University of the Aegean

School of Engineering

Department of Financial and Management Engineering

Autonomous Vehicles: Basic Concepts in Motion Control and Visual Perception

Georgios V. Tepteris

Supervisor: Prof. Ioannis Minis

Committee Members: Assistant Prof. Vasileios Zeimpekis

Assistant Prof. Vasileios Koutras

Chios, June 2020

To my Family

Acknowledgments

First and foremost, I would like to pay my special regards to my supervisor Professor Ioannis Minis for giving me the opportunity to write this exciting and interesting thesis. From the beginning he encouraged me not only for the completion of the thesis but has also given advises and shared interesting thoughts through all the years of my studies. Furthermore, I would like to thank him for the continuous support, and for his patience. His guidance helped me in all the time to the writing of this thesis.

Besides my advisor, I would like to express my deep gratitude to Dr. Vasileios Zeimpekis, Assistant Professor of the University of the Aegean for our interesting discussions and his advises. Moreover, I am grateful to Dr. Vasileios Koutras, Assistant Professor of the University of the Aegean for his insightful comments and encouragement. Also, I want to thank the members of DeOPSys Lab of Department of Financial and Management Engineering.

Finally, I am indebted to all my professors during my studies who have contributed to my educational and personal development.

Last but not the least, I would like to thank my family: my parents and my brothers for supporting me spiritually throughout writing this thesis and my life in general. I am also grateful to my partner who supported me throughout this venture.

Abstract

This thesis focuses on two important aspects of autonomous vehicles: Control of the vehicle's longitudinal and lateral motion, and recognition of the objects in the vehicle's drivable space. In terms of vehicle control, we first drilled down to the existing kinematic and dynamical models used to describe the motion of the vehicle. Subsequently, we developed appropriate lateral and longitudinal controllers to be used in the CARLA vehicle simulator. For longitudinal control we developed a PID controller, while for lateral control we developed a Stanley controller, and we implemented both in the Python language. The controllers generated brake, throttle and accelerator commands to drive the vehicle dynamical model in the CARLA environment. For longitudinal control, we tuned the gains of the PID controller to achieve satisfactory performance. The results indicated that a PD controller appropriately tuned resulted in good performance. In terms of visual perception, we drilled down on aspects of existing related methods and techniques and outlined how they can be used to achieve this very complex task. Subsequently, we developed Python routines to process the semantic segmentation output of a deep neural network and perform relatively simple tasks, such as ground plane estimation, lane marking identification, object recognition within predefined bounding boxes and distance estimation between the recognized objects and the vehicle. Possibly the most significant contribution of this thesis is the systematic presentation of existing fundamental knowledge in the above two areas in a way that one can build upon to develop new, improved concepts for vehicle control and visual perception.

Περίληψη

Η παρούσα διπλωματική εργασία εστιάζεται σε δύο σημαντικές πτυχές των αυτόνομων οχημάτων: α) Στον έλεγχο του οχήματος σε διαμήκη και πλευρική κίνηση και β) στην αναγνώριση του πεδίου οδήγησης του οχήματος.

Όσον αφορά στον έλεγχο του οχήματος, αρχικά εξετάσαμε τα υπάρχοντα κινητικά και δυναμικά μοντέλα που χρησιμοποιήθηκαν για να περιγραφεί η κίνηση του οχήματος. Στη συνέχεια, αναπτύξαμε κατάλληλους πλευρικούς και διαμήκεις κατευθυντές που χρησιμοποιήθηκαν στον προσομοιωτή οχημάτων CARLA. Για το διαμήκη έλεγχο αναπτύξαμε κατευθυντή PID, ενώ για τον πλευρικό έλεγχο αναπτύξαμε κατευθυντή Stanley και υλοποιήσαμε και τους δύο κατευθυντές σε γλώσσα Python. Οι τελευταίοι υπολογίζουν τις εντολές επιτάχυνσης και επιβράδυνσης (θέση των πεντάλ γκάζι και φρένου) καθώς και κατεύθυνσης (γωνία τιμονιού) που δίδονται στο δυναμικό μοντέλο του οχήματος στο περιβάλλον του CARLA. Για τον διαμήκη κατευθυντή, επιλέξαμε τις παραμέτρους PID για να επιτευχθεί ικανοποιητική απόδοση. Σύμφωνα με τα αποτελέσματα, ο κατευθυντής PD με κατάλληλες παραμέτρους οδηγεί σε καλή απόδοση.

Όσον αφορά στην οπτική αναγνώριση του πεδίου οδήγησης, εξετάσαμε τις πτυχές βασικών υπαρχόντων σχετικών μεθόδων και τεχνικών και σκιαγραφήσαμε πώς μπορούν να χρησιμοποιηθούν για την επίτευξη αυτού του πολύπλοκου στόχου. Στη συνέχεια, αναπτύξαμε συναρτήσεις Python για την επεξεργασία της εξόδου από υφιστάμενο νευρωνικό δίκτυο εννοιολογικής κατηγοριοποίησης (semantic segmentation) και για να υλοποιήσουμε σχετικά απλές δράσεις, όπως η εκτίμηση του επιπέδου του εδάφους, η αναγνώριση σήμανσης της λωρίδας, η αναγνώριση αντικειμένων εντός προδιαγεγραμμένων πλαισίων οριοθέτησης και η εκτίμηση της απόστασης μεταξύ των αντικειμένων αυτών και του οχήματος. Πιθανόν η πιο σημαντική συνεισφορά της διπλωματικής αυτής εργασίας είναι η συστηματοποίηση του υφιστάμενου γνωστικού υπόβαθρου στους παραπάνω δύο τομείς ώστε να χρησιμοποιείται στην ανάπτυξη νέων βελτιωμένων προσεγγίσεων για τον έλεγχο του οχήματος και την οπτική αναγνώριση πεδίου.

Table of Contents

Table of Figures

List of Tables

Chapter 1 Introduction

In recent years, autonomous vehicles are becoming progressively important in the automotive industry and oftentimes monopolize the interest of the media. In this area, significant research is being conducted in both academia and industry. Furthermore, several companies are testing models of autonomous vehicles on the road such us Tesla[1] and Waymo[2].

It has been ascertained that the launch of autonomous vehicles will have many positive effects. For example, accidents may decrease due to sophisticated safety systems embedded in the vehicles and the absence of human errors. Passenger convenience, less ecological impact and the more efficient use of road infrastructure are also considerable advantages. However, there is a long way to go for the commercial launch of autonomous vehicles, and much more research, development and testing are needed.

Active areas of research and development include dynamic motion planning, control of the motion of the vehicle, visual perception, safety concepts in urban areas, real time decisions and others. The present thesis focuses on two such areas of self-driving vehicles; dynamic control and visual perception. Both areas are complex, highly promising, and fit the background of the students in the Financial and Management Engineering Department of the University of the Aegean.

In dynamic vehicle control, the fundamentals include a solid analysis of the vehicle longitudinal and lateral dynamics and the use of relevant approaches in control systems. In visual perception, deep neural networks are used to detect the road infrastructure such as lanes, obstacles, traffic signs and traffic signals, as well as moving objects, such as other vehicles, bicycles, pedestrians.

[1] https://waymo.com/
[2] https://www.tesla.com/autopilot

In order to gain the particular necessary knowledge in these areas, we followed relevant Coursera courses[3] in:

- Introduction to Self-Driving Cars (University of Toronto)
- State Estimation and Localization for Self-Driving Cars (University of Toronto)
- Visual Perception for Self-Driving Cars (University of Toronto)
- Machine learning (Stanford University)

We also used the open source autonomous vehicle simulator CARLA to conduct relevant experiments (Dosovitskiy, et al., 2017).

This thesis

- presents and explains the basics of vehicle dynamics
- uses established control approaches to control these dynamics both in the longitudinal and lateral sense. Simulation experiments are used to tune the parameters of the longitudinal speed controller and draw interesting conclusions
- presents and explains the basics of visual perception
- develops software using relevant available functions to perform visual perception tasks, based on CARLA visual inputs.

This thesis contributes to the understanding of the complex technological background, upon which the vehicle control and visual perception tasks are based. Furthermore, it explores aspects of both areas and indicates how simulation and the related software development environment may be used to develop and test important concepts towards contributions in these areas.

The structure of the remainder of the thesis is as follows: Chapter 2 provides an introduction to self-driving hardware and software architectures, discusses safety assurance and overviews the need of sensors and computing power. Chapter 3 introduces the two types of vehicle control, lateral and longitudinal control. Furthermore, it presents the implementation of an appropriate controller and the

[3] https://www.coursera.org/

tuning of its parameters through simulation. In Chapter 4, convolutional neural networks are described, and a case study is presented, involving three major tasks: drivable space estimation, lane estimation and object detection. The conclusions of the work are given in Chapter 5.

Chapter 2 The need of autonomous vehicles

2.1 The impact of autonomous vehicles

The basic role of technology is to facilitate humanity with the necessary precondition for respect to the environment. For several decades now, as technology evolves and the planet's population has increased, the need to create new innovative ventures is becoming more pressing. One of them is the creation of autonomous vehicles. To achieve this, science and technology have to collaborate.

The presence of driverless vehicles will solve many everyday difficulties as well as will improve universal problems such as global pollution. This will be achieved by removing congestion, especially in urban centers, reducing accidents and pollution. Of course, such an action requires hard and precise work as well as many experiments and studies.

The software of autonomous vehicle must be precise, fast and perfectly programmed to grasp the space in which it moves and to prevent or avoid the intentions of other vehicles. It is clear that improper design of such a vehicle can be fatal for its user and also for vehicle users.

Impact on traffic congestion

It is worth considering the hours everyone spends driving. A typical example is the Americans who, according to the American Society of Civil Engineers, spend more than 6.9 billion hours per year on the road (American Society of Civil Engineers, 2017). Also, various university studies report the so-called “phantom traffic jam”, in which drivers create a "stop-and-go" traffic, regardless of lane changes, merges or other disruptions.

The key to resolve the above phenomenon is the communication between the self-driving vehicles and their surroundings in order to be able to identify the ideal route. Needless to say, the reduction of traffic congestion will lead to reduction of accidents and traffic deaths. For the above reasons, the pace control of the autonomous vehicles will smooth the flow of traffic for all cars.

Impact on mobility

One of the main issues that arise during driving is the movement of highways and the speed at which the vehicle will reach its destination as quickly as possible. Of course, arrival requires the safety of all drivers. Research from time to time on the launch of the autonomous vehicle in market has shown that driverless vehicles could increase lane capacity (vehicles per lane per hour) on highways.

To begin with, the ability to monitor every single second surrounding traffic and respond with finely tuned braking and acceleration adjustments should enable autonomous vehicles to travel safely at higher speeds and with reduced headway between each vehicle. Furthermore, autonomous vehicles will greatly facilitate people who have difficulty to drive or cannot drive, such as seniors and people with disabilities.

Impact on safety

One notable reason that urges researchers to study more and everything related to self-driving cars is the plethora of accidents. Worldwide, 1.35 million people are killed each year in car crashes and the majority of these accidents are due to fatal human errors while driving (e.g. driver inattention, malaise, alcohol, cell phone use, speed limit transgression) (WHO, 2018). A cooperative autonomous driving environment will be incapable of preventing all accidents.

Autonomous vehicles will reduce traffic accidents, deaths and injuries especially those that result from driver distraction. However, there are some risk factors that deserve attention. First of all, passengers of AVs (Autonomous Vehicles) should not become overtly sure of themselves and neglect to take elementary precautions, such as fastening their seat belt. In addition to, pedestrians should cross streets carefully and not have the mentality that AVs pose no threat to them. Last but not least, driverless vehicles could be targeted by hackers or terrorists since their networks could become a breeding ground for the distribution of ransom ware or malware. Therefore, a foolproof system is needed which will be immune to such attacks.

Impact on the environment

As the years go by, nature gives us more and more signs. Climate change is now a fact and phenomena such as the greenhouse effect and deforestation need to be controlled. Science and technology must collaborate and allow the planet to breathe.

Autonomous vehicle technology could reduce fuel emission by accelerating and decelerating more smoothly than a human driver. Further improvements could be had from reducing distance between vehicles and increasing roadway capacity enabling lower peak speeds (improving fuel economy) but higher effective speeds (improving travel time). Thus, by reducing exhaust emissions, both time and money savings are achieved.

2.2 Fundamental concepts of autonomous vehicles

The Driving Task consists of three sub-tasks. The first one is perception, which pertains to the necessary mapping and comprehension of key elements in the driving environment, including the road, road signs, traffic signals, other vehicles, pedestrians and other elements that constitute the world around the vehicle. In addition to recognizing all moving objects of relevance, there is a need to predict their future state. The second sub-task is motion planning, that allows the vehicle to move from point A to point B successfully. For example, to travel from home to the university, the route the vehicle will follow should be decided. Finally, the third sub task, is to control the vehicle itself, that is control the vehicle's position and speed through breaking, steering and acceleration decisions. All these three sub-tasks of the driving task are necessary in order to drive a vehicle successfully.

Another important aspect of vehicle automation is the Operational Design Domain or ODD, which defines the operating conditions the vehicles are designed to operate. ODD encompasses characteristics such as time of day, roadways and other components upon which the self-driving car performance depends. It also affects the level of autonomy of the vehicle.

Level 0: A human driver has full control of the vehicle, that is all driving sub-tasks: perception, planning and control.

Level 1: The vehicle's systems help the driver through lateral or longitudinal control tasks. A characteristic example of longitudinal control is the Adaptive Cruise Control or ACC function, which controls the speed of the vehicle at a predefined level. An example of lateral control is lane keeping assistance, which warns the driver when s/he changes lane inadvertently.

Level 2: The vehicle's systems exercise lateral and longitudinal control under special driving scenarios. However, the driver is always needed to control the vehicle.

Level 3: The vehicle's systems can operate partly Object and Event Detection. In this case, on condition of malfunction, the driver should take control of the vehicle. In contrast to Level 2, in Level 3 and under certain scenarios consideration of the driver is not necessary.

Level 4: This is one step before full autonomy: The system may handle an emergency situation when the driver doesn't intercede quickly. In some cases, the system notifies the driver to take control of the vehicle.

Level 5: Full autonomy, under which ODD is unlimited; that is the system can handle any situation of the traffic environment. (U.S. Department of Transportation, 2017)

2.3 Hardware and software architecture of autonomous vehicles

To achieve vehicle autonomy above Level 2 up to Level 5, there are significant interventions required to any vehicle both in hardware and software. In this Section, we provide a descriptive introduction to both aspects. Later in this Thesis we drill down on technical details as needed.

2.3.1 Major hardware of autonomous vehicles[4]

There are two hardware categories in autonomous vehicles: Sensors and Computing Hardware. The latter perform certain important computations based on the inputs provided by the former.

[4] Inspired by (Bussemaker, 2014)

Sensors

The sensors sense and measure objects in the environment of the vehicle, or detect the alterations of dynamical objects in this environment.

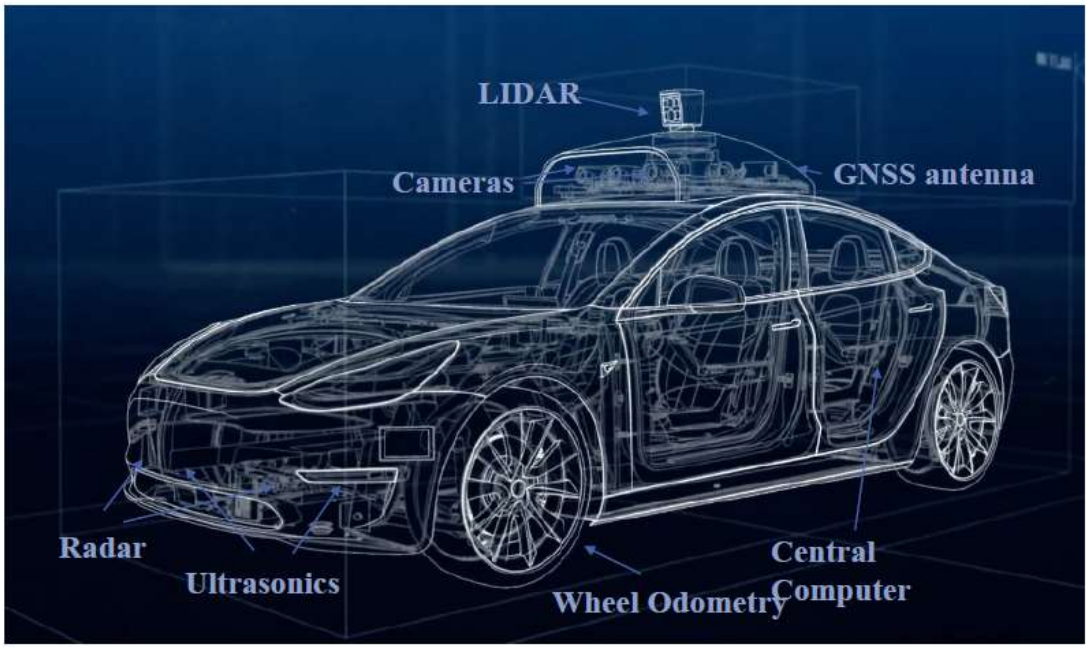

Figure 2.1 Typical sensors of an autonomous vehicle (Coursera(2019a))

Typical sensors of an autonomous vehicle are shown in Fig. 2.1. The camera is the eyes of the autonomous vehicle and is positioned on the upper part of the vehicle. For example, on the roof (as in most autonomous vehicles) or in the middle of the windshield. The Lidar sensor is placed at the highest level of the vehicle, since the plot of the environment around the vehicle that Lidar produces should not be obstructed by parts of the vehicle. The Radar sensor is placed at the front bumper and tracks the position of other vehicles, which are nearby. The Ultrasonic sensors are placed sideways in the front bumper and the rear bumper. They are used to measure objects that are located very close to the vehicle, such us curbs or other vehicles which are parked. The Wheel odometry sensor measures the number of wheels turns and estimates the distance travelled by the vehicle. This sensor can combine with other sensors, such as the GNSS/IMU antenna, in order to improve GPS information. The GNSS/IMU sensor receives signals from the GPS satellites and combines the measurements from tachometers, altimeters and gyroscope for accurate positioning of the autonomous vehicle.

The above sensors are classified into two types: Exteroceptive (or extero) sensors that are used to identify objects in the vehicle's environment, and proprioceptive sensors, which sense ego properties, that is, properties of the vehicle of reference. These two types are further examined below.

Exteroceptive Sensors

This sensor type includes cameras, that is passive, light-collecting sensors that capture detailed visual information of the environment (see for example Fig. 2.2). A camera uses three metrics: resolution, dynamic range and field of view. The resolution metric is the number of pixels that create an image; the higher number of pixels, the better the image quality. The dynamic range of a camera relates to the difference between the darkest and the lightest tones in an image. For autonomous vehicles high dynamic range is critical, due to the significant changes in lighting status, for example, at night. The field of view is defined-by the horizontal and vertical angular extent that is visible to the camera and can be varied through the lens selection and zoom.

Figure 2.2 A typical camera with three camera lens (Autonomous vehicle international (2019))

The other important extero sensor is LIDAR (Light Detection and Ranging), which implements a surveying method that measures the distance of objects by emitting light beams, receiving the reflected return, and performing the necessary calculations

involving the related time difference and the speed of light. It usually comprises a spinning item that sends laser beams. The output is a three-dimensional point cloud map (3-D Map), which is an approximation of the geometry of the environment. LIDAR is not affected by environmental lighting. One important aspect of LIDAR is the number of laser beams used, that is 8, 16, 32 or 64 beams. A second important aspect is the number of points per second the sensor may collect and process. The faster processing, the more accurate the 3D map. One more component is the rotation rate. The higher the rate, the faster the 3D marks are updated. A fourth aspect is the detection range and is guided by the power of the light source. Finally, like the camera, LIDAR is characterized by the field of view, which is the angular expansion visible to it.

Figure 2.3 presents an example of a 3D map. Figure 2.4 presents examples of LIDAR sensors.

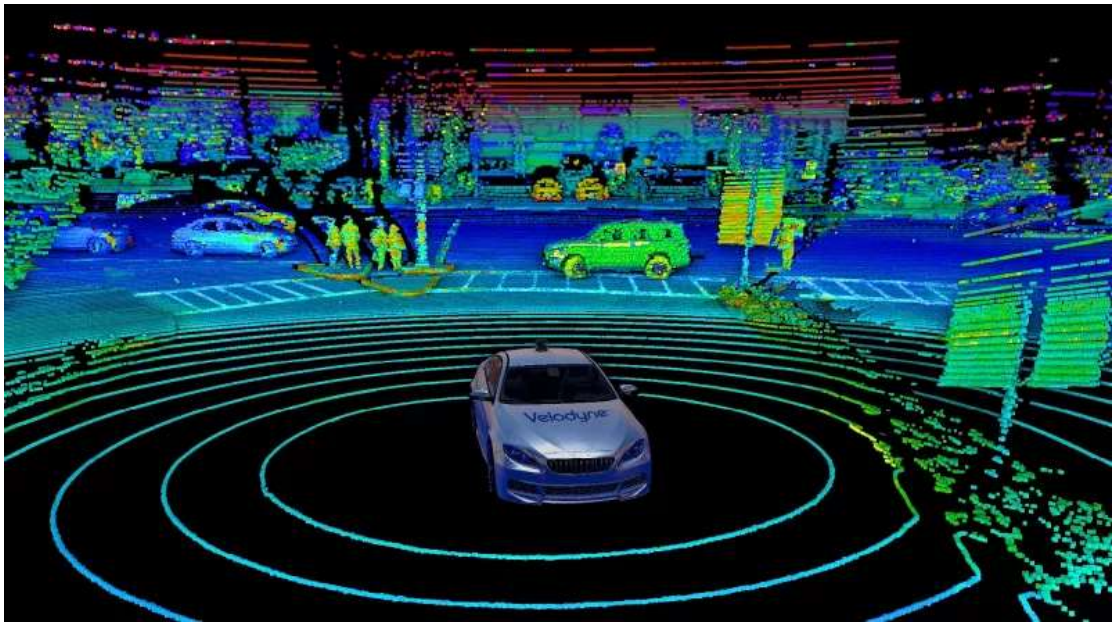

Figure 2.3 A 3D map example (Coursera (2019a))

Figure 2.4 Typical LIDAR sensors and examples of mounting on various vehicles (Coursera (2019a))

A third important sensor is the RADAR (Radio Detection and Ranging) that can detect larger objects in a shorter distance than LIDAR – see Fig. 2.5. RADAR has the distinct advantage of not been affected by rain. The characteristics of the RADAR sensor include detection range, field of view, position and speed measurement accuracy. Also, RADAR may cover a wide angular field in short range, a narrow field a longer range.

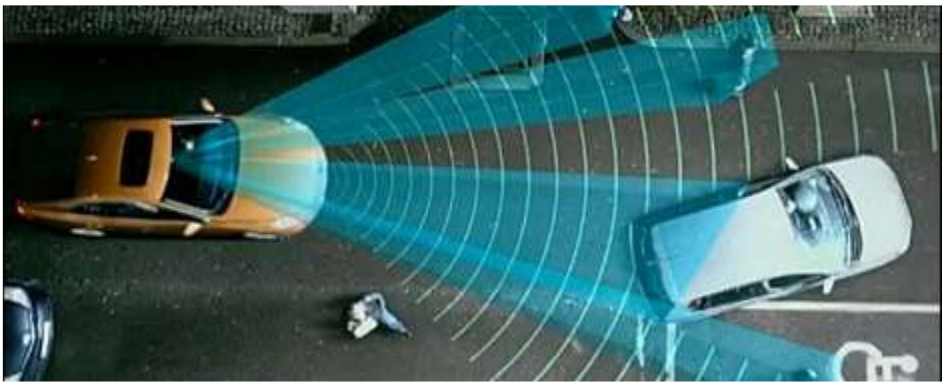

Figure 2.5 Radar detects other objects (Coursera (2019a))

The final extero sensor is the ultrasonic sensor or SONAR (Sound Navigation and Ranging), which measures the distance between the ego car and the car ahead or behind or next to it, using sound waves. SONAR is used in short range applications; for example, in parking scenarios in which the reference vehicle needs to manoeuvre close to other vehicles (see Fig. 2.6). Its characteristics include maximum range and the detection field of view.

