant information, which is then passed through a number of hidden layers to compute the two targets (speed and distance).

Recurrent neural network with long short-term memory

Recurrent neural networks are popular in regression problems because of their potential to learn temporal relationships in a sequence of data. Long short-term memory (LSTM) is a widely used design of deep recurrent networks, which are designed in particular to learn long-term dependencies. This may lead to a smoother behaviour of the target calculation and a better generalisation to unseen environments.

Figure 4.12 shows a diagram of the design of an LSTM cell. It has three gates, which together form one cell that has some intrinsic memory. The first gate is the *forget gate layer*, which determines how much of the previous state is considered for computing the new state. The *input gate layer* determines how much of the current input is used for the computation of the new state. The last gate is the *output gate layer*, which filters the new cell state to obtain the desired output [19].

Similar to most recurrent networks, the LSTM network is trained using backpropagation through time, which unrolls the temporal relation over a given number of time steps. The length of the unrolled system is the backpropagation length. It is one key parameter of the model, as it decides how many past inputs must be remembered. While a short backpropagation length does not convey strong temporal relationships, a backpropagation length which is large may lead to vanishing gradient problems [19] and hence a loss of context.

Other design choices of the neural network include the size of each state, and the number of interconnected LSTM cells, which determine the depth of the network. All other parameters are similar to the ones explained for the fully connected net. In order to obtain comparable results, the same loss function was used and the networks were trained using the same cross-validation method.

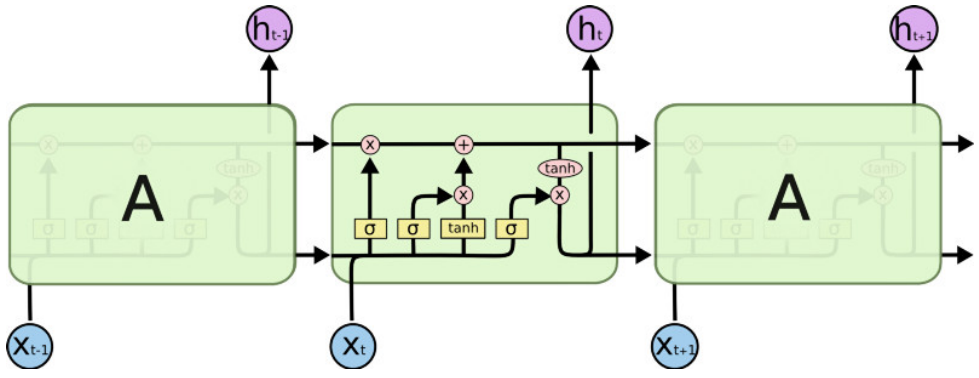


Fig. 4.12 Diagram of a long short-term memory (LSTM) cell. It contains a forget layer, an input layer and an output layer. The layers determine how much of the previous state and the input state is used to compute the next state, and how much of the state is output to the next layer. [19]

5 Implementation

In this chapter, the interaction of the different components of this project and their individual implementations are detailed. This includes the development of the simulation as specified in section 3.2 that was carried out as part of this project, as well as the implementation of the different controller designs detailed in chapter 4. As the simulation was started from scratch in January 2017, a large amount of effort was spent on implementing structures and methods for handling the data which is sent to and received from the simulator in a universal way so that it may be used for future work with the simulator. The structure of this chapter aims to represent the logical order of the implementation process.

Figure 5.1 shows how these individual system components interact to form a system that is capable of controlling a vehicle to drive autonomously in the way a human would. In order to show the system interaction, the two stages of human and autonomous driving were separated, shown in purple and green, respectively.

During human driving, the user drives a set of laps on a set of tracks and gives manual input to throttle and brake pedals and the steering wheel hardware. The Unreal Engine simulation publishes a set of outputs including the human inputs and the vehicle state (see table 3.1 for full list) to ROS at each time step. These outputs are logged and used off-line for training of the driver model in the *Personalisation*-block. During autonomous driving, the controller receives the outputs from the simulator and computes the vehicle's control signals (see table 3.2 for a full list) using the personalised driver model. The performance is evaluated by a quantitative assessment of the user's driving logs.

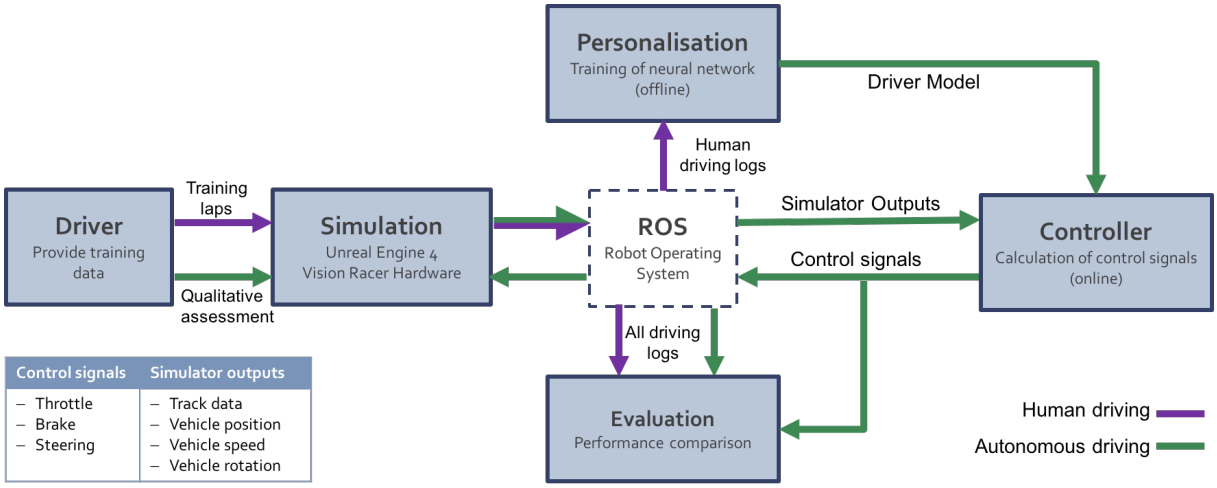


Fig. 5.1 Overview of the system components and their interaction. During human driving (purple), the driving logs of the user are sent from the simulator via ROS, recorded and then used for offline model training. The model trained offline is used by the controller in the testing phase for online target computation. During autonomous driving (green), the vehicle state is sent from the simulator to the controller, which computes the driving commands and sends them back to the simulator via ROS. An evaluation module computes driving statistics from the driving logs of training and testing phases.

Except the simulation, all code elements were written in python. Python was chosen as a programming language as it allows a simple interface with ROS and supports versatile machine learning applications such as tensorflow.

5.1 Simulation and ROS

Following from the simulation requirements specified in section 3.2, several development tasks on the Unreal Engine were carried out. They enabled the creation of a racing game which is entertaining for the user, assures a reliable communication between ROS and the simulator and sends and receives all required data in a useful format. The simulation is based on C++ code and uses blueprints to interface between the code and the game.

This section focuses on the elements of the simulation and its integration with ROS which have been developed within the scope of this project. More details on the complete simulation and how it has been implemented can be found in the User Guide in Appendix A.

5.1.1 Game Development

The practical testing of this project was performed in a racing game set-up. Thus, the development of a game, which is challenging and entertaining for the users, is one key prerequisite. The development of the game was started from the video tutorial series, which can be found in [36].

The tutorial series consists of 15 videos that explain the use of blueprints to set up a game. The resulting game consists of a track with checkpoints that mark the progress in the game, a vehicle that is reset to the current checkpoint if it crashes and flips, and a simulation head-up-display (HUD) that shows the progress of the user on the track.

In order to obtain realistic test data, the simulation was connected to the driving hardware, in particular the throttle and brake pedals, the steering wheel and the gearshift paddles. As the connection of the hardware components differs between steering wheel models, the mapping had to be manually established to assure that the user inputs achieve the desired action of the car. Further, it is possible to adjust the weightings of the inputs. This is used to control the sensitivity of the steering, throttle and brake response. The correct input mapping and the selected weights are shown in Appendix C.

Furthermore, the HUD was enhanced in order to provide better statistics for the driver, improving the driving experience and offering additional rewards for user. Not only the overall race time, but also the current lap time is shown, in addition to the best and last lap and the best race time to be beaten. Figure 5.2 shows a screenshot of the enhanced HUD. The current race and lap times are updated in real time, by binding the HUD to the respective variables. The records to be beaten are saved at the end of each race, and loaded at the start of the next race.

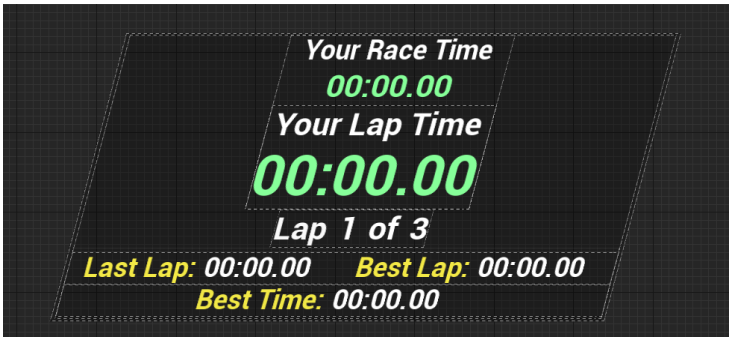


Fig. 5.2 Screenshot of the head-up display of the simulation, which shows information about the current race and lap times of the driver, as well as the records to beat. The information is updated in real-time.

Track Information

Two methods of sending the spline data were implemented. They send the track data either in the representation of the spline points and tangents, as shown in figure 3.2, or as a vector of densely interpolated points. Both methods were implemented using blueprints.

The spline points are available as an array from the "track generator", a blueprint used to create the race track. Every array contains two elements, namely a location and a tangent. Both are 3-dimensional vectors. In order to send the data via ROS, it must be converted to a suitable ROS message.

As both vectors should be sent in the same message, a *twist*-message was chosen as a data format. The ROS *twist*-message contains two vectors (representing linear and angular velocity originally), which are used to send the position and tangent vectors, respectively. The messages are published consecutively, so that they can be picked up by the server script, as explained in the next section, and saved to a text file. The text file can the be loaded to compute an interpolated version of the spline and a number of properties, as explained in section 5.2.1. Appendix C shows the blueprint that implements the publishing of the spline points.

In order to send an interpolated version of the spline directly via ROS, a function was created that generates a vector holding the interpolated points. This vector is then published similarly to the publishing of the spline points as shown in Appendix C. Figure 5.3 shows how this function was designed using blueprints. The logic that is implemented here is a while-loop, which adds points to the array until the end of the spline is reached.

The purple *Interpolate Spline* box represents the function call, and has the Step Size as a function input. The step size, given in cm, defines the distance between consecutive points. Knowing the distance between points is very useful for the algorithms used for the controller, as it simplifies finding points on the interpolated spline.

The spline has some intrinsic properties and functions, which are used for the blueprint implementation. The *Spline Length* property is used to obtain the stopping criterion for the while-loop, i.e. to detect when the total distance of all points in the array equals the length of the spline. The function *Get Location at Distance Along Spline* returns a vector (indicated as yellow connector),

which defines a global position at a given distance from the origin of the spline. This is the data that is stored in the array. At every iteration of the while-loop, the distance is calculated as the product of the length of the array (which is increasing by 1 at every iteration) and the step size.

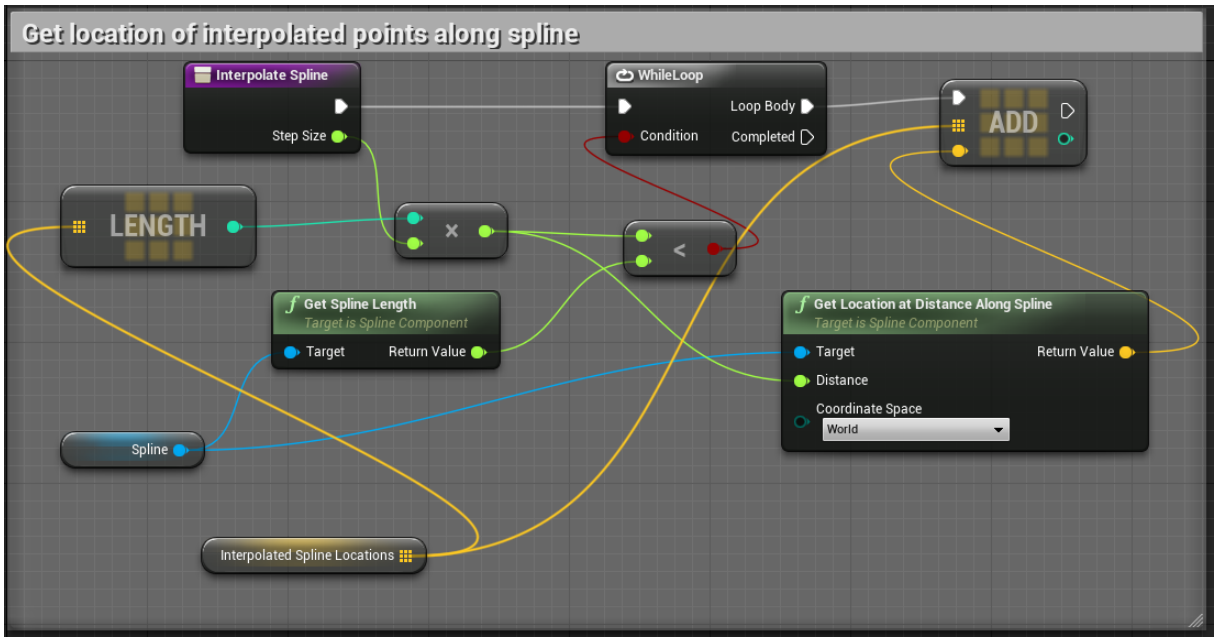


Fig. 5.3 Blueprint design of the function which interpolates the spline data. The function iterates along the entire spline at distances given by the interpolation distance. The points are stored in an array, which is published when queried.

There are a few details to consider in order to allow for a seamless integration of the interpolated spline array with the ROS backend (i.e. the server) that processes the data and makes it available to the controller. Firstly, the spline data should only be sent when necessary, and in the desired format. Therefore, a command was implemented to trigger the sending of the spline. This command is issued by the backend when data is to be received. Additionally, the track should be designed such that the vehicle's driving direction is equal to the direction of the track. This allows efficient searching of directions along the track and is necessary in order to compute the track's properties correctly (see section 5.2.1 for details).

5.1.2 ROS Communication

In order to enable communication with the controller via ROS for all simulator projects, a ROS plug-in was installed outside of this project. The plug-in implements functions in C++ to create ROS subscribers and publishers, which were used for communication between the controller and the simulator. ROS subscribers are all elements which *listen* to a topic, i.e. which receive data, while publishers are used to *send* data.

The script which handles the sending and receiving of data in the backend (i.e. on the controller side) is referred to as the *Server* in the following sections. The server forms the core of the

Listing 5.1 ROS message for publishing driving inputs

```
function publishInputs(data):
    message = {'msg':{'header':data.header, 'steering':data.steering,
        'throttle':data.throttle, 'brake':data.brake,
        'gear':data.gear, 'handbrake':data.handbrake,
        'autonomous':data.autonomous, 'autogear':data.autogear},
        'op':'publish',
        'topic':'/CarInput', 'type':'car_simulator/CarInput'}
    send_message(json.dumps(message).encode('utf-8'))
    return message
```

controller implementation, as it handles all receiving and sending of messages, as well as all computational tasks which are carried out on the controller side of the system.

In order to allow for a successful communication between the simulator and the controller, it is necessary that both sides can "understand" the same message, i.e. interpret messages in the correct way. Thus, whenever a new message type is added, it must be added to both sides of the network. One example for a message which was added within this project is the `CarInput` structure, which handles all the information that is sent from the controller to the simulator.

For this, firstly a new message type must be created, which contains all the inputs as shown in table 3.2 with their correct data types. Listing 5.1 shows how the publisher is filled with input data and encoded into json format. The data is published straight from the low-level controller (see section 5.4.2 for details). Hence, no additional data structure is necessary to handle the online-commands which are sent by the server.

As the `CarInput` is a custom message and not a default message, there is no existing subscriber function in the blueprint-level of Unreal Engine already. Thus, this function must be added from the underlying C++ code, and then the entire game must be re-built. This makes the function available to the blueprint-level, so it can then be used to receive driving instructions from the controller.

The way the server is set up is such that all incoming data that the server has subscribed to is saved in a *queue*, so that no data is lost. However, in the situation of online driving, it is more important to receive the latest data, rather than obtaining the next data in the queue, which might already be deprecated. Thus, in the autonomous driving mode, the logic for the popping of messages from the *queue* is different from other subscribers. In this case, always the latest element in the queue is selected, and all possibly deprecated data is ignored, so the controller is always most up-to-date.

5.2 Data Representation

In order to make the data that is sent by the simulator useful for the controller, the model training and the evaluation, two classes were created that handle the data and perform useful calculations.

One class is the *DataStruct* class, that handles the driving logs, and the other one is the *Splines* structure, that contains various methods to manipulate the track data.

5.2.1 Track Data

As the track data may be either represented as spline points with 3-dimensional position coordinates and tangents, or as a set of interpolated points, a *Spline* object needs to be capable of handling both formats. The track information is stored as a set of densely interpolated points and its properties, which are necessary for the later implementation of other methods.

The internal variables of the class are the *positions*, *gradients* and *curvatures* of each point. Additionally, an *accumulated distance* is computed when the track is loaded, which is a measure of distance along the track. This information is used to speed up the process of finding points along the spline during simulation time. Generally, the class is structured in order to allow a maximum amount of computation to be performed offline and reduce the online computational time.

There are several tasks performed by the spline, which are used frequently by other objects and are thus detailed in the following.

Load track

Two different methods for loading the track were implemented: Computing the track from the spline positions and tangents and reading the interpolated spline directly from a text file. The computation of the spline uses an existing class for computing cubic Bezier splines that had been developed in the Personal Robotics Lab. The methods from this class are applied to create an interpolated representation of the track, which is then re-sampled to the desired number of points.

Either of the two methods are used to create a track, where the reading of a spline from a text file is the default option. In fact, it is only necessary to compute the interpolation of the spline once, which is then stored in a text file which contains the interpolated points along with its properties. This reduces the required computational time whenever a track is loaded.

Compute spline properties

The spline properties that are relevant for most operations on tracks are the gradients and curvatures of the spline at each point. These properties are approximated using the interpolated spline positions. The gradients are computed as the directions of the vectors between adjacent points of the spline, as shown in listing 5.2.

Figure 5.4 shows the approach for computing the curvature. It is defined as the inverse of the radius of a circle passing through 3 points. The three points are placed at a distance L apart

Listing 5.2 Algorithm for gradient computation

```
function computeGradients(track)
    delta = differences(track) % find vectors btw adjacent points
    gradients = 4_quadrant_inverse_tangent(delta)
    return gradients
```

from each other on the track. Choosing points at a distance further than 1 sample away from each other results in more distinctive curvatures, hence the larger distance L was chosen for this implementation. This is shown as $10m$ in the sketch, but in reality a much smaller value of $1m$ ($\approx$ 4 samples) was used in order to reduce the error of the approximation.

