MINI AUTONOMOUS CAR ARCHITECTURE FOR URBAN DRIVING SCENARIOS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÖKHAN KARABULUT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2019

Approval of the thesis:

MINI AUTONOMOUS CAR ARCHITECTURE FOR URBAN DRIVING SCENARIOS

submitted by **GÖKHAN KARABULUT** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Tolga Can
Supervisor, **Department of Computer Engineering, METU** _____

Assist. Prof. Dr. Selim Temizer
Co-supervisor, **Dept. of Computer Science, Nazarbayev Univ.** _____

Examining Committee Members:

Assoc. Prof. Dr. Yusuf Sahillioğlu
Department of Computer Engineering, METU _____

Prof. Dr. Tolga Can
Department of Computer Engineering, METU _____

Assist. Prof. Dr. Mehmet Tan
Department of Computer Engineering, TOBB-ETU _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Gökhan Karabulut

Signature :

ABSTRACT

MINI AUTONOMOUS CAR ARCHITECTURE FOR URBAN DRIVING SCENARIOS

Karabulut, Gökhan
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Tolga Can
Co-Supervisor: Assist. Prof. Dr. Selim Temizer

September 2019, 70 pages

Autonomous cars capable of driving in city traffic have been long studied in architectures decomposed into perception, planning, and control components. Recent advances in deep learning techniques considerably contributed to the perception component of this approach. These techniques also laid the groundwork for the progress of other approaches such as end-to-end learning of steering commands and driving affordances from camera images. Though these approaches are promising to simplify the overall architecture, the decomposed architectures are found more persuasive, constituting the majority of today's state-of-the-art, market-oriented driverless cars. However, studies on small-scale autonomous cars, which are considered low-cost and rapid prototyping platforms, are not on a par with research on the modern decomposed architectures. These studies often remain limited to end-to-end approaches or resort to traditional image processing techniques in over-simplified traffic scenarios. In this thesis, we present a decomposed architecture for small-scale cars covering extended traffic scenarios with seven traffic signs, traffic lights, lane changes, cloverleaf interchange, pedestrian crossings, and parking. To realize this architecture, we created

segmentation and classification datasets. We trained two deep learning models for learning lane semantics and classifying traffic signs and lights. We developed a behavior planner to decide on the best behavior primitives for traffic scenes. Based on these behavior primitives, we implemented a trajectory planner to find optimal trajectories along the lanes and a controller to follow these trajectories. With our novel lane segmentation scheme, 97% accurate classifier, robust planner and controller algorithms, we achieved successful drives on simulated and real courses.

Keywords: autonomous car, traffic scene parsing, traffic sign classification, optimal trajectory planning, path tracking

ÖZ

ŞEHİR İÇİ SÜRÜŞ SENARYOLARI İÇİN MİNİ OTONOM ARAÇ MİMARİSİ

Karabulut, Gökhan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Tolga Can
Ortak Tez Yöneticisi: Dr. Öğr. Üyesi Selim Temizer

Eylül 2019 , 70 sayfa

Şehir trafiğinde sürüş yapabilen otonom araçlar algılama, planlama ve kontrol bileşenlerine ayrılan mimariler olarak uzun zamandır çalışılmaktadır. Derin öğrenme tekniklerindeki son gelişmeler bu yaklaşımın algılama bileşenine önemli ölçüde katkıda bulunmuştur. Bu teknikler ayrıca kamera görüntüleri üzerinden uçtan uca yönelim komutlarının ve sürüş sağlarlıklarının öğrenilmesi gibi diğer yaklaşımların ilerleyişi için de zemin hazırlamıştır. Her ne kadar bu yaklaşımlar genel mimariyi sadeleştirmek için ümit verici olsa da bugünün en gelişmiş, pazara yönelik sürücüsüz araçlarının çoğunluğunu oluşturan ayrışmış mimariler daha ikna edici bulunmaktadır. Bununla birlikte, düşük maliyetli ve hızlı prototip oluşturma platformları olarak düşünülen küçük ölçekli otonom araçlar üzerinde yapılan çalışmalar, modern ayrışmış mimariler üzerine yapılan araştırmalarla aynı düzeyde değildir. Bu çalışmalar genellikle uçtan uca yaklaşımlarla sınırlı kalmakta veya aşırı basitleştirilmiş trafik senaryoları içinde geleneksel görüntü işleme yöntemlerine başvurmaktadır. Bu tezde, küçük ölçekli araçlar için yedi trafik işareti, trafik ışıkları, şerit değişikliği, yonca yaprağı kavşağı, yaya geçitleri ve park işlemi ile genişletilmiş trafik senaryolarını kapsayan ayrışmış bir mimari

sunuyoruz. Bu mimariyi gerçekleştirmek için bölütleme ve sınıflandırma veri setleri oluşturduk. Şerit anlamlarını öğrenmek ve trafik levha ve ışıklarını sınıflandırmak için iki derin öğrenme modeli eğittik. Trafik sahnelerine göre en iyi davranış temellerine karar veren bir davranış planlayıcısı geliştirdik. Bu davranış temellerine dayanarak, şerit boyunca en uygun yörüngeleri bulan bir yörünge planlayıcısı ve bu yörüngeleri takip etmek için bir denetleyici gerçekledik. Özgün şerit bölümleme tasarımız, %97 doğru sınıflandırıcımız, dayanıklı planlayıcı ve kontrolcü algoritmalarımızla benzetimli ve gerçek güzergahlarda başarılı sürüşler gerçekleştirdik.