Figure 2.6 Self-driving car detects a free parking space with sonar (Coursera (2019a))

Proprioceptive Sensors

The second category/ type of sensors includes proprioceptive sensors, which sense ego properties, that is, properties of the reference vehicle. A key sensor is the GNSS (Global Navigation Satellite Systems), such as GPS, which measures the ego position, velocity and sometimes heading.

Another important proprioceptive sensor is the IMU (Inertial Measurement Unit) – see Fig. 2.7. It measures the angular accelerations of the ego vehicle. The IMU synthesizes the outputs of three gyroscopes and three accelerometers to monitor the motion of the vehicle. The gyroscope is very accurate but generates noisy measurements. It computes the angular rotation rate, actually taken from three gyroscopes, that is the angular speed of the body structure relative to an inertial frame of reference. The accelerometer measures body accelerations which are important in

estimating the accurate position of the ego-car. All these sensor measurements estimate the 3D orientation of the car, that is the vehicle's, a most necessary variable for vehicle control.

The last type of proprioceptive sensors includes the wheel odometry sensors, which measure the wheel rotation rate. This sensor records the frequency of rotation, which is used in estimating the speed and the rate of change of the autonomous vehicle's heading. It is also the sensor that calculates the range of kilometres of any vehicle.

Figure 2.7 Typical IMU sensors (Coursera (2019a))

Computing hardware

The central computer is shielded in the car body and it is connected with the sensors. It analyses the sensor outputs in order to operate steering, acceleration and brake. For example, Nvidia's central computer is called Drive PX and Intel's computer is called Mobileye's EyeQ. Computing hardware performs serial and parallel computations. For example, the LIDAR and image processing used for segmentation mapping and object detection would use GPUs, FPGAs, and custom ASICs, which are special hardware used to perform complex computations, such as image processing.

2.3.2 Software architecture of autonomous vehicles[5]

The software architecture of autonomous vehicles runs on the computing hardware overviewed above. The main software modules of autonomous driving perform a) environment perception, b) environment mapping, c) motion planning, d) vehicle control and e) system supervision. All are essential for full vehicle autonomy.

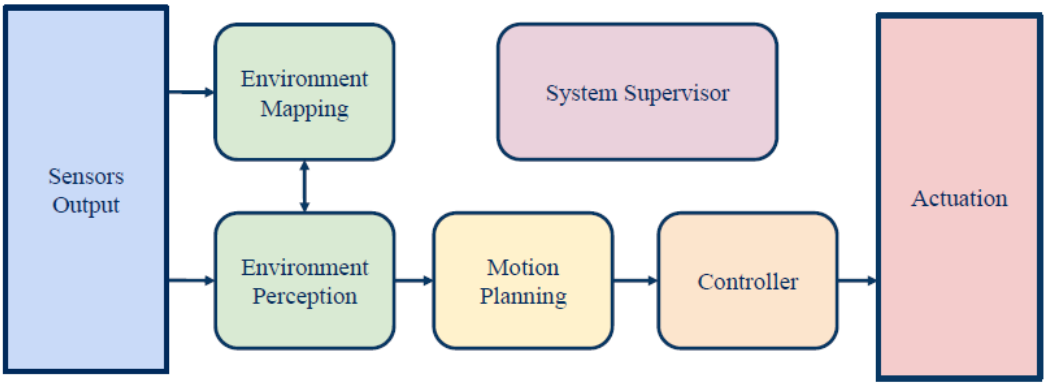

Figure 2.8 Software architecture of autonomous vehicles (Coursera, 2019a)

Modules (a) and (b) are used to recognize the environment around the vehicle based on the outputs of the appropriate sensors. Module (a) recognizes where the autonomous vehicle is in space by detecting and categorizing necessary objects in the environment. These objects could be other vehicles, pedestrians, road marking, road signs, bikes, motorbikes and the road; that is everything that affects the driving behaviour of a vehicle.

Module (b) constructs necessary maps to locate objects in the immediate environment of the ego car. These maps are used for motion planning, tracking, collision prevention, etc. There are three types of such maps:

- The occupancy grid map: It is used to capture objects in the vehicle's environment. The latter is mapped based on grid cells; a probability is related to each grid cell to indicate whether the cell is occupied or not.

[5] Inspired by Coursera (2019a)

Certain objects are excluded from the grid map, such as the drivable surface, and dynamic objects

- The localization map: It is used to track the exact position of the ego vehicle within its environment by combining LIDAR or camera data with other sensor data
- The detailed road map: It is a fusion of existing data and incoming static data regarding the ego vehicle's environment. Such important data include line markings and road signs, which support motion planning.

Modules (a) and (b) interact; the static data about the environment of the ego vehicle that are necessary to update the detailed road map are provided by the perception module. The detailed road map is, in turn, used by the perception module in its object detection task.

The motion planning module (c) uses the outputs of modules (a) and (b) to ensure that the vehicle moves from its origin to its destination in a safe and effective way. Motion planning comprises the following:

- long term planning of the entire driving task between the origin and destination along the predefined route; the latter comprises the on the best path synthesized by road segments
- behavioural planning, which accepts the above long-term plan as input and constructs a short term plan, that is a set of actions and manoeuvres to be followed along the path of the long term plan. For example, a lane change may be allowed considering the behaviour of the surrounding vehicles
- Immediate or reactive planning: It defines the route and the speed profile to be followed, satisfying all the constraints of the ego vehicle's environment. The inputs used by this planning level include the output of the behavioural planner, the occupancy grid, etc.

The controller (module d) uses as input the path of the motion planning module (c) and executes it in the best possible way by controlling four variables: steering angle,

brake pedal position, throttle pedal position, and gear settings. The module performs two control actions:

- Longitudinal control: It regulated the break and throttle pedal positions, and the gear settings to achieve the desired speed set by motion planning
- Lateral control: It regulates the steering angle to follow the planned path also set by motion planning

The system supervisor (module e) ensures that all systems are functioning properly and notifies the driver of any malfunctions or issues. It includes

- The hardware supervisor that monitors data from all ego vehicle's hardware. The data of each monitored piece of hardware are examined whether they fall within the expected range for the environment in which the ego vehicle operates
- The software supervisor monitors all software outputs and detects any inappropriate values, actions, or dissimilarities between outputs of different software modules

2.4 The planning hierarchy

In this Section, we discuss the decision making that is necessary to be carried out in an autonomous vehicle. There are three types of related decisions a) long-term planning b) short-term planning and c) immediate-term planning (Urmson, et al., 2008). The first type concerns long-term planning and is responsible for the autonomous vehicle to travel from the one point (origin) to another (destination). For example, from Athens to Thessaloniki or from the university to home. It results in a high-level plan for the entire driving task, including which roads to travel, which lanes, which turns to make, etc.

The second type of decisions concerns short-term planning, including when the autonomous vehicle must change lane with safety, when it should execute a right turn, etc. These decisions involve control and trajectory planning and answer questions of the type, how do I follow a lane on this curved road? What steering input should I apply? Should I accelerate or brake? If so, by how much.

To clarify the above two types of decisions, consider the following example. Let's assume that the vehicle will enter an intersection, and then turn left at this intersection (see Figs. 2.9 and 2.10). This task falls in the short-term planning type as the related decisions involve lane changes and stopping locations. Given that the intersection is controlled by traffic lights, the vehicle should decide if it should change lanes in order to turn left. Then, as it approaches the intersection it should choose to slow down smoothly in order to respect passenger comfort.

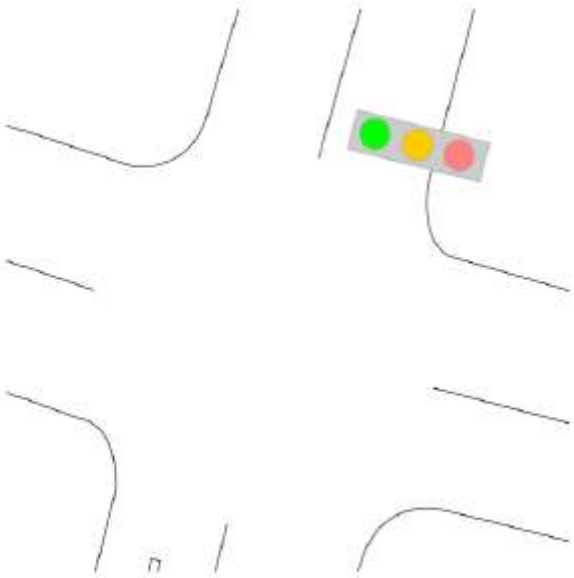

Figure 2.9 Turning left at an intersection: The vehicle should also stop just before the pedestrian crossing. (Coursera 2019a)

Moreover, there situations that may arise along the way. The decisions to address these situations fall into the immediate decision type and require safe reactions from the planning system. Object detection and event detection and response play a very critical role here. What if another vehicle pulls into the turn lane in front? The ego vehicle would need to stop earlier to make room for the other vehicle. What if the stop lines weren't marked? In this case, the ego vehicle would have to approximately

judge where the implied stop line is and stop before the pedestrian crossing. What if there were other vehicles behind the ego vehicle or even stalled in the intersection? How does the decision to execute a left turn change based on the many possible scenarios that can rapidly arise in normal driving? The final result is an enormous list of possible decisions that must be taken and evaluations that have to be made on different timescales. In every scenario, there is a need to have a reliable set of choices that can be evaluated in real time and be updated as new information comes in. Furthermore, even a seemingly simple driving scenario requires three or four levels of decisions and must then be executed with careful vehicle control. This example illustrated the constant stream of decisions needed for motion planning.

To represent the immediate type decisions in the software of the autonomous vehicle involves two methods. The first method is reactive planning. In this method we define sets of rules that only consider the current state and not future predictions. These rules take into account the current state of the ego vehicle and other objects in the environment and produce immediate actions. An example of such rules would be, if there is a pedestrian crossing the road, stop, or if the speed limit changes, you have to adjust your speed to match it. In both examples, the system observes current events and makes a decision based on immediately available information.

Another method of planning is predictive planning.

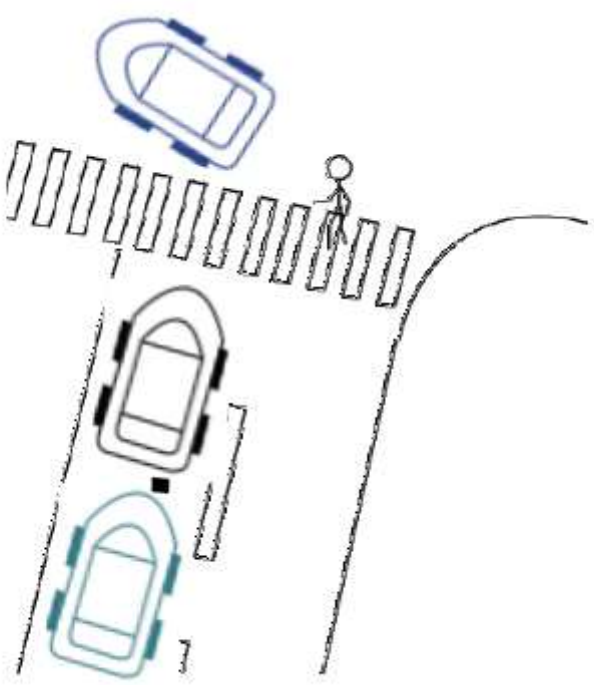

Figure 2.10 Intersection, decelerate smoothly to the stop line (Coursera 2019a)

In predictive planning we make predictions on how other agents in the environment will move over time, including vehicles and pedestrians. This prediction information affects all involved decisions. This is a more natural way to generate decisions and relates closely to how humans operate vehicles. An example of rules in predictive planning is if the car has stopped for the last 10 seconds, then it's probably going to stay stopped for the next few seconds. So, maybe there is a way that the ego vehicle can move past it safely. Or consider a pedestrian jaywalking: the pedestrian will enter the ego vehicle lane by the time it gets close to it. Then, the vehicle needs to slow down and give the pedestrian a chance to cross the road ahead of the ego vehicle. The system predicts where other objects on the road will be in the future before it makes the needed decisions. Accurate predictions of the actions of the other actors in the environment adds a significant layer of complexity to the perception tasks. Nonetheless, the scenarios of a safe handling of self-driving vehicles, are expanded by predictive planning.

2.5 Safety concerns

Prior to the deployment of autonomous vehicles, Governments have been active in setting safety rules to prevent accidents from hardware or software malfunctions. Furthermore, the parties involved in developing autonomous vehicles are developing and testing safety systems. Testing may involve creating traffic scenarios and testing them in simulators and in a real environment. Developers should be considering all the safety standards set by the National Highway Traffic Safety Administration (NHTSA).

In this Section, we overview this critical topic.

2.5.1 Examples of safety incidents involving autonomous vehicles

Unfortunately, over the last five years several accidents have taken place during this initial and limited deployment period of autonomous vehicles. A characteristic example involved the autonomous vehicle of Waymo (Google), which in the Spring of 2016 ran into the side of a bus in order to avoid an obstacle. Specifically, a bus was approaching the vehicle from the rear and intended to pass it, while the Waymo vehicle was prepared to turn (see Fig. 2.11). As the distance between the two vehicles was limited, the vehicle's software estimated that the bus would not attempt to pass it. It turns out that buses usually pass through smaller gaps than the software anticipated in this case. By the time the software updated the bus location, it was too late, and the conflict was inevitable. Figure 2.11 shows how the autonomous vehicle perceived the environment. The path of the vehicle is indicated by a green background and right next to it, the bus with purple attempting to pass the vehicle. The red circle shows a piece of the autonomous vehicle wedged on the bus door.

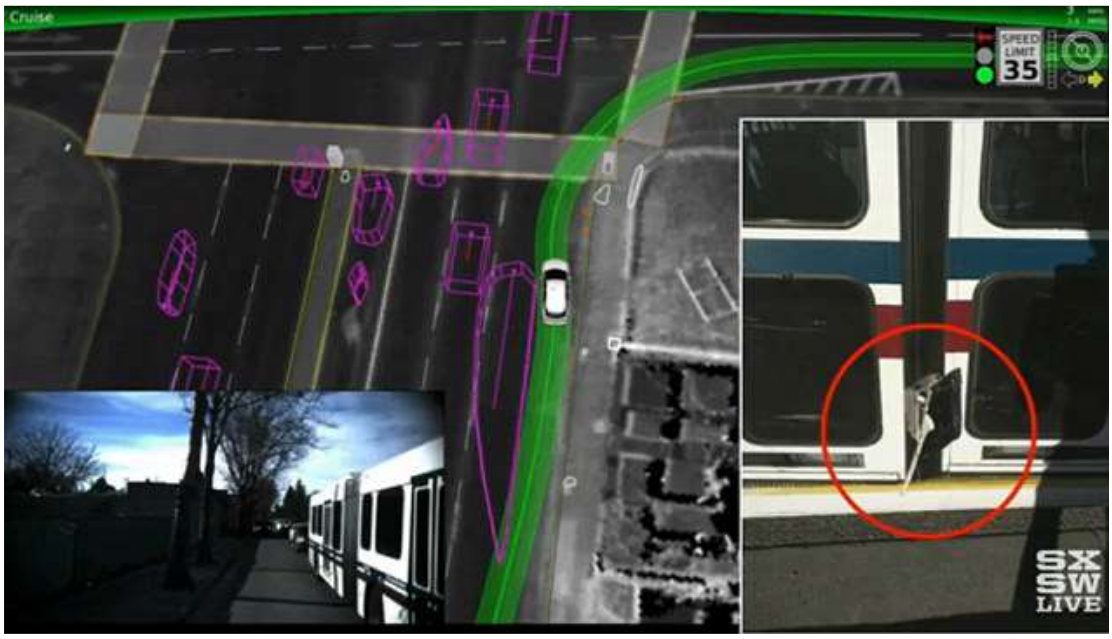

Figure 2.11 Crash of Waymo car with bus (Coursera (2019a))

A year later, an Uber self-driving vehicle overreacted during a minor collision caused by another vehicle and ended up overturning (see Fig. 2.12). As it turned out, the controller had not been tested for such a scenario and overreacted because the dynamic models of the vehicle did not anticipate significant disturbance forces from other vehicles acting on the car. The vehicle's controller was faced with the following dilemma; to crash into two cars in the adjacent lane or to collide with a motorcycle. It's not clear what specific decision was made, since a lawsuit has buried the details of the case.

In general, since decisions in self-driving vehicles are made by multiple agents, it is very challenging to determine what the right action is in many unusual situations. The above sample cases illustrate the need for robustness integrated into the control systems of the vehicle, and for exploratory testing that covers as many foreseeable events as possible.

Figure 2.12 Uber's autonomous vehicle crash (Coursera (2019a))

In 2018, Uber instituted an extensive program that involved safety drivers monitoring the autonomy software of vehicles been tested in the region of Tempe Arizona. Within this program, a fatal accident took place; that is, an autonomous vehicle travelling on a wide multilane divided road at night collided with a person riding a bicycle (Fig. 2.13).

Several causes were found to be relevant to this unfortunate incident. First, the safety driver did not perform proper checks. It is rumoured that the driver was watching Hulu. However, Uber did not have a way to assess the driver's carelessness. In such testing programs, there is need to use a safety driver monitoring system, due to the intrinsic difficulty of the driver to stay focused in monitoring the operation of the autonomous drive system.

Second, there was significant confusion in the detection software system. During the initial detection at six seconds to impact, the rider (and the bicycle) was first considered as an unknown object. Then the rider was misclassified as a vehicle, and then misclassified again as a bicycle. At the end, the decision made by the autonomy software was to ignore the detections, since they were considered as untrustworthy (National Transportation Safety Board, 2018).

Finally, 1.3 seconds before impact, the Volvo emergency braking system detected the rider and would have applied the brakes immediately to reduce the impact speed, potentially saving the life of the victim. However, Uber had disabled the Volvo system when in autonomous mode because it is unsafe to have multiple collision avoidance systems operating simultaneously during testing. Eventually, the autonomous vehicle did not react to the presence of the rider and the inattentive driver was unable to react quickly enough to avoid the collision.

The combination of a) the failure of the perception system to correctly identify the rider with a bicycle b) the failure of the planning control system to avoid the detected object due to uncertainty, c) the human inattentiveness, and d) the disconnection of the emergency braking backup system led to this fatal accident.

2.5.2 Safety and hazards in autonomous vehicles

From the above cases, it is evident that design, perception and control may all lead to failures. Oftentimes the simultaneous operation and interaction of multiple systems or multiple decision-makers can lead to unanticipated consequences.

Figure 2.13 Uber's deadly crash with pedestrian *(National Transportation Safety Board, 2018)*

In general, safety assurance is the process of avoiding unnecessary risk of harm to a living being. For example, driving into an intersection when the traffic signal is red would be unsafe as it leads to unreasonable risk of harm to pedestrians crossing the

intersection, to the occupants of the vehicle and to those of other vehicles moving through the intersection. Furthermore, a hazard is a potential source of unreasonable risk of harm or a threat to safety.

The most common sources of hazards for autonomous vehicles include: a) typical mechanical faults, such as incorrect assembly of a brake system causing a premature failure; b) typical electrical faults, such as incorrect internal wiring leading to a loss of indicator lighting; c) failure in computing hardware used in autonomous driving; d) errors or bugs in software, or in bad or noisy sensor data, or in inaccurate perception; e) incorrect planning or decision-making, inadvertently selecting hazardous actions ; f) failures in the fallback interaction with a human driver, that is, not providing enough warning to the driver to resume responsibility; g) hacking , whereby a self-driving car is hacked by some malicious entity .

2.5.3 Emerging safety regulation and current approaches

The National Highway Traffic Safety Administration (NHTSA) of the US requires industry to develop a complete strategy addressing twelve areas in order to make autonomous vehicles safer (U.S. Department of Transportation, 2017). The first area concerns the system design, which should provide the foundation for the entire safety concept. Especially software design requires careful planning and control and any existing SAE and ISO standards from aerospace and other industries related to automotive should be applied.

The other 11 areas can be categorized in two large groups. The first group concerns the autonomy design and the second group concerns testing of the autonomy function and finding ways to decrease failure.

The first group of areas in the NHTSA framework promotes a precise practical design that provides designers with a clear insight of errors and constraints of the system. Also, it permits designers to determine supported safe cases before testing. Designers should also secure the autonomous vehicle has a fallback mechanism that is friendly to the driver and informs him for potential risks or the vehicle may return in the safety mode. It is very important to note that the driver may not be attentive when the vehicle turns to the safety mode. Thus, the designers should consider minimum risk

settings, until the driver takes control of the vehicle. In addition to the above, the driving system design should abide by all federal, state and local laws for traffic within ODD. The NHTSA framework also encourages designers to consider cybersecurity threats and the related solutions to protect the system from such threats. Finally, the human machine interface should be paid attention to. As a result, the driver should be fully aware at all times of the condition of the autonomous vehicle, for example if the sensors are functional, the current route, the environmental factors that may affect the state of the vehicle, etc.

The second group of areas in the NHTSA framework supports severe testing and the development of exceptional assurance programs that are more stringent than any other system. Simulation, close track testing and public road driving are the three principal foundations of this part of the framework. Autonomy systems with special components that reduce crash energy and guarantee passengers' welfare like restraints, airbags and crash worthiness should be widely adopted. An important issue is the immediate transition of the vehicle after an accident to a safe state. All autonomous vehicles must have a black box, much like aircraft which records all necessarily data to provide information on what went wrong. Finally, the NHTSA recommends that consumers must be mentality trained to operate an autonomous vehicle.

The work of Waymo provides good insights into the application of the NHTSA framework. Waymo's safety report (Waymo, 2018) covers all twelve areas of NHTSA and classifies them in five safety levels shown in Fig. 2.14.

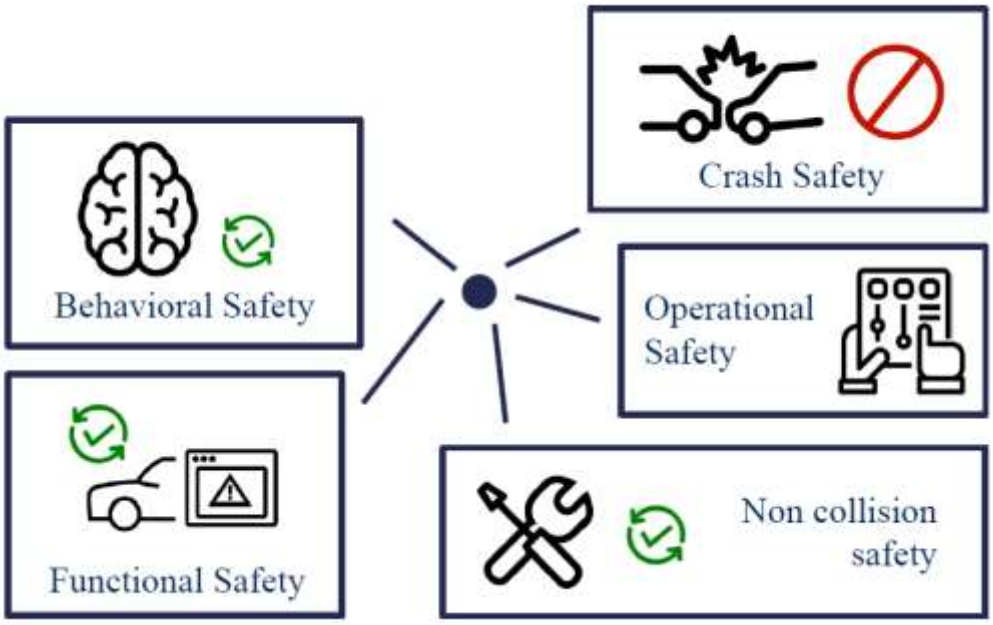

Figure 2.14 Architecture safety levels of Waymo industry (Coursera (2019a)

Firstly, the systems of Waymo safeguard the safety of behavioural driving under key scenarios; for example when the traffic light goes from green to red what operations should the autonomous vehicle software perform in order to stop the vehicle while managing all traffic rules?. Secondly, the functional safety of the system includes backups and redundancies; that is, if the primary system fails then the vehicle switches to a secondary backup process to lead the vehicle towards a safe state.