Equation 5.1 shows how this geometry can be used to compute an approximation of the curvature as outlined by Calabi*et al.* [37]. The approximate curvature ($\tilde{\kappa}$) is computed as the quotient of 4 times the area of the triangle (A) spanned by the three points divided by the product of the sides of the triangle (denoted by l_1, l_2, l_3).

The area of the triangle is found from the widely used *Shoelace*-formula. The formula describes a method to compute the area of a polygon from the coordinates of its corners, and the formula for a 2-dimensional triangle is applied here. The three points used for the computation are denoted as $\mathbf{x_i} = (x_i, y_i)$. The lengths l_i are computed as the euclidean distances between the points.

$$\tilde{\kappa} = \frac{4 * A}{l_1 * l_2 * l_3} = \frac{2 * |(x_1 - x_3) * (y_2 - y_1) - (x_1 - x_2) * (y_3 - y_1)|}{\|\mathbf{x_1 x_2}\| * \|\mathbf{x_1 x_3}\| * \|\mathbf{x_2 x_3}\|} \tag{5.1}$$

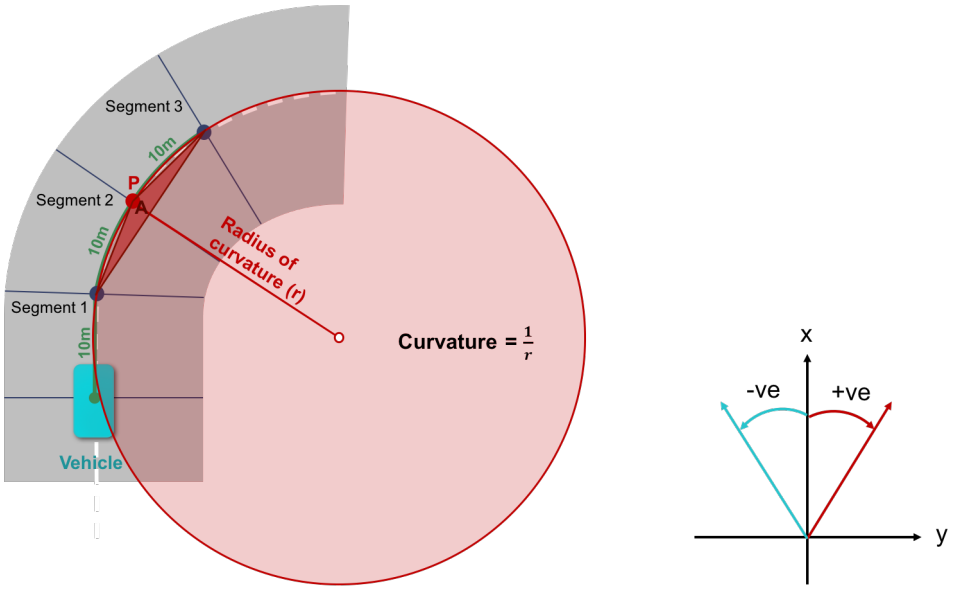


Fig. 5.4 Segment representation of the track and curvature computation. The curvature in point P is computed as the inverse of the radius of a circle through the point P and two points that are each a segment length L (here 10m) away from P. On the right, the direction and angle convention is shown.

Listing 5.3 Algorithm for curvature computation

```
function computeCurvatures(track, distance)
    for point in track:
        x, x_up, x_down = findTrianglePoints(track, distance, point)
        area = shoelaceFormula(x, x_up, x_down)
        l1, l2, l3 = euclideanDistances(x, x_up, x_down)
        curvature_sign = sign(gradient(x_up)-gradient(x_down))
        curvature(point) = curvature_sign*4*area/(l1*l2*l3)
    return curvature
```

Listing 5.4 Algorithm for computation of accumulated distance

```
function computeAccumulatedDistances(track)
    for point in track:
        distance += euclidean_distance(point,next_point)
        accumulated_distance(point) = distance
    return accumulated_distance
```

However, the above formula does not contain any information on the sign of the curvature, i.e. whether it is a left curve or a right curve. This information however is contained in the gradients of the reference points ahead and behind P. If the difference between the gradients ($P_{ahead} - P_{behind}$) is positive, the curvature is positive, and vice versa. This approximation is valid as L is small and the points are thus close. Listing 5.3 shows the algorithm summarising the steps explained above.

The angle and direction convention followed throughout all of this analysis is shown on the right of figure 5.4. This is based on the convention used by Unreal Engine. In this convention, the local orientation of the vehicle always corresponds to $x = 0$. Positive angles are equivalent to right turns (in the direction of the vehicle movement), while negative angles correspond to left turns. Similarly, positions on the right of the track are denoted as having a positive distance to the track center, while positions left of the track have a negative distance.

Finding the triangle points is one application where the third property, namely the accumulated distance, is used. The accumulated distance is a list of distances, which contains the total distance from the first interpolated point up to the indexed point. The last element of the list is thus equivalent to the total length of the track. The logic for this is shown in listing 5.4.

Thus, the distance between any two points on the track can be computed by subtracting the accumulated distances at the individual points. If the resulting distance is negative, the result can simply be wrapped, i.e. the total track length is added to give the required distance.

Using this property for curvature computation, the track is simply searched in positive and negative directions, starting from P, until the points at distance L are found. This is necessary if the distance between the interpolation of the points is not known. If the distance is known, this step is not necessary and the indices can be simply altered by the required number of steps ($k = L/step$).

Listing 5.5 Algorithm for computation of mean curvature vector

```
function getCurvatureVector(curvatures,index,numAhead,numBehind,L):
    curvatures = append(curvatures, curvatures)
    bounds = (index-get_idx(numBehind*L):L:index+get_idx(numAhead*L))
    for n in bounds:
        lowerbound, upperbound = next_bounds()
        segments(n) = mean(curvatures(lowerbound:upperbound))
    return segments
```

Spline applications

Finally, the interpolated spline contains methods that require external input, and which are useful for various applications explained in the next sections. These methods are the computation of the mean curvature vector and the computation of the (signed) distance to the track center. As enabler of any application, it is necessary to find the point of the track which is closest to a given vehicle position. The point is given by the index. As this is a query that is performed so frequently, it was also embedded in the *Splines* structure. Thus, one can simply call `getIndex(position, track)` to obtain the index to the closest point on the track.

For the computation of the mean curvature vector, which is used for the model input as described in section 4.3.1, the following external inputs are needed: The index of the query point on the track, the number of segments ahead and behind the vehicle, and the length of each segment. The index of the query point indicates the point on the track which is closest to the current position of the vehicle.

With this information, the indices of the boundaries of each segment are found, using the accumulated distances explained above. Again, if the exact distance between the interpolated spline points is known, the accumulated distance is not needed. The mean curvature is then equal to the mean of all curvature values within the boundaries of each segment, ahead of the vehicle and behind. In order to avoid wrapping, the curvature vector is doubled at the start of computation. The algorithm is shown in listing 5.5

Figure 5.5 visualises the logic: Based on the index, which is passed to the function, the bounds of the segments ahead and behind are found (by searching the accumulated distance in positive and negative directions). The mean of all elements between consecutive bounds form one entry in the *segments*-vector.

For the computation of the distances to the track center, two components must be determined. The absolute distance is equal to the euclidean distance between the current position of the vehicle (which is an input to the function) and the closest point on the track. The sign of the distance is determined by rotating the vector from the track to the current position of the vehicle by the tracks orientation. Thus, if the vehicle is left of the track, this vector has a negative direction, and if it is right of the track, the vector has a positive direction. Again, the directions adhere to the convention shown in figure 5.4 and are calculated using the 4-quadrant inverse tangent, as shown in listing 5.6.

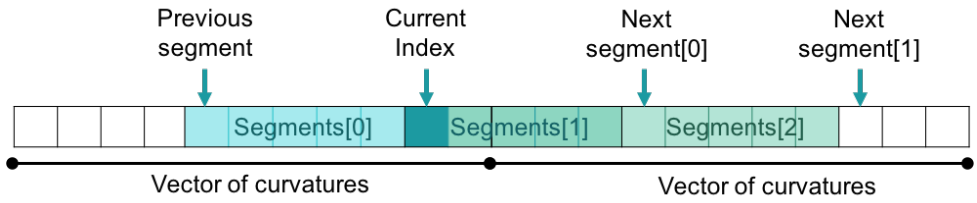


Fig. 5.5 Illustration of the mean curvature vector used as track sensor. From the number of segments ahead (here 2) and behind (here 1) of the track, the mean curvature is calculated for each range, giving a vector of means as the sensor data. For computational speed-up, the vector of curvatures is doubles.

Listing 5.6 Algorithm for computation of signed distance of vehicle to track center

```
function getDistanceFromTrack(vehiclePosition,trackPoint):
    perpendicular_vector = vehiclePosition - trackPoint
    direction = 4_quadrant_inverse_tangent(perpendicular_vector)
    rotated_direction = direction-gradient(trackPoint)
    distance = euclidean_distance(vehiclePosition - trackPoint)
    return sign(rotated_direction)*distance
```

For all operations involving angles, it should be noted that angles are always truncated to $\pm 180°$. This assures that the *sign*-operator can be used, as any angle is always truncated to the multiple of $360°$ that gives the smallest value for the angle.

5.2.2 Test Data

For the simulator outputs, two types of structures were implemented. One is targeted to model training and evaluation, and holds the entire set of outputs including some parameters that are calculated additionally. This object is called *DataStruct*. Another object, called *Vehicle*, is used to hold the simulator outputs during simulation time. Both objects have a similar structure, but as the *Vehicle* only holds single values and interfaces closely with the online controller, it was separated from the *DataStruct* to make it more user-friendly. This way, the interaction in the controller between the vehicle, the personalised driving model and the low-level controller becomes very intuitive to understand, as shown in the next section.

Data Struct

A *DataStruct* object contains the fields specified in table 5.1 and provides the necessary information for model training and evaluation. The simulator outputs are loaded from the driving and testing logs and converted from json format to the respective variables using an *add_json*-method. Every row of the structure corresponds to one test point.

In order to use the logged data for model training and evaluation, some additional information other than the simulator outputs given in table 3.1 must be computed. For this, the methods shown in listings 5.5 & 5.6, which were embedded in the *Spline* structure, are used. These

complete the necessary information for model training. For purposes of evaluation, a method to add the target speed and target distance to the track center was added. The total length of the data set is a counter used to keep track of the number of elements that have been added.

Table 5.1 Variables in DataStruct object. The variables contain all outputs from the simulator and properties which are relative to the spline, as well as the target values used for evaluation.

Simulator outputs	From Spline	Additional Information
See table 3.1	Index (of closest point on track)	Length of data set
	Current distance to center	Target speed
	Segment curvature vector	Target distance to track center

Vehicle

In a way, the *Vehicle* is a simplified version of the *DataStruct*. It does not store the target trajectory, as this is performed by a different object during the simulation, and only keeps the last N simulator outputs. N is the memory of the vehicle, and is set to 1 by default. Only when recurrent networks are used, N is set to the backpropagation length (see section 4.3.2. At every time step, the vehicle's past states are shifted by 1, and the new input is inserted in the last position of the state vector. Similarly to the *DataStruct*, the new information is decoded from a json string into the respective variables.

In addition to the vehicle data, the object is intialised with a copy of the track on which the vehicle moves, so that the index of the closest track point and the current distance to the track center can be computed independently, without having to pass the track every time. This layout was mainly chosen for convenience and to obtain a clear level of abstraction in the code.

5.3 Neural Network Training

The training of the neural networks is performed offline using TensorFlow. TensorFlow is a library designed specifically for deep learning applications that is now widely used in research and industry. What distinguished TensorFlow from other deep learning libraries is that it uses *tensors* to build powerful *computationalgraphs*.

A tensor is a node in the computational graph holding values such as variables, placeholders (equivalent to empty arrays) or the results of mathematical operations. The computational graph determines how the tensors interact and can represent large and complex networks easily. The graphs can be saved and re-loaded using a simple API, which makes it particuarly suitable for this project as training is performed offline, while testing is performed in a different application online.

Two network designs were implemented and tested: Feed-forward neural networks of fully connected layers and recurrent neural networks using a multi-layer LSTM design as explained in section 4.3.2. Both methods use a common dataset-object, which selects and prepares the data for model training.

5.3.1 Training Data Preparation

The data used for training the neural networks is the driving data from the manual driving of the users. The driving logs are saved in text files containing the json strings which are sent via ROS. Using the *DataStruct*-object explained in section 5.2, this data is loaded into a python structure. The data is passed as a list of files, such that any number of test runs can be used for training.

Next, the variables which are used as features (model inputs) and labels (correct model outputs) are determined. The labels are the target speed and target distance from the track center, while the features can be flexibly designed. Within the scope of this project, two combination of features were tested. The first combination contains only the information on the track, i.e. the vector of mean segment curvatures. Different segment lengths and numbers of segments (amount of total "look-ahead" were tested. The second set of data included the curvature vector as well as the vehicle's speed and distance from the track center, forming a weak form of feedback. This is the feature set shown in figure 4.11.

In order to train the model to predict a label a few time steps ahead, the labels were offset by a *delay*-variable. Thus, the correct output at a given time is actually the delayed speed and distance. Appropriate choices for the delay are discussed in 6.2.

For model training, it is often beneficial to scale the input and output features, as this results in equal weighting of all features and allows to use activation functions to introduce non-linearity to the data. Two methods of scaling were considered: Rescaling to a given range and normalisation. Rescaling, as described by equation 5.2, truncates the entire data-set to the range $[a,b]$. Normalisation, on the other hand, scales the data to have zero mean and unit variance as shown in equation 5.3. σ denotes the standard deviation of the data here.

$$x_{rescaled} = a + \frac{x - min(x)}{max(x) - min(x)}(b-a) \tag{5.2}$$

$$x_{norm} = \frac{x - mean(x)}{\sigma(x)} \tag{5.3}$$

To allow for maximum design flexibility of the neural net, both statistics were computed for the feature and label sets separately, and made available as a property of the dataset object.

Once the features and labels of the test data have been created, the data is split into training and validation sets. The training set is used to adjust the network weights. The validation set is used for cross-validation and to detect overfitting. The training and validation data is chosen at

Listing 5.7 Tensorflow implementation of a fully connected two-layer neural network using the tensorflow.contrib.slim tools (fully_connected and stack). It takes the network input tensor, the number of outputs, the architecture of the layers (as a list of layer sizes) and the optional activation function as inputs and returns the output tensor of the network, z.

```
function neural_net(x, n_labels, layer_architecture, activation_fn=None):
  net = tf.stack(x, tf.fully_connected, layer_architecture, scope="stack")
  z = tf.fully_connected(net, 2, activation_fn, scope="fc_layer")
  return z
```

random, proportionally to the size of the data set. Unless specified otherwise, 80% of the driving data was used for testing, and 20% was used for validation.

Finally, the training and validation features and labels are split into batches, which are then used for network training. This is performed using a *generator* function, which returns a batch of features and labels out of the randomly sorted arrays at every iteration of the algorithm.

5.3.2 Feed-forward Neural Network

The feed-forward neural network is implemented as a deep network of fully connected layers which are stacked on top each other, as it was shown in figure 4.11. It uses the tensorflow *contrib.slim* library, which is a library designed for deep learning that contains powerful tools to simplify the construction of neural nets.

The functions *fully_connected* and *stack* can be used to define a fully connected network of any architecture in only 2 lines as shown in listing 5.7. In the first line of the algorithm, a number of fully-connected layers is stacked on top of the input tensor x. The number of layers is given by the length of the *layer_architecture*, while the value of each element in the vector gives the size of the respective layer.

The second line builds the output layer by connecting the stacked net to the output nodes, which are always 2 for the implementations in this project. The activation function determines whether the output nodes are multiplied by a nonlinear function such as *tanh*. The return variable z is a tensor which holds the outputs of the neural net.

Instead of passing the features directly to the neural net, they are scaled so that all features have equal weighting and so that an activation function can be used in the output layer of the net. As the most common activation functions, namely the *tanh* and *sigmoid* functions, give outputs in the range $[-1, 1]$ and $[0, 1]$, respectively, the output data is re-scaled to this range. For reasons of consistence, the input is also scaled to the same range.

Listing 5.8 shows the definition of all tensors necessary for the model training. The placeholders x and $y\_$ are used to feed the features and labels to the neural net. The tensor *x_scaled* is a scaled-down version of the feature tensor, using the formula shown in equation 5.2. This scaling operation is performed within the computational graph so that the scaling is always consistent.

Listing 5.8 Definition of tensors required for training and testing of the network. x and y_ hold the features and labels, respectively. From the scaled features, the network is created and the outputs are scaled back to the original scope. The network is trained by minimising the mean squared error of the unscaled labels.

```
x = tf.placeholder(tf.float32, [None, FEATURE_SHAPE[1]])
y_ = tf.placeholder(tf.float32, [None, OUTPUT_SHAPE[1]])
x_scaled = scale[0]+(x-xmin)/(xmax-xmin)*(scale[1]-scale[0])

# Build the graph for the net
y = nn(x_scaled, activation_fn=tf.tanh)
output = tf.add((y-scale[0])*(ymax-ymin)/(scale[1]-scale[0]), ymin)

loss = tf.reduce_mean(tf.reduce_sum(tf.square(y_-output),1))
training_step = tf.train.AdamOptimizer(LR).minimize(loss)
```

The minima, maxima and the scaling range are saved as variables to the computational graph, thus they are restored when the model is loaded for online application.

The last layer of the neural network as defined in listing 5.7 gives the output tensor y, which is scaled back up in the *output*-tensor. The loss is defined as the mean squared error between the original labels and the scaled outputs of the network, which represent the true targets. The network weights are optimised using the *Adam*-algorithm. It is an improved version of the gradient descent algorithm which is computationally efficient and does not require much tuning [38].

The neural net is then saved using a tensorflow *Saver* object. The *Saver* allows to store the entire computational graph built by tensorflow and all variables and parameters, so that the entire network can be rebuilt later for online testing. Additionally, the training and testing accuracies are written into a log-file. These logs can be used for visualising the training process in TensorBoard, a set of visualisation tools developed specifically for this purpose. With TensorBoard, a number of training processes can be compared with respect to their training and testing accuracies, so that the network which performed best can be selected.

After the graph is built and stored, the neural network is ready to be run. For this, a tensorflow *Session* is initialised. In order to improve the network performance and achieve convergence, the training is performed in multiple epochs, i.e. multiple iterations of the training data. Within every epoch, the network is optimised in batches, which are subsets of the total data set.