In terms of non-collision safety, Waymo's systems protect the passengers of the vehicle in the event of a crash. Waymo has introduced a multiple back-up system which includes collision avoidance systems. These systems slow down or stop the vehicle in the infrequent occurrence that the primary system does not detect or respond to objects in the route of the vehicle.

In terms of operational safety, the system lets the passengers assume partial control of the vehicle, when it is in the safe mode. Finally, Waymo's advanced non-collision

safety system reduces risks that relate to the system in some way. For example, sensor hazards or electrical system failures that could cause injuries to the passengers, vehicle technicians, test drivers, etc. The Waymo team produced and tested many hazard scenarios to analyse moderation strategies for each of these risks. They used a range of methods to analyse these scenarios, such as fault trees that work from top down in terms of dynamic driving tasks and bottom-up in terms of small subsystem failures.

For securing the above requirements, Waymo developers use simulators to test all changes in software (see Fig. 2.15); for example, each change or new scenario, is tested by simulating operation during ten million miles in the simulation software. During these tests, Waymo's team performs methodical scenario investigation by varying multiple parameters, such as speed and position of other cars and pedestrians to test whether the ego-car behaves with safety.

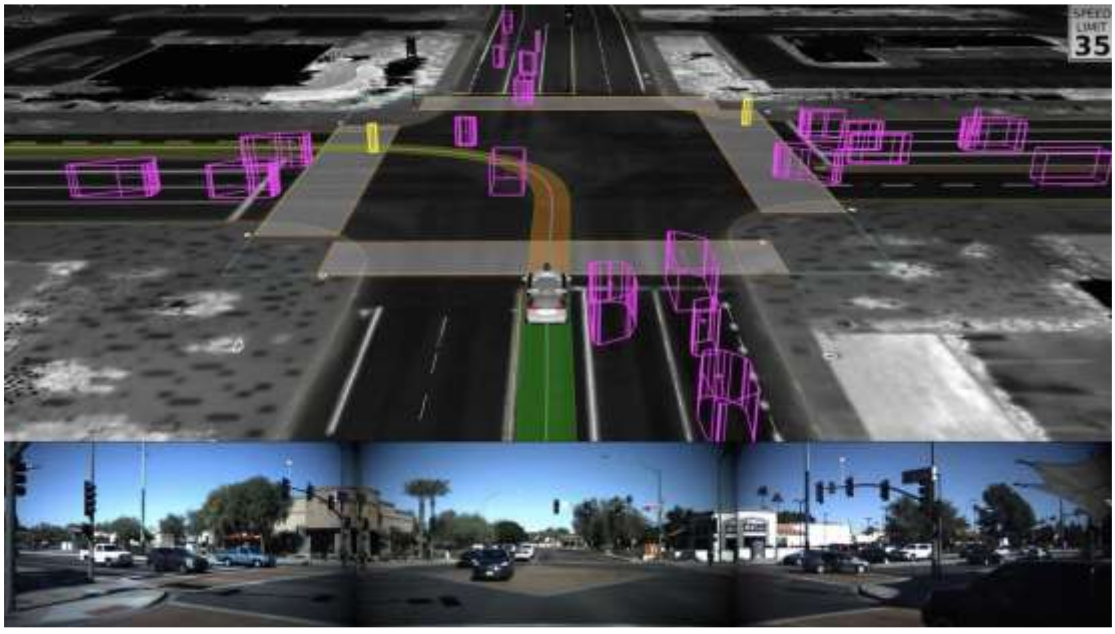

Figure 2.15 Simulation tasting of Waymo's industry (Coursera (2019a)

Post simulation, the team performs physical tests in closed tracks to confirm specific goals in safety performance. The final step comprises tests in an actual traffic environment to explore more challenging scenarios.

In 2017, Waymo in California drove 563,000 km and has encountered 63 disconnections (i.e. one disconnection every 9,000 km.) Disconnections or

disengagements are the deactivations of the autonomous mode when failures are detected in the autonomous software or hardware or when the safe operation demands from the test driver of the autonomous vehicle to take control . General Motors (GM) in California drove 210,000 km with 105 disconnections or approximately one every 2,000 km. This performance is far away from the target of both companies of 400,000 km between successive accidents or disconnections.

Considering that accidents are unique events, the report of DMV (Department of Motor Vehicles, 2018) states that more than 8 billion miles would be required to verify the safety case for an autonomous vehicle. With a fleet of 100 vehicles travelling 24 hours a day, seven days a week, it would take more than 400 years to launch the first autonomous vehicle. For this reason, companies need to increase their fleets to earn more experience with the autonomous systems on the road.

Chapter 3 Vehicle control

3.1 Introduction to vehicle control

Vehicle control provides the actuation commands to drive the vehicle based on the decisions made by the perception of the environment and short-term planning.

The architecture of vehicle control is shown in Fig. 3.1 This architecture comprises four major parts/layers, which are interconnected. The first layer is perception and recognizes the road and the environment around of the autonomous vehicle. Perception is synthesized by the outputs of sensors and the perception intelligence built-in the software of the autonomous vehicle. Perception provides input information to the short-term planning layer.

This second layer generates the speed profile and the vehicle's path, collectively called the drive cycle. The latter comprises the short-term motion plan of the vehicle and provides the critical inputs to the third layer, vehicle control. These inputs are the reference velocity, and an array of consecutive coordinates that define the short-term path of the vehicle (reference path).

The third layer controls the vehicle dynamics in order to execute these reference inputs as best as possible. The third layer comprises the longitudinal and lateral control, that regulate the longitudinal and lateral dynamics of the autonomous vehicle, respectively. Longitudinal control aspires to achieve the reference speed, and lateral control aspires to achieve the reference path. Both control tasks are necessary to follow the short-term plan as accurately as possible by minimizing the errors between the reference inputs and actual outputs. To do so the two control schemes provide three control commands: Throttle angle, break position, and steering angle (the same control settings as in any non-autonomous vehicle driven by a human driver).

The final layer of the architecture is the actuation layer, i.e. the vehicle throttle/engine, break and steering systems, which receive the signals from the control layer and implement these commands in the actual vehicle.

The lateral dynamics have an impact on the longitudinal dynamics and the reserve. The coupling variable is the actual vehicle velocity. It affects the lateral dynamics of the vehicle (due to centrifugal forces, etc.); the lateral dynamics, in turn, affect the actual vehicle velocity and thus the longitudinal dynamics.

The vehicle dynamics, both longitudinal and lateral, are modelled by differential equations, which regulate the state of the vehicle. The forward speed and displacement are determined by the longitudinal forces. Lateral speed and displacement are determined by lateral forces and moments. During vehicle control, it is essential that the desired route and the desired speed be maintained stable.

Due to the importance of vehicle control in the focus of this thesis (and especially of this chapter), we provide in Sections 3.2, and 3.3 an overview of longitudinal and lateral dynamics and control based on (Rajamani, 2012). In Section 3.4 we focus on the tuning of the longitudinal controller via a well-known simulation software of autonomous vehicle operation.

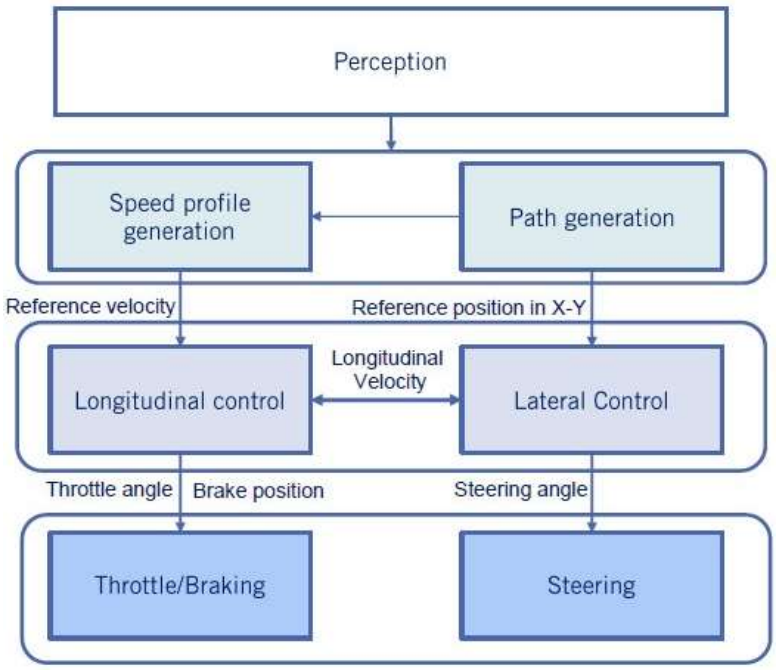

Figure 3.1 The strategy architecture of vehicle control (Coursera (2019a))

3.2 Lateral control[6]

Lateral control is responsible to execute the motion plan for a predetermined route. This is done by adjusting the steering angle, also considering possible differences between the actual and the planned routes (route errors). Lateral control selects a strategy to eliminate these errors taking into account the dynamics of the vehicle, the desired drive characteristics, as well the steering angle constraints

The planned route can be described in numerous ways. A simple description is through a sequence of straight-line segments that connect a sequence of points along the route. These points may be determined by the GPS from earlier runs of the route. Another, improved, way to define the route is by using a sequence of continuous parameterized curves, which can be identified from a fixed set of motion primitives.

To describe the lateral control strategies, in this section we present a simple model for the kinematics of the vehicle, a related model for the vehicle dynamics, and the controllers that drive the vehicle dynamics towards the desired state.

Vehicle kinematics: The bicycle model

In the bicycle model, the vehicle geometry is simplified as follows: The rear axle and the two rear wheels are represented by a single wheel; a similar representation is used for the front wheels and axle (see Fig. 3.2).

To analyze the kinematics of the bicycle model, three points may be used a) the centre of the rear axle, b) the centre of the front axle and c) the centre of gravity of the autonomous vehicle. Thus, to develop the kinematic equations, it is important to define the reference point. More specifically, consider the coordinates of the center of the rear axle to be x_r and y_r, the heading angle of the bicycle model to be θ and the distance between the rear axle and the front axle to be L (see Fig. 3.2). The steering angle is δ and the speed is v. Note that instantaneous centre of rotation (ICR) is the intersection of the straight line of the rear axle with the perpendicular line of the front wheel. In the absence of slip forces and considering the radius of rotation from ICR to be R, then Eq. (3.1) holds.

[6] Inspired by (Snider, 2009) and (Rajamani, 2012)

$$\dot{\theta} = \omega = \frac{v}{R} \tag{3.1}$$

Considering the triangle of Fig. 3.2

$$\tan\delta = \frac{L}{R} \tag{3.2}$$

Thus, Eq. (3.1) becomes

$$\dot{\theta} = \omega = \frac{v}{R} = \frac{v\tan\delta}{L} \tag{3.3}$$

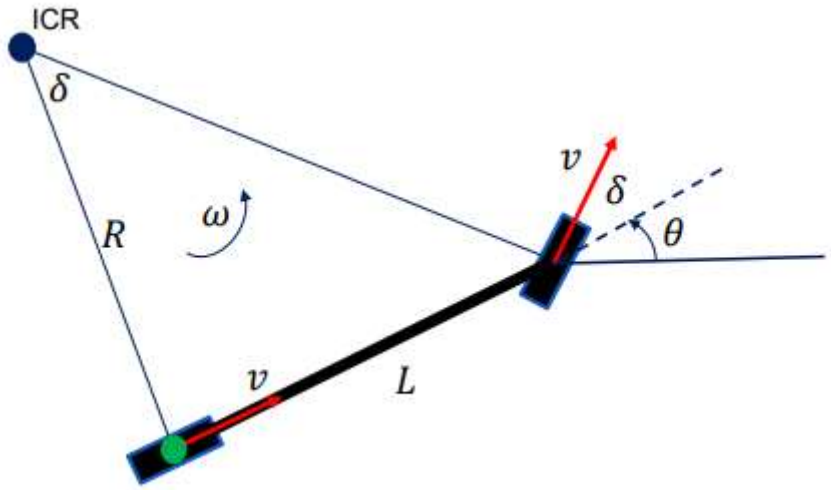

Figure 3.2 Kinematics of bicycle model with respect to (x_r, y_r) (Coursera (2019a))

The equations of motion for the reference point (x_r, y_r) and the rotation angle θ are:

$$\dot{x}_r = v\cos\theta \tag{3.4}$$

$$\dot{y}_r = v\sin\theta \tag{3.5}$$

$$\dot{\theta} = \frac{v\tan\delta}{L} \tag{3.6}$$

For the center of the front axle (x_f, y_f) these equations according to the triangle of figure 3.2 become (refer also to Fig. 3.3)

$$\dot{x}_f = v\cos(\theta + \delta) \tag{3.7}$$

$$\dot{y}_f = v\sin(\theta + \delta) \tag{3.8}$$

$$\dot{\theta} = \frac{v\sin\delta}{L} \tag{3.9}$$

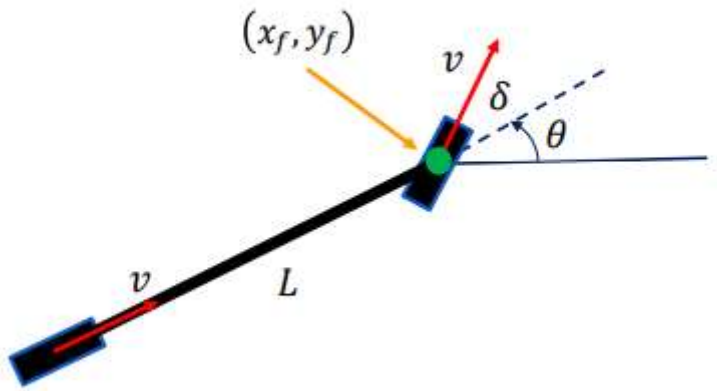

Figure 3.3 Kinematics of bicycle model with respect to (x_f, y_f) (Coursera (2019a))

The difference between Eq. (3.6) and (3.9) result from approximations of the radius of rotation.

For the centre of gravity c_g, with coordinates (x_c, y_c), the slip angle β needs to be considered (see Fig. 3.4). Let l_r be the distance between the centre of the rear axle and the centre of gravity. From Fig. 3.4 and considering Eq. (3.2)

$$tan\beta = \frac{l_r}{R} \Longrightarrow \beta = tan^{-1}(\frac{l_r}{R}) \Longrightarrow \beta = tan^{-1}(\frac{l_r}{L}tan\delta) \tag{3.10}$$

If there is no slip rate, then the kinematic equations with respect to the centre of gravity become:

$$\dot{x}_c = v\cos(\theta + \beta) \tag{3.11}$$

$$\dot{y}_c = v\sin(\theta + \beta) \tag{3.12}$$

$$\dot{\theta} = \frac{v\cos\beta\tan\delta}{L} \tag{3.13}$$

Again the differences between Eq. (3.6) and (3.13) are due to the radius of rotation.

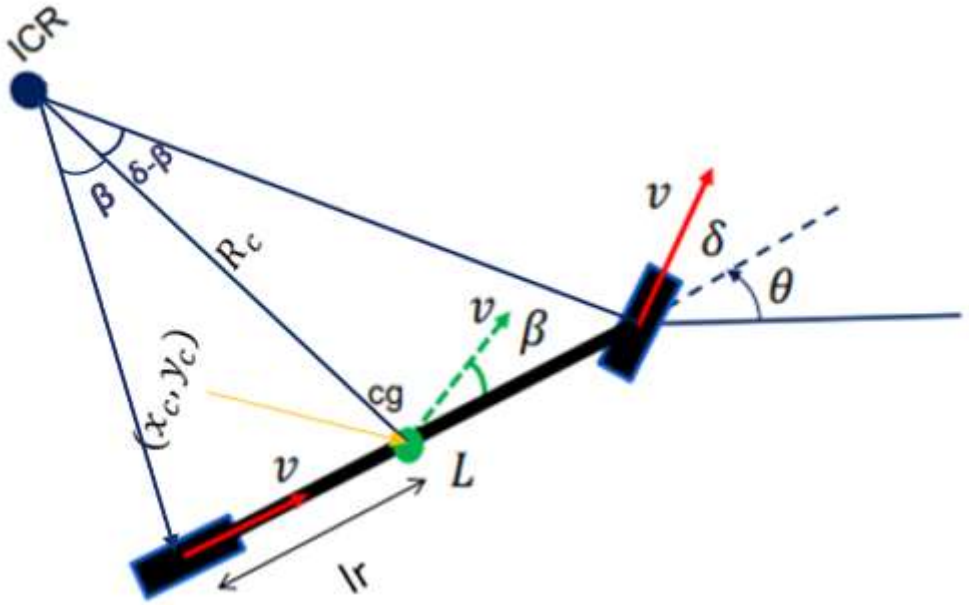

Figure 3.4 Kinematics of bicycle model with respect to (x_c, y_c) (Coursera (2019a))

The dynamics of the bicycle model

To describe the lateral dynamics of the bicycle model let's make the following simplifying assumptions:

- The forward speed is constant
- Other nonlinear effects, such as suspension movement, aerodynamic forces and road inclination are assumed unimportant.

Consider the centre of gravity (c_g) of the autonomous vehicle as the reference point. With reference to Fig. 3.5, the total acceleration comprises the lateral acceleration $\ddot{y}$ and the centripetal acceleration from rotation of the vehicle.

$$a_y = \ddot{y} + \omega^2 R_c \tag{3.14}$$

Considering v to be the longitudinal speed of the vehicle

$$v = \omega R_c \tag{3.15}$$

Under certain simplification assumptions, the rate of the heading angle $\dot{\theta}$ can be considered equal to the angular speed of vehicle.

$$\omega = \dot{\theta} \tag{3.16}$$

The vehicle sideslip angle β may be defined through the triangle of Fig. 3.5. The second relationship in Eq. (3.17) holds for small angles β..

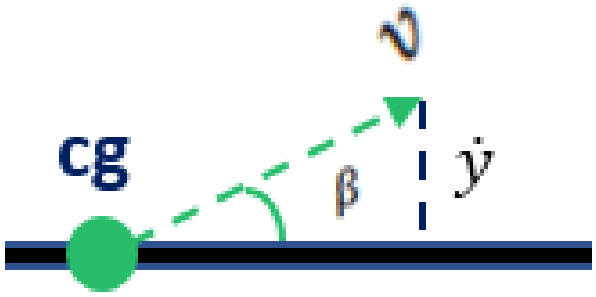

Figure 3.5 Relationship between the speed v and its component $\dot{y}$

$$\sin\beta = \frac{\dot{y}}{v} \Rightarrow \beta = \frac{\dot{y}}{v} \Leftrightarrow \dot{y} = \beta v \tag{3.17}$$

Using Eqs. (3.15), (3.16) and (3.17) into Eq. (3.14), we obtain

$$a_y = v\dot{\beta} + v\dot{\theta} \tag{3.18}$$

Thus, for the lateral dynamics we may write the second law of Newton as:

$$mV(\dot{\beta} + \dot{\theta}) = F_{yf} + F_{yr} \tag{3.19}$$

where m is the mass of the vehicle, and F_{yf} F_{yr} are the forces applied on the front and the rear tires respectively (in the y direction of course).

The moments generated by the tire forces have opposite directions and the equation becomes

$$I_z\ddot{\theta} = l_f F_{yf} - l_r F_{yr} \tag{3.20}$$

where I_z is vehicle inertia term and l_f and l_r are the distances between the center of gravity and the front and the rear tires respectively (See Fig 3.6).

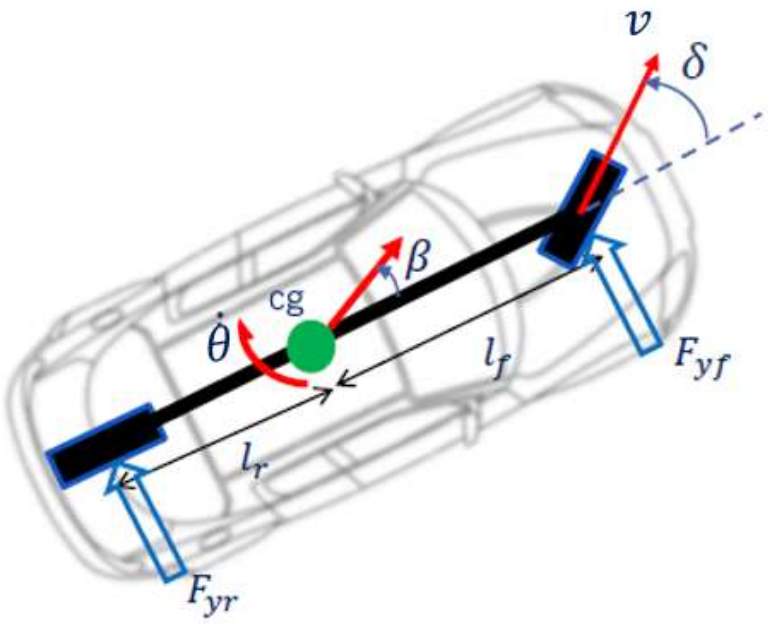

Figure 3.6 Lateral dynamic model with respect to cg (Coursera (2019a))

Forces F_{yf} and F_{yr} are given from:

$$F_{yf} = C_f a_f = C_f(\delta - \beta - \frac{l_f \dot{\theta}}{v}) \tag{3.21}$$

where C_f is the linearized cornering stiffness of the front wheel and a_f is the slip angle of front tire

$$a_f = \delta - \theta_{vf} = \delta - \beta - \frac{l_f \dot{\theta}}{v} \tag{3.22}$$

where θ_{vf} is the angle between the actual speed vector and the longitudinal axis of the vehicle and δ is the front wheel steering angle (Rajamani, 2012) – see also Fig 3.6.

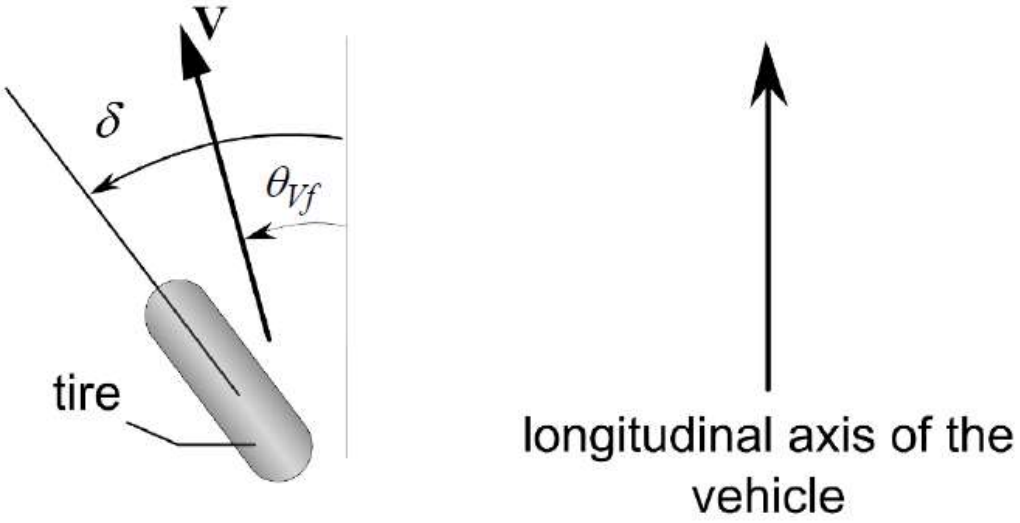

Figure 3.7 Front tire sleep angle (Rajamani, 2012)

$$F_{yr} = C_r a_r = C_r(-\beta + \frac{l_r \dot{\theta}}{v}) \tag{3.23}$$

where C_r is the linearized cornering stiffness of the rear wheel and a_r is the slip angle of the rear tire. The equation of a_r is similar given by

$$a_r = -\theta_{Vr} = -\beta + \frac{l_r \dot{\theta}}{v} \tag{3.24}$$

Substituting the lateral forces F_{yf}, F_{yr} into to the dynamic equations for the bicycle model (3.21) and (3.23), we obtain the following state equations:

$$\dot{\beta} = \frac{-(C_r + C_f)}{mv}\beta + \left(\frac{C_r l_r - C_f l_f}{mv^2} - 1\right)\dot{\theta} + \frac{C_f}{mv}\delta \tag{3.25}$$

and

$$\ddot{\theta} = \frac{(C_r l_r + C_f l_f)}{I_z}\beta + \frac{C_r l_r^2 - C_f l_f^2}{I_z V}\dot{\theta} + \frac{C_f l_f}{I_z}\delta \tag{3.26}$$

Assuming that the states of the system are y, β, θ and $\dot{\theta}$, the input is the steering angle δ and the outputs are, y and θ,the system lateral dynamics (along the y axis) is given by Eq. (3.17) to (3.13), considering Eqs. (3.18), (3.25) and (3.26).:

$$\dot{X}_{lat} = A_{lat}X_{lat} + B_{lat}\delta \tag{3.27}$$

where

$$A_{lat} = \begin{bmatrix} 0 & v & v & 0 \\ 0 & -\dfrac{C_r + C_f}{mv} & 0 & \dfrac{C_r l_r - C_f l_f}{mv^2} - 1 \\ 0 & 0 & 0 & 1 \\ 0 & \dfrac{C_r l_r - C_f l_f}{I_z} & 0 & -\dfrac{C_r l_r^2 + C_f l_f^2}{I_z v} \end{bmatrix} \tag{3.28}$$

$$B_{lat} = \begin{bmatrix} 0 \\ \dfrac{C_f}{mv} \\ 0 \\ \dfrac{C_f l_f}{I_z} \end{bmatrix} \tag{3.29}$$

$$X_{lat} = \begin{bmatrix} y \\ \beta \\ \theta \\ \dot{\theta} \end{bmatrix} \tag{3.30}$$

Controllers

Two types of lateral control are typically used in autonomous vehicles. The first type comprises geometric controllers, which are based on the geometry and coordinates of the desired route and the kinematic models of the vehicle. There are two sub-types of geometric controllers: a) pure pursuit and b) Stanley controllers. The second type of control concern dynamic controllers, such as the model predictive controller (MPC). In this thesis we will discuss geometric controllers.

The main idea of pure pursuit control is that a reference point is defined in the vehicle and a target point is set on the desired route (trajectory) at a fixed distance from the reference point. The steering angle is then determined so that the vehicle turns

smoothly towards the route. Typically, the center of the rear axle is used as the reference point and is connected to the target point on the route ahead of the vehicle with line of length (l_d), which is defined as the look-ahead line. The angle between the axis of the vehicle and the look-ahead line is α (see Fig 3.8).

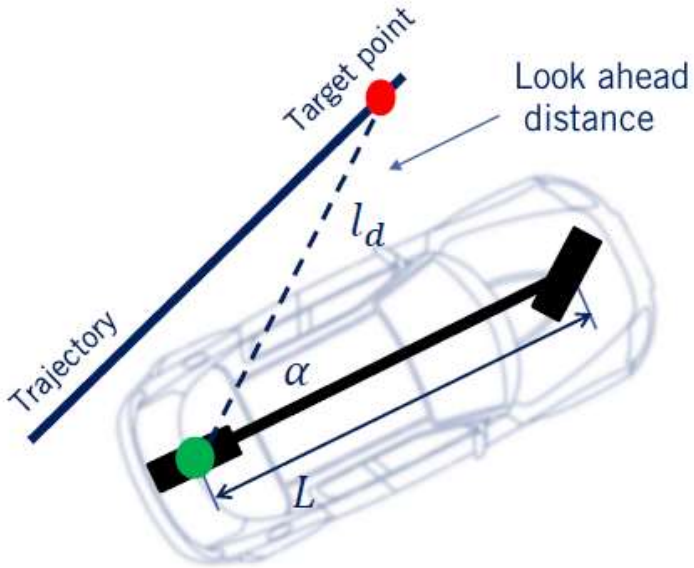

Figure 3.8 Connecting the reference point with target point (Coursera (2019a))

The target point on the route (trajectory), the instantaneous centre of rotation and the centre of the rear axle form a triangle with two sides of length R and a third of length l_d (see Fig. 3.8). Using the law of sines for the triangle if Fig. 3.9, and after some algebra

$$\frac{1}{R} = \frac{2 \sin a}{l_d} \tag{3.31}$$

where $k = \frac{1}{R}$ is the path curvature. Then, from the bicycle model, the steering angle δ required for the vehicle to stay on route is calculated as (see Eq. 3.3)

$$\delta = \tan^{-1} \frac{L}{R} \tag{3.32}$$

where L is the distance between the rear and front axles and

From Eqs. (3.27) and (3.28) we have

$$\delta = \tan^{-1}(\frac{2L \sin a}{l_d}) \tag{3.33}$$

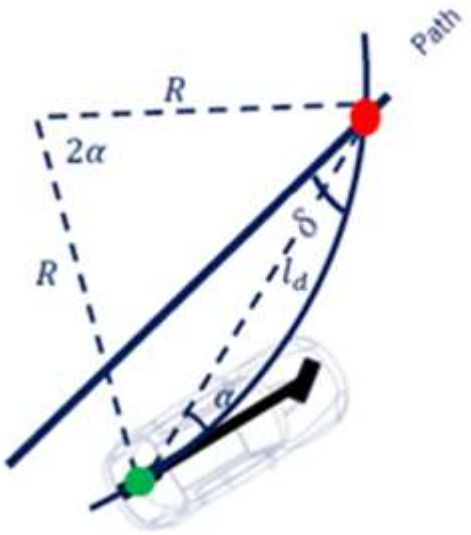

Figure 3.9 Steering angle needs to follow the arc towards the target point (Coursera (2019a))

The Stanley controller is a geometric path tracking controller. The main idea is to change the reference point. To do so, three modifications are required (see also Fig. 3.10):

- Change of the reference point from the centre of the rear axle to the centre of the front axle.
- Do not consider the look ahead distance but take into consideration the heading alignment and cross track errors. The heading alignment error is shown as $\psi(t)$ in Fig. 3.10 where the heading relative to the trajectory is $\theta(t)$. The cross-track error is shown as e.

This type of control strategy has the following advantages vs. the pursuit strategy:

- It directly eliminates the heading error relative to the route

- The steering angle its set equal to the heading directly.

For cross-track error dynamics the following equation holds (see Fig. 3.10):

$$\dot{e}(t) = -v(t)\sin\big(\theta(t) - \delta(t)\big) \tag{3.34}$$

For the heading error dynamics, the following equation holds (see also Fig. 3.10):

$$\dot{\theta}(t) = \frac{-v(t)\sin\delta(t)}{L} \tag{3.35}$$

To eliminate the heading error relative to the planned route, the steering angle should be set equal to the heading

$$\delta_1(t) = \theta(t) \tag{3.36}$$

To eliminate the cross-track error a proportional control is added. This is scaled by the inverse of the forward speed (v). If the the cross-track error is e then the steering angle is given by the following equation, where function tan^{-1} maps the proportional control signal to the angular range $(-\pi, \pi)$ and k is the gain of the proportional controller

$$\delta_2(t) = \tan^{-1}\left(\frac{ke(t)}{v(t)}\right) \tag{3.37}$$

Combining Eqs. (3.35), (3.36) and (3.37) the control law providing the steering angle of the vehicle is as follows

$$\delta(t) = \theta(t) + tan^{-1}\left(\frac{ke(t)}{v(t)}\right), \qquad \delta(t) \in [\delta_{min}, \delta_{max}] \tag{3.38}$$

For large cross-track error:

$$\tan^{-1}\left(\frac{ke(t)}{v(t)}\right) \approx \frac{\pi}{2} \rightarrow\ \delta(t) \approx \theta(t) + \frac{\pi}{2} \tag{3.39}$$

In this case the heading error increases in the opposite direction and the steering command will drop to zero when the heading error achieves $(-\frac{\pi}{2})$. Then the vehicle continues to the route until the cross-track error becomes 0.

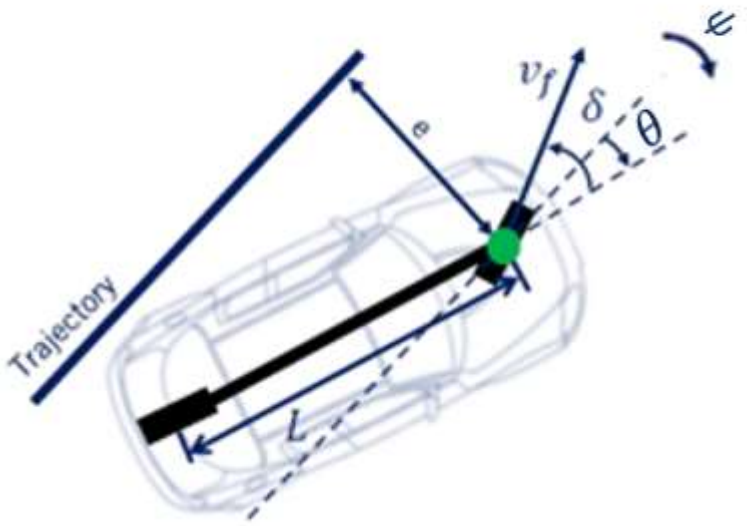

Figure 3.10 Geometry of the Stanley controller (Coursera (2019a))

3.3 Longitudinal control

Longitudinal control generates the input commands, or the actuator signals, that drive the vehicle, i.e. the throttle and the brake commands. Let us first review the vehicle longitudinal dynamic model, which is responsible for generating the forward motion of the vehicle.

Vehicle longitudinal dynamics[7]

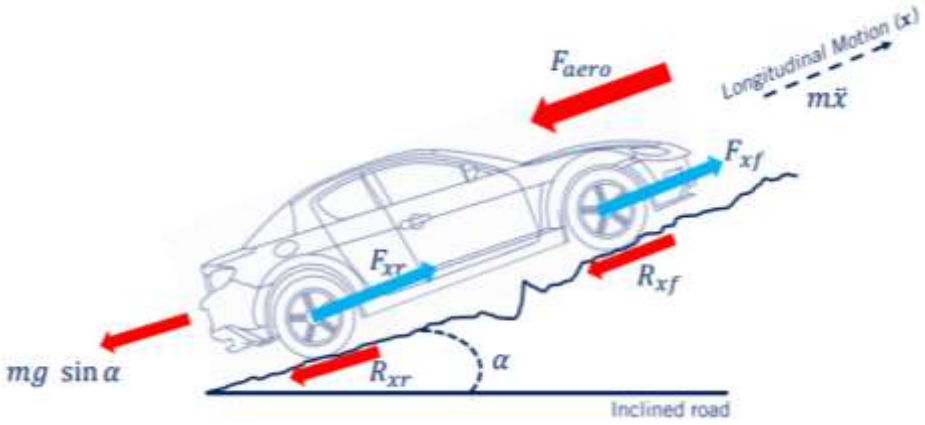

Figure 3.11 A typical vehicle on an inclined road (Coursera (2019a))

[7] Inspired by (Rajamani, 2012)

According to Fig. 3.11, the forces acting on the vehicle are the following: the driving force at the front wheels of the vehicle F_{xf}, the driving force at the rear wheels of the vehicle, F_{xr} (if any), the aerodynamic drug force F_{aero}, gravity $mgsina$, and finally the rolling friction forces R_{xf} and the R_{xr}. Based on Newton's second law the longitudinal dynamic equation is therefore

$$m\ddot{x} = F_{xf} + F_{xr} - F_{aero} - R_{xf} - R_{xr} - mg\sin a \tag{3.40}$$

By grouping the driving forces $F_x = F_{xf} + F_{xf}$ and the rolling resistance forces $R_x = R_{xf} + R_{xf}$ Eq. (3.40) becomes

$$m\ddot{x} = F_x - F_{aero} - R_x - mg\sin a \tag{3.41}$$

Let

$$F_{load} = F_{aero} + R_x + mg\sin a \tag{3.42}$$

The aerodynamic resistance force is provided by

$$F_{aero} = \frac{1}{2}C_a\rho A\dot{x}^2 = c_a\dot{x}^2 \tag{3.43}$$

where, C_a is the aerodynamic drag coefficient, ρ is the mass density of air, A is the frontal area of the vehicle, which is the projected area of vehicle in direction of travel, and $\dot{x}$ is the longitudinal speed.

The rolling resistance depends on the tire normal force on the rear tires and the autonomous vehicle speed $\dot{x}$:

$$R_x = N\left(\hat{c}_{r,0} + \hat{c}_{r,1}|\dot{x}| + \hat{c}_{r,2}\dot{x}^2\right) \approx c_{r,1}|\dot{x}| \tag{3.44}$$

where $c_{r,1}$ is the linear rolling resistance factor.

Thus,

$$F_{load} = c_a\dot{x}^2 + c_{r,1}|\dot{x}| + mg\sin a \tag{3.45}$$

and Eq. (3.41) becomes

$$m\ddot{x} = F_x - F_{load} = F_x - (c_a\dot{x}^2 + c_{r,1}|\dot{x}| + mg\sin a) \tag{3.46}$$

F_x is the driving or traction force that is generated by the power train. The power is generated by the combustion of fuel in internal combustion engines or

electrochemical reactions in batteries for electric vehicles. The engine torque is passed to the automatic transmission system, which includes a torque converter placed between the engine shaft and the gearbox. The gears in the gearbox change accordingly to the desired speed. Thus, through the differential, the power generates the wheel torque which finally generates the traction forces.

The relation between the engine speed and the wheel speed its modeled as a kinematic constraint. The wheel rotation speed ω_w is the result of the engine angular speed ω_e modulated by several gear ratios including those of the torque converter, transmission and differential. If this combined gear ratio is symbolized as GR, the equation of the wheel rotational speed ω_w is

$$\omega_w = \text{GR}\omega_e \tag{3.47}$$

where ω_e is the engine angular speed. The vehicle longitudinal speed is provided by

$$\dot{x} = r_{eff}\omega_w = r_{eff}\text{GR}\omega_e \tag{3.48}$$

where r_{eff} is the tire effective radius.

Considering now the practical power train inertia J_e , we can write a dynamic equation that balances the engine torque $T_e$with the total torque needed to generate the engine acceleration and to overcome the momentum due to the load. Thus, the engine dynamic model simplifies to

$$J_e\dot{\omega}_e = T_e - (GR)(r_{eff}F_{load}) \tag{3.49}$$

The engine torque T_e is a function of the accelerator pedal position, x_θ and the engine speed ω_e (in RPM - revolutions per minute) as shown in Fig. 3.12. It may be approximated by a second-order polynomial equation

$$T_e(\omega_e, x_\theta) \approx x_\theta(A_0 + A_1\omega_e + A_2\omega_e^2) \tag{3.50}$$

where A_0, A_1, A_2 are engine-dependent parameters. Thus,

$$J_e\dot{\omega}_e = x_\theta(A_0 + A_1\omega_e + A_2\omega_e^2) - (GR)(r_{eff}F_{load}) \tag{3.51}$$

Equations (3.46), (3.48), (3.51) and (3.45) model the vehicle longitudinal dynamics, where x_θ is the input.

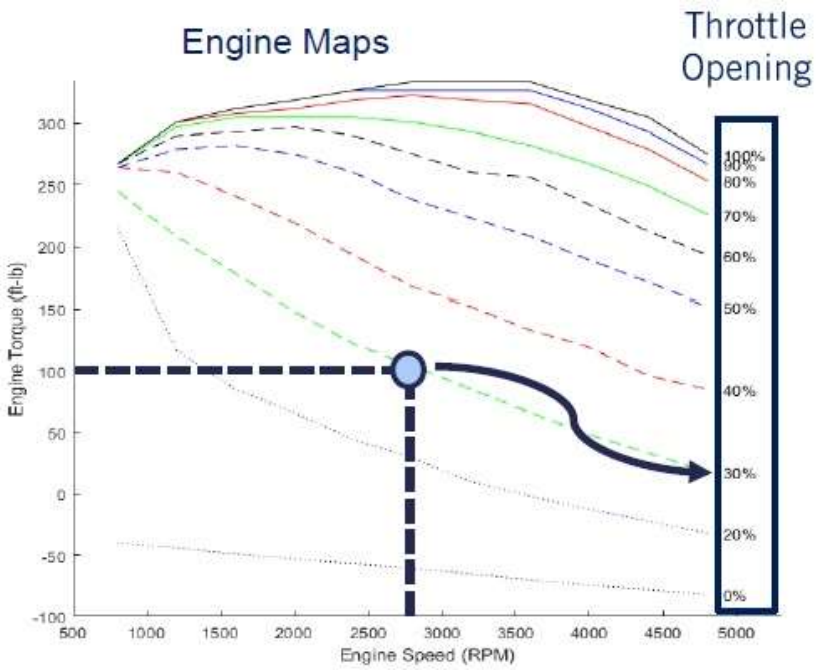

Figure 3.12 Typical engine maps (Coursera (2019a))

Longitudinal speed control system [8]

The longitudinal speed control system receives as reference input the desired speed from the navigation system and tries to minimize the error between the reference speed and the actual speed. To do so, it uses the control architecture of Fig. 3.13. The reference speed is compared to the actual speed, and the error is the input to the controllers that set the throttle (and brake) positions. The latter are inputs to the vehicle dynamics described above. The output is the actual speed.

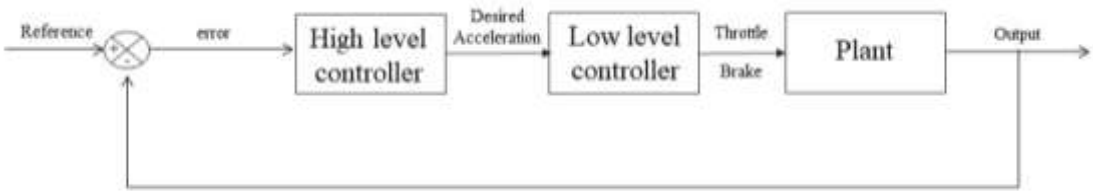

Figure 3.13 A longitudinal speed control feedback system

[8] Inspired by (Francis, 2015)

The high level controller computes the difference between the set reference speed and the autonomous vehicle actual speed to produce the required vehicle acceleration from:

$$\ddot{x}_{des} = K_p(\dot{x}_{ref} - \dot{x}) + K_I \int_0^t (\dot{x}_{ref} - \dot{x})\, dt + K_D \frac{d(\dot{x}_{ref} - \dot{x})}{dt} \qquad (3.51)$$

where K_p, K_I, K_D are the PID controller gains, $\dot{x}_{ref}$ is the reference speed and $\dot{x}$ is the vehicle velocity.

Based on the desired acceleration, the low level controller generates the throttle (or braking) commands. Figure 3.14 explains the action of the lower level controller.

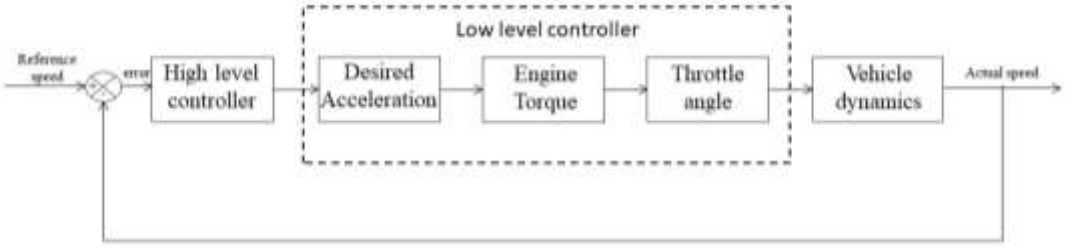

Figure 3.14 Low level controller

Equation (3.49) can be re-written as

$$T_{engine} = \frac{J_e}{(r_{eff})(GR)} \ddot{x}_{des} + T_{load} \qquad (3.52)$$

The lower level controller uses the above equation to compute the required engine torque from the desired acceleration provided by the upper level controller. It then computes the throttle angle (opening) form the steady-state engine map of Fig. 3.12 based on the required torque and the engine angular speed in RPM. This is then the input to the longitudinal vehicle dynamics discussed above.

3.4 Experiments on longitudinal control

3.4.1 The CARLA (Car – Like – To – Act) autonomous vehicle simulator

General description of CARLA

CARLA is an open source simulator for autonomous driving and has been produced by a team from the Computer Vision Centre at the Autonomous University of Barcelona, Intel and the Toyota Research Institute using the Unreal computer game engine.

CARLA has been developed in order to render and simulate with flexibility and realism. The simulation is applied as an open-source layer over the Unreal Engine 4 (UE4), which makes possible future extensions by the community. State – of –the - art rendering quality, realistic physics, standard NPC (non-player-character) logic, and an ecosystem of interoperable plugins are supplied by the engine.

Thus, CARLA is designed as a server-client system, where the server runs the simulation and depicts the scene (UE4) and the client is used to record data, control the autonomous vehicle scenarios and send commands to the vehicle. The client API regulates the interaction with the server via sockets and is developed in Python. It sends commands and meta-commands to the server and then receives sensor readings. The commands are used to control the vehicle and the meta – commands regulate the behavior of the server. Steering, accelerating, and braking are considered as commands. Meta-commands reset the simulation, modify the properties of the environment, such as the weather conditions, the illumination and the density of the cars and pedestrians.

Figure 3.15 CARLA simulation environment (Screenshots from the DeOPSys Lab system)

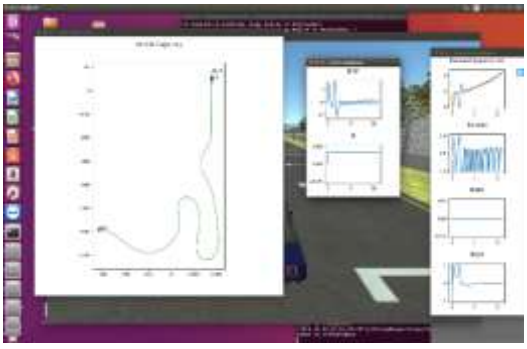

Figure 3.16 CARLA simulation graphs (Screenshots from the DeOPSys Lab system)

The environment (see Fig. 3.15 and Fig. 3.17) includes 3D models of static objects (buildings, vegetation, traffic signs, infrastructure) and of dynamic objects (vehicles and pedestrians). All models are designed thoroughly in order to combine visual quality and rendering speed.

The simulation graphs (see Fig. 3.16) includes the route of the autonomous vehicle and the result graphs (forward speed, steering, throttle, brake, and the error between the desired and the actual speed). All these graphs are shown the behavior of the vehicle on the default route.

CARLA permits flexible configuration of the client's (agent's) sensor suite. Sensors are limited to RGB cameras and to pseudo-sensors, which provide ground-truth depth and semantic segmentation. The client determines the type, the position and the number of the cameras. Thus, the user specifies the 3D location, the 3D orientation with respect to the vehicle's coordinate system, the field of view, and the depth of field. Regarding pseudo-sensors, semantic segmentation is an image processing algorithm that identifies objects from the camera pixels. It uses twelve sematic classes: road, lane-marking, traffic sign, sidewalk, wall, building, vegetation, vehicle, pedestrian and other. Thus, the pseudo-sensor displays the orientation and the position of static and dynamic objects.

Except from the sensor and pseudo-sensor readings, CARLA provides a variety of measurements related to the state of the client, such as the vehicle location, the speed and the acceleration vector, the impact from collisions and the vehicle orientation. CARLA also provides measurements of the traffic environment, including the state of the traffic lights, the speed limit at the current location of the vehicle along with the percentage of the vehicle's footprint that overlaps with wrong-way lanes or sidewalks. Last but not least, CARLA provides information about the exact locations and bounding boxes of all dynamic objects in the environment. These signals are crucial when the driving policies are assessed.

The simulation is performed in discrete steps. One-time step is referred as a frame and the frequency rate is 30 frames per second.

Figure 3.17 CARLA simulation environment with sematic segmentation and depth cameras (Screenshots from the DeOPSys Lab system)

Figure 3.18 CARLA simulation environment with semantic segmentation and depth cameras (Screenshots from the DeOPSys Lab system)

Figures 3.17 and 3.18 show the environment of the CARLA simulator. The top window of the right corner of both pictures shows the semantic segmentation (right) pseudo-sensor which includes the classes of lanes (light green colour), sidewalks (purple colour), traffic lights (with yellow) and cars (light blue colour). The second pseudo-sensor shown next on the semantic segmentation pseudo-sensor is the ground-truth depth which is important in the initial supervised classification of an image.

CARLA client components used in this thesis

The autonomous vehicle in the CARLA simulator constantly receives steering, throttle and brake commands. For the work in this thesis we provide the server with appropriate settings. For example, for the control study, we have eliminated pedestrians and other vehicles in order to investigate the intrinsic properties of the longitudinal speed controller. We have also programmed the lateral and longitudinal controllers in the client software (in Python). The controllers compute the current position, the speed and the heading angle of the vehicle. Based on the control logic implemented in the client, the controllers return the commands for the steering angle, throttle and brake to the server. This is repeated every time frame.