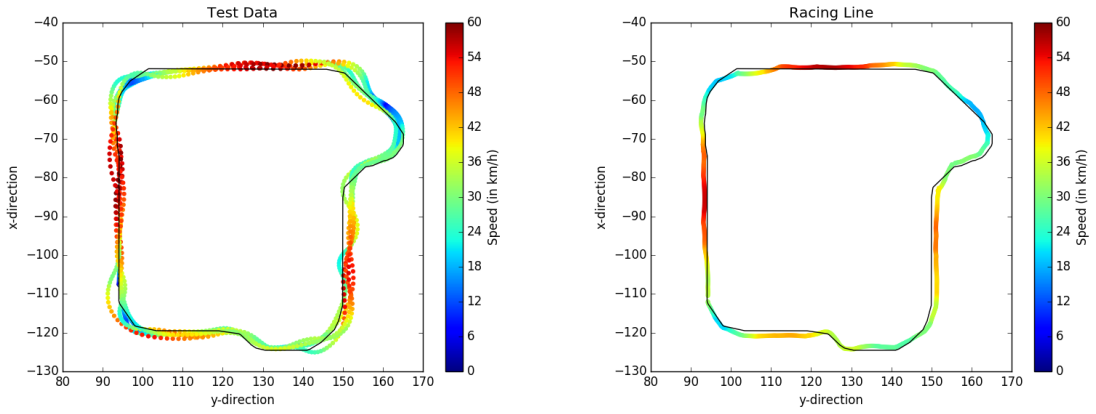
Listing 5.9 shows the procedure for the training of one epoch: The batches are loaded within a for-loop, and used to create a *dictionary*. This is equivalent to filling the placeholders x and $y\_$ with an array of size $(batch\_size, n\_features)$ and $(batch\_size, n\_labels)$, respectively. The dictionary is used to compute the training accuracy of the step and to optimise the weights using the Adam optimiser. The training accuracy of each epoch, which is stored in the *Saver*, is the mean accuracy of all batches within that epoch.

An additional visual representation of the trained network performance is generated by running the entire data of a specified test file through the network. The test file can either be the data on

Listing 5.9 Training of the neural net. For every batch of training data, the current loss is computed, and the weights are updated in a next training step.

```
for _ in range(dataset.train_batches_per_epoch):
  features, labels = dataset.next_train_batch()      # Get the next batch
  fd = {x: features, y_: labels}                     # Fill placeholders

  training_accuracy += loss.eval(feed_dict=fd)    # Evaluate performance
  train_step.run(feed_dict=fd)                       # Train the network
training_accuracy /= dataset.train_batches_per_epoch
```



(a) Human driving trajectory *(b) Model with track data only*

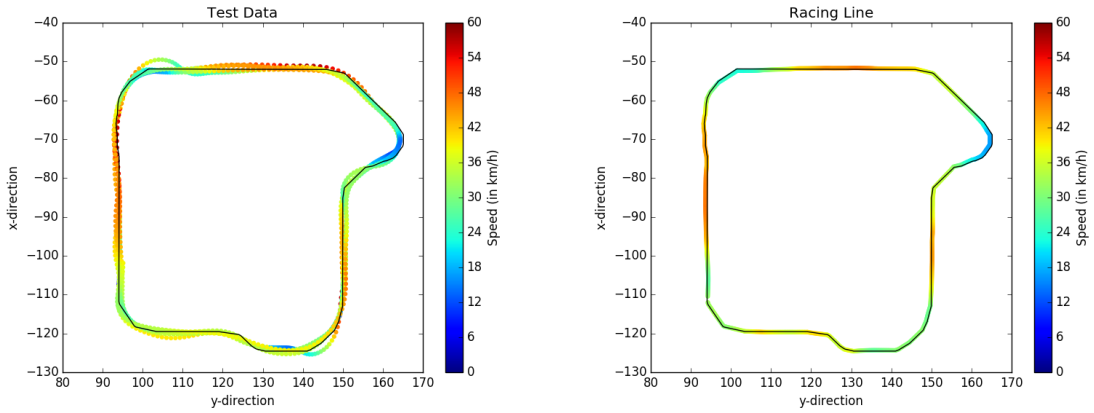
Fig. 5.6 Visualisation of the results of the model training as performed at the end of each training session. From the human driving (left) a model is build, which represents the learned trajectory (right). The model shown here only takes the track data as model inputs.

which the network was trained, or a different track. The target trajectory is then plotted, and this plot is saved together with the model and a plot of the original test data.

Figures 5.6 & 5.7 show examples of the generated plots for two different drivers. The left plots show the training data from 5 laps performed by the two drivers. The right plots show the trajectories generated by the neural network models. The models were trained using the same parameters, only taking the information on the course of the track as model inputs. The target trajectory is predicted from these inputs with a prediction horizon of 5 time steps. Hence, for every segment of the track the targets will be exactly equal, giving a single "racing line".

A comparison of figures 5.6 & 5.7 shows that the resulting models are indeed capable of capturing the different driving styles. The driver shown in figure 5.6a drives rather slow, but remains close to the center of the track, while the driver in figure 5.7a drives faster, breaks harder and deviates much stronger from the center.

This is just an example to show that the neural net as implemented above is capable of reproducing certain characteristics of a human driving style, and to provide familiarity with the visualisation of the performance. A more complete testing, evaluation and discussion of the performance of the neural networks with different parameters is provided in chapter 6.



(a) Human driving trajectory *(b) Model with track data only*

Fig. 5.7 A comparison of the network model outputs for an aggressive driver to a more careful driver (see fig 5.6 shows that the network can capture some key driving characteristics.

5.3.3 Recurrent Neural Network

The recurrent neural network was implemented using long short-term memory (LSTM) cells, which were explained in section 4.3.2. The computational graph of the recurrent neural network is thus fundamentally different from the fully connected structure created in listing 5.7. Many elements of the set-up of the required tensors and the running of the training and validation sessions however can be adapted for the recurrent network.

The fundamental difference to the training of feed-forward networks is that the recurrent network is trained to recognise temporal relationships. Thus, the training data must contain time series of a given length, which is defined as the $backpropagationlength$. Consequently, the placeholders for the features and labels are expanded by one dimension to the shape $[n\_batches, backpropagation\_length, n\_features]$.

Changing the shape of the input and output tensors also requires to change the *generator* function, which prepares the training and testing batches. The generator for the recurrent net was thus adapted so it yields 3-dimensional batches. Every sample is hence a time-series of features or labels.

The computational graph is based on the TensorFlow rnn_cell class, which offers a number of functions to implement recurrent LSTM cells efficiently. The construction of the net is shown in listing 5.10 and was adapted from [39]. The initial state is a placeholder, which is filled with zeros at the start of each new training epoch, or initialised with the last state during the training phase or during online testing.

The state of an LSTM cell has the form of a tuple, which has two elements: A cell state and a hidden state. The cell state represents the memory of the network, which is computed from a combination of the previous cell state, passed through the forget gate, and the current input, passed through the input gate. The hidden state is computed by passing the current state through

Listing 5.10 Definition of the computational graph for a recurrent neural network using the dynamic RNN architecture. The current state is a tuple of the hidden state and the cell state, whereby the hidden state is used to compute the output [39]. NUM_LAYERS gives the depth of the network, while STATE_SIZE defines the length of each state within a layer.

```
def rnn(x):
    init_state = tf.placeholder([NUM_LAYERS, 2, None, STATE_SIZE])
    rnn_tuple_state = create_tuple(init_state, NUM_LAYERS)
    # Initialise weights for computation of the output tensor
    W = tf.Variable(np.random.rand(STATE_SIZE, OUT_LENGTH))
    b = tf.Variable(np.zeros((1,OUT_LENGTH)))
    # Create computational graph for the recurrent LSTM network
    cell = tf.nn.rnn_cell.LSTMCell(STATE_SIZE, state_is_tuple=True)
    cell = tf.nn.rnn_cell.MultiRNNCell([cell]*NUM_LAYERS, state_is_tuple)
    hidden_states, current_state = tf.nn.dynamic_rnn(cell, x)
    y = tf.matmul(hidden_states, W) + b
    return y, init_state, current_state
```

the output gate and multiplying it with an activation function (typically *tanh*) [40]. Thus, the hidden state is used to calculate the output tensor y.

The benefits of using TensorFlow for this implementation is that it has a comprehensive API for the creation of LSTM networks. The *LSTMCell* command creates a cell which has the layout shown in figure 4.12. With *MultiRNNCell*, multiple of these cells can be connected in order to increase the depth of the network. With $dynamic_rnn$, the inputs can be dynamically unrolled to give a list of hidden states and the current cell state, which is convenient for training.

The definition of the training step, the loss function and the training, validation and testing phases is equivalent to the feed-forward neural net, with the exception that the features must be re-shaped into the required format (which is handled by the *generator* function, and that the initial state is another placeholder that must be filled in order to execute the network.

5.4 Controller Implementation

Figure 5.8 shows a block diagram of the modules that were implemented for the controller. The interaction of the modules shown here is handled by the *Server*, the structure of which was explained in section 5.1.2. The server coordinates the setup of the modules, the sending and receiving of data via ROS, and it calls the *TargetCalculator* and *Controller* modules at every time step when the vehicle is driving autonomously.

The *TargetCalculator* module contains different methods for calculating the target speed and target distance. It takes the inputs appropriate for the model, i.e. either a network graph for the feed-forward and recurrent neural networks or the relevant test data for K-nearest neighbor regression. Additionally, a method that achieves a simple following of the track is implemented as a reference for validation and comparison. Finally, the *Controller* module contains separate

PID controllers for the speed and steering control and computes the driving inputs that are then encoded and sent to the simulator via ROS.

The *Track* module holds the information on the track in the form of interpolated points and their properties. The *Vehicle* module stores the current simulator outputs, which are converted from the json message sent via ROS. These structures were explained in section 5.2.

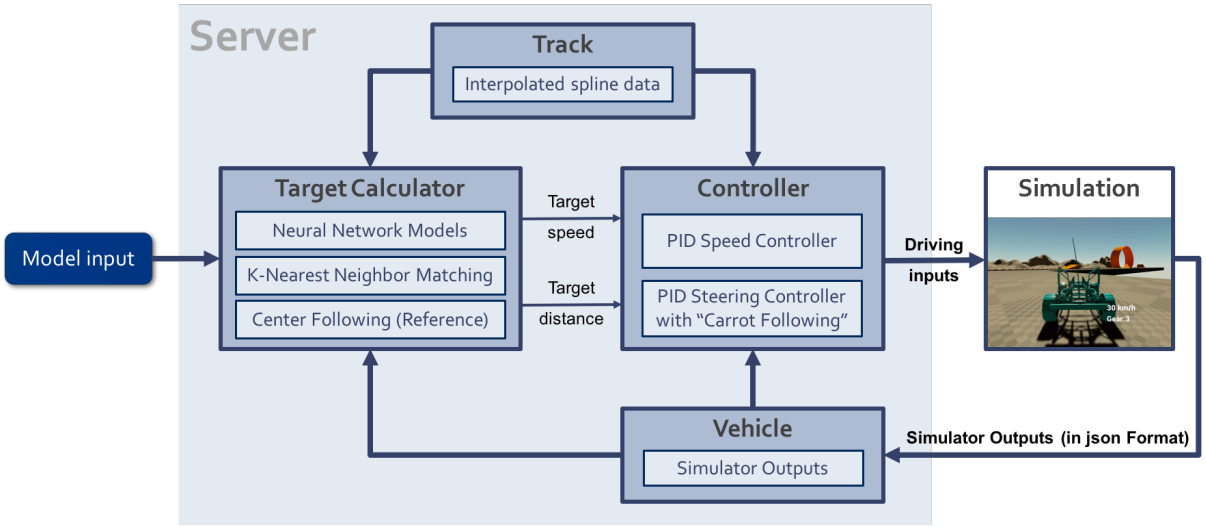


Fig. 5.8 Block diagram of the controller modules handled by the server. The target calculator uses the information on the track, the model and the simulator outputs (stored in Vehicle) to calculate target speed and target distance using one of the shown methods. The controller then uses this information to compute the driving inputs that are sent to the simulator.

5.4.1 Target Calculator Module

The target calculator consists of 4 different classes, one for each implemented learning method, which all have the same structure and properties. Two classes are the previously explained neural network models (the feed-forward model and the recurrent model are treated as seperate models, as they require a different setup. Additionally, k-nearest neighbor regression was implemented to compare against the neural nets. A reference implementation which follows the center of the track was used for tuning of the PID controller, as it provides a steady reference frame.

Every class is initialised with the necessary set-up parameters (which differ between learning methods) and contains a function *get_target*, which is called with the vehicle structure as input. It computes the target properties. The properties are the measures which represent the human behaviour, namely the target speed and target distance to the track center. These properties are accessed by the PID controller afterwards to compute the low-level driving commands.

K-nearest Neighbor Regression

The k-nearest neighbor model, as explained in section 4.3.2, is formed directly from the test data. Thus, the model fitted with a set of features, which form a look-up table for prediction. The

Listing 5.11 Class overview for k-nearest neighbor regression. In the initialiser, the model is fit with the driving data. During driving, the next targets are predicted from the most similar instances of human driving data.

```
class KnnSearch():
    def __init__(data, k):
        knnRegressor = KNeighborsRegressor(n_neighbors=k,method='distance')
        features, labels = compute_features(data)
        knnRegressor.fit(features, labels, prediction_horizon)

    def get_target(vehicle):
        features = compute_features(vehicle)
        target_speed, target_distance = knnRegressor.predict(features)
```

predicted target is then simply a (weighted) mean of the lables of the k most similiar feature instances in the look-up table.

The k-nearest neighbor regressor is implemented using the $scikit-learn$ toolbox for k-nearest neighbor search. It contains three core elements, namely the initialiser, which sets the necessary parameters, the fitting function and the prediction function (see listing 5.11). While the initialisation and the fitting can be performed once when the model is loaded, the prediction is performed at every time online step, and thus it is called in *get_target*.

The parameter choices for the k-nearest neighbor search are the number of nearest neighbors, k, and the averaging method. For this implementation, the method *distance* was chosen, which forms a weighted average of the k most similar instances based on the euclidean distance of the features. Thus, more similar features are given a higher weighting.

The features that were used for the fitting of the model are the arrays of mean curvatures ahead and behind the vehicle. These are computed offline for the entire dataset using the algorithm shown in listing 5.5. Similarily, the current features are computed from the state of the vehicle during autonomous driving.

The labels of the model are the target values, i.e. the target speed and target distance. Particularly, these are the targets at a given time in the future, which is the prediction horizon. If the prediction horizon is 1, the immediately next time step $t+1$ is predicted by the model. However, in practice a larger time step is used in order to account for the lag in the reaction of the PID controller (given by the time constant). The prediction is implemented through an offset of the labels by the prediction horizon.

As the model is only based on past human data, it is not expected to diverge largely from the track. In unknown situations, a weighted average of the most similar situations is computed, which results in a value that is within the bounds of the human driving behaviour. However, the resulting trajectory in unknown situations might be very different from the one that the human driver would follow. The testing and evaluation of the performance of this model is discussed in chapter 6.

Neural Network Models

For both types of networks, the same logic is followed for the target calculation. Two separate classes were only used in order to account for the fact that the input data is required to have a different format for the two networks, and because the recurrent net requires that the current state is passed to a placeholder.

In the initialisation of the class, the model that was saved during the offline training phase is restored, together with the parameters which are required for the computation of the features. These are specifically the parameters for the computation of the curvature vector, i.e. the length of each segment and the total number of segments ahead and behind the vehicle (denoted by L, $Npos$, $Nneg$, respectively). Saving these parameters in the computational graph and restoring them at the start of a driving session assures that always the same parameters are used and avoids mistakes.

During autonomous driving, the features are computed at each time step using the restored parameters. The feed-forward network always returns one value, which is a list of the target speed and the target distance, while the recurrent net returns a vector which has the length of the memory of the network, which was denoted as $backpropagation\_length$ in section 5.3.3. The last element in the vector is used as the target state.

The prediction using neural networks is much faster than the KNN implementation, as all weights are pre-computed, and it is not necessary to search the entire data-set at every time step. This allows fast sampling and can achieve precise control.

Center following

The policy for following the track center aims to follow the track at a constant speed on straight tracks, but slows down before tight curves. Using such a policy, it is possible to test high driving speeds without making the vehicle drift far outside the track in tight curves. This would happen without a slow-down rule, as there is a physical limit to the turning radius of a vehicle at a given speed.

The implementation uses the rule given in equation 5.4. The $reference\_speed$ is the maximum speed that the vehicle will reach, which is a parameter passed during set-up of the module. Another parameter is the $slowdown$, which is given as percentage of the maximum curvature. If slowdown is large, the vehicle will brake strongly before curves, while a small value means that the speed is nearly unchanged.

A high slowdown is appropriate when large reference speeds are tested, while a small slowdown is useful when testing small speeds. As this implementation was mainly used for tuning the PID controller, both regimes were of interest.

$$speed = reference\_speed - slowdown * \frac{current\_curvature}{maximum\_curvature} \tag{5.4}$$

Listing 5.12 Logic for finding low-level control signals

```
function getSignals(vehicle, target):
    check_gears(target.speed, vehicle.speed)
    targetAngle = carrot_following(vehicle, target)

    steering = steering_PID(vehicle.rotation, targetAngle)
    throttle, brake = speed_PID(target.speed, vehicle.speed)

    adjust_controls(vehicle, target)
```

5.4.2 Low-level Controller Module

The low-level controller module computes the low-level control signals for throttle, brake and steering from the target speed and target distance to the center of a given track. If necessary, it also adjusts the gears of the vehicle.

In order to do so, the function *get_signals*, which interfaces directly with the server, executes a number of key functionalities that are summarised in listing 5.12. Firstly, it checks whether the gears must be set manually, which is the case if the vehicle is to move backwards. Next, the carrot-following algorithm is executed, which finds a target orientation of the vehicle.

The target orientation, together with the current rotation of the vehicle, forms the error for the PID controller for the steering control. A similar controller handles the generation of throttle and brake commands in one common speed controller. Finally, the control signals are checked again and overruled in the case of a large error as explained below.

PID Controller

The PID controller used to calculate both the steering and the throttle and brake commands was implemented from scratch, as this allowed more flexibility to amend the controller for the purposes of the project. The calculation of the correction signal is implemented using a standard PID approach for discrete-time signals, as shown in equation 5.5. u_t denotes the correction signal, e_t is the error at time t and the controller gains are defined as K_p, K_i, K_d.

$$e_t = target_t - state_t \tag{5.5}$$

$$derivative = \frac{e_t - e_{t-1}}{t_s} \tag{5.6}$$

$$integral = integral + t_s * e_t \tag{5.7}$$

$$u_t = K_p * err + K_i * integral + K_d * derivative \tag{5.8}$$

Similar to available PID controller implementations, the integral value is limited to a range which is set in the initialiser of the controller. What is different from many implementations is that the controller gains, as well as the reference to be tracked, are changed at every time step in the

Listing 5.13 Conversion of acceleration signal into brake and throttle

```
if gear == -1:
    u = -u  % invert acceleration command for backwards movement
if u>0: throttle=min(u,1)
else:   brake=min(-u,1)
```

function which calculates u. This is appropriate as both values are continuously changing, and it simplifies the function interface. Furthermore, the steering is given in degrees and hence must be normalised to $\pm 180°$ to avoid wrapping errors.