The hardware used for the Carla simulator is as follows:

- CPU: Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
- Ram: 32GB DDR3 1333MHz

- Graphics card: GeForce GTX 1060 6GBF
- Hard drive: SSD 256GB and HDD 500GB

The above hardware specifications support efficient performance of the simulator.

The operational system used in this thesis is Ubuntu 16.04 with CUDA drivers and python 3.6. For more information about to install CARLA simulator check in Appendix A.

3.4.2 Experimental study of motion control

For this thesis, we used CARLA to develop and implement the longitudinal and lateral controllers of the autonomous vehicle. The operation system used with the CARLA simulator was Ubuntu 16.4. CUDA, which was also employed to improve the performance of the simulator by enabling the hardware and software of the Graphics Processor Unit (GPU) to achieve better and faster computing performance. Also, we used Python version 3.6 to compile and run all client programs used by the CARLA server. More information on the simulator installation steps are provided in Appendix A.

Controller development in CARLA

For the longitudinal control we developed the code for the PID controller, which was described in Section 3.3. The PID controller takes as reference inputs the desired speed at the waypoints of the simulator, as well as the position that the vehicle should attain. The waypoints provide both the vehicle's trajectory and the speed that become the reference signals for vehicle navigation along its trajectory and for the PID controller.

This output of the controller is formed by the sum of three parts: The proportional part that multiplies gain K_P with speed error, the integral part with gain K_I, which multiplies the accumulated past errors (error integral) in order to eliminate the steady state error (deviation between the desired speed and the actual speed), and the derivative part with gain K_D, which reduces the overshoot caused by the integration (K_I) term. The output of the PID controller constitutes the throttle and brake

commands. Positive outputs correspond to appropriate throttle positions and negative outputs correspond to appropriate brake positions.

For the lateral control that provides the steering angle we developed the Stanley controller, which was described in Section 3.2. The reference signal is given at the waypoint positions and the control routine computes the cross track and heading errors. To do so it uses the current way point and the previous one to produce the trajectory line that is expressed by the following equation (see Fig. 3.19).

$$ax + by + c = 0 \tag{3.53}$$

Based on this line, the cross-track error is computed from the following equation

$$e = \frac{ax_c + by_c + c}{\sqrt{a^2 + b^2}} \tag{3.54}$$

This is the perpendicular distance between the reference trajectory and the front axle point (see Fig. 3.19). Eq. (3.54) is proven by geometric arguments.

Then, using Eq. (3.37) of Section 3.2, the steering angle to eliminate the cross-track error is provided by the following expression.

$$\tan^{-1}(\frac{ke}{v}) \tag{3.55}$$

where k (for any $k > 0$) is a proportional gain of proportional and v is the forward speed.

The heading error ψ is computed by

$$\psi = \tan^{-1}(\frac{-a}{b}) - \theta_c \tag{3.56}$$

where a and b are the coefficients of the trajectory line and θ_c is the heading angle.

By adding Eqs. (3.55) and (3.56), the steering angle is given as

$$\delta = \psi + \tan^{-1}(\frac{ke}{v}) \tag{3.57}$$

The lateral controller provides the steering angle of the vehicle.

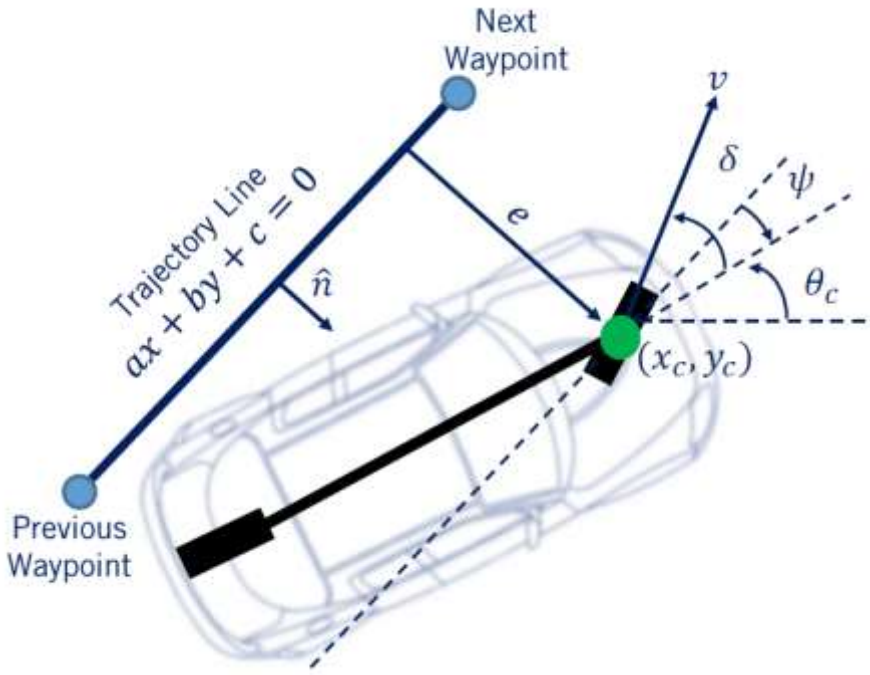

Figure 3.19 Geometry of Stanley Controller (Coursera (2019a))

Experimental set up

We conducted experiments to investigate the effects of the gains K_P, K_I, K_D of the PID longitudinal controller of Section 3.3 on the vehicle speed and trajectory. In the experiments we varied systematically the values of the gains and observed the resulting effect. It is noted that when $K_I = K_D = 0$, then the PID controller becomes a P controller. Similarly, when $K_I = 0$ a PD controller is obtained and when $K_D = 0$ a PI controller is obtained (see Table 3.1). Overall, we performed twenty-two (22) experiments.

Table 3.1 The various controllers tested and the related gains

PID			PI		PD		P
K_P	K_I	K_D	K_P	K_I	K_P	K_D	K_P

The reference route of the autonomous vehicle is shown in Fig. 3.20. The measurement unit used in the x-axis and y-axis is meters (m). The total distance travelled by the autonomous vehicle is 1,755 m.

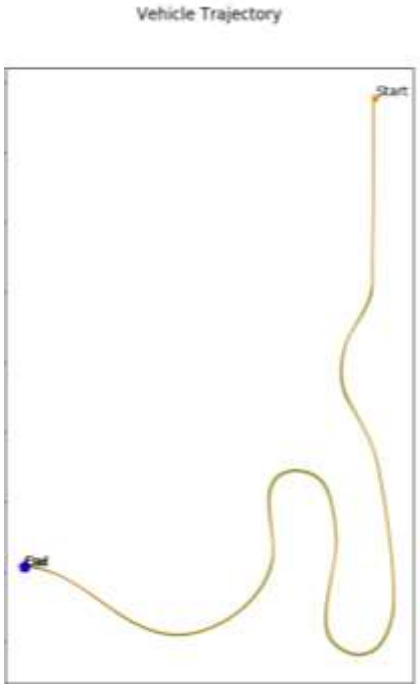

Figure 3.20 The reference route of the autonomous vehicle

For the experiments we firstly set up Carla using the following parameters:

- Weather: sunny.
- Vehicle: Ford Mustang.
- The total runtime before simulation lasts 200sec.

Moreover, important libraries have been imported:

- Library math
- Library NumPy. This library accelerates the math operations used in the simulation by performing array multiplications, fast linear algebra operations, etc.
- Library matplotlib used for plotting all graphs (e.g. the graph for the forward speed of the autonomous vehicle)
- Library Time it is used to handle the time-related tasks. For example, the steps used to calculate the Frames per second (FPS). Also, it is used to compute elapsed time, and for gathering timestamps to update the PID controller.

Regarding the control commands steering, throttle and brake:

- The steering wheel angle is represented it by a real number between (-1,1), where -1 corresponds to full left and 1 corresponds to full right
- The angle (pressure) of the throttle pedal is represented by a real number between (0,1), where 1 corresponds to full pressure on the pedal and 0 corresponds to no action
- The brake pedal angle (pressure) is represented similarly to the throttle pedal.

All above parameters were provided to the main program module7.py which interacts with the class controller2d.py.

The performance of the controller is assessed through the mean squared error (MSE) provided in Eq. (3.58)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(v_{desired} - v_{actual})^2 = \frac{1}{n}\sum_{i=1}^{n} {v_{error}}^2 \quad (3.58)$$

where n are the points crossed by the vehicle.

Experimental investigation and results

Case 1: Investigating the effect of the proportional gain K_P on all four controller cases

The main goal of case 1 is to assess the performance of the autonomous vehicle under the following values of the proportional gain $K_P = 0.1$, 1, 5, and 10 in all four controller variations (PID, PD, PI, P). In all cases in which K_D and K_I were not zero, these gains assumed the values $K_D = 0.01$ and $K_I = 0.2$. Thus, we performed 16 experimental runs (4 K_P values x 4 controllers).

Table 3 presents the MSE for each case. These results are also presented in Fig. 3.29. From the results of the Table and the Figure, it is clear that

- The lowest value of the proportional gain $K_P = 0.1$ corresponds to the most inferior performance across all controllers
- The value of $K_P = 1$ corresponds to the most superior performance across all controllers
- As expected, the PID controller performs better than the other controllers, since it offers a greater number of parameters for the designer to tune.

Thus, the best performance is obtained by the PID controller with $K_P = 1$ and the worst by the P controller with $K_P = 0.1$.

Table 3.2 MSE for various values of K_P and all four controllers

K_P	Mean Squared Error (MSE)			
	P	PD	PI	PID
0.1	35.14	35.12	2.09	1.9
1	0.86	0.86	0.13	0.12
5	1.72	1.35	0.77	0.77
10	2.16	2.14	1.11	1.15

Figures 3.21 to 3.22 drill down on the performance of the best performing PID controller with $K_P = 1$. The x-axis of all graphs represents the waypoint number.

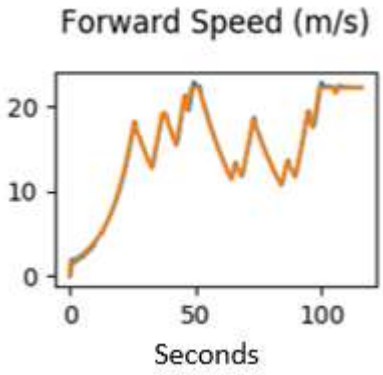

Figure 3.21 The relationship between desired (orange) and actual (blue) speeds – best case

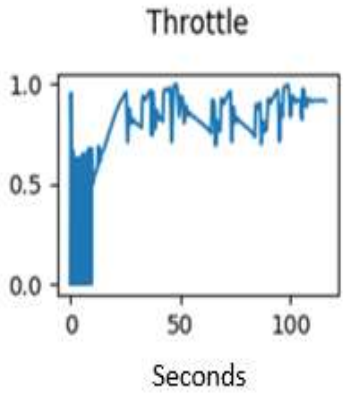

Figure 3.22 The angle (pressure) of the throttle pedal – best case

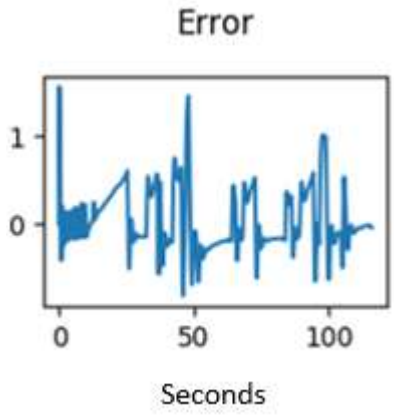

Figure 3.23 The error between the desired and the actual speed – best case

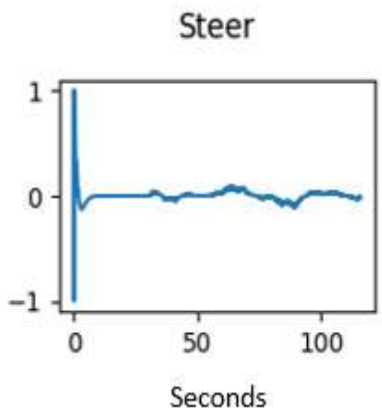

Figure 3.24 The steering angle – best case

From Fig. 3.21 it is evident that the actual speed follows the desired speed very closely. This is validated by the very limited values of the speed error (in m/s) displayed in Fig. 3.23. At the start of the trajectory, the speed error is higher and, thus, the throttle and the steering commands assume significant values until the transition is complete (Figs. 3.22 and 3.24, respectively).

Figures 3.25 to 3.28 drill down on the performance of the worst performing P controller with $K_P = 0.1$. In this case as well, the x-axis of all graphs represents the duration (in seconds).

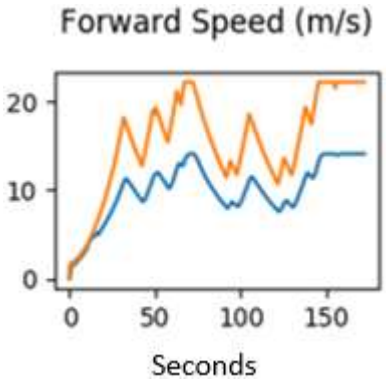

Figure 3.25 desired (orange) and actual (blue) speeds – worst case

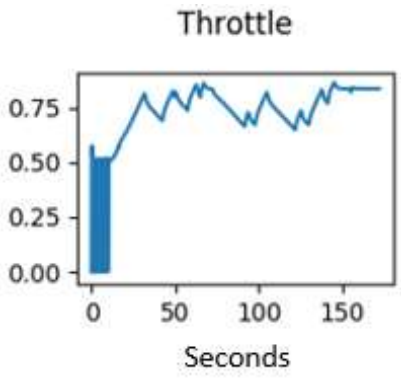

Figure 3.26 The angle (pressure) of the throttle pedal – worst case

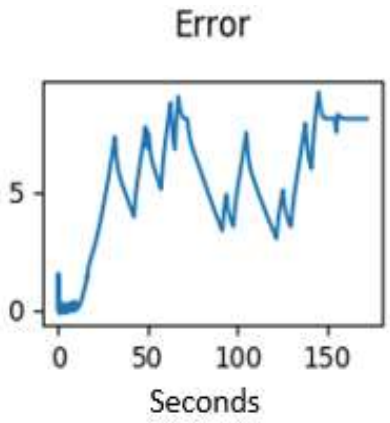

Figure 3.28 *The error between the desired and the actual speed – worst case*

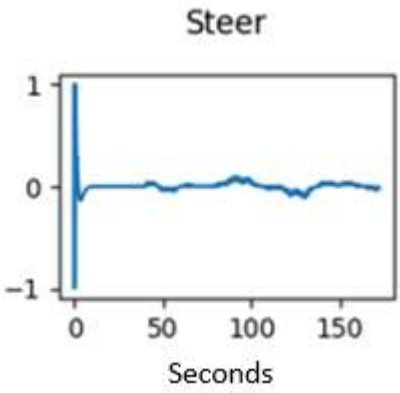

Figure 3.27 The steering angle - worst case

In Fig. 3.25 the desired speed and the actual speed have a significant difference, although, the pattern of the forward actual speed follows almost the pattern of the forward desired speed.

Figure 3.26 shows that the pressure on the throttle pedal is less than the previous case (best case). That's why the actual forward speed is in this case lower than the forward desired speed.

Concerning the error (Fig-3.27) it is obvious that there is a large deference between actual forward speed and the desired forward speed at any waypoint.

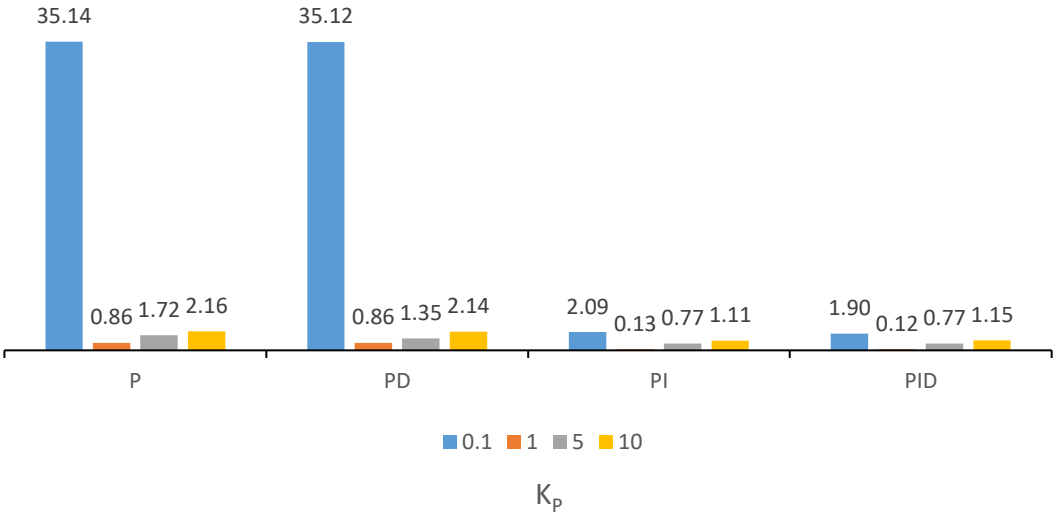

Figure 3.29 MSE for the various K_P values for all four controllers

From Fig. 3.29 it can be seen that the PID and PI controllers have similar performance. One may think that this is possibly due to the small value of $K_D = 0.01$ in the PID controller. This is investigated further below.

Case 2: Investigating the controller gains K_D and K_I for the PD and PI controllers

The main goal of case 2 is to assess the performance of the autonomous vehicle under various values of the gains K_I and K_D for the PI and PD controllers, respectively. The values tested are those of Table 3.3 and in all cases K_P assumed its nominal value of 1.

Table 3.3 K_I and K_D values

K_D	K_I
0.001	0.02
0.05	1
0.1	2

Thus, the procedure for this case included six (6) experiments. Tables 3.4 and 3.5 present the MSE results for the two cases. From Table 3.4 it is clear that MSE remains

almost the same for each value of K_D. Thus, the suspicion that the low value of K_D is the cause of the PID and PI controllers displaying the same performance in Case 1 above, is not true.

In the PI controller case, the value $K_I = 2$ leads to a very high value of MSE, indicating instability. This is validated by the simulation results. In this case indeed the control system entered instability, and the violent oscillation of the throttle, the steering, and the speed resulted in the vehicle getting completely off course (see Figs 3.30 and 3.31).

Table 3.4 MSE for the PD controller for different K_D values

K_D value	MSE
	PD
0.001	0.90
0.05	0.88
0.1	0.87

Table 3.5 MSE for the PI controller by different K_I values

K_I value	MSE
	PI
0.02	0.29
1	0.16
2	217.75

Figure 3.31 The autonomous vehicle just before getting off course (Screenshot from DeOPSys Lab PC)

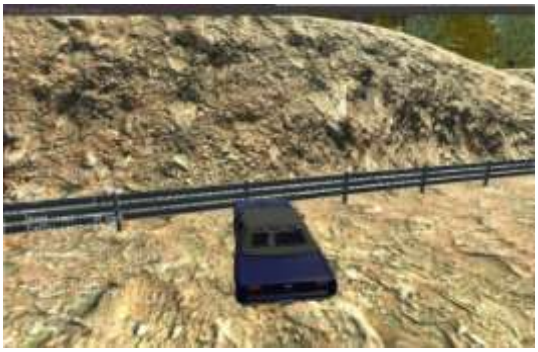

Figure 3.30 The autonomous vehicle end position (Screenshot from DeOPSys Lab PC)

The variation of the various parameters in this interesting case is presented in Figs. 3.32 to 3.35

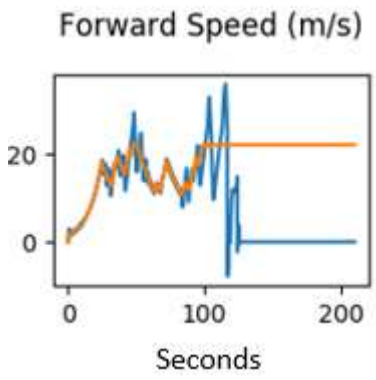

Figure 3.32 The relationship between desired speed and actual speed

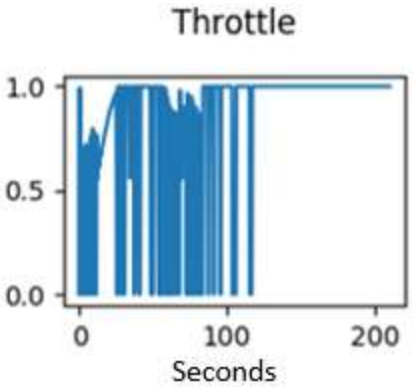

Figure 3.33 The pressure of the throttle pedal

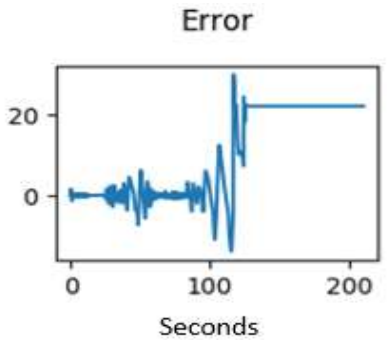

Figure 3.35 The difference of the desired and the actual speed

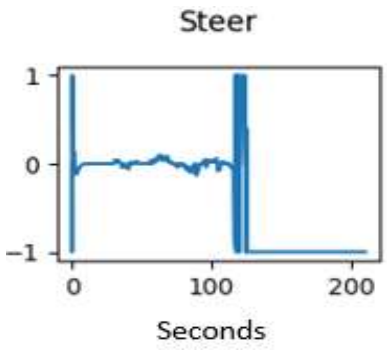

Figure 3.34 The steering angle

From Fig. 3.32 it is evident that the system becomes unstable, the amplitude of the speed oscillation increases, and the speed goes to 0 after the vehicle stops off course. The error of Fig. 3.35 presents similar characteristics.

This instability is evident from Figs. 3.33 and 3.34 (especially the latter one). In the former Figure the throttle oscillates between its limits (0 and 1) and locks at 1 at the end of the trip.

In Fig. 3.35, the steering angle oscillates between its limiting positions (-1 and 1) in a fruitless attempt to keep the vehicle close to the desired path.

Major takeaways from the above experimental study include the following:

- in case 1, it is observed that the PID controller with proportional gain of 1 had a superior error behavior. This leads to better vehicle behavior with respect to longitudinal control
- in case 2, it is clear that MSE is similar for all examined values of the derivative gain for the in PD controller. On the contrary, in the PI controller MSE is very sensitive to the integral gain. High gain values may lead to instability.

Chapter 4 Object detection in autonomous vehicles

4.1 Introduction to neural networks

The main perception tasks in autonomous vehicle self-driving is to recognize static and dynamic objects. To detect these objects most automakers investing in the autonomous vehicles (AVs) use cameras. As already discussed in Section 2.3, the camera is a passive sensor which provides detailed information regarding objects in the environment. This visual information is useful to understand the scene by performing tasks such as object detection, segmentation and identification. Through such processes, an AV may detect traffic signs or signals, other vehicles, pedestrians, driving lanes and other objects.

AV perception is based on artificial neural networks that comprise layers, or groups of so-called neurons, which relate to other layers. The task of these layers is to convert the input data to outputs; this is done by computing the weighted sum of inputs and by normalizing using the activation functions that are allocated to the neurons (Mani, 2019).

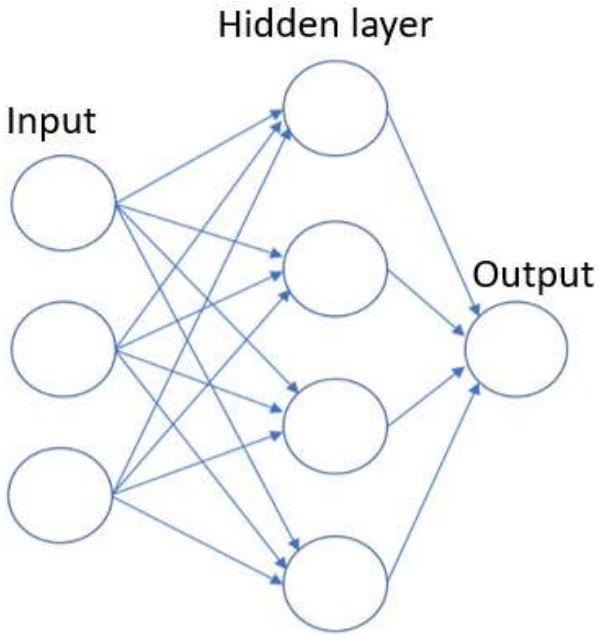

Figure 4.1 A neural network with two layers

Feedforward neural networks

A feedforward neural network (FNN) is a basic model of deep learning. The main goal of an FNN is to approximate some function f. Thus, it defines a mapping from input x to output y through a function of x and θ

$$y = f(x; \theta) \tag{4.1}$$

For example, an FNN may receive an image as an input (x) and then using the network connectivity and the parameters θ may classify the image to a category y (i.e. car, pedestrian, etc.).