Speed Control

For the implementation of the PID controller for speed control, the current speed and the target speed are passed to the PID controller described above as the *state* and the *target*, respectively. The controller gains are calculated at each time step as a function of the target speed. These functions were determined during tuning of the PID controller, which is explained in section 6.2.

The resulting correction signal u is then converted into a throttle or brake command, depending on the sign of u. Additionally, driving backwards requires an inverse of the accleration control. "Faster", when going backwards, means that the speed of the vehicle decreases. Inversely, braking means that the speed increases. Thus, the correction signal u must be inverted if the car is going backwards. Listing 5.13 shows the implementation of this logic.

Steering Control

The steering control was implemented in two steps, as shown in listing 5.12. Firstly, the target orientation is determined using a carrot following approach, and then a PID controller is used to minimise the orientation error.

Listing 5.14 shows the algorithm which calculates the target orientation, based on the concept shown in figure 4.4. The distance covered within a lookahead of $1s$ (which was used initially, see section 6.2 for a discussion), is computed as the mean of the current speed and the target speed. The target location of the vehicle is hence a point which is a *carrot_distance* away from the vehicle along the track, with a given target distance to the track center.

The target angle is computed as the direction of the vector between the current location and the target location of the vehicle. Similar to the speed control, a correction is needed for backwards motion, as the car's orientation is opposite to its direction of motion. See section 4.2.2 for a furher explanation.

The function to obtain the steering command from the PID controller signal is implemented in analogy to the speed controller. The only difference here is that the steering signal is divided by the maximum steering angle of the vehicle, and truncated to the range $[-1, 1]$ rather than $[0, 1]$, which is the case for the throttle and brake.

Listing 5.14 Algorithm for computation of the target orientation of the vehicle using a carrot following approach

```
function carrot_following(location, speed, target, lookahead):
    carrot_distance = lookahead*mean_speed()
    targetLoc = findTargetLoc(carrot_distance, target, location)
    angle = inverse_tangent(targetLoc-location)
    if gear == -1:  angle = 180+angle   % invert for backwards steering
    return angle
```

Additional amendments

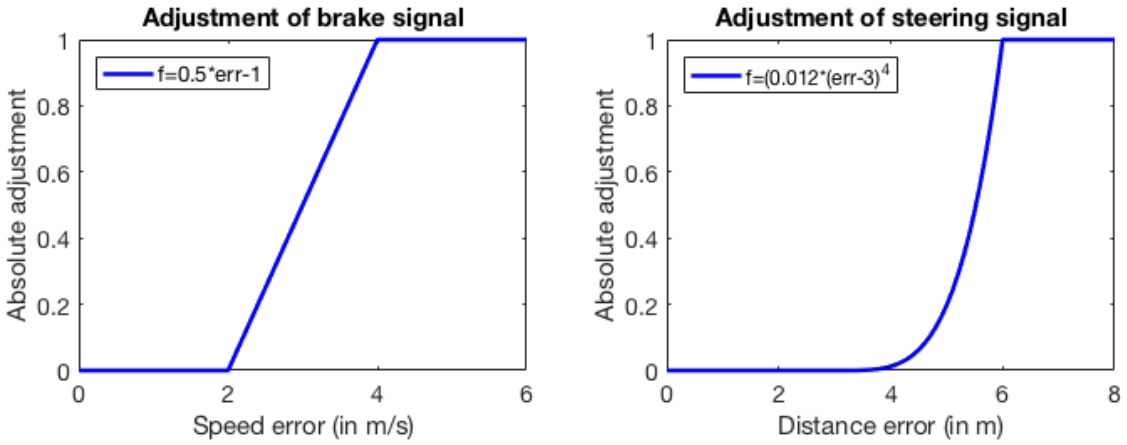
In order to improve the performance of the speed and steering control, two types of amendments were implemented into the low-level controller. The first amendment is the control of the gear, if necessary, and the second amendment is a *posteriori* amendment of the control signals in specific situations. The amendments to the control signals were inspired by Melder's discussion of possible enhancements of PID controllers for vehicle control [30].

The amendment of the gears is necessary as otherwise it is impossible to drive backwards. Thus, if the target speed is (sufficiently) negative, the gear is set to -1 and the *autogear* is deactivated, overruling any gear setting of the simulator. Furthermore, automatic gear changing can hinder the PID controller to stabilise at a given speed in some situations. This is particularly the case for low speeds in the range of $20-30km/h$. Thus, in this range the gear is forced to 2.

The amendment of the controller signals are used to handle large errors effectively. A faster convergence for large errors could be achieved by changing the gains of the PID controllers, however this would lead to larger overshoot or oscillation, which is not desirable. Hence, an additional control mechanism has been implemented, which allows to quickly react to large errors while keeping the fine-tuning properties of the PID controller.

The function *adjust_controls* in listing 5.12 handles this additional control mechanism by increasing the control signals for the brake and steering by a linear function, depending on the error. An appropriate function was found experimentally. The experiments also showed that the throttle reacts more quickly than the brake, so these amendments were only implemented for the brake and the steering commands.

Figure 5.9 shows the linear functions which are added to the existing signals in case of large errors in speed or distance to the track center. The amendment for the steering was implemented as a polynomial function in order to increase smoothness of the steering, which has a high priority.



(a) Adjustment to the brake control signal *(b) Adjustment to the steering control signal*

Fig. 5.9 Adjustments to the brake (left) and steering (right) control signals if the error exceeds an allowed limit. For the brake control, the error is given by the difference to the target speed, while the steering error for this purpose is defined as the difference to the target distance from the track center.

6 Testing and Results

The following chapter details the testing that was carried out to assess the performance of the different implementations. It details the set-up which was used for testing, and the results of the testing process. Firstly, the tuning of the PID controller is explained, followed by an extensive testing of each of the personalised models. A discussion of the results and a comparison of the models can be found in chapter 7.

6.1 Test Setup and User Data

For all experiments presented in this chapter, three example tracks were used (unless specified otherwise). Different drivers drove each of the tracks for a specified number of laps, and the data for each run was recorded. These logs form the basis of the model training.

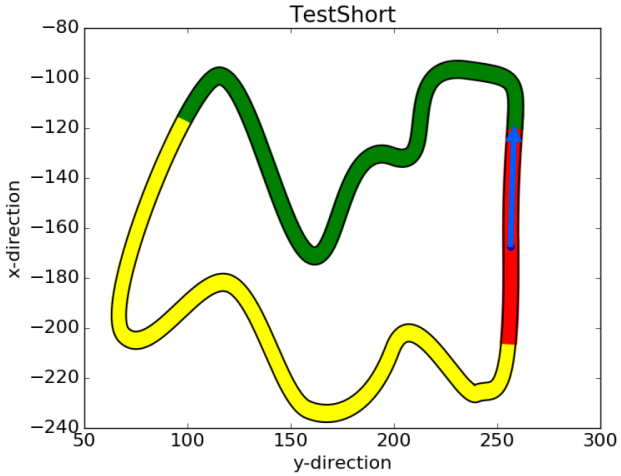
6.1.1 Test Tracks

Figure 6.1 shows the three test tracks, which were built in Unreal Engine. Each track has a width of $7m$. For the purposes of this analysis, the tracks were split into segments of similar difficulty. The difficulty is categorised into 4 classes, where A denotes an easy segments (shown in red) and D denotes a hard segment, shown in green. Each track has a different length and contains different characteristics. This is useful because it allows to test the model for different characteristics, and to assess the potential of the controller to generalise to unseen situations.

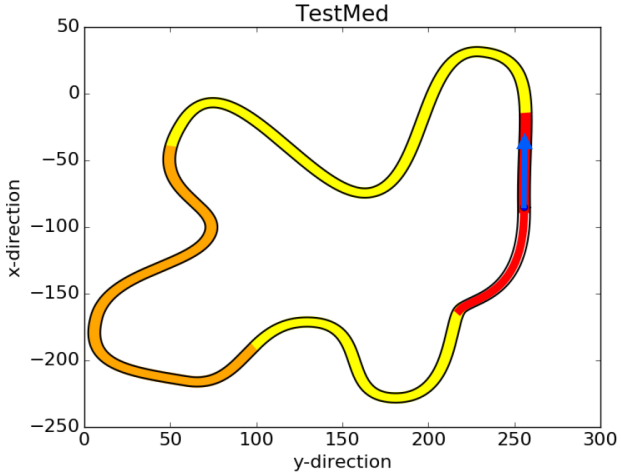
Each of the tracks contain both left an right turns, so that the controller is capable of moving in both directions. The blue arrows in each of the graphs in figure 6.1 have the same length of $50m$ and point in the direction of travel. This gives an indication of the scaling of the tracks.

The shortest track (fig. 6.1a) has a total length of $710.5m$ and is the most difficult. It consists of many sharp turns in both directions, in which a driver must slow down strongly in order to remain on track. The track shown in figure 6.1b has a length of $1016.7m$, and consists of longer and smoother curves. Those curves can be taken much faster. Finally, the long track has a total length of $1539.2m$. It contains both tight and flat curves, and two long straight segments in which the driver can reach the maximum speed of the vehicle. The maximum speed was capped at $90km/h$.

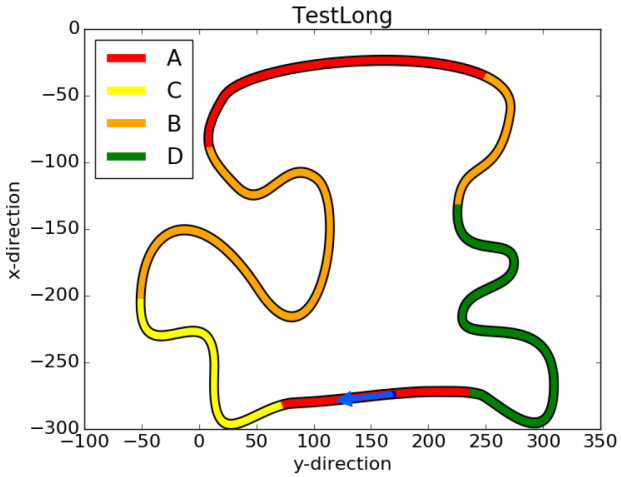
In order to obtain comparable results from the training of different models, the total number of laps to be driven was determined to give an approximately equal total race time. Thus, the number of logged data points should be in the same order of magnitude. Every driver was asked to drive 10 laps on the short and medium track (as the short track is more difficult and requires a slower speed), and 7 laps on the long track. This gave approximately $4000 - 5000$ test data points per track.



(a) Short track (710.5m, most difficult)



(b) Medium track (1016.7m, easiest)



(c) Long track (1539.2m, easy and difficult parts)

Fig. 6.1 The three tracks used for testing the implementations. The short track is very difficult with many sharp turns. The medium track is easier with flatter curves. The long track has some long straight segments, and a number of turns of varying difficulty. The level of difficulty is indicated by the color of the segment (where red means easy and green means hard). The arrows indicate the direction and are each of length 50m.

6.1.2 Human Driving Performance

Five different drivers were asked to drive the three courses. Every driver was allowed one round to practice on each track. No constraints were given to how they should drive the track. Purposely, the drivers were allowed to drive outside of the track boundaries, as this may also form part of the personal driving style. Lifting the constraint of making the users drive only on the track means that personal styles of different users can be more clearly distinguished, as some drivers might aim to always stay on the track, while others might, for example, try to cut corners.

The selected drivers all had different driving experience. One driver was a complete beginner (B), two were intermediates (I1,I2) and two were expert drivers (E1,E2). Every user drove the tracks in a different order, so that there is no bias towards the tracks which were driven last. Out of the 5 drivers, the three most distinguishable styles were chosen for the further analysis in the next sections.

In order to quantify the driving style, the following metrics were observed to fulfil the requirements given in section 3.1.2:

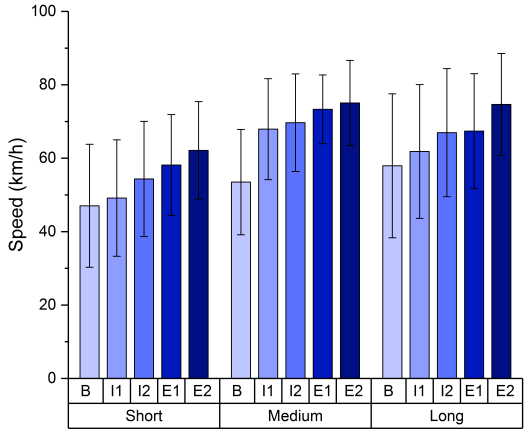
- Mean and variance of speed (in m/s)
- Mean and variance of absolute distance to the track center (in m)
- Total progress in 100 seconds (in m)

Figure 6.2 visualises the driving styles of the users along each of the three test tracks. Each of the values shown in the figured have been computed from the mean trajectory of the respective driver. The mean trajectory was found by averaging the speed and distance to the track center for all laps. For this, the track was split in many small segments, and the average of the output values in that segment was computed. The mean across all segments of one track is shown in figure 6.2. The numerical results of the analysis are shown in table D.1.

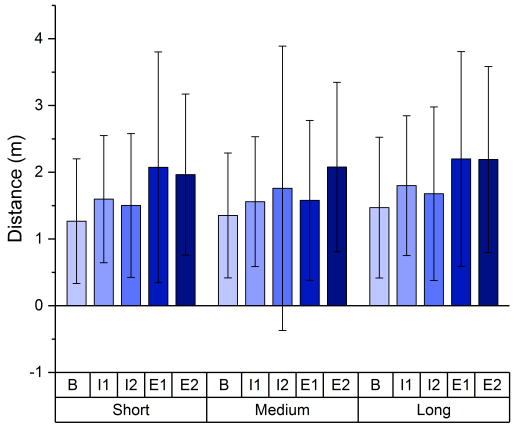
The data clearly shows a correlation between the difficulty of the track and the mean driving speed: All drivers drove at a significantly lower speed on the short track (which was most difficult), and approached similar mean speeds on the other two (easier) tracks (see figure 6.2a).

The figure also shows the significant difference between the 5 test drivers. The more experienced drivers drove at a higher speed, which also resulted in a larger progress in the first $100s$ as shown in fig. 6.2c. The deviation of the speed is larger for the more inexperienced drivers. This allows the conclusion that more experienced drivers drive more efficiently and avoid strong breaking, for example.

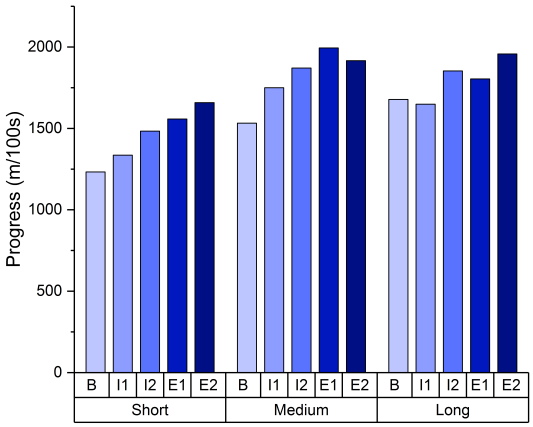
The distance to the track center shown in figure 6.2b also shows a significant differences between experienced and inexperienced drivers. The clear tendency which can be observed here is that the more experienced drivers deviate stronger from the center of the track. The deviation also tends to be larger, meaning that they deviate largely in some parts of the track (e.g. when approaching turns), but stay close to the track centers on other parts.



(a) Mean and deviation of user speed



(b) Mean and deviation of absolute distance



(c) Total progress in 100s

Fig. 6.2 Comparison of the data from 5 test drivers on the different tracks. The performance is compared in terms of the mean and deviation of the speed along the track (top), the mean and deviation of the absolute distance to the center of the track (center) and the total progress within the first 100s of the simulation (bottom). For the top two graphs, the mean is denoted by the colored bar, and the error bar indicates the standard deviation ($\pm\sigma$).

Based on the above analysis, three drivers were chosen for future comparison: One beginner, **B**, one intermediate driver, **I2** and one expert, **E2**. These drivers exhibit clearly distinguishable driving characteristics, which should be imitated by the model. Thus, these are the metrics that the models will be compared against.

6.2 PID Tuning

The PID controllers for the control of speed and steering have a number of parameters, including the controller gains and the lookahead parameter. As the analysis in section 4.2.2 showed, different vehicles and environments exhibit very different characteristics, and thus the PID parameters should be adjusted for any new environment that the controller is deployed in.

A trial-and-error method was chosen for the tuning of the PID controllers. Implementing a method for auto-tuning the controller was considered, but it was not feasible within the scope of the project. This is due to the current set-up of the simulation, which must be started manually for each test run, then stopped so that new parameters can be computed, and restarted again manually. That makes automatic iterative approaches infeasible, as these often require 100 or more iterations, and the implementation of an online tuning method was not a priority for this project.

The tuning for the speed and steering control were carried out separately, as both controllers act independent of each other. The speed controller was tested first, as a stable speed control is necessary to test the steering control.

6.2.1 Speed Controller

For the PID controller implementation, the throttle and brake control is treated as one single contoller, as it was explained in section 4.2.2. Initial estimates of the PID controller gains were obtained from a Matlab simulation of the acceleration behaviour. The Matlab simulation contained a state-space model of a simplified system model and a feedback-loop for the controller.

As the brake was rarely used on the given test tracks, the state equation was based only on the relationship between speed and throttle, given in equation 4.4. The equation was derived from an analysis of driving data from Unreal Engine.

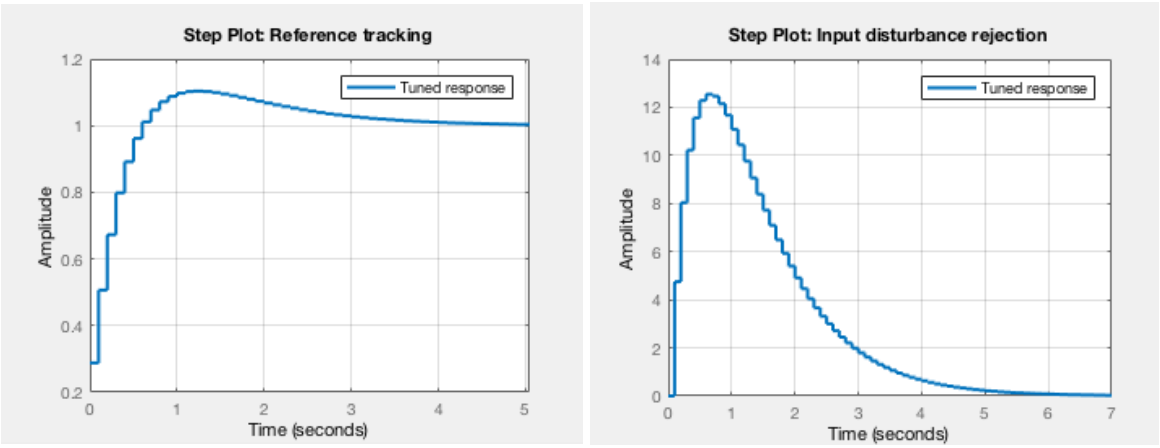
The state-space system takes the throttle as input, has the next speed as an output and the current speed as a state (together with a constant offset). Thus, rearranging equation 4.4 gives the following state-space representation of the simplified system:

$$\begin{bmatrix} speed_{t+\Delta t} \\ const. \end{bmatrix} = \begin{bmatrix} 1 & -\frac{0.7}{0.15} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} speed_t \\ const. \end{bmatrix} + \begin{bmatrix} \frac{1}{0.15} \\ 0 \end{bmatrix} throttle_t \tag{6.1}$$

Additionally, the constraint that $throttle \in [0,1]$ was applied, which limits the maximum possible rise time of the PID controller. This implementation allowed to use the Matlab `pidtool`, which shows the response of a system for different parameters. For an initial estimate, parameters were chosen that assured a rise time of $< 0.5s$ and a maximum overshoot of $< 10\%$.

It is to be noted that this is the rise time of the unconstrained system. It is applicable for relatively small changes in the target speed. For large changes, in particular for the initial acceleration of the vehicle, the rise time is longer due to the limitation in the maximum acceleration of the car (as $throttle \leq 1$).

Figure 6.3 shows the system step response with the chosen controller that fulfills the given requirements. The parameters that were used for initial testing, which achieve the shown response, are $K_p = 0.065, K_i = 0.044, K_d = 0.006$.



(a) Reference tracking performance *(b) Disturbance rejection performance*

Fig. 6.3 Response of tuned PID controller according to the simplified model given by equation 4.4. A trade-off that provides a response time of $< 0.5s$ and an overshoot of $< 10\%$ was chosen for an initial estimate. For this performance, $K_p = 0.065, K_i = 0.044, K_d = 0.006$

For the testing of the speed controller in the simulation, a long straight track was chosen, as it does not require any steering commands. Thus, the steering can be set to a constant of 0. In every test, the vehicle is accelerated from 0 to a given speed. The time taken to reach the target speed and the maximum overshoot were recorded for each trial.

A trial was found to be *good* if it converged to the target speed *quickly* and with an overshoot of at most 10%. A *quick* convergence means that the vehicle accelerates fully until the target speed is reached, and does not have a steady-state error. This naturally takes a few seconds, as an acceleration from 0 to a high speed is physically limited to this constraint.

The initial conditions performed well within the range of $40-50 km/h$, but resulted in a large steady-state error for small speeds and significant oscillation for other speeds. This suggests that it is appropriate to vary the parameters depending on the speed, as proposed by Melder *et al.* [30].

A slowly converging behaviour with a significant steady-state error is a clear indicator that the chosen parameter gains are too small, while a large overshoot indicates that the gains should be reduced. Furthermore, testing showed that the derivative gain K_d introduces oscillation into the system, and it was hence set to $K_d = 0$ (which is equivalent to reducing the PID controller to a PI controller). This is in line with the Matlab simulation, which suggested very small values for K_d.

Based on the findings explained above, a range of parameters were tested for speeds from $10-90km/h$, in steps of $10km/h$. As the vehicle is regulated to a maximum speed of $90km/h$, it was not necessary to test speeds which are higher than that.

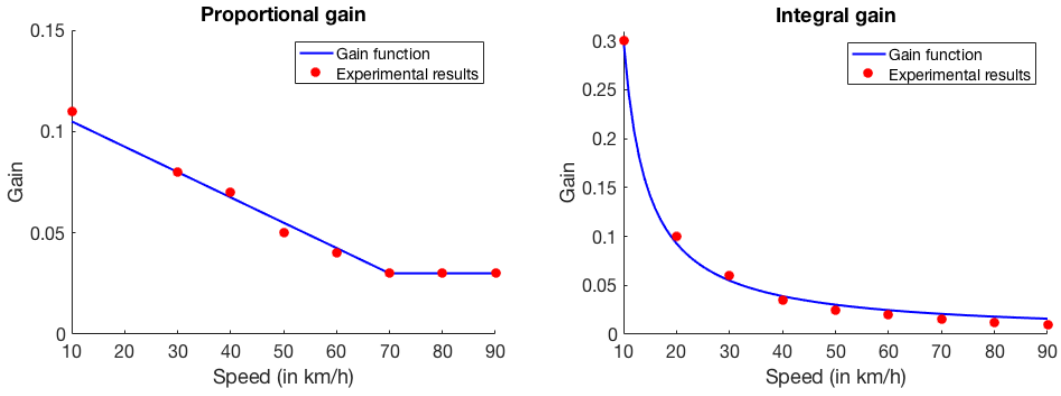
The testing procedure was performed as suggested in [30]: Firstly, the proportional term was determined as the highest gain that does not result in overshoot. Then, the integral gain was adjusted to reduce the steady-state error and improve the convergence to the final speed.

Figure 6.4 shows the *best* controller gains (as defined above) for the proportional gain (fig. 6.4a) and the integral gain (fig. 6.4a). The test points (shown in red) were used to fit a function that determines the gain as a function of the target speed.

The proportional gain showed a clearly linear tendency. Thus, it was implemented as a linearly decreasing function, which however was cut-off at a value of 0.03 in order to avoid the gain becoming arbitrarily low or even negative. The integral gain was fit to a hyperbolic function using the python $curve\_fit$-tool. This accounts for the fact that small speeds require particularly large gains, which is important when driving backwards or after a crash, for example. The resulting formulas for K_p and K_i are shown in equation 6.2. The target speed in m/s is used for the computation. To increase the meaningfulness of figure 6.4, this was scaled up to km/h

$$K_p = max(0.1775 - 0.0045 * target\_speed, 0.03) \tag{6.2}$$

$$K_i = \frac{1.73}{4.63 * target\_speed - 7.11} \tag{6.3}$$



(a) Proprtional gain function *(b) Integral gain function*

Fig. 6.4 Functions for proportional and integral speed controller gains, which were fitted to the data points shown in red. While the function for the proportional gain is linear, the integral gain follows a hyperbolical trend.

6.2.2 Steering Controller

As the steering controller has been implemented using a "carrot following" approach, there is one additional parameter to be determined, which is the tuning of the lookahead. The lookahead of the "carrot" is given as a time in seconds, and thus is a variable of the mean of the current and the target speed. As a basis, a lookahead of $1s$ was chosen.

For the steering control, no linear model was implemented in Matlab, as the next position is a nonlinear function of the throttle and speed inputs, which cannot be easily simplified. The nonlinear functions can be obtained by rearranging equations 4.3 & 4.4 to a similar system as shown in equation 6.1 (but with the position as additional states).

Furthermore, the $carrotfollowing$ approach also means that the target of the PID controller (which is the position relative along a given track) is constantly changing, and hence no similar plot to figure 6.4 can be obtained. Thus, the trial-and-error method for tuning this controller was focused on the analysis of any oscillatory behaviour, and the behaviour of the controller in wide and sharp turns.

As this controller was exclusively tested on straight tracks, no lateral control of the vehicle was necessary, which has been identified as the core purpose of the integral gain of the steering controller in [29]. Indeed, it was found that the integral gain introduced undesired oscillation to the system, and hence for the steering only a PD controller was used.

After extensive testing of multiple gains, the parameters were chosen as $K_p = 1.2$, $K_d = 0.15$. These parameters were found to give the best tradeoff between a stable performance that avoids oscillation, and a responsive behaviour when taking sharp turns. The gains were chosen to be constant as for this controller the *lookahead* is a parameter which varies with the speed. A lookahead of 1s was found to result in a reliable controller.

6.2.3 Evaluation of final PID design

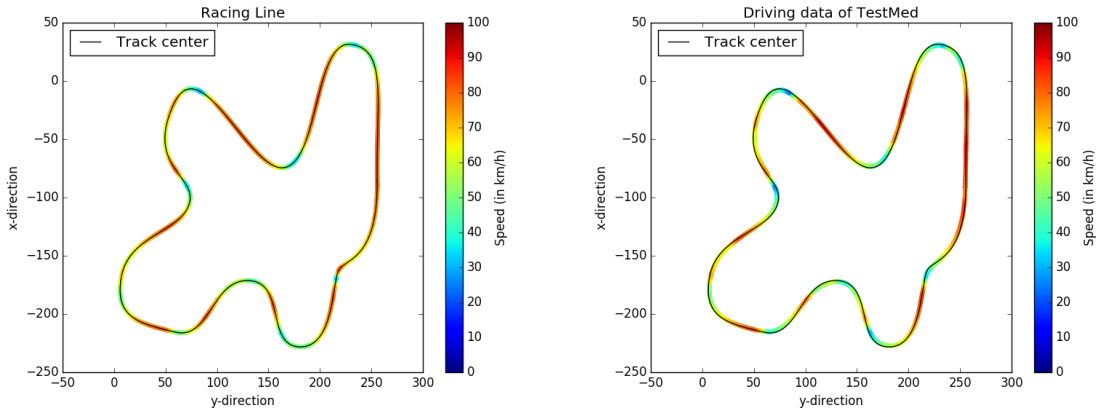
In order to assess how well the PID controller performs, it was tested on the reference model that follows the track center. Thus, the target position is always on the track. The maximum speed was given as $80km/h$ for testing on the *TestLong* and *TestMed* tracks, and $60km/h$ for testing on the *TestShort* track. The speed was linearly reduced depending on the curvature of the track, up to a maximum of 80%.

Figure 6.5 shows an image of the qualitative performance of the controller on the *TestMed* track. From this analysis, the controller seems to accurately follow the given target, however with a small time lag. Additionally, in sharp turns, the controller actually brakes slightly stronger than the model and has a small overshoot on the long straight track elements.

Table 6.1 provides the overall statistics of the vehicle that follows the center of the track. The error is clearly largest for the fast track, as the maximum given speed is never fully reached,

Table 6.1 Evaluation of baseline performance of the tuned PID controller. The average statistics of the speed comparison are ≈ 5% of the target speed, and the mean steering distance is ≈ 0.8m.

Track	TrackLong				TrackMed				TrackShort			
	Speed		**Dist**		**Speed**		**Dist**		**Speed**		**Dist**	
	m	**σ**	**m**	**σ**	**m**	**σ**	**m**	**σ**	**m**	**σ**	**m**	**σ**
Target	68.5	10.9	0	0	63.7	12.7	0	0	50.4	10.1	0	0
Vehicle	65.2	14.3	0.92	0.56	61.3	14.5	0.74	0.43	48.9	13.4	0.73	0.57
Error	3.3	3.4	0.92	0.57	2.4	1.8	0.74	0.43	1.5	3.3	0.73	0.57



(a) Target trajectory *(b) Vehicle trajectory*

Fig. 6.5 Performance of the speed and steering PID controller to follow the center of the track. Qualitatively, the trajectories are very similar, indicating a good baseline performance of the controller.

and reduces if the mean speed is smaller overall. On average, it can be concluded that the PID controller remains within a speed error of $< 5\%$ of the target speed, and the mean distance to the center of the track is approximately equal to $0.8m$. For the further analysis, a distance error of $0.8m$ is assumed as a baseline performance that the models will be compared against.

As the controller gains were chosen to achieve a lag of the system of $\approx 0.5s$, a prediction horizon of the same length was chosen for all personalised models. This approach is aimed at minimising the error introduced by the controller, as it offsets the controller delay by the prediction horizon.

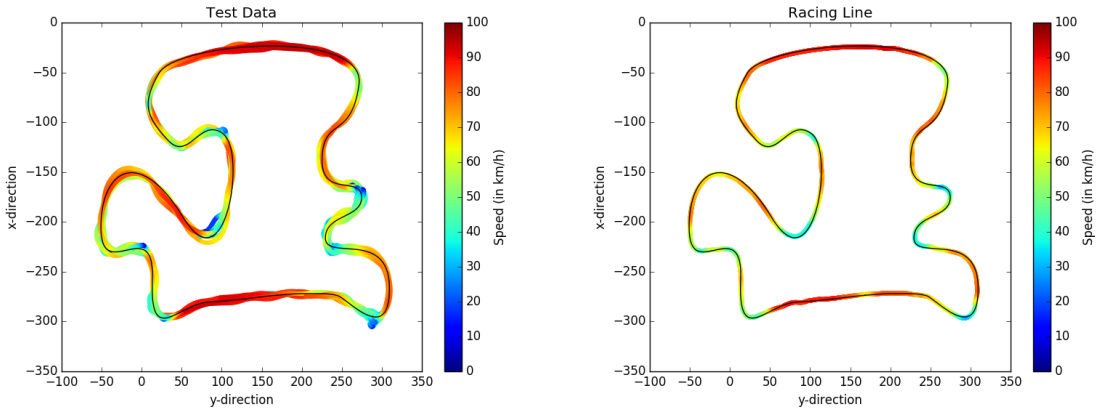
6.3 Model testing

In order to assess the training performance of the different models that were implemented within the scope of the project, the mean squared error between the output of the chosen behavioural model and the true labels of the test data was chosen as a method of comparison. It was found that methods based on the track data only result in much larger errors, especially for validation on other tracks. Methods that feed back the state show a much reduced error, but exhibited limited performance during testing in the simulator.

6.3.1 K-nearest Neighbor Regression

For the k-nearest neighbor regression, no model training is required, as the model directly represents the model. However, the performance of the mean-squared error can still be computed and used to compare the results against the neural network approaches.

Figure 6.6 shows an example of the racing line of the k-nearest neighbor regression on one of the test tracks. As the k-nearest neighbor approach only takes the feature vectors as inputs, this a single racing line can be computed. The mean-squared error of this approach was always in the range of 10-20, which is a consequence of removing all variations along the track.



(a) Test data used for knn fitting *(b) Evaluation of racing line computed by knn*

Fig. 6.6 Comparison of driving data and model performance of the k-nearest neighbor approach. As only the information on the track is taken as inputs, a unique racing line can be defined.

6.3.2 Feed-forward Neural Network

The feed-forward neural networks were tested for two different input feature vectors, one which is just formed of the track information (see figure 6.7a for an example output) and one feature vector which was expanded by the vehicle state, leading to a much lower mean-squared error and a more distinctive training output, as shown in 6.7b.

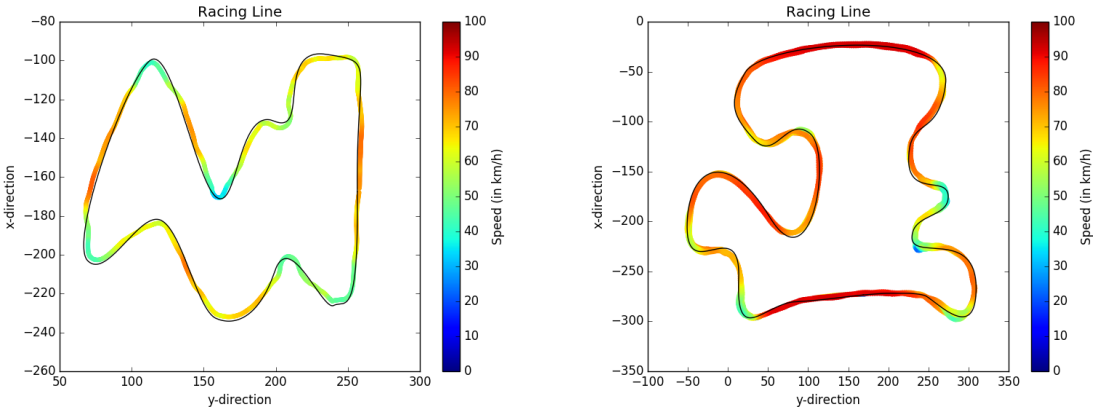
A wide range of possible parameters and combinations were tested for the implementation of the neural network. Firstly, different layer architectures were tested, including single-layer and two-layer implementations with different sizes of each layer. Due to the randomness of each training run it was difficult to determine a unique best design, but out of the combinations of two layers of sizes 5, 10, 20, 50, 100, a combination that showed relatively small errors across all tracks was the layout [20, 5], which was hence chosen for all further implementations. This is effectively a neural network with two hidden layers, where the first layer has a size of 20, and the second layer has a size of 5.

In a second step, different delays and feature vector parameters were determined. Again, no unique best implementation could be determined (for this many more tests would have to be run

and averaged), but the error clearly reduces as a more expressive feature vector is used. A size of the feature vector of 10 was found to lead to small training errors. For the delay, a value of 5 was chosen due to the PID tuning properties as explained in 6.2.

As figure 6.7 suggests, the training error using the expanded feature vector is much smaller, particularly for short delays. This is the case for both the validation and testing accuracies. Validation accuracy is hereby defined as the error of the cross-validation on the same track, whereas testing accuracy was defined as the mean-squared error found when testing the implementation on unknown tracks.

Using final parameters of [8,2,10] for the number of positive segments, negative segments and the segment length, respectively, the mean-squared error for the feature-only neural net was found to be typically around 10-12, while the error for the expanded feature vector was found to be only around 1-5. This however does not guarantee an acutal better performance for online-training, as will be shown later.



(a) Racing line of track-only features *(b) Racing line of expanded feature vector*

Fig. 6.7 For two different tracks, the outputs of the neural network training are shown. The track-only feature vector gives a single line only, while the expanded vector accurately computes the user's trajectories.

6.3.3 Recurrent Neural Network

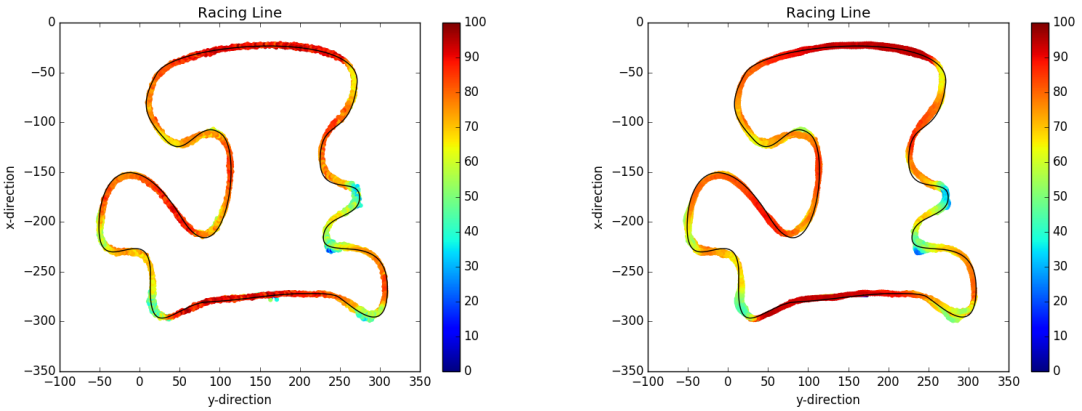
For the recurrent neural network, a similar analysis as described for the feed-forward networks was followed. For two different types of features, namely the absolute labels and the change in the label values between time steps, different recurrent network designs were tested.