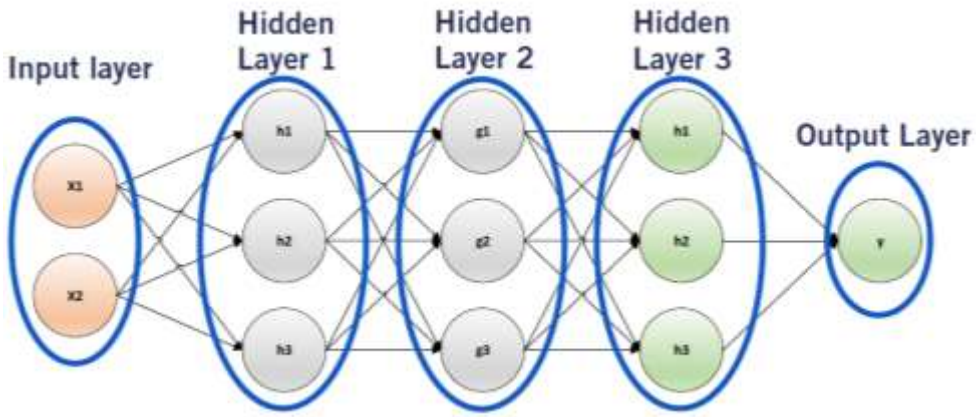

Figure 4.2 Four-layer feedforward neural network (Coursera (2019c))

In the four-layer feedforward neural network of Fig. 4.2 the input x can be a vector or a tensor, such as one that comprises the pixels of a photo image. Then, the input is processed by the first layer of FNN through with function $f^{(1)}(x)$. Likewise, the second hidden layer takes as an input the output of the first hidden layer and operates on this output through function $f^{(2)}(x)$; similarly, the output of the second layer passes through at the third hidden layer and function $f^{(3)}(x)$. The final layer obtains the output of the last hidden layer and converts it to the output y. This model is a Forward NN since the input data x are processed by the intermediate computations

(using the related functions f) to obtain the output y without feedback connections. Thus, the four-layer feedforward neural network may be represented by

$$f(x;\theta) = f^{(4)}\left(f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)\right) \quad (4.2)$$

where x is the input layer $f^{(1)}, f^{(2)}, f^{(3)}$ are the functions of the hidden layers and $f^{(4)}$ is the function of the output layer.

During the training process, the function of the neural network $f(x;\theta)$ should be tuned in order to represent the true function $f^*(x)$ by estimating the parameters of θ. The hidden layers of the neural network are the most important ones. Each of these layers transform the output of the previous layer h_{n-1} using a non-linear function g, which called activation function, as well as multiplicative weight matrix W and the bias b:

$$h_n = g(W^T h_{n-1} + b) \quad (4.3)$$

These weights and bias values are the learning parameters θ of the neural network. The activation function g may be one of ReLU, sigmoid, tan, Maxout Unit. For example, ReLU (Rectified Linear Unit) is used often as an activation function for FNN (see Eq. 4.4 and Fig. 4.1).

$$g(z) = \max(0, x) \quad (4.4)$$

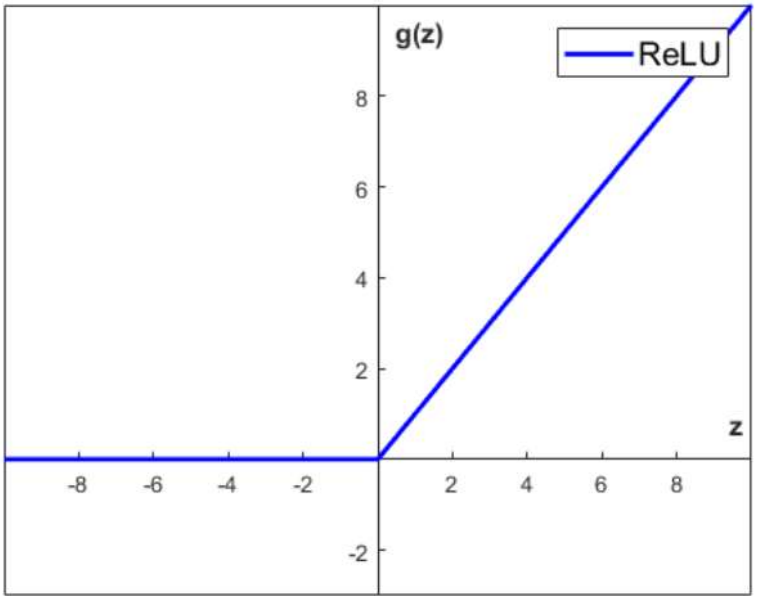

Figure 4.3 ReLU activation diagram

Feedforward neural networks are used for perception tasks associated with autonomous vehicle applications such as object classification; this process identifies with labels or bounding boxes objects in the picture. Furthermore, the object detection estimates the location as well as the objects in the area. The other task is the depth estimation, which called pixel-wise task. This task helps the autonomous vehicle to determine where the objects are by estimate the depth value for each pixel in the picture. However, the semantic segmentation specifies which class each pixel of picture belongs to.

4.2 Convolutional neural networks and object recognition

Convolutional Neural Networks (CNN) execute a great number of perception tasks for autonomous vehicles. A CNN is a special type of neural network suitable for processing data such as 1D time series, 2D pictures as well as 3D videos. For the purpose of this thesis we will discuss the two-dimensional case, which is central to image processing and object recognition (see Section 4.3).

Network architecture

The two main types of layers in a CNN are the convolutional layers and the pooling layers. For example, VGG 16, which is a CNN for classification and detection, receives

a picture as input and processes it through a set of convolutional layers, then through a pooling layer and this process continues until the fully connected layers FC(i) and the Softmax output layer (see the two equivalent representations of Fig. 4.4).

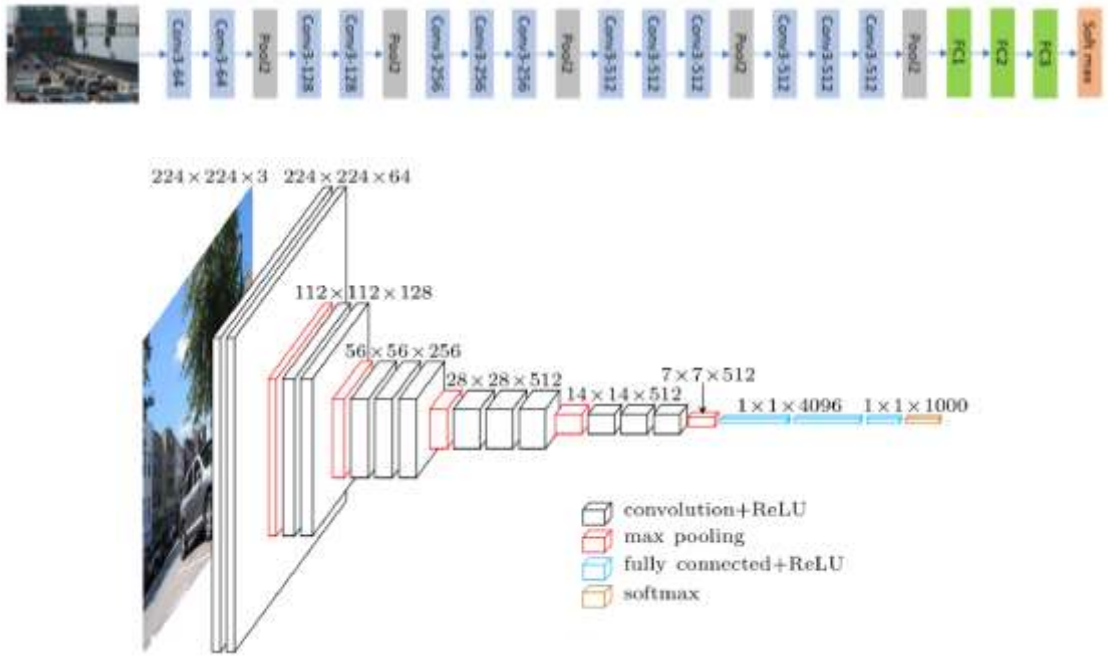

Figure 4.4 VGG 16 architecture (tryolabs, 2020)

For 2D object recognition the input picture is firstly processed using the VGG feature extractor which is built by alternate convolutional layers and pooling layers. VGG 16 accepts a 256x256x3 pixel image (3 representing the 3 RGB colors) and includes two convolutional layers with 64 filters each (for the meaning of a filter see next paragraph) followed by a pooling layer, where pooling is applied on a 2x2 pixel window with a stride of 2, thus reducing the height and width of the image by a factor of 2 (see also below for all these terms). Subsequently, the architecture includes two more convolutional layers with 128 filters followed by a pooling layer of the same functionality, and so on, till a set of fully connected layers (FC1, FC2, FC3 see Fig. 4.4). The output of the concatenation of the convolutional and pooling layers is a feature map that contains the features of the image. The fully connected layers perform classification of the significant features contained in each bounding box of the image (for the bounding boxes see the sub-section on object recognition below). Finally, for

the final detection the Softmax output layer is used which is a vector with a single score per class. The highest score usually defines the class of the contents of each bounding box.

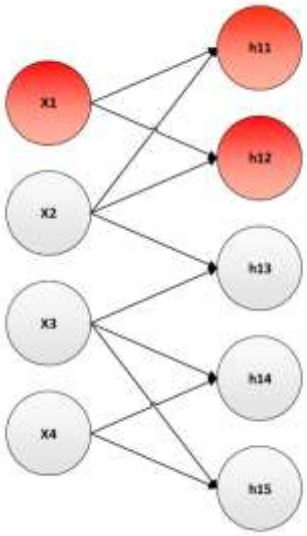

Figure 4.5 Sparse connectivity between the nodes of CNN (Coursera(2019c))

Convolution and pooling operations

Consider a picture (image) inserted to the CNN that comprises pixels in a matrix form of $M \times N \times 3$ array of pixels. Every pixel represents the projection of a 3D point into the 2D picture plane. The width of the input picture is in the horizontal dimension, the height is its vertical dimension and the depth is the number of channels. When the input is a colored one, it relates to three channels: Red, Green and Blue. In a grayscale picture the data is a matrix of dimension $M \times N$. At the boarder of the picture zero pixels are added, an operation called padding, which is important to preserve the picture's form in order to execute the convolution operations. The zero padding helps the information at the borders not be lost after each convolutional layer.

The convolutional layers use cross-correlation much as a linear operator. In this case there is sparse connectivity between the nodes of previous and the next layer (see Fig. 4.5). Each convolution operation uses convolutional filters or kernels. The filters (kernels), which are set during initialization and cannot be modified afterwards, act as

feature detectors taking values during the training process. For example, a kernel could recognize the horizontal edges in an image. The resulting array is called feature map.

The kernel is a matrix which scans (or slides) across the picture and multiplies the matrix of the input picture. A typical choice is to keep the kernel size at 3×3 or 5×5. In our case we choose three 3×3 Kernels, each corresponding to a channel RGB (Red, Green, Blue).

Every kernel includes a set of weights and a single bias. The cross-correlation operation is given by the following equation

$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{4.5}$$

where I is the input matrix (the 2D picture), K is the kernel, i, j are the pixel indices on which the convolution is applied, and m and n are the width and the height of the kernel (Ian Goodfellow, 2016).

In the case of the 2D- colored picture with three input channels, each one of them is convoluted with the corresponding kernel; in this case Eq. (4.5) is executed three times, one for the red channel, one for the green channel and one for the blue channel.

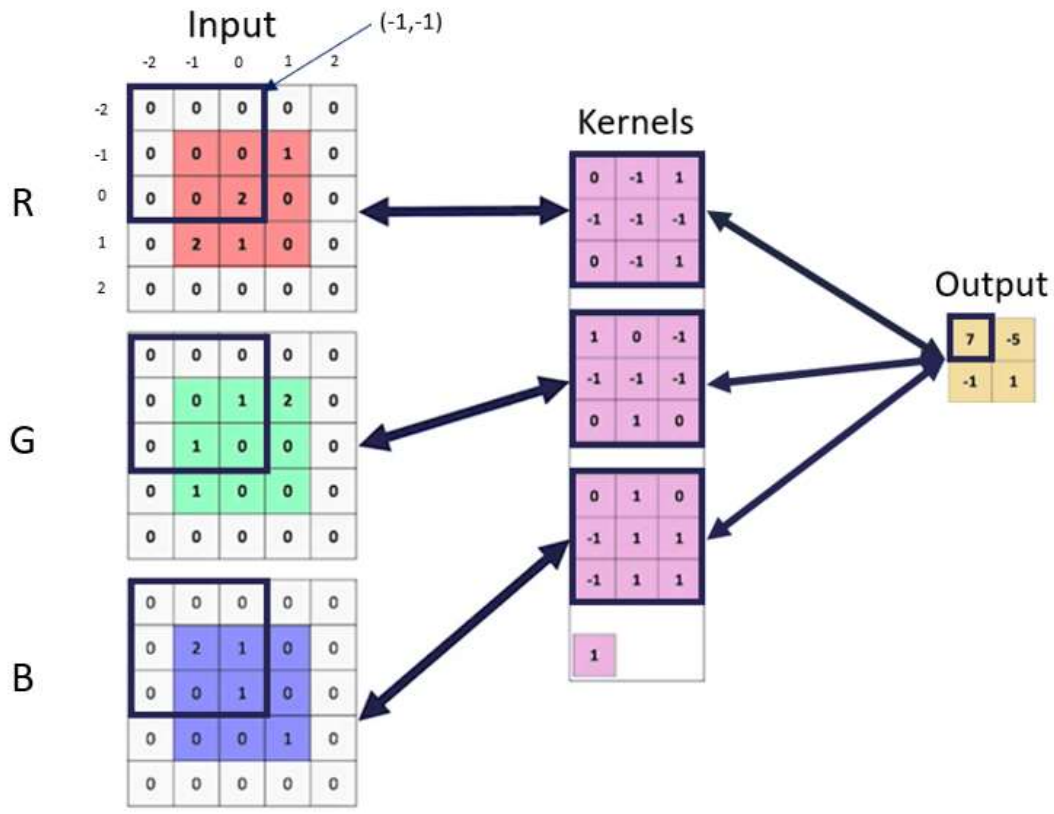

Figure 4.6 A 3×3 kernel (per channel) moves over the input to generate the output (Coursera, 2019c)

Referring to Fig. 4.6, the padding is represented by the zeroes at the edges of the matrix of each channel. The first operation that involves the red channel and the corresponding kernel is

$$(0x0) + (0x(-1)) + (0x1) + (0x(-1)) + (0x(-1)) + 0x(-1)) + (0x0) + (0x(-1)) + (2x1) = 2 \quad (4.6)$$

Similarly, the first operation involving the green channel with its kernel is

$$(0x1) + (0x0) + (0x(-1)) + (0x(-1)) + (0x(-1)) + (1x(-1)) + (0x0) + (1x1) + (0x0) = 0 \quad (4.7)$$

Finally, the first operation involving the blue channel with its kernel is

$$(0x0) + (0x1) + (0x0) + (0x(-1)) + (2x1) + (1x1) + (0x(-1)) + (0x1) + (1x1) = 4 \quad (4.8)$$

The bias is 1.

Adding the results of the three operations with the bias, the output result is 7. To find the other outputs, the dark blue box moves by a stride of 2 pixels horizontally, vertically, and horizontally again (for each color).

In the VGG 16 convolutional network, each convolutional layer includes a large number of kernels (filters) e.g. 64 filter (sets of three) in the first convolutional layer of Fig. 4.4.

The other structural element of convolutional networks is the pooling layer, which is important for object recognition. Max pooling is the most common pooling method for convolutional networks. It summarizes the output using the max function. A similar process as the one described above is used. In the case of Fig. 4.6 a 2×2 max filter is used and a stride of two. Therefore, the output is an array 2×2.

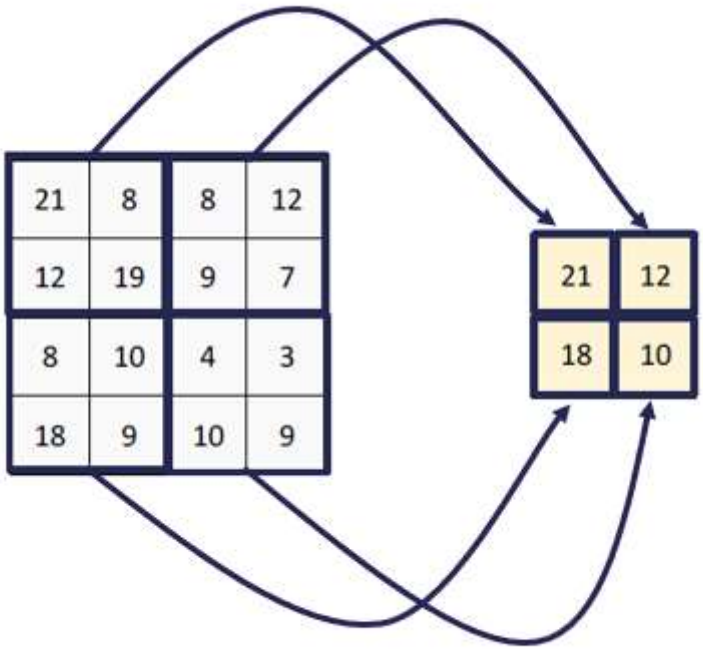

Figure 4.7 Max pooling (Coursera, 2019c)

In the first 2×2 part of the image the max function is max (21, 8, 12, 19) so the max of these four number is the 21. Thus, 21 goes at the first element of the output and the same process continues in the next three outputs.

In the VGG 16 convolutional network, pooling is performed as above; i.e. in a 2x2 pixel sliding window with a stride of 2. Thus, the dimensions of the image are reduced by 2 after each pooling step.

In VGG 16, the three fully connected layers (FC1, FC2 and FC3) have different depths: The first two have 4096 channels each and the third executes classification and includes 1000 channels (one for each class). Finally, the Softmax layer produces the output by applying the Softmax function as activation function (see Eq. 4.11 in Section 4.3). The Softmax function is a form of logistic regression which normalizes the input value into a vector of values that follows the probability distribution between zero and one. The output of the Softmax function is equivalent to a categorical distribution, which is the probability that any of the classes are true.

Training the network

For training the network, first we define the input picture as x and $f^*(x)$ the bounding box locations and class (for the bounding boxes see below). The first step to be done is to evaluate a loss function $L = L(f(x;\theta), y)$ that quantifies the similarity between the predicted bounding boxes and the ground truth bounding boxes (see also the object detection paragraph below). The result of the loss function is provided to the optimizer which outputs a new set of parameters θ. In order to adjust the parameters of the convolutional network the most common method used is the gradient decent. Gradient decent it is an iterative optimization procedure which uses the first order derivative to improve the parameters θ. Once the iterative process begins the algorithm calculates the gradient of the loss function with respect to theta from the Eq. (4.9). The gradient decent of the training loss function with respect to parameter vector θ can be written as

$$\nabla_\theta J(\theta) = \nabla_\theta [\frac{1}{n}\sum_{i=1}^{n} L[f(x_i;\theta), f^*(x_i)]] = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L[f(x_i;\theta), f^*(x_i)] \qquad (4.9)$$

The parameters θ are updated based on the computed gradient from

$$\theta \leftarrow \theta - \varepsilon \nabla_\theta L(f(x;\theta), y) \qquad (4.10)$$

where the learning rate ε is set in an appropriate a way to avoid taking too large or too small steps in the parameters space. For example, a large learning rate ε may cause the process to diverge and a small rate may cause slow convergence. To terminate the algorithm of gradient decent, a stopping criterion is defined, and the algorithm returns the last set of parameters.

Object recognition

This operation identifies objects such as signs, traffic lights etc. in a picture. It usually detects objects independently in each picture. In this sub-section we will describe the Faster R-CNN approach, an efficient object recognition method/architecture shown in Fig. 4.8.

Figure 4.8 Faster R-CNN architecture (tryolabs, 2020)

After insertion, the picture is processed using a feature extractor (orange shape in Fig. 4.8). This extractor is the most expensive part of a 2D object detector. The output of the extractor commonly has much lower width and height than the input image but much greater depth. Typical feature extractors include VGG, ResNet and Inception. In this discussion, the feature extractor is based on at the convolutional and pooling layers of the VGG 16 classification network discussed above. Note that each convolutional layer of VGG 16 generates abstractions of the layer's input. Thus, each layer focuses on different shapes. The output of the convolutional and pooling layers is a feature map that has encoded the information (features) of the image along its depth. The location of the features is maintained with respect to the original image. Note that Faster R-CNNs use an intermediate output of VGG 16; for example, the 14x14x512 tensor of Fig. 4.4. This is because VGG 16 is used only for feature extraction in Faster R-CNN and not for final classification. The fully connected layers and the Softmax functionality of VGG 16 (see Fig. 4.4) are not used in Faster R-CNN.

The next step in the architecture of Fig 4.8 concerns the identification of regions of interest in the image in order to classify these regions. This is done by identifying appropriate bounding boxes of rectangular shape in the image that contain features; bounding boxes are inevitably of different sizes and aspect ratios. This process starts by centering at each point of the (14x14 point) layer of the feature map a set of anchor boxes of different sizes and aspect ratios (several different anchor boxes are centered at each point). Usually the anchor boxes are defined by their size (e.g. 64, 128px and 256 pixels) and the aspect ratios (e.g. 0.5, 1 and 1.5). The anchor boxes reference the original image (picture). In the case of the 14x14x512 feature map of VGG 16, the

spacing of the centers of anchor boxes in the original image will be 16 pixels, since 14x16=256, the width and height of the original image.

The feature map augmented by the bounding (anchor) boxes is the input to the Region Proposal Network (RPN) which is used to identify proposed objects. To do so, it

- determines the probability of each anchor box to contain an object or not (i.e. object or background)
- adjusts the shape and size of each anchor box that contains an object to better fit that object

RPN comprises three convolutional layers: a layer with 512 channels and 3×3 kernel size and two parallel convolutional layers the channel number of which depends on the number of anchors *k* per point and an 1×1 kernel (see Fig. 4.9). For each anchor: a) the output of the classification layer provides the score for the anchor containing an object and the score for the anchor containing just background; b) there are four outputs resulting from the adjustment layer $\Delta_{x_{center}}$, $\Delta_{y_{center}}$, Δ_{width} and Δ_{height} that are applied to the anchors to better fit the objects they contain.

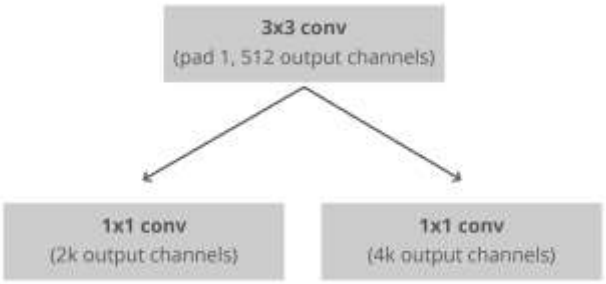

Figure 4.9 RPN architecture, where the two parallel layers perform classification, and bounding box refinement. k is the number of anchors per point (tryolabs, 2020)

The next step after RPN is to classify the identified object proposals into categories. A method to do this is to cut the convolutional feature map using each object proposal and then resize each piece to a fixed size tensor of dimensions $14 \times 14 \times convdepth$ using interpolation. Subsequently, max pooling is applied to this tensor

(corresponding to an object proposal) with a 2×2 kernel to obtain the final $7 \times 7 \times convdepth$ feature map for each object proposal.