Again, single-layer and two-layer structures were evaluated, for state sizes of 5,10 and 20. For reasons of consistency with the results for the neural network, a feature vector size of [8,2,10] for the track data was chosen again.

What was noticeable about this implementation is that it achieved very low testing errors, i.e. errors on new, unknown tracks, for all implemented structures. A significant difference however was found when testing the networks on different tracks. If trained on the Medium and Long

track, and tested on the Short track, the errors were in the range of 4-6, for both validation and testing, while for the other combinations of validation and testing tracks, the testing error was approximately double of the validation error. Nonetheless, the testing error is reduces by a factor of 5 compared to the KNN and NN implementations.

The network structure which yields the best results was found to be a two-layer RNN with a state size of 5 for each layer. Figure 6.8 shows the qualitative results of this approach. Despite the low prediction error, these do not seem very satisfactory, as the data does not seem smooth. Using the differences as inputs to the network can partly reduce this.



(a) Recurrent neural net using absolute values *(b) Recurrent neural net using differences*

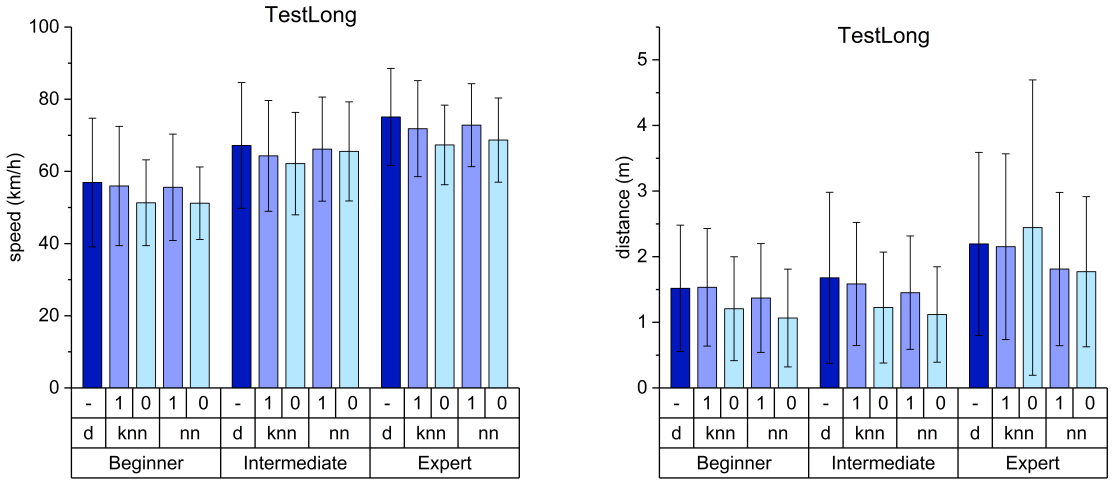
Fig. 6.8 Output of the recurrent neural net using two different input formats, absolute values and differences. The output of the absolute values is not smooth, despite the low mean-squared error. The differences show a much smoother picture.

6.4 Performance Comparison

For a comparison of on-line driving performance in the simulation, KNN and NN were selected as the most promising models within the current set-up. Thus, both approaches were tested against the user. In order to assess the performance of the models, a set of models of each type was trained and then driven on each track for 3 rounds (for *TrackMed* and *TrackShort*) or 2 rounds (for *TrackShort*).

Each of the models was trained using the data from two tracks, so that one track is always unknown to the model. For each of the three drivers, thus 3 models were trained, giving 9 sets of testing data from the simulator for each method that was used (three networks, tested on three tracks). Similar to the evaluation of the driver performances in figure 6.2, a range of statistics was collected from the logs of the autonomous driving sessions. These statistics were then compared to the data of the users. The key findings from the analysis are presented with aid of the following plots.

For all types of analysis conducted within this section, the data from the models is split in the following way: Knn and nn denote the method that was used for training, respectively. The 0



(a) Speed performance on TestLong *(b) Distance performance on TestLong*

Fig. 6.9 Performance of the KNN and NN models compared to the human driver, in known and unknown environments.For the beginner, both the KNN and NN models nearly perfectly replicate the driving performance metrics. It can be noted that generally, the model of the expert keeps a larger distance to the center of the track than the beginner.

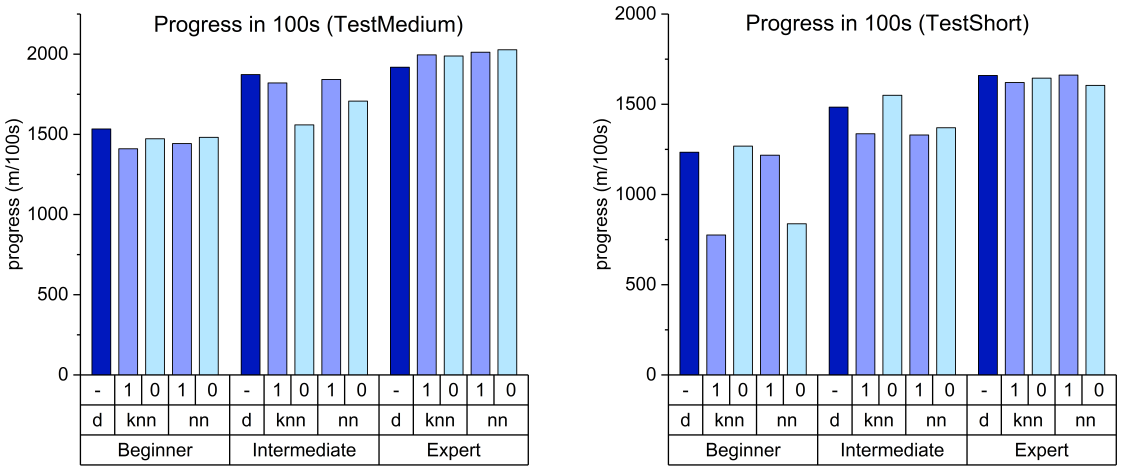
and 1 above the learning method indicates whether the model was trained on the shown track, or whether the track was unknown to the model. 1 means known, and 0 means unknown. For the visualisation, the mean of the performance of all known tracks was taken.

Figure 6.9 compares the statistics of total speed and distance variation along the track, in analogy to figure 6.2. These statistics show that the models which "know" the track out-performed the models to which the track was unknown in all cases. Not only the mean speed is correctly predicted, but also the variance of the speed is comparable to the human example.

Comparing the knn and nn models shows hardly any difference in the variation of the speed, but in this example it appears theat the knn approach is better capable of imitating the distance profile. The knn model, which was trained for the expert driver on the short and medium tracks, and does not know the long track, showed a significantly large variance in the distance. This is an indication that this model may have oscillated or deviated far from the track for some other reason.

Similarly to the analysis above, figure 6.10 showed the total progress over 100 time steps. What is interesting to note is that for this example, all neural network approaches exceed the performance of the expert driver. For the other two drivers, the error is larger, which may be explained by the fact that the other two drivers drove less smoothly, and hence made it more difficult for the model to learn a deterministic trajectory.

Especially for the short track, the difference between the beginner and two of the models is large. Nonetheless, if the data from the driver would be omitted, in most cases it would still be clear which driver the model was trained on.



(a) Total progress of driver and models on TestMed *(b) Progress of driver and models on TestShort*

Fig. 6.10 Progress in 100s of the driver and the models for TestMed (left) and TestShort (right). On the easier Medium track, the expert is even outperformed by the model. On the more difficult short track, the variation between the model and the user is larger, in particular for the beginner. Interestingly, the model that does not know the track outperforms the known model in some cases.

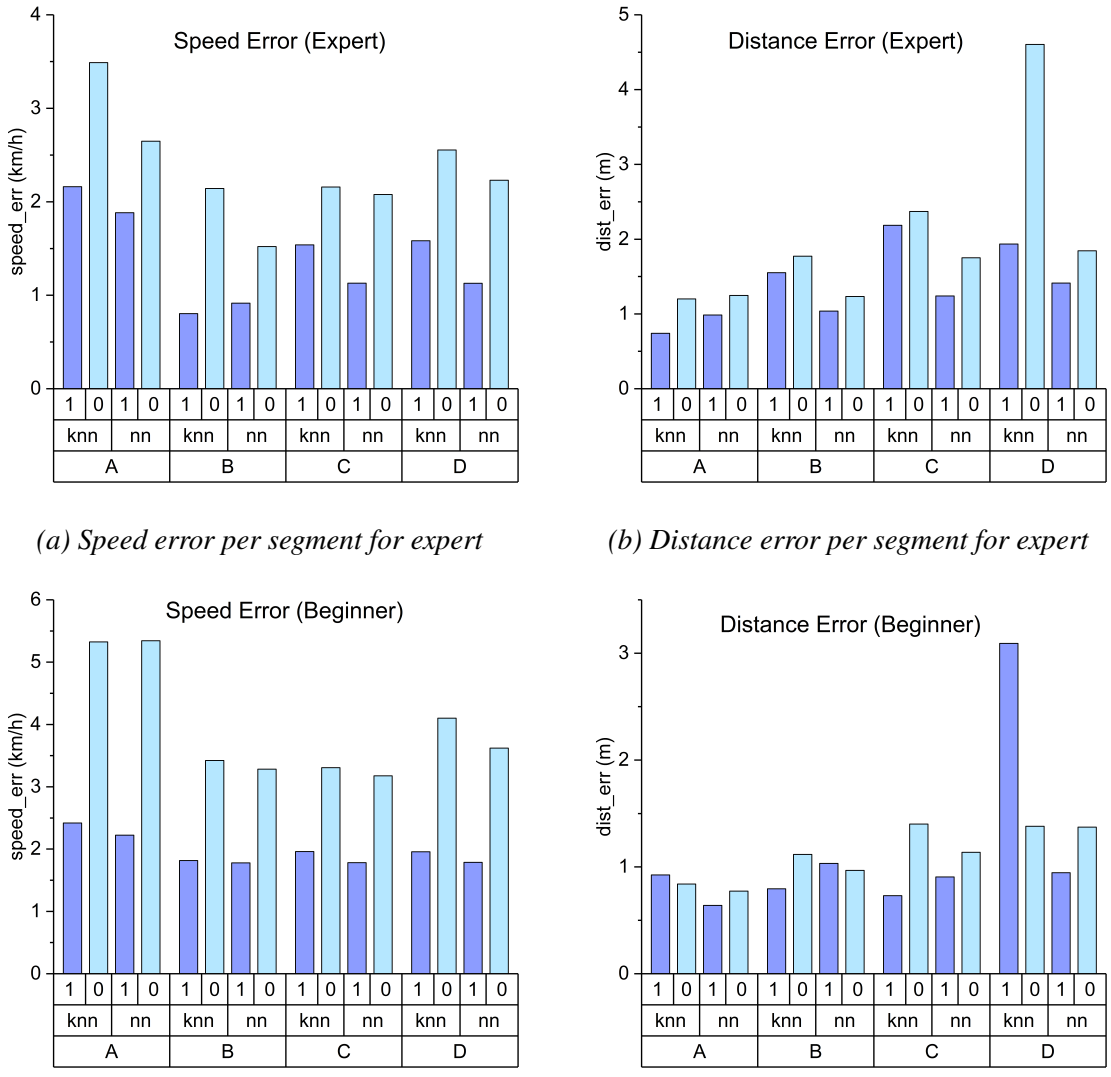
Figure 6.11 shows the mean error along the track for the different difficulty levels, both for the beginner and for the expert. The mean error here is not defined as the difference in the statistics as shown in figure 6.9, but instead as the mean of the absolute differences of the mean trajectories. This measure is stricter than the just taking the difference in the overall statistics, as it also penalises lag in the system.

From the graphs shown in figure 6.11, three dependencies can be observed: Firstly, the error in the speed profile of models which do not know a given track (indicated by green bars) is significantly larger than the error of the models which were trained on that track. This is particularly the case in easy segments. Overall, the speed error however is still within the range of $2-3km/h$, which is comparable to the baseline performance of the PID controller defined in section 6.2.

Secondly, the tendency towards an inversely proportional relationship between the error of the speed and the error of the distance can be observed: For easy segments (A), the model follows a nearly identical trajectory to the users (with respect to the PID baseline performance), even if the track is not known. However, this error increases significantly for more difficult track segments. This result is somewhat expected, as the controller might approach a tight curve too fast, for example, so that a deviation from the track cannot be avoided.

Furthermore, it is interesting to know that this trend is much stronger for the expert. This may be explained by the fact that the expert's mean speed is far larger than the beginner's speed (see figure 6.2. The tendency that the speed error is larger for easier segments may seem counter-intuitive at first sight. However, the speed error is given as an absolute value, not a percentage of the speed of the vehicle. For segments with a higher mean speed, the error may thus also be proportionally higher.

Thirdly, it should be noted that for nearly all cases shown here, the neural network out-performed the knn applications. Even though these are just experimental results, which would need to be verified on a much larger basis to be statistically relevant, this suggests that the neural network is actually better at following the precise trajectory of the user, despite obtaining nearly identical overall performance scores.



(a) Speed error per segment for expert

(b) Distance error per segment for expert

(c) Speed error per segment for beginner

(d) Distance error per segment for expert

Fig. 6.11 Mean trajectory error of the expert driver and the beginner on different types of segments. The models that know the track clearly outperform the the models that do not, and in most cases the error is smaller for the neural network.

Drawing together the findings from the individual graphs shown above, a number of conclusions can be drawn. Firstly, the controllers implemented here are capable of imitating both the speed profile and the lateral deviation profile of a particular user accurately. Generally, models to which a track is new do perform worse than the models which know the respective track, however the comparison is relatively small. This is likely to be due to the fact that even though some parts of the track may be unknown, the models are in fact capable of recognising similar environments.

Comparing the neural networks to k-nearest neighbour regression, the overall performance scores are nearly identical in many cases, or might even support the choice for k-nearest neighbor search. Only when examining the mean element-wise difference along the track, neural networks out-perform the KNN algorithm in most cases.

Finally, the performance of the model to imitate human driving behaviour is also dependent on the difficulty of the track, as it is easier to stay on the target trajectory in straight segments. The error of the lateral deviation from the user's mean trajectory increases for difficult segments.

7 Discussion

In the previous chapter, the testing and results for the individual system components, namely the PID controller and the machine learning models, as well as the complete system were provided. In this chapter, the results are discussed, and areas for future enhancements and applications of the project deliverables are detailed.

7.1 Model Performance

The primary objective of this project was the development of a personalised controller for an autonomous vehicle. Such a controller has been designed and implemented successfully, using three different methods of learning the human driving behaviour. Furthermore, a framework for comparing the human model using 5 different methods of comparison was developed. In the following, the key components which enabled the successful implementation are discussed, and areas for future developments are highlighted.

The controller implementation was based on a two-stage design approach suggested by Cardamone *et al.* [9], which connects a high-level behavioural model to a low-level controller. The high-level behavioural model predicts a personalised target trajectory for the vehicle, which is converted into low-level commands for steering, throttle and brake by the low-level controller.

7.1.1 PID Controller Evaluation

The low-level PID controller was implemented using two PID controllers, one for speed control (i.e. for throttle and brake) and one for steering control. The steering controller implementation is designed using a "carrot-following" approach which computes a sub-goal for the vehicle at every time step. The sub-goal is used to compute the steering output of the vehicle.

The PID controllers have an inherent lag in their response, which was designed to be $\approx 0.5s$. In cases where the targets change significantly, this lag may be larger, as there is a physical limit to the maximum acceleration, deceleration and steering of the vehicle.

The lag which is introduced by the controller is offset by the prediction horizon used for the training of the driver model. The model does not predict the target at the next time step, but instead at $0.5s$ in the future. Thus, the controller is given time to react to the changes in the target, which reduces the error introduced by the PID controller.

In order to increase the response time of the PID controller to large changes in the target, some adjustments have been made to the controller. Hence, if the controller error is very large, the respective signal (throttle, brake or steering) is increased to reduce the error as quickly as possible.

Another source of error of the PID controller are tight curves, where the deviation from the target trajectory is large if the speed is too high. Again, there is a physical limit to how fast a vehicle can take a tight curve. Within the scope of this project, no alterations were made to the target to account for the physical limits, but instead it was assumed that the driver model should reflect these limits and thus control the car appropriately. However, this is one aspect that may be interesting to consider if this project was taken further.

The carrot following approach, which was implemented for the steering control of the vehicle, partly counteracts this effect through oversteering slightly in curves. As the lookahead distance is dependent on the speed of the vehicle, the lookahead is larger if a curve is approached quickly, which leads to stronger oversteering and contributes to a reduction in the error.

Despite these drawbacks of the PID controller, it can be concluded that it is a very robust implementation which reliably followed the given target and was capable of recovering from disturbances, such as getting stuck on the edge of the track, or loss of signal. Visually, the driving style exhibited by the controller appeared smooth and with hardly any oscillation in the steering, which was an important requirement.

This PID controller formed the basis of the online testing of the machine learning methods in the driving simulation, as it translates the high-level targets into the actual commands which drive the car. From testing the controller with the reference targets of following the track center at a given speed, the mean error of the PID controller was found to be a speed of $\approx 2km/h$ and a distance of $\approx 0.7m$. The speed error thus is in the range of $< 5\%$, which is very satisfactory. The distance error of $\approx 0.7m$ is satisfactory considering that the track has a width of $7m$.

7.1.2 Machine Learning Model Evaluation

Three different models were implemented for learning a personalised driving model and predicting the vehicle targets during the driving simulation. These methods are k-nearest neighbor regression, feed-forward neural networks and recurrent neural networks. A discussion of each implementation, and a comparison of the different methods are provided below.

K-nearest Neighbor Regression

K-nearest neighbor is the simplest of the considered approaches, and is based on searching for the most similar control signals within a given set of testing data. The targets are then computed as a weighted sum of the most similar instances in the testing data.

For the k-nearest neighbor model, only the information on the course of the track was used as training data, which means that a unique racing line can be computed that represents the target performance of the model, as it was shown in figure 6.6. The mean squared error of the difference between the racing line and the actual human data can be computed, which gives a score that is comparable to the scores used for the training of the neural networks.

Despite it's simple algorithm, the k-nearest neighbor model showed surprisingly good results, especially in known environments. But even on unknown tracks with some similar characteristics, the KNN controller could sometimes complete with the driver who's model it was using.

One significant drawback of the implementation using KNN is its computational efficiency. For the scope of the analysis carried out here, where the data set used for testing was relatively small, this did not limit the performance of the algorithm significantly, which is why it was classified as an efficient algorithm in table 2.2, but it does significantly reduce the potential to scale the approach up to much larger datasets. Thus, a faster, more efficient approach should be chosen when testing the controller on a larger scale.

Feed-Forward Neural Networks

Feed-forward neural networks are one implementation which is much faster than KNN during on-line testing, as the required computation can be performed off-line during training of the model weights.

Fully-connected neural nets of different layouts were tested on two sets of input data, one of which is based only on the information about the track, similarly to the KNN implementation, and one set which appends the track features by the vehicle state. A preferred layout was selected based on the validation and testing accuracy of the model in the training stage.

The neural net that was only trained on the data of the track, and thus outputs a unique racing line, performed similarly well to the KNN approach. In most situations, it outperformed the KNN by a small margin. As for the KNN approach, a difference can be noted between the performance in seen and unseen situations.

Using the extended state, which included the target speed and the target distance as additional (feed-back) inputs, resulted in significantly lower training errors. This means that during testing, the next state was accurately computed for a set of test data, even if this data was not seen previously. However, the controller using this method showed a poor performance when employed on the track.

Possible reasons for the poor on-line performance is that the controller actually learns undesired relationships between the inputs and outputs and relates the outputs directly back to the input. Despite the high correlation between the current state and the target state, this relation should not be learnt by the net, as it will result in a "lazy" neural network that only changes marginally.

This behaviour could be observed during testing, as the controller started off very slowly and did not react well to changes in the course of the track. These results show that the mean-squared error does not necessarily represent the most appropriate loss function for this application. Furthermore, the network quickly found itself in a new situation, and due to the volatile inputs of speed and target distance computes unreasonable targets. Once it has hit this state, the system was not able to recover.

Recurrent Neural Networks

The recurrent neural network was implemented in order to exploit the time-dependency of human driving and exhibit some human-like variations of how the track is driven. Furthermore, it would allow to learn situations like crash recovery or other unexpected disturbances, as this behaviour cannot be learned by a model that only considers the track data for target computation.

The recurrent neural networks were implemented using long short-term memory units, and use a sequence of past states to predict a new output. Two different sets of model outputs were considered: One set that computes the target speed and the target distance from the track center, and one model which computes the required change in speed and distance. This second approach removes any bias in the system.

The resulting testing accuracies were particularly notable as they showed a smaller or nearly equal error for testing compared to validation, i.e. performed better or equally well on an unknown track compared to a known track. While the mean-squared error for both methods was comparable, the difference method yielded a qualitatively smoother result (see figure 6.8).

However, this model performed similarly to the feed-forward neural net with extended feature vector, and completed the lap slowly and without much similarity to the human driver. In particular, the initial acceleration from 0 to the target speed introduced a large disturbance. This may be explained by the fact that there is only very little data available that is similar to these low-speed states, which makes it difficult to learn the desired target sequence.

Comparison of learning methods

From the tests carried out within the scope of this project, two machine learning methods were identified that are very well capable of imitating the driving style of a particular user: k-nearest neighbor regression and feed-forward neural networks. Both methods use only the information about the track as inputs to the neural network.

Further methods were implemented to explore the potential to learn time-dependencies in the training data. For these methods, the features are expanded by the current state of the vehicle. While these approaches exhibited much better training performance, they drove poorly when tested in the driving simulation. Main reasons for that are likely to be the relatively small number of training data overall, or possibly a cost function which does not accurately represent the priorities of the training.

Further improvements to this method could thus be made by increasing the data available for testing by training a general controller from all the available data, and using the user-specific data just to "fine-tune" the network.

Another method of exploiting the potential of the recurrent neural nets to predict the driving behaviour in unknown environments is to implement a feed-forward structure that returns a

prediction and its confidence bound, and to include the predictions from the recurrent net whenever the confidence of the underlying model is low.

Overall, it can be concluded that feed-forward neural networks are the preferred implementation of the three methods that were analysed in this project. The controller that was implemented using this network exhibits a robust performance, that accurately models the driver's behaviour in known and some unknown situations, and is very computationally efficient in the on-line application.

Probabilistic Models

All the methods considered in the above analysis are deterministic methods, which compute one target trajectory based on the given features. Human driving however is not deterministic; a human driver never takes the same curve in exactly the same way twice. This suggests to use probabilistic methods, which compute the mean and variance of a Gaussian process representing the system behaviour at every point of time.

Two such methods, which were analysed in section 2.2.1, are Gaussian Process Regression (GPR) and Generative Adversarial Imitation Learning (GAIL). The implementation of GAIL is also based on a neural network, and could therefore be an interesting way of developing the driver model further. The measure of confidence could then be used, for example, to determine whether a different model should be used for comparison.

Taking samples of a stochastic model of the human driver should create trajectories that are not always equal, but vary slightly between consecutive laps. This would make the performance more similar to actual human driving.

7.1.3 Comparison to Human Drivers

The framework for assessing similarity to human driving data, which was developed for the analysis carried out in section 6.1.2, consists of five core metrics: The total progress within the first 100s of the race, the mean and standard deviation of the speed along the track, and the mean and standard deviation of the absolute distance to the track center.

For each of the data sets of the human drivers and the driving data of the models, a mean trajectory over the different laps is calculated. This makes the analysis more comprehensive for a deterministic method as used here, as large variations of the human trajectories of consecutive laps are smoothed out. Based on the 5 given metrics, the performance of the models was compared for different types of segments, which were classified by difficulty, and for the different users.

The differences between the Beginner, Intermediate and Expert drivers are accurately replicated by the models, so based on these metrics it is possible to classify the user from which the model was trained. A comparison between the statistics for the target speed and the target distance

showed that the model captures the speeds more easily, while an accurate distance to the center of the track is difficult to model. However, most of the errors observed are within a range of $\pm 1m$, which is reasonable considering the total track width of $7m$.

Another noteworthy fact is that the speed error actually increases more significantly when the controller is implemented on an unknown track, whereas the error in the distance remains similar. However, this may also be due to the fact that the mean distance of the user does not change much on different tracks, while the mean speed between the test tracks is quite significant.

The provided framework makes an attempt of representing the driving style of a user in a comprehensive way, using 5 metrics of comparison for the analysis. The evaluation of the data obtained from different users and models shows that the models can be used to accurately identify the human driver, whereby the model is more accurate if the data about the track is known previously.

This method is entirely based on performance metrics. As one of the main concerns of the research into personalised vehicle controller designs however is to increase the user comfort, an interesting way to continue the research conducted within this project could be to explore new ways to measure the comfort of a driver when being driven by an autonomous controller.

This task could for example be performed by using the eye tracking, skin response or heart rate of the human when exposed to the driving style of "his" controller. In the virtual environment, using a virtual reality headset could make the feeling of actually being inside an autonomously driven vehicle more realistic.

Measuring the imitation performance based on the emotional response of the user is likely to highlight the need for some constraints to be implemented in the controller. This may be due to the fact that a certain amount of steadiness is expected of an autonomous controller, in order to for example not go off-track.

7.2 Simulation Platform Development

In order to provide a test platform for the controller implementations discussed above, a Simulation platform based on an Unreal Engine virtual car racing simulation was developed.

The contributions to the simulation development included the creations of blueprints to enable the sending of the track data to the user, the connection of the hardware inputs to the game and the development of the racing environment to make the simulation more challenging and enjoyable for the test persons.

The main contribution from this project however was the development of a platform that handles the data which is being sent from the simulator and makes it available for use in the controller. This included a range of structures that hold the data (see sections 5.2.2 and 5.2.1) and convert it into a format that is convenient for off-line model training, on-line application in the controller

and data analysis. This platform was made available for anyone taking on related projects in the Personal Robotics Lab.

7.3 Project Applications

The controller developed in the project follows a flexible design that combines machine learning with "classic" control approaches. For the feasibility of this project, the implementation was tested in a virtual driving simulation, but the system design allows some flexibility to expand the application of the controller.

In the simulation, many difficulties occurring in real-world scenarios have been neglected. However, the simulation could gradually be expanded to include more realistic real-world circumstances. For example, additional constraints, such as a maximum speed or a safety distance to the track boundaries could be easily implemented by introducing an additional element between the target computation and the low-level controller, which adjusts the predicted control signals if necessary.

Introducing another vehicle to the simulation, to investigate overtaking behaviour or lane changing, for example, could be of interest particularly if rule-based approaches were to be developed further. In this case it may be interesting to only investigate the learning module of the implementation further, and apply common methods of autonomous vehicle control to convert the target trajectory into the desired motion of the car.

Finally, another exciting application could be to adapt the controller developed here to develop a wheelchair that learns to adapt to different user's driving styles. Similar to real-world driving, safety concerns would mark the highest priority, but the given trajectory could be adapted based on the learned preferences of the user. In this case, the throttle, brake and steering commands would be replaced by the joystick commands, but the same two-level approach can be applied. Similarly to the considerations taken within this project, it is unlikely to be of concern for the user how the joystick is moved precisely, but rather how the wheelchair moves as a result.

8 Conclusion and Further Work

8.1 Achievements

In this project, a two-stage controller for the imitation of human driving behaviour was implemented. The design includes a robust PID controller that generates the low-level control signals of the vehicle, and a high-level behavioural model that generates a target trajectory for the controller from the current state of an autonomous vehicle.

Three different methods of imitiation learning were implemented, namely k-nearest neighbor regression, feed-forward neural nets and recurrent neural nets. The first two approaches proved to be robust and accurately imitated the driving style of a particular user, especially in known environments.

The drawbacks of these approaches is that the simple model cannot learn complex behavioural patters, such as a recovery from a crash. In order to learn such complex behaviours, a recurrent neural network was implemented, which showed low prediction errors during training. However, at the current state the model is not capable of accurately reproducing human driving behaviour, which may be due to a lack of data.

In order to test the implementations described above, further contributions were made to the development of a simulation platform. Thus, a new racing game was implemented and used for testing, and tools were developed to handle the interfacing with the simulation seamlessly.

A comprehensive analysis to assess the similarity of the developed controller to the human demonstration was suggested. It is based on a combination of metrics including the speed and lateral movement of the car, as well as the total progress of the vehicle within a given time.

8.2 Future Work

There are a number of different ways in which this project could be taken further. These may be based on using the simulator platform that has been developed, or on the controller implementation. Three interesting ways of developing the ideas presented in this report further may include the implementation of a probabilistic model, the development of an emotional model to assess the driver's comfort, or the adaption of the approach followed here in a new environment.

Probabilistic models allow the determination of a confidence interval of the predictions which are made by the model to imitate human driving behaviour. This information could be used in order to improve predictions in a number of ways, and potentially develop a model that is capable of imitating complex behavioural pattern.

Within this project, a quantitative analysis based on statistical measures of the difference between the trajectory of the user and the model was carried out. However, as a human model should finally be able to provide more comfort for the users, methods of measuring the emotional response of a user could be explored.

Finally, it would be exciting to see whether this approach could be deployed in a different environment, such as for the control of an autonomous wheelchair, by adapting the PID controller to control a joystick and developing the behavioural model further.

References

[1] D. Muoio, "19 companies racing to put self-driving cars on the road by 2021," *Business Insider UK*, October 2016. [Online; last accessed 28/01/2016].

[2] M. Kuderer, S. Gulati, and W. Burgard, "Learning driving styles for autonomous vehicles from demonstration," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2641–2646, 2015. ID: 1.

[3] E. de Gelder, I. Cara, J. Uittenbogaard, L. Kroon, S. van Iersel, and J. Hogema, "Towards personalised automated driving: Prediction of preferred acc behaviour based on manual driving," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1211–1216, June 2016.

[4] B. Shi, L. Xu, J. Hu, Y. Tang, H. Jiang, W. Meng, and H. Liu, "Evaluating driving styles by normalizing driving behavior based on personalized driver modeling," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 12, pp. 1502–1508, 2015.

[5] L. Xu, J. Hu, H. Jiang, and W. Meng, "Establishing style-oriented driver models by imitating human driving behaviors," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 5, pp. 2522–2530, 2015.

[6] X. Fu, H. H. Mosebach, D. Gamrad, K. Lemmer, and D. Söffker, "Modeling and implementation of cognitive-based supervision and assistance," in *2009 Mathmod Conference Vienna*, 2009.

[7] T. Padir, "Towards personalized smart wheelchairs: Lessons learned from discovery interviews," in *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 5016–5019, 2015. ID: 1.

[8] T. Carlson and Y. Demiris, "Collaborative control for a robotic wheelchair: Evaluation of performance, attention, and workload," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 42, no. 3, pp. 876–888, 2012. ID: 1.

[9] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Learning drivers for torcs through imitation using supervised methods," in *2009 IEEE Symposium on Computational Intelligence and Games*, pp. 148–155, 2009. ID: 1.

[10] N. V. Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber, "Robust player imitation using multiobjective evolution," in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pp. 652–659, IEEE, 2009.

[11] J. Togelius, R. D. Nardi, and S. M. Lucas, "Making racing fun through player modeling and track evolution," 2006.

[12] J. Togelius, R. D. Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pp. 252–259, IEEE, 2007.

[13] H. J. Williams, "Real-life applications: The business of making learning fun," September 2005. [Online; accessed 28/01/2017].

[14] M. Rodrigues, A. McGordon, G. Gest, and J. Marco, "Adaptive tactical behaviour planner for autonomous ground vehicle," in *2016 UKACC 11th International Conference on Control (CONTROL)*, pp. 1–8, Aug 2016.

[15] B. Gorman, C. Thurau, C. Bauckhage, and M. Humphrys, "Bayesian imitation of human behavior in interactive computer games," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 1, pp. 1244–1247, 2006. ID: 1.

[16] scikit learn, "Nearest neighbors." http://scikit-learn.org/stable/modules/neighbors.html. [Online; accessed 11/06/2017].

[17] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[18] H. B. Demuth, M. H. Beale, O. De Jess, and M. T. Hagan, *Neural Network Design*. USA: Martin Hagan, 2nd ed., 2014.

[19] C. Olah, "Understanding lstm networks." https://colah.github.io/posts/2015-08-Understanding-LSTMs/, August 27 2015. [Online; accessed 11/06/2017].

[20] J. Ko, D. J. Klein, D. Fox, and D. Haehnel, "Gaussian processes and reinforcement learning for identification and control of an autonomous blimp," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 742–747, 2007. ID: 1.

[21] D. Loiacono, A. Prete, P. L. Lanzi, and L. Cardamone, "Learning to overtake in torcs using simple reinforcement learning," in *IEEE Congress on Evolutionary Computation*, pp. 1–8, July 2010.

[22] J. Ho and S. Ermon, "Generative adversarial imitation learning," *CoRR*, vol. abs/1606.03476, 2016.

[23] A. Kuefler, J. Morton, T. A. Wheeler, and M. J. Kochenderfer, "Imitating driver behavior with generative adversarial networks," *CoRR*, vol. abs/1701.06699, 2017.

[24] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels, *Evolution of Reactive Rules in Multi Player Computer Games Based on Imitation*, pp. 744–755. Advances in Natural Computation: First International Conference, ICNC 2005, Changsha, China, August 27-29, 2005, Proceedings, Part II, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[25] C. E. Rasmussen, "Gaussian processes for machine learning," 2006.

[26] T. Georgiou and Y. Demiris, "Predicting car states through learned models of vehicle dynamics and user behaviours," in *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1240–1245, 2015. ID: 1.

[27] P. Falcone, F. Borrelli, J. Asgari, H. E. Tseng, and D. Hrovat, "Predictive active steering control for autonomous vehicle systems," *IEEE Transactions on Control Systems Technology*, vol. 15, no. 3, pp. 566–580, 2007. ID: 1.

[28] X. Qian, A. de La Fortelle, and F. Moutarde, "A hierarchical model predictive control framework for on-road formation control of autonomous vehicles," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, pp. 376–381, 2016. ID: 1.

[29] J. Mange, S. Pace, and A. Dunn, "Optimization of pid controller parameters for automated ground vehicle control on dynamic terrain," in *Proceedings of the International Conference on Scientific Computing (CSC)*, p. 3, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

[30] N. Melder and S. Tomlinson, "Racing vehicle control systems using pid controllers," *COLLECTED WISDOM OF GAME AI PROFESSIONALS*, p. 491.

[31] P. Sujit, S. Saripalli, and J. Sousa, "An evaluation of uav path following algorithms," in *Control Conference (ECC), 2013 European*, pp. 3332–3337, IEEE, 2013.

[32] A. L. Rankin, C. D. Crane III, and D. G. Armstrong II, "Evaluating a pid, pure pursuit, and weighted steering controller for an autonomous land vehicle," in *Intelligent Systems & Advanced Manufacturing*, pp. 1–12, International Society for Optics and Photonics, 1998.

[33] G. Reynoso-Meza, J. Sanchis, X. Blasco, and R. Z. Freire, "Evolutionary multi-objective optimisation with preferences for multivariable pi controller tuning," *Expert Systems with Applications*, vol. 51, pp. 120 – 133, 2016.

[34] S. Tomlinson and N. Melder, "Representing and driving a race track for ai controlled vehicles," *COLLECTED WISDOM OF GAME AI PROFESSIONALS*, p. 481.

[35] T. Georgiou and Y. Demiris, "Personalised track design in car racing games," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, Sept 2016.

[36] Epic Games, "Bluepring time attacker racer." https://docs.unrealengine.com/latest/INT/Videos/PLZlv_N0_O1gYdhCvvMKGpCF6LCgBz9XeS/. [Online; accessed 11/06/2017].

[37] E. Calabi, P. J. Olver, and A. Tannenbaum, "Affine geometry, curve flows, and invariant numerical approximations," *Advances in Mathematics*, vol. 124, no. 1, pp. 154 – 196, 1996.

[38] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[39] E. Hallström, "How to build a recurrent neural network in tensorflow." https://medium.com/@erikhallstrm/using-the-dropout-api-in-tensorflow-2b2e6561dfeb. [Online; accessed 17/06/2017].

[40] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *CoRR*, vol. abs/1506.00019, 2015.

A User Guide

This guide aims to support anyone who might continue the project to set up the software and understand the structure of the provided code easily. It further gives a rough overview of the Unreal Engine Simulation at its current state. The simulation is still in development so some of the details given here might change in the future.

All code is saved in the Gitlab repositories of the Personal Robotics Lab. The simulation is saved in the *UnrealPRLCarSimulator* repository. The controller code is stored in the *DriveLMe* folder of the *CarSim-RosNodes* repository and can be cloned from there.

Software Requirements

As explained in section 3 of the report, the system consists of two elements: The Unreal Engine Car Racing Simulation (which is also being used for other projects) and the code for the implementation and testing of the autonomous driving controller (the core of this project).

All controller code has been implemented in Python and requires common libraries such as `numpy`, `scipy` and `matplotlib` as well as the learning libraries `scikit-learn` and `tensorflow`. All of these are widely available and can be installed using `pip`. If Gaussian Process Regression is to be investigated further, it is additionally necessary to install `GPflow` which has been used in the initial attempts made in this project.