The final step of the Faster R-CNN architecture is to classify each object proposal. This is performed by using a Region-based convolutional neural network (R-CNN) which

- classifies the proposals into one of the predetermined classes as well as a background class for inferior suggestions
- adjusts the bounding box of the proposed object based on the predicted class.

This is performed by using two fully connected layers for each proposal, one for classification and the other for box size adjustment.

Extension of 2D object recognition to 3D[9]

An autonomous vehicle is required to understand the scene in 3D in order to be capable to safely cross its environment knowing where the pedestrians, vehicles lanes and signs are. For that reason, 2D object detectors should be extended to 3D. The typical method involved is to employ LiDAR point clouds. The 2D bounding box in a picture, the LiDAR point cloud, and the inverse of the camera projection matrix are used to project the corners of the bounding box as rays into the 3D space. This intersection of these lines is called a frustum and commonly include points in 3D that correspond to the object in the picture. Subsequently, a small neural network is used to predict the seven parameters required to define the bounding box in 3D.

Another important issue in using 2D object detection in 3D is object tracking. Object tracking involves monitoring a sequence of detections of the same object and synthesizing a trajectory that determines the object motion over time. In addition, object tracking incorporates a predicted position commonly through known object dynamic models. Object tracking requires a set of assumptions limiting how fast a scene changes. For example, a key assumption is that the camera and tracked objects cannot move instantly to different locations in an unrealistically short time.

[9] (Forsyth & Ponce, 2011) and (Qi, et al., 2018).

Based on these assumptions, the detected object in an image accompanied by appropriate speed vectors that are used to predict where the object will end up in subsequent images. Thus, the first step (prediction) is to define the position and speed in picture space. Every object will have a motion model that updates its state. For example, the constant speed motion model may be used to move each bounding box to the new locations. After this first step, every detection is correlated to the prediction by calculating the Intersection over Union (IoU) between all measurements and the prediction. Each measurement compares to the prediction, and the one with the highest IoU is assigned to the prediction. The final step consists of using a Kalman filter to merge the measurement and prediction updates. This filter updates the total object state, including speed and position, which can be used in a subsequent prediction step. For further details on this complex process refer to (Barfoot, 2019)

The traffic signs and signals should be detected from a long distance in order for the autonomous vehicle to react suitably. For that reason, the traffic signs and signals occupy a very limited number of pixels in the picture. Furthermore, traffic signs include multiple classes that should be identified (and thus classified). On the other hand, traffic lights change their state as the autonomous vehicle moves. To detect traffic signs and lights, two stages are followed. The first one creates a special output class termed agnostic bounding boxes, which identifies all traffic signs in the picture without defining which class every bounding box belongs to. The second stage, processes the bounding boxes from the first stage and categorizes them into categories like stop signs, yellow, red and green signals etc.

4.3 Visual perception for autonomous vehicles: Semantic segmentation and case study

Having set the foundations for CNN and object recognition in Section 4.2, in this Section we focus on relevant tasks for autonomous vehicles. First, we introduce the process of semantic segmentation, which is central to visual perception. Secondly, we present a case study that concerns the visual perception of an autonomous vehicle. The case study uses the output of semantic segmentation to

- Determine the drivable space of the autonomous vehicle

- Recognize the lanes
- Recognize the valid objects
- Estimate the distance of the valid objects from the autonomous vehicle

This process permits the autonomous vehicle to recognize where it can move on the road, as well as what objects are into its view, thus supporting the vehicle's decision making. Particularly, estimating the drivable space for an autonomous vehicle is important for safe operation.

4.3.1 Semantic segmentation

Semantic segmentation is a picture (image) processing process that detects and recognizes objects at pixel level. The semantic segmentation problem identifies pixels belonging to predetermined categories, such as static objects e.g. sidewalks, roads, traffic lights and traffic signs, as well as dynamic objects e.g. vehicles, cyclists, and pedestrians. The semantic segmentation neural network accepts an image as input and examines each pixel separately; the output is a vector of class scores per pixel. A pixel becomes a part of the class with the highest score. For that reason, it is necessary for the estimator to assign the highest result to the correct class for each pixel in the picture. For example, a vehicle pixel should have a very high vehicle result and significantly lower results for other classes (Everingham, et al., 2009) (Badrinarayanan , et al., 2017).

The semantic segmentation problem can be modeled as a function approximation problem and can be addressed by the architecture shown in Fig. 4.10.

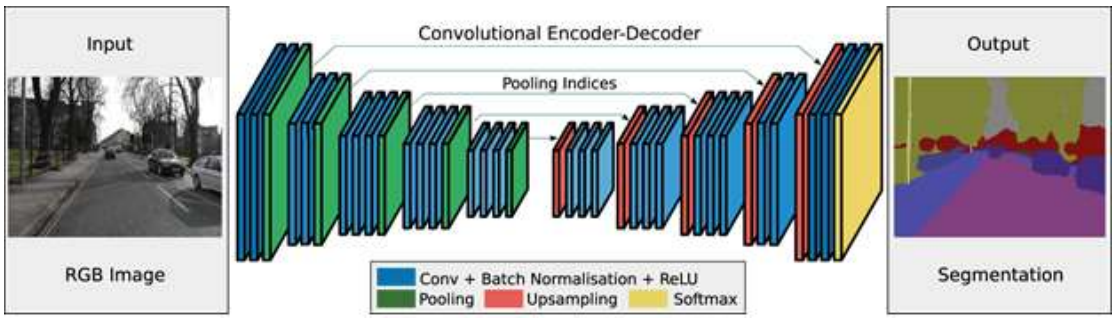

Figure 4.10 Basic architecture of semantic segmentation (Playment, 2018)

The input is the RGB image, and the output is the semantic segmentation, that is the classification of each pixel in the image in a pre-determined class (signified in the above example by a separate color). The approach to obtain the output comprises three major stages. The first stage generates the feature map from the RGB image. This may be done by the use of convolutional neural networks as the ones used in object detection, for example VGG 16. As discussed in Section 4.2, the feature map corresponds to the activation of different sections of the picture, where high activation means that a determined feature was found such as road, vehicle etc. (see Section 4.2). The dimensions of the feature map tensor may be 14x14x512 as discussed in Section 4.2.

The second stage upsamples the downsampled feature map back to the original picture resolution. For upsampling, the nearest neighbor process is used on a $2x2$ image patch of the downsample feature map as follows (the color of the image patch represents the different values of each pixel.)

- The nearest neighbor upsampling produces an empty initial upsample mesh.
- Every pixel in the upsample mesh is filled with the value of the nearest pixel in the original image patch (see Fig. 4.11).
- This procedure is repeated until all the pixels in the unsample mesh are filled with values from the image patch.

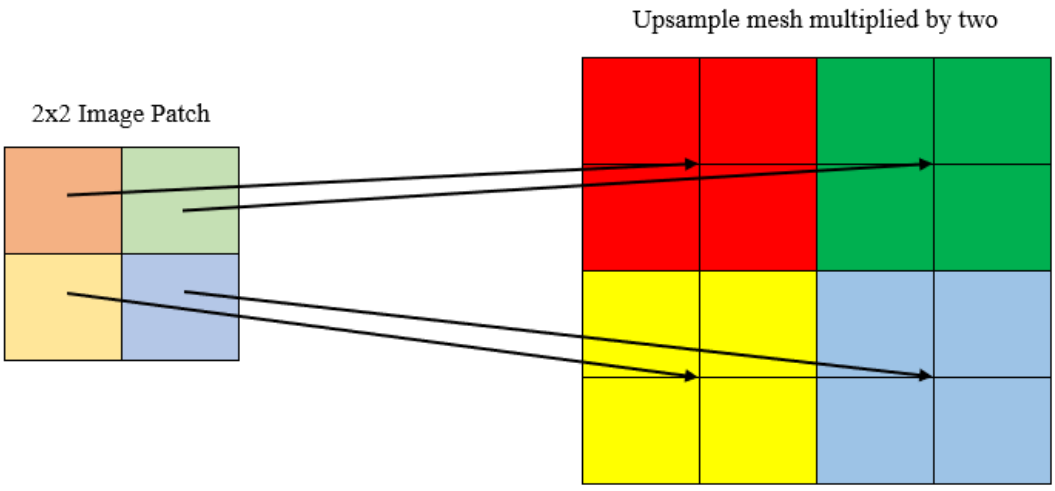

Figure 4.11 Upsampling layer

For better results many researchers use a procedure called feature decoder. This process is the opposite of the feature extractor process, which downsamples the resolution of the image. Instead, the feature decoder uses upsampling layers followed by convolutional layers (see Fig. 4.10). The depth of the semantic segmentation neural network is controlled by how many filters are defined for both the feature extractor and feature decoder. Note that the upsampling convolution block is referred as deconvolution block.

- In the first deconvolution block, the feature map is unsampled to twice the input resolution (28x28x512). The upsampling layer is followed by convolutional layers which are used to correct the features in the upsampled feature map with already existing data in which the neural network has been trained (learnable filter banks). These corrections very often determine the recommended smooth boundaries.
- As the feature map passes through to the rest of the decoder the output feature map becomes of similar resolution to the input image.

After, the processing of the feature extractor and feature decoder, the output passes through the third stage, the Softmax layer. The Softmax output layer is used more often as an output classifier to provide values as close to one as possible for the correct class and as close to zero as possible for the other classes for every pixel. The index of the maximum score is the recommended output representation:

$$Softmax(z_i) = \frac{\exp\,(z_i)}{\sum_j \exp\,(z_j)} \tag{4.11}$$

where z_i is a vector transformed into a discrete probability distribution of the class and i is the index of the class.

The segmentation output includes a plot index for every pixel. To visualize the semantic segmentation output we define these mapping indices and their corresponding colors, as provided in the following table (see also Fig. 4.12).

Table 4.1 Mapping indices and visualization colors

Category	Mapping index	Visualization color
Background	0	Black
Buildings	1	Red
Pedestrians	4	Teal
Poles	5	White
Lane markings	6	Purple
Roads	7	Blue
Side walks	8	Yellow
Vehicles	10	Green

Figure 4.12 Semantic segmentation visualization colors

4.3.2 Case study: Estimation of the drivable space

As discussed above, the case study uses the output of semantic segmentation to

- Determine the drivable space of the autonomous vehicle
- Recognize the lanes

- Estimate the distance of objects from the autonomous vehicle using the filtered 2-D object detection.

To determine the drivable space, we first identify the ground plane. This is done using as inputs the semantic segmentation data to evaluate the equation of the ground plane (Forsyth & Ponce, 2011).

Subsequently, the process of the case study estimates the lane boundaries using again the semantic segmentation output.

Finally, an object detection neural network is used which recognizes the object class. This network is its high recall but low precision. Recall (sensitivity) is determined by the number of truly positive results divided by total sum of truly positive and false negative results. On the other hand, precision (positive predictive value) focuses only on correct positive predictions. Consequently, this network provides some incorrect results. For this reason, the output of semantic segmentation is used again to filter out the errors from the object detection network

To implement this case study, the Jupyter programming environment and pictures from the CARLA simulator were used. Jupyter allows to create documents that contain computer code, equations, visualizations etc. Also, supports over 40 programming languages including Python, R, and other. Furthermore, the neural network for object detection was VGG 16 which is mentioned at chapter 4.2. The output of this neural network is included to a NumPy (python library) array.

Estimating the drivable space

Estimating the drivable space is equivalent to estimating pixels belonging to the ground plane in the scene. To do so, firstly we estimated the x_c, y_c, z_c coordinates of every pixel in the picture. To calculate these coordinates, we used the following equations:

$$x_c = \frac{(u - c_u)z}{f} \tag{4.12}$$

$$y_c = \frac{(v - c_v)z}{f} \tag{4.13}$$

$$z_c = depth \tag{4.14}$$

Where u, v are the coordinates of each pixel in the picture, c_u, c_v and f are the intrinsic calibration parameters such as the camera geometry and the camera lens characteristics, as found in the camera calibration matrix K.

$$K = \begin{pmatrix} f & 0 & u_c \\ 0 & f & v_c \\ 0 & 0 & 1 \end{pmatrix} \tag{4.15}$$

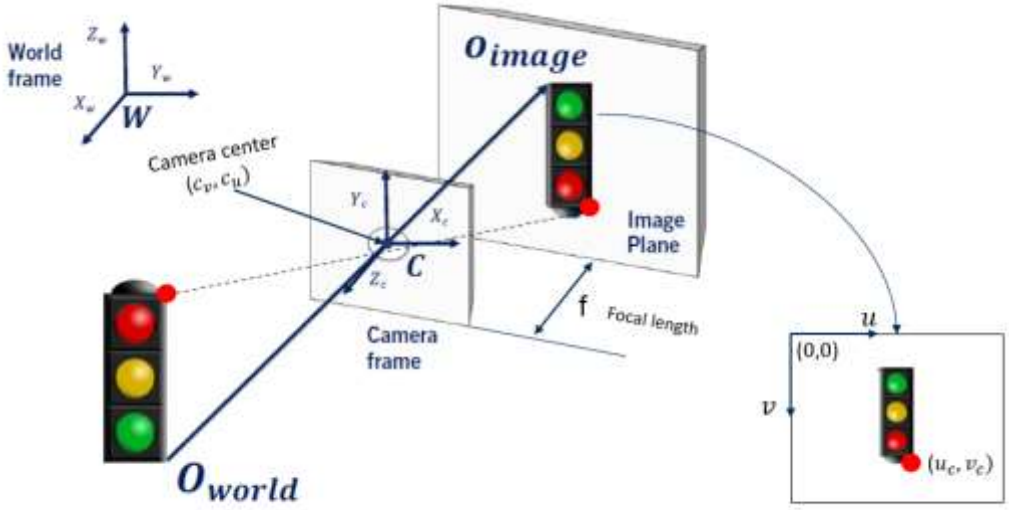

Figure 4.13 Pinhole camera model (Coursera (2019c))

Using the pinhole camera model of Fig. 4.13:

- The focal length f is the distance between the camera center and the image plane
- The piercing point (c_u, c_v) are the coordinates of the camera frame center
- The piercing point (u_c, v_c) is the intersection of the optical axis with the image plane provided in pixel coordinates.
- The z_c coordinate is the optical axis of the camera (with points in front of the camera in the positive z_c direction). Depth is the distance, along the z axis, between the nearest and farthest objects in the image that appear acceptably in focus.

After estimating the coordinates of each pixel, we used the RANSAC (Random Sample Consensus) algorithm to estimate the ground plane (Forsyth & Ponce, 2011). The RANSAC algorithm is a general parameter estimation approach, for robust fitting of models in the presence of many data outliers. The algorithm comprises the following six steps:

1. Choose randomly a minimum of 3 points and obtain their x_c, y_c, z_c coordinates from Eqs. (4.12), (4.13), (4.14)
2. Calculate the ground plane model using the 3 selected points with equation

$$ax_c + by_c + cz_c + d = 0 \quad (4.16)$$

   by using the function compute_plane() (Coursera (2019c))

3. Calculate for every pixel and its (x_c, y_c, z_c) coordinates the distance of the respective point from the ground plane and with the help of function dist_to_palane (Coursera (2019c)) which computes the distance , and calculates the number of inliers based on a distance threshold. This is the end of an iteration
4. Check if the current number of inliers is larger than the calculated number of inliers in the previous iteration and keep the inlier set with the largest number of points
5. Repeat this process for a thousand iterations or until the number of inliers is larger than the minimum number of outliers
6. Recompute and return a plane model using all inliers in the final inlier set

```
Function ransac_plane_fit(xyz_data)

    Step 0: Set the thresholds of RANSAC

    maximum number of iterations ←100

    minimum number of inliers ← 10,000

    maximum distance from point to plane for point to be considered ←0.3

    maximum_inliers_counter ←0

    maximum_inliers_set_index←None
```

```
For each number of iterations do
    Step 1: Choose a minimum of three points from xyz_data randomly
    index ←np.random.choice(range(xyz_data.shape[1]), 3, replace=False)
    current_data← xyz_data[:, index]
    Step 2: Compute the plane model
    plane_parameter←compute_plane(Current data)
    Step 3: Find the number of inliers
    distance_list ← dist_to_palane(plane_parameter.T, xyz_data[0, :].T,…
    xyz_data[1, :].T, xyz_data[2, :].T)
    Step 4: Check if the current number of inliers is larger than the
    calculated number of inliers in the previous iteration and keep the inlier
    set with the largest number of points
    inliers_count ← np.sum(distance_list < distance_threshold)
    If inliers_count > maximum_inliers_count then
        maximum_inliers_count←inliers_count
        maximum_inliers_set_idx = np.where(distance_list  <…
        distance_threshold)[0]
    If inliers_count > minimum number of inliers OR i > maximum…
    number of iterations
        Break
End For
Step 6: Recompute and return a plane model using all inliers in the final
inlier set
final_data = xyz_data[:, max_inliers_set_idx]
output_plane = compute_plane(final_data)
```

```
        return output_plane

End Function
```

Figure 4.14 RANSAC pseudocode

The pseudocode for the algorithm is provided in Fig. 4.14. The code that implements this process is provided in Appendix B. For the semantic segmentation input of Fig. 4.12, the ground plane computed is given by the following array:

$$Ground\ Plane: [0.02, -1.00, 0.01, 1.4] \quad (4.17)$$

where $a = 0.02, b = 1.00, c = 0.01$ and $d = 1.4$

Lane estimation

For lane estimation, we used the output of semantic segmentation for the current lane the autonomous vehicle is using. For reliable implementation, this task was divided in two subtasks: lane line estimation and post-processing. The latter subtask consists of horizontal line filtering and similar line merging for those lines that are not part of the drivable space of the autonomous vehicle.

In the first subtask, we examine any line that is characterized as a lane boundary in the output of semantic segmentation. These lines are called 'proposals' and to examine them three steps were followed:

1. Create a picture that includes the pixels corresponding to the lane boundaries (as characterized by semantic segmentation)
2. Implement the edge detection process on the above lane boundary picture to derive the drivable space of the autonomous vehicle. To do so, firstly, we extract a binary mask of pixels which belong to classes that appear as lane separators. The binary mask defines a region of interest (ROI) of the original image. Mask pixel values of 0 nominate the image pixel that is part of the background and mask pixel values 1 nominate the image pixel that belongs to the region of interest (Jain, et al., 1995). Thus, these classes include lane marking lines as well as road rails (if any). Subsequently, using the binary mask we employ an edge detector. In this thesis we used the *canny* edge detector which is a multi-process algorithm that can detect edges in the presence of

noise (CANNY, 1986). A Gaussian filter is used to normalize the image to reduce the noise as well as any unwanted details and textures. In this way all the edge elements are kept, while most of the noise is eliminated.

3. Implement the line estimation process using the output of the edge detection process. This output contains pixels classified as edges, which are used to estimate the lanes. To detect lines in the output edge map, we used the Hough transform line detection algorithm (Forsyth & Ponce, 2011),which is capable of detecting multiple lines in the edge map. The Hough transform produces a set of lines that connect pixels which belong to edges in the edge map. The minimum length of the required lines can be set as a hyperparameter to force the algorithm to detect only lines that are very long to be part of lane markings.

The pseudocode of the algorithm that implements this process is provided in Fig. 4.15 below. The code is provided in Appendix B

For the semantic segmentation input of Fig. 4.12, the algorithm's output consisted of 1 line and is shown in Fig. 4.16.

```
Function estimate_lanes_lines (segmentation_output)

        Step 1.1: segmentation==6 OR segmentation==8
        Step 1.2: Perform edge detection using cv2.Canny()
        Step 1.3: Perform line estimation using cv2.HoughLinesP()
        Return lines

End function
```

Figure 4.15 Pseudocode for the algorithm that detects lines

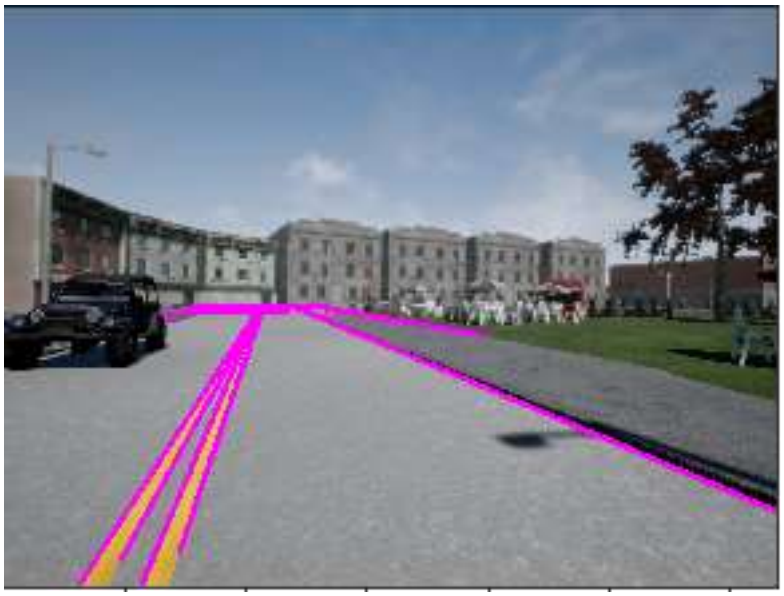

Figure 4.16 Lane line estimation with purple color

The second step, estimation of the lane boundary, merges lane lines and filters out any unnecessary horizontal line that is shown in the picture (see Fig. 4.16). This is performed by the following sub-steps:

2.1 Define lines with slope lower than the limit that characterizes a horizontal line

2.2 Cluster lines based on intercept and slope

2.3 Merge all lines in each cluster using average slope and average intercept

The pseudocode of the algorithm that implements this process is provided in Fig. 4.15. The related code is presented in Appendix B.

For the input of Fig. 4.16, the algorithm's output is shown in Fig. 4.18

```
Function merge_lane_lines(lines)
        Step 0: Define the thresholds
        similarity threshold of slope ← 0.1
        minimum threshold of slope ← 0.3
        similarity threshold intercept← 40
```

```
Step 2.1: Get slope and intercept of lines
slopes, intercepts ← get_slope_intecept(lines)
iterations ← 0
cluster_lines ← []
current_index ←[]
Step 2.2: Determine the lines with slope less than horizontal slope threshold
filter_lines ←  lane_lines[absolute value(slopes) > minimum threshold of
        slope]
Step 2.3: Iterate over all remaining slopes and intercepts and cluster lines
For each slope, intercept in (zip(slopes, intercepts)) do
        existing_cluster_lines ← np.array([iterations in current for each…
        current in current_inds])
        If not exists_in_clusters.any() then
        cluster_slope = np.logical_and(slopes < (slope + similarity_threshold
        ..._of slope), slopes > (slope - similarity_threshold_of slope))
        cluster_intercept ← np.logical_and(intercepts < (intercept +
        ...similarity_threshold_intercept), intercepts < (intercept – similarity_
                ...threshold intercept))
        index ← np.argwhere (cluster_slope & cluster_intercept &
                filter_lines).T
            If index.size then
            current_index.append(inds.flatten())
            cluster_lines.append(lines[inds])
            End If
```

```
                End If

                iterations ← iterations + 1

        End For

        Step 2.4: Merge all lines in clusters using mean average

        filter_lines ← [np.mean(cluster) for cluster in cluster_lines]

        filter_lines ← np.squeeze(np.array(filter_lines)

        Return filter_lines

End Function
```

Figure 4.17 Pseudocode of merge lines algorithm

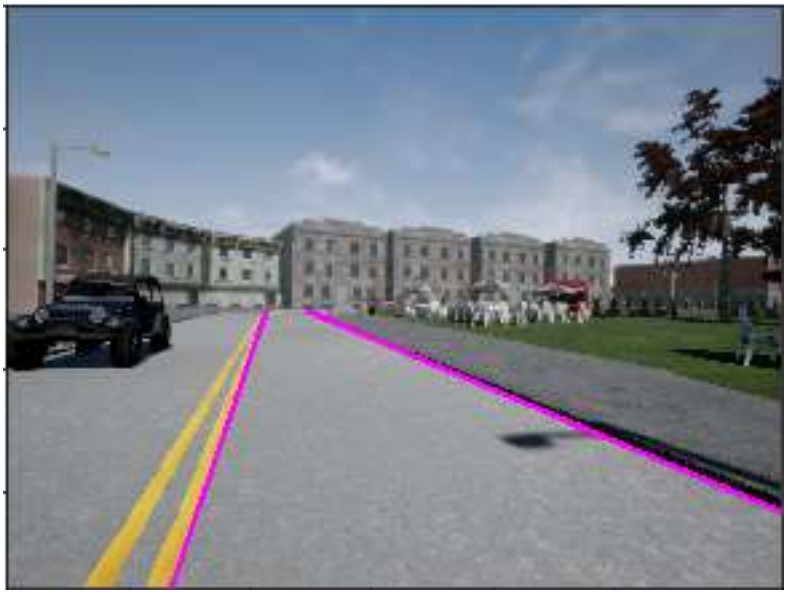

Figure 4.18 Lane estimation for the space where it is legally allowed for the autonomous vehicle to drive

Object detection

In image of Fig. 4.19, the bounding boxes are created by the VGG 16 neural network and our provided as inputs to the current analysis. More specifically, along with the output categories, we are given the limits of the bounding boxes, such as 'vehicle', and $[x_{min}, y_{min}]$ and $[x_{max}, y_{max}]$.