In order to set up a connection to the game engine for data collection and testing, it is further necessary to run ROS on an Ubuntu machine. This machine however does not need to be the same as the computer from which the controller code is run (see below for more details). An introduction to ROS can be found here: http://wiki.ros.org/ROS/Introduction

Unreal Engine Simulation

The driving simulation used for all results in this project was started from scratch using the online Blueprint Time Attack Racer tutorial (https://docs.unrealengine.com/latest/INT/Videos/PLZlv_N0_O1gYdhCvvMKGpCF6LCgBz9XeS/). This tutorial consists of 15 videos and is very useful for anyone with little or no experience in game development. It explains the use of blueprints as interfaces for a visual implementation of functions and shows which blueprints exercise which functionalities. The following explanations assume familiarity with the concepts explained in the tutorial series and basic knowledge of ROS.

In order to enable the communication with ROS, a plug-in has been installed which implements functions to allow publishing and subscribing to information sent via ROS. For this, a number of blueprints were amended to allow for the communication.

A number of ROS topics have been defined to transmit the necessary information between the controller and the simulation: `/CarSimUnreal` contains all available information being sent by the simulator, while `/CarInput` contains all information being sent by the controller to autonomously control the vehicle. Additionally, there are two commands that are used to send track information: `/TrackCmds` is a float number that is used to trigger the publishing of the track spline (see section 3 for details), which is then sent under the topic `/TrackData` or `/TrackDataInterpolated`.

The new blueprint *UDP_Publisher* handles the publishing of `/CarSimUnreal` and needs to be adapted if additional outputs from the simulation are required. The current inputs and outputs that are being sent and received can be found in tables 3.1 and 3.2.

The *TrackGenerator* blueprint (still in development) is used to automatically create new tracks. The event graph of the blueprint handles the sending of track information according to the the `/TrackCmds`: If 0 is sent, the function will publish `/TrackData` containing the spline points and tangents for a calculation of the track. Otherwise, track data interpolated by the specified value (in cm) will be published as `/TrackDataInterpolated`, i.e. if the value 100.0 is sent, a sequence of points on the spline will be published which are 1m apart from each other.

For this project, a number of maps have been created. These maps can be found in the *VehicleAdvBP/Maps* folder and should be duplicated in order to create new race tracks. It is important to note that each map has its own instance of the *Vehicle Controller*, *Tracker* and *Track Generator* and thus all setting of these blueprint instances should be checked when a new map is used.

Step-by-Step Guide

In order to run the system, some preliminary steps are required. These assume that the *CarSimRosNodes* repository has been cloned to a catkin workspace and the game is installed on an available windows machine. Furthermore, all path names in scripts in the *DriveLMe* folder need to be corrected to an adequate local path.

1. Appropriately source the ROS setup files on the Ubuntu machine.
2. Launch a UDP Rosbridge from the terminal using:
   `roslaunch rosbridge_server rosbridge_udp.launch`
3. Start the **Epic Games Launcher** and open the **PRLCarSim** Project.
4. Assure that the **Ros IP** in the tracker on the used map is set to the IP of the machine that runs the ROS.

Now, the user can perform one of the following tasks:

Manual Driving

In order to record the manual driving data, the following actions are required:

1. Open the `testSendReceiveForUnreal.py` script.
2. Assure that the track name is set to the name of the map.
3. If track data is to be collected, set `getSpline` to *True* and set `trackDist` to the desired value (as explained above).
4. Run the script and input the test ID (such as the name of the driver or any other identification name) and type *n* to identify the manual driving mode.
5. Start the simulation and drive. The data will be stored in a folder inside *testData* labelled by the test ID. The file has the format: *[trackname]_[idx]_manualLog.txt*

Neural Network Training

The script `net_regress.py` runs the training session for the neural networks. Check that all global variables (i.e. the network parameters) are set to the desired values and run the session. The `user` (i.e. the test ID in manual driving), `TRACKNAME` and `TEST_IDX` parameters are used to identify the correct datasets for training. The neural net model is stored in a subfolder in the user's folder together with plots of the test data and the resulting model performance.

Autonomous Driving

Autonomous driving is performed similarly to manual driving. The data inputs now must also contain the correct learning method to be used as well as the parameters that are relevant for the chosen model (Refer to section 5.4.1 for details). These are:

- **Center Following**: For the reference implementation of following the track center, the target speed (in m/s) as well as the maximum "slowdown", i.e. the maximum percentage of speed reduction in the steepest curve, are required.
- **K-nearest Neighbor Search**: As this approach uses the entire training data set as reference, a list of tracks and indices of the manual logs is required. Further, the parameters for curvature computation as well as the number of neighbors are given.
- **Neural Network**: For this implementation it is sufficient to give the folder name of the network to be used. All parameters are stored in the model and will be loaded at start-up.

B rFactor 2 Variables and Matlab Code

Relevant output variables

The following table lists all relevant outputs of the rFactor 2 simulation used for an analysis of the initial controller application. For the analysis, only the motion in the horizontal plane is considered (i.e. the x and z directions). All movement in vertical direction is omitted from the analysis as it was found to be negligible. The global orientation of the vehicle was inferred from the local velocities by applying the rotation detailed in the next section.

Table B.1 Relevant outputs from rFactor data set in the horizontal plane

Output	Variable Name	Range
Throttle	Telemetry_mFilteredThrottle	$[0;1]$
Brake	Telemetry_mFilteredBrake	$[0;1]$
Steering	Telemetry_mFilteredSteering	$[-1;1]$
Position	Telemetry_mPos_x Telemetry_mPos_z	$(X,Z)m$
Local Acceleration	Telemetry_mLocalAccel_x Telemetry_mLocalAccel_z	$(X,Z)m/s^2$
Local Velocity	Telemetry_mLocalVel_x Telemetry_mLocalVel_z	$(X,Z)m/s$
Speed	Telemetry_Speed_KPH	$\geq 0km/h$

Calculating the global vehicle orientation

This code was developed together with Theodosis Georgiou based on the work he has completed on the simulator.

Listing B.1 Steering Control

```
function steer = getSteer(orientation, targetOrientation)
% [breaklines=true, caption={Computation of the global vehicle orientation
    }]
function orientation = calc_angle(modelStruct)

%% Rotation matrix
rightVector = [modelStruct.Telemetry_mOri_0_x; ...
        modelStruct.Telemetry_mOri_1_x; modelStruct.Telemetry_mOri_2_x];
upVector= [modelStruct.Telemetry_mOri_0_y; ...
        modelStruct.Telemetry_mOri_1_y; modelStruct.Telemetry_mOri_2_y];
forwardVector= [-modelStruct.Telemetry_mOri_0_z; ...
```

```
        modelStruct.Telemetry_mOri_1_z; -modelStruct.Telemetry_mOri_2_z];

% Local velocity vectors
vz = modelStruct.Telemetry_mLocalVel_z;
vx = modelStruct.Telemetry_mLocalVel_x;

%% Global Variables
global_vX = vz .* rightVector(3,:) + vx .* rightVector(1,:);
global_vZ = vz .* forwardVector(3,:) + vx .* forwardVector(1,:);

%car orientation
orientation=atan2(global_vX,global_vZ);
end
```

Linear controller implementation

Listing B.2 Steering Control

```
function steer = getSteer(orientation, targetOrientation)
% This function computes the steering required to reach target orientation
% Inputs:
% - orientation: current angle of the car in the global plane
% - targetOrientation: estimated car angle in the global plane after deltaT
% Output:
% - steer: Value in [-1; 1] corresponding to steering by -45 to 45 degrees

angle = targetOrientation-orientation;  % Obtain angle difference

if(angle > pi)                          % Handle cases close to -pi/pi
    angle = angle - 2*pi;
elseif(angle < -pi)
    angle = angle + 2*pi;
end

steer = 4*angle/pi;        % Normalise to value in [-1; 1] (1 == 45 degrees)
if steer>0                 % Cut off any value > 1 to maximum steering
    min(steer,1);
elseif steer <0
    max(steer,-1);
end
end
```

Listing B.3 Throttle Control

```
function throttle = getAccel(targetspeed, actualspeed, ts)
% Function computes required throttle for reaching target speed
% Inputs:
% - targetspeed,actualspeed: speed at time t, estimated speed at t+ts (m/s)
```

```
% - ts: Time between actualspeed and targetspeed (delta t)
% Output:
% - throttle: Value in [0,1], where 1 means full pressing of the throttle

acceleration = (targetspeed - actualspeed)/ts;
throttle = 0.3+0.09*(acceleration);    % linear model
throttle = max(min(throttle,1),0);    % limit between 0 and 1
end
```

Listing B.4 Brake Control

```
function brake = getBrake(targetspeed, actualspeed, ts)
% Function computes required braking for reaching target speed
% Inputs:
% - targetspeed,actualspeed: speed at time t, estimated speed at t+ts (m/s)
% - ts: Time between actualspeed and targetspeed (delta t)
% Output:
% - brake: Value in [0,1], where 1 means full pressing of the brake

acceleration = (targetspeed - actualspeed)/ts;
brake = -0.06*(acceleration); % linear model
brake = max(min(brake,1),0);    % limit btw 0 and 1
end
```

C Unreal Engine Blueprint Screenshots

Gear changing logic

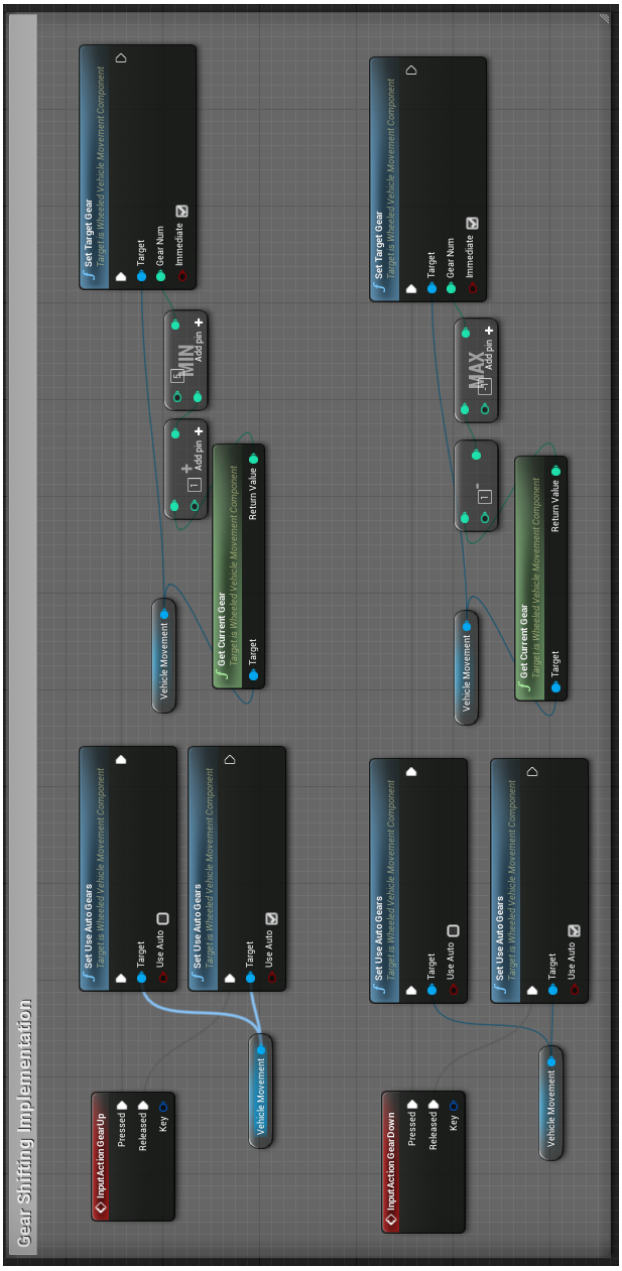


Fig. C.1

ROS advertiser and publisher of spline points

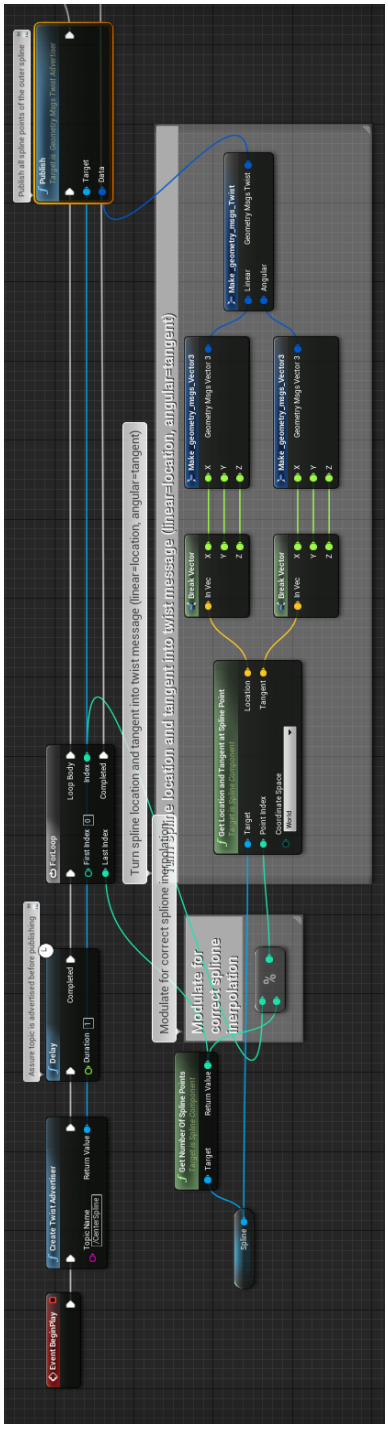


Fig. C.2

Motion control

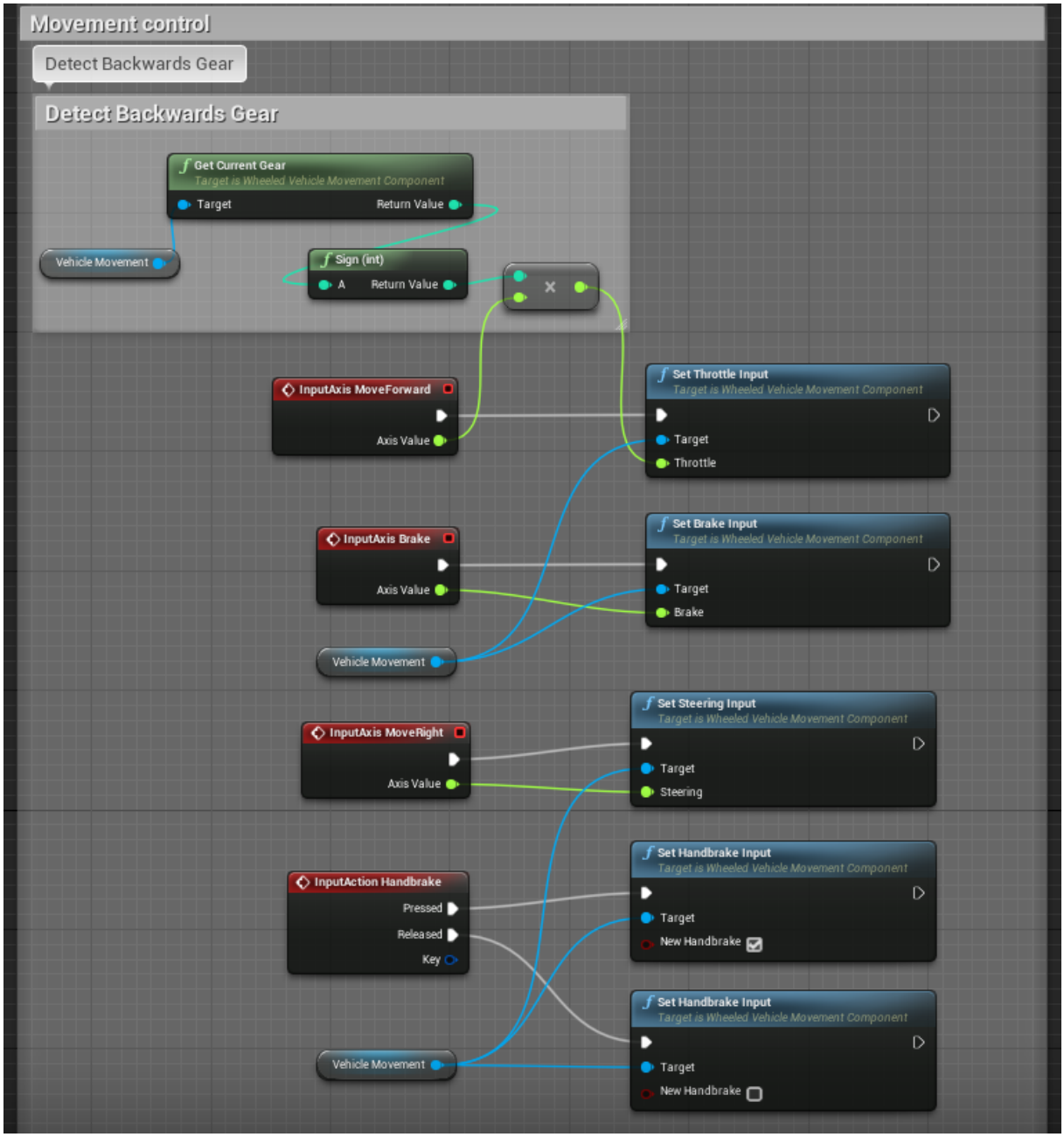


Fig. C.3

D Performance Metrics of Human Driving Data

Table D.1 Driving statistics of 5 test drivers on 3 test tracks (see fig 6.1). Mean speed and total progress in 100s contain most information on how "good" a driver is, while the distance to the track center carries most information on how curves are approached.

Track	**Speed** (km/h)		**Acceleration** (m/s^2)		**Distance to center** (m)		**Progress**
	Mean	Deviation	Mean	Deviation	Mean	Deviation	($m/100s$)
Beginner							
Short	47.044	16.748	3.584	4.691	1.267	0.934	1232.45
Med	53.499	14.345	1.986	2.739	1.352	0.935	1532.5
Long	57.933	19.613	2.632	3.642	1.47	1.054	1677.87
Intermediate 1							
Short	49.147	15.868	3.596	4.408	1.597	0.952	1335.91
Med	67.927	13.756	2.595	3.162	1.559	0.973	1749.56
Long	61.837	18.226	2.328	3.051	1.799	1.047	1649
Intermediate 2							
Short	54.357	15.707	3.675	4.837	1.502	1.077	1483.13
Med	69.692	13.288	2.491	3.516	1.759	2.131	1870.56
Long	66.96	17.445	2.45	3.772	1.678	1.300	1853.07
Experienced 1							
Short	58.15	13.734	2.856	4.076	2.074	1.729	1557.89
Med	73.319	9.360	2.023	2.935	1.579	1.197	1994.34
Long	67.386	15.658	2.665	4.506	2.199	1.609	1804.03
Experienced 2							
Short	62.136	13.298	3.063	4.605	1.964	1.208	1658.09
Med	75.043	11.597	2.093	2.931	2.078	1.269	1915.92
Long	74.67	13.871	1.992	2.934	2.191	1.394	1957.11