For object detection the convolutional neural network used has been developed at Stanford University to serve the purposes of Coursera course. The network results are saved in a NumPy array in class dataset_handler(). The function filter_detections_by_segmentation (see Fig. 4.20) takes as inputs a) the initial detections of neural network using the command detections = dataset_handler.object_detection (which is loaded before the execution of the function) (see Fig. 4.19) and b) the output of semantic segmentation (see- Fig. 4.12).

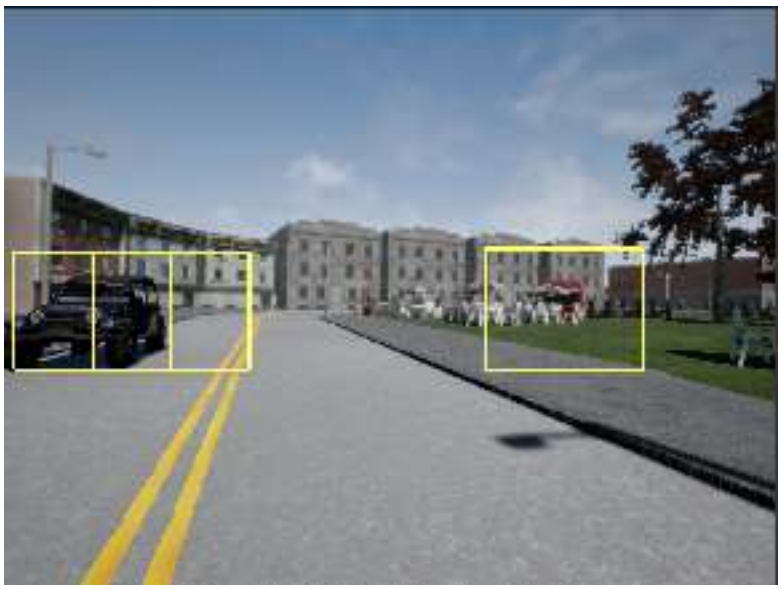

Figure 4.19 Input image with bounding boxes

Four steps were followed:

- For each crafted bounding box in the input image, calculate how many pixels in the bounding box belong to the category predicted by semantic segmentation
- Crop the segmentation output to pixels which are located inside of the bounding box
- Divide the calculated number of pixels by the area of the bounding box. The number of pixels with the same category as the detection output is counted and then normalized by the number of total pixels in the bounding box area
- If the ratio is larger than a lower limit, keep the detection. Else, remove the detection from the list of the detections. In our case study, the bounding boxes are filtered out when the compute normalized count is less than a threshold of

0.3. This means that, if less than a 30% of the bounding box area is occupied by the predicted class pixels, the bounding box has to be eliminated.

The pseudocode of the algorithm that implements this process is provided in Fig. 4.20. The related code is presented in Appendix B.

For the input of Fig. 4.19, the algorithm's output is shown in Fig. 4.21, i.e. only one car is detected.

```
Function filter_detections_by_segmentation(detections, segmentation_output):
    Filtered_detections ←[]
    Ratio_threshold ← 0.3
    For each detection in detections do
        Step 1: Compute the number of pixels which belong to the category
        for every detection
        Class_name, x_min, y_min, x_max, y_max, score ←detection
        x_min, y_min, x_max, y_max ← np.asfarray(x_min),
        np.asfarray(y_min), np.asfarray(x_max), np.asfarray(y_max)
        x_min, y_min, x_max, y_max ← int(x_min), int(y_min), int(x_max),
        int(y_max)
        If class_name equal to car then
            Class_sed_idx ← 10
        Elif class_name equal to pedestrian then
            Class_sed_idx ← 4
        End If
        category_pixel_cnt ← np.sum(segmentation_output[y_min:y_max,
        x_min:x_max] == class_sed_idx)
```

```
            Step 2: Divide the computed number of pixels by the area of the
            bounding box (total number of pixels).

            category_ratio ← category_pixel_cnt / ((x_max - x_min) * (y_max -
            y_min))

            Step 3: If the ratio is greater than a threshold keep the detection. Else,
            remove the detection from the list of detections.

            If category_ratio > ratio_threshold then

                filtered_detections.append(detection)

            End If

        End For

        return filtered_detections

End Function
```

Figure 4.20 Filtering the bounding boxes algorithm

Figure 4.21 Visualizing the car in the picture with the bounding box

Finally, for estimating the distance to the objects in the significant bounding boxes of the picture, we calculated the minimum distance from the pixels of the bounding box to the camera center. To calculate this distance, we used Eqs. (4.12), (4.13) and (4.14) and (4.18) below.

$$d = \sqrt{x^2 + y^2 + z^2} \tag{4.18}$$

The pseudocode of the algorithm that implements this process is provided in Fig. 4.20 The related code is presented in Appendix B.

```
Function find_min_detection(detections, x, y, z):
        min_distances←[]
        For detection in detection:
                Step 1: Compute the distance of each pixel in the detection bounds
                Step 2: Find the minimum distance with the eq. (4.18)
        End For
        min_distances ←np.array(min_distances)
        min_distances ← min_distances.reshape([-1, 1])
        return min_distances
End Function
```

Figure 4.22 Find the minimum distance algorithm

For the input of Fig. 4.19, the algorithm's output is shown in Fig. 4.23, i.e. the calculated distance from the detected car is 8.52 m.

Figure 4.23 The distance between the center of the camera and the car (8.52m)

To sum up, this case study uses the output of a (ready) semantic segmentation to perform four necessary tasks to:

1. Determine the drivable space of the autonomous vehicle. This task is important for recognizing the whole road, including the wrong-way part
2. Recognize the lanes. This task is important for recognizing the lanes along the road and, thus, the permissible driving space
3. Recognize the objects (Object recognition). This task is important for recognizing the objects in the environment of the autonomous vehicle
4. Estimate the distance of the recognized objects from the autonomous vehicle using the filtered 2-D object detection. This task is important for estimating the distance between ego-vehicle and other vehicles or objects.

Chapter 5 Conclusion

This thesis focused on two interesting and critical topics of current research on autonomous vehicles: Control of the vehicle dynamics and visual perception. The intention has been to drill down on the technical background that is necessary in order to conduct research and development in these two areas, present the essentials of this background and conduct limited experiments to indicate how researchers in the DeOPSys lab may proceed in their work using the available knowledge and tools.

In terms of vehicle dynamics control, we focused on speed control and steering control. The system reference inputs are the desired speed and the desired trajectory, respectively. The controller uses the current longitudinal speed, the driving direction, the current position of the vehicle from the IMU-sensor, GPS, and radar, the curvature of the path and the current yaw rate and steering angle. The outputs of the two schemes attempt to follow the reference inputs with as little deviation as possible. For both speed and lateral control, the corresponding controllers use appropriate dynamic models in order to generate the accelerator, brake or steering wheel commands. In our experiments we used a typical PID controller for longitudinal control and a Stanley controller for lateral control. We tested, through CARLA simulation, the effects of various values of the longitudinal PID controller parameters K_P, K_D and K_I (including values of zero in order to test P, PD and PI controllers). PID controller has many advantages, some of these are a) zero steady state error, b) moderate peak overshoot and stability, and c) its use for controlling both fast and slow process variables. That's why the results indicated that the vehicle tested had better performance using a PID controller than the other three (PD, PI, P controllers). Furthermore, high values of the integral gain K_I result to violent oscillation of the throttle, steering and speed and to instability.

In terms of visual perception, the inputs are camera images, and the system recognizes the different objects in the environment along its path. To do so, deep neural networks

are used. The steps for processing the camera images are several and highly complex. These steps have been overviewed systematically in this thesis. Subsequently, four significant concepts were tested, ground plane estimation, lane marking identification, object recognition and object distance. In these tests we recognized the importance of semantic segmentation, which provided the essential inputs to all cases. Given the output of semantic segmentation, for all four cases we developed code in Python that leverages important functions to perform these operations in order to recognize where the autonomous vehicle can drive on the road as well as what obstacles are into its route.

The two areas examined in this thesis, and the related experiments are supplemental and highly important in order to realize autonomy. Both tasks above require significant technical background, which both researchers and developers need to acquire. Furthermore, the CARLA simulation environment has been an invaluable tool, which can be used to develop new concepts in the PYTHON language and test them in a straightforward manner. Thus, it is recommended that future researchers in our lab familiarize themselves with the advanced background required for autonomous vehicle work and use CARLA as an excellent tool to develop and test their concepts. Multiple areas exist for further investigation. For example, one possible improvement in visual perception should be the combination of the camera and LIDAR outputs to obtain improved results, using real time object recognition in the CARLA environment. As far as vehicle control, research in lower level control could be performed. In lateral control, instead of a Stanley controller, the advanced model predictive controller (MPC) could be tested. This could open new opportunities for improved vehicle behavior, especially under complex dynamic states.

References

American Society of Civil Engineers, 2017. *Conditions & Capacity,* s.l.: American Society of Civil Engineers.

Badrinarayanan, V., Kendall, A. & Cipolla, R., 2017. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, December, pp. 2481-2495.

Barfoot, T. D., 2019. *STATE ESTIMATION FOR.* s.l.:Cambridge University Press.

Bender, P., Ziegler, J. & Stiller, C., 2014. Lanelets: Efficient Map Representation for Autonomous Driving. *Intelligent Vehicles Symposium*, 8-11 June, pp. 420-425.

Billington, J., 2018. *ZF and Mobileye jointly develop ADAS camera technology.* [Online]
Available at: https://www.autonomousvehicleinternational.com/news/adas/zf-and-mobileye.html
[Accessed 21 September 2019].

Bussemaker, K., 2014. *Sensing requirements for an automated vehicle for highway and rural environments,* Delft: Delft University of Technology.

CANNY, J., 1986. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, p. 679–698.

Department of Motor Vehicles, 2018. *Autonomous Vehicle Disengagement Reports,* California: Department of Motor Vehicles.

Dosovitskiy, A. et al., 2017. CARLA: An Open Urban Driving Simulator. *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1-16.

Everingham, M. et al., 2009. The PASCAL Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision 88*, 9 Septemper, pp. 303-338.

Falcone, P. et al., 2007. Predictive Active Steering Control for Autonomous. *IEEE TRANSACTIONS ON CONTROL SYSTEMS TECHNOLOGY*, May, pp. 566-580.

Forsyth, D. A. & Ponce, J., 2011. *Computer Vision A Modern Approach.* Second Edition ed. s.l.:Pearson Education, Inc..

Francis, B. A., 2015. *Classical Control.* s.l.:University of Toronto.

G. Neuhold, T. O. S. R. B. a. P. K., 2017. The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes. *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 5000-5009.

General Motors, , 2018. *SELF-DRIIVING SAFETY REPORT,* s.l.: General Motors.

Ian Goodfellow, Y. B. A. C., 2016. *Deep Learning.* Vol. 1 ed. s.l.:MIT press.

Jain, R., Kasturi, R. & Schunck, B. G., 1995. *MACHINE VISION.* s.l.:McGraw-Hill.

Jiang, J. & Astolfi, A., 2018. Lateral Control of an Autonomous Vehicle. *IEEE TRANSACTIONS ON INTELLIGENT VEHICLES*, June, pp. 228-237.

Kelly, J. & Waslander, S., n.d. *State Estimation and Localization for Self-Driving Cars,* Toronto: Coursera.

Lin, X., Sánchez-Escobedo, D., Casas, J. R. C. R. & Pardàs, M., 2019. Depth Estimation and Semantic Segmentation from a Single RGB Image Using a Hybrid Convolutional Neural Network. 15 April.

Liu, W. et al., 2016. *SSD: Single Shot MultiBox Detector,* s.l.: s.n.

Mani, K., 2019. *Medium.* [Online]
Available at: https://medium.com/datadriveninvestor/what-is-a-neural-network-9ca88b29f7cb
[Accessed 3 March 2020].

Martínez-Díaz, M. & Soriguera, F., 2018. Autonomous vehicles: theoretical and practical challenges. *Transportation Research Procedia*, pp. 275-282.

Mashadi, B. & Crolla, D., 2012. *Vehicle Powertrain Systems.* 1st Edition ed. s.l.:John Wiley & Sons, Ltd.

National Transportation Safety Board, 2018. *PRELIMINARY REPORT,* s.l.: National Transportation Safety Board.

Qi, C. R. et al., 2018. *Frustum PointNets for 3D Object Detection from RGB-D Data.* Salt Lake city, Conference on Computer Vision and Pattern Recognition.

Rajamani, R., 2012. LATERAL VEHICLE DYNAMICS. In: *Vehicle Dynamics and Control.* s.l.:Springer US, p. 498.

Rey, J., 2020. *Faster R-CNN: Down the rabbit hole of modern object detection.* [Online]
Available at: https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/
[Accessed 16 April 2020].

Snider, J. M., 2009. *Automatic Steering Methods for Autonomous Automobile Path Tracking.* Pittsburgh, Pennsylvania: Carnegie Mellon University.

Szeliski, R., 2010. *Computer Vision: Algorithms and Applications.* s.l.:Springer.

U.S. Departement of Transportation, 2017. *NHTSA.* [Online]
Available at: https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-

ads2.0_090617_v9a_tag.pdf
[Accessed 15 September 2019].

U.S. Department of Transportation, 2016. 2015 Motor Vehicle Crashes: Overview. *TRAFFIC SAFETY FACTS*, August.

U.S. Department of Transportation, 2017. *Automated driving systems: A vision for safety*, s.l.: U.S. Department of Transportation.

Urmson, C. et al., 2008. Autonomous Driving in UrbanEnvironments: Boss and the Urban Challenge. *Journal of Field Robotics*, 22 February, pp. 425-466.

Waslander, S. & Kelly, J., n.d. *Introduction to Self-Driving Cars*, Toronto: Coursera.

Waslander, S. & Kelly, J., n.d. *Visual Perception for Self-Driving Cars*, Toronto: Coursera.

Waymo, 2018. *On the Road to Fully Self-Driving*, s.l.: Waymo.

WHO, 2018. *Global status report on road safety 2018: Summary*, Geneva: World Health Organization.

Appendix A. Installing the CARLA simulator

Technical aspects of Carla simulator - Hardware

The hardware used for the Carla simulator is as follows:

- CPU: Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
- Ram: 32GB DDR3 1333MHz
- Graphics card: GeForce GTX 1060 6GBF
- Hard drive: SSD 256GB and HDD 500GB

The above hardware specifications support efficient performance of the simulator.

Technical aspects of Carla simulator - Software

The operational system used in this thesis is Ubuntu 16.04. Before installing the simulator, a certain process should be followed. The first step concerns the terminal of ubuntu and testing the firewall status for allowing Carla to have default access to ports 2000, 2001 and 2002 (TCP and UDP). The command in the terminal is:

$ sudo ufw status

After running this command the response of the system should be

Status: inactive

For the graphics card drivers, OpenGL 3.3 or above is required. The Carla Python client runs on Python 3.5.x or Python 3.6.x (where x is any number). The installed version of pip for the Python client needs to be checked. When both python3 and pip versions are available, the NumPy library should be installed. NumPy is the fundamental package for scientific computing in Python, which facilitates all array operations, including mathematical shape manipulation, logical, discrete Fourier transforms and much more. The library for package NumPy can be installed with the following command:

$ pip3 install numpy

Download and Extract the CARLA Simulator[10]

Download the CARLA simulator (CarlaUE4Ubuntu.tar.gz) from Coursera (Introduction to self-driving cars, week 7)

Extract the contents of CarlaUE4Ubuntu.tar.gz to any working directory.

Install Python Dependencies for Client

The CARLA Simulator client files require additional modules to be installed, which are detailed inside the $HOME /opt/CarlaSimulator/requirements.txt file.

[10] The version of CARLA simulator is 0.8.4

$ python3 -m pip install -r $HOME /opt/CarlaSimulator/requirements.txt --user

Loading the Simulator with the race track map

After completing the above and in order to load the simulator with the race truck map (server mode) used in this thesis, the following command needs to be entered with the terminal in window mode:

$ cd $HOME /opt/CarlaSimulator

The above command locates the Carla simulator on the system. The next step is to enter

$ ./CarlaUE4.sh /Game/Maps/RaceTrack -windowed -carla-server -benchmark -fps=30

The above command opens the racetrack map (server) with thirty (30) frames per second (fps). The fps argument is used to tune the simulator for a given frame-per-second rate. Upon entering this command, the simulator waits for a Python client code to respond. Subsequently, editing or writing a file to the Python client in the terminal of Ubuntu the following command is used:

$ nano name_of_program.py

where name_of_program can be changed by user. For example,

$ nano module_7.py

In another terminal, the user should run the Python client program using the following command, which calls the server that is motioned above:

$ python3 manual_control.py

If the Python client successfully connects, a new pygame window should appear (see Fig. A.1)

Figure A. 1 Carla Environment

Appendix B. Python code for the visual perception case study

The Python code and functions used in the case study of visual perception of Section 4.3 are provided below:

To import the results of the semantic segmentation network should be loaded the below dataset (2 lines)

```
colored_segmentation = dataset_handler.vis_segmentation(segmentation)
plt.imshow(colored_segmentation)
```

1. For drivable space estimation

```
def xy_from_depth(depth, k):
  # Get the shape of the depth tensor
  M, N = depth.shape
  # Grab required parameters from the K matrix
  f = k[0, 0]
  c_u = k[0, 2]
  c_v = k[1, 2]
  # Generate a grid of coordinates corresponding to the shape of the depth
   map
  u_mtx, v_mtx = np.meshgrid(np.arange(N), np.arange(M))
  # Compute x and y coordinates
  x = (u_mtx - c_u) * depth / f
  y = (v_mtx - c_v) * depth / f
  return x, y
```

For estimating the ground plane with the RANSAC algorithm

```
def ransac_plane_fit(xyz_data):
  # Set thresholds:
  num_itr = 100  # RANSAC maximum number of iterations
  min_num_inliers = 10000  # RANSAC minimum number of inliers
```

```
distance_threshold = 0.3
max_inliers_cnt = 0
max_inliers_set_idx = None
for i in range(num_itr):
    # Step 1: Choose a minimum of 3 points from xyz_data at random.
    idx = np.random.choice(range(xyz_data.shape[1]), 3, replace=False)
    curr_data = xyz_data[:, idx]
    # Step 2: Compute plane model
    plane_param = compute_plane(curr_data) # (1, 4)
    # Step 3: Find number of inliers
    distance_list = dist_to_plane(plane_param.T, xyz_data[0, :].T, xyz_data[1,
    :].T,     xyz_data[2, :].T)
    # Step 4: Check if the current number of inliers is greater than all previous
    iterations and keep the inlier set with the largest number of points.
    inliers_cnt = np.sum(distance_list < distance_threshold)
    if inliers_cnt > max_inliers_cnt:
        max_inliers_cnt = inliers_cnt
        max_inliers_set_idx = np.where(distance_list < distance_threshold)[0]
    # Step 5: Check if stopping criterion is satisfied and break.
    if inliers_cnt > 10000 or i > num_itr:
        break
# Step 6: Recompute the model parameters using largest inlier set.
final_data = xyz_data[:, max_inliers_set_idx]
output_plane = compute_plane(final_data)
return output_plane
```

For the lane estimation

```
def estimate_lane_lines(segmentation_output):
    # Step 1: Create an image with pixels belonging to lane boundary
    categories from the output of semantic segmentation
```

```
road_mask = (segmentation==6) | (segmentation==8).astype(np.uint8)

# Step 2: Perform Edge Detection using cv2.Canny()
mask_canny = cv2.Canny(road_mask * 255, 50, 100)

# Step 3: Perform Line estimation using cv2.HoughLinesP()
lines = cv2.HoughLinesP(mask_canny, rho=10, theta=np.pi/180*1,
threshold=100, … minLineLength=200, maxLineGap=100)
lines = lines.reshape([-1, 4])
return lines
```

For merging the estimated lines

```
def merge_lane_lines(lines):
    similarity_threshold_of_slope = 0.1
    minimum_threshold_of_slope = 0.3
    similarity_threshold_of_ intercept = 40
    Step 2.1: Get slope and intercept of lines
    slopes, intercepts = get_slope_intecept(lines)
    iterations = 0
    cluster_lines = []
    current_index = []
    Step 2.2: Determine the lines with slope less than horizontal slope
    threshold
    filter_lines =  lane_lines[abs(slopes) > min_slope_threshold]
   for slope, intercepts in (zip(slopes, intercepts)):
```

```
            existing_cluster_lines = np.array([iterations in current for current in
            current_index])

        if not existing_cluster_lines.any():

            cluster_slope = np.logical_and(slopes < (slopes +
                    similarity_threshold_of_slope), slopes > (slopes -
                    similarity_threshold_of_slope))

            intercept_cluster = np.logical_and(intercepts < (intercept +
                    similarity_threshold_of_ intercept), intercepts < (intercept -
                    similarity_threshold_of_ intercept))

            index = np.argwhere (cluster_slope & cluster_intercept &
                    filter_lines).T

                if index.size then

                    current_index.append(index.flatten())

                    cluster_lines.append(lines[inds])

            iterations = iterations + 1

        Step 2.4: Merge all lines in clusters using mean average

            filter_line = [np.mean(cluster) for cluster in cluster_lines]

            filter_line = np.squeeze(np.array(filter_line)

            return filter_line
```

For object detection

```
    def filter_detections_by_segmentation(detections, segmentation_output):

        ratio_threshold = 0.3

        for detection in detections:
```

```
# Step 1: Compute number of pixels belonging to the category for
every detection.

class_name, x_min, y_min, x_max, y_max, score = detection

        x_min, y_min, x_max, y_max = np.asfarray(x_min),
        np.asfarray(y_min), np.asfarray(x_max), np.asfarray(y_max)

x_min, y_min, x_max, y_max = int(x_min), int(y_min), int(x_max),
        int(y_max)

if class_name=="Car":

    class_sed_idx = 10

elif class_name=="Pedestrian":

    class_sed_idx = 4

category_pixel_cnt = np.sum(segmentation_output[y_min:y_max,
        x_min:x_max] == class_sed_idx)

# Step 2: Devide the computed number of pixels by the area of the
bounding box (total number of pixels).

category_ratio = category_pixel_cnt / ((x_max - x_min) * (y_max
  y_min))

# Step 3: If the ratio is greater than a threshold keep the detection.
Else, remove the detection from the list of detections.

if category_ratio > ratio_threshold:

        filtered_detections.append(detection)

  return filtered_detections
```

For estimating the distance of detected objects

```
def find_min_distance_to_detection(detections, x, y, z):

    min_distances = []
```

```
for detection in detections:

        # Step 1: Compute distance of every pixel in the detection
    bounds

        class_name, x_min, y_min, x_max, y_max, score = detection

        x_min, y_min, x_max, y_max = np.asfarray(x_min),
          np.asfarray(y_min), np.asfarray(x_max), np.asfarray(y_max)

        x_min, y_min, x_max, y_max = int(x_min), int(y_min),
          int(x_max), int(y_max)

        # Step 2: Find minimum distance

        mtx_dist = np.sqrt(x**2 + y**2 + z**2

        min_distances.append(np.min(mtx_dist[y_min:y_max,
    x_min:x_max]))

min_distances = np.array(min_distances)

min_distances = min_distances.reshape([-1, 1])

return min_distances
```