Anahtar Kelimeler: otonom araç, trafik sahnesi ayrıştırma, trafik işareti sınıflandırma, optimal yörünge planlama, yol takibi

To those who cannot do without Vim key bindings.

ACKNOWLEDGMENTS

I would like to start with the members of team Robocodes: Berkant Bayraktar, Berker Acır, Ilker Ayçiçek, Yunus Emre Saçma, and Asst. Prof. Dr. Selim Temizer. Their patience against all the odds made this thesis possible. Lack of space and budget to build a race course did not stop them. They dared to develop a mini autonomous car in a simulated environment and see it in a competition without ever testing in a real environment against competition rules. Thank you all.

Supervisor of our team and co-supervisor of my thesis, Asst. Prof. Dr. Selim Temizer, did more than supervising. I truly enjoyed the enlightenment moments of how complicated-looking subjects turned into intuitive ideas after talking to him. I am grateful to him for encouraging me to study autonomous cars in which I had the chance to gain hands-on experience in both machine learning and robotics. Aside from supervising the team and my thesis, I cannot thank him enough for his time and effort to find a sponsor to afford a mini autonomous car platform.

I am thankful to our sponsor, Acasus, for providing us with a mini autonomous car platform while we were desperately looking for a sponsor.

I would like to thank my supervisor, Prof. Dr. Tolga Can, for his support and help with all the thesis procedures. It would not be easier without his support.

I also would like to thank my friends at Turkish Aerospace for tolerating my unplanned absence when I had to take annual leave to work on my thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

2D	2 Dimensional
3D	3 Dimensional
AP	Average Precision
DARPA	Defense Advanced Research Projects Agency
ESC	Electronic Speed Controller
GPS	Global Positioning System
IMU	Inertial Measurement Unit
IoU	Intersection over Union
LIDAR	Light Detection and Ranging
mAP	Mean Average Precision
MARC	Mini Autonomous Racecar Competition
MIT	Massachusetts Institute of Technology
RDDF	Route Definition Data Format
RGB	Red Green Blue
RNDF	Route Network Definition File
ROS	Robot Operating System
SSD	Single Slot Detector
UKF	Unscented Kalman Filter
YOLO	You Only Look Once

CHAPTER 1

INTRODUCTION

Autonomous cars have received a great deal of attention over the last two decades and started to be a reality in the last few years with several level of autonomy from driving assistance level to full autonomy [1]. In quest of fully autonomous vehicles, a number of competitions were organized to stimulate researchers' interest [2, 3]. These events started with autonomous cars driving on desert roads with only static obstacles and evolved to a point where the cars survived a simple form of everyday traffic scenarios including highway driving, overtaking, intersections, and parking. These challenges led to state-of-the-art software architectures which decompose the driving problem into components such as perception, planner, and controller. Each of these components was also powered by state-of-the-art algorithms that shaped today's autonomous vehicles.

Meanwhile, the progress in deep learning techniques along with the increase of storage and computational power in the computer market made a breakthrough in computer vision. These advances not only boosted the decomposition-based approaches in perception side, but also revived the existing idea of learning a mapping from raw camera images to steering commands [4], which is followed by the idea of learning driving affordance indicators such as distance to the center of the ego lane, orientation of the car relative to the road, and distance to the other cars so that steering and speed commands can be computed with a dedicated controller based on these affordances [5].

Being an elegant and promising solution, learning affordances from raw images requires additional tooling for data acquisition in order to compute and record affordance indicators per image. The choice of affordance indicators for a smooth driving

experience also presents its own challenges.

Learning a direct mapping to the steering commands quickly reaches its limit when the driving scenario becomes more complicated than tracking a curvy road. Introduction of traffic regulations, overtaking, and lane keeping policies remain too abstract to be captured in such a mapping. Furthermore, human drivers tend to take different actions at different times even for the same scenarios. Different actions on similar raw images in the training set easily confuses the model [5]. Last but not least, end-to-end nature of this approach presents difficulties in understanding and debugging the behavior of the vehicle.

Existing decomposition-based approaches heavily depend on a high-definition map of the driving environment, which often loses its validity due to changing streets or constructions on the roads. Autonomous driving in urban scenarios without an accurate map has been studied extending an existing traffic scene segmentation dataset with ego lane, parallel lane, and opposite lane annotations [6], but it is yet to be deployed and tested on a car. For novel methods like this instance, a low-cost and risk-free solution for initial on-road testing could be the use of a small-scale car.

Present small-scale driverless car studies either focus on directly learning steering commands from the images [7, 8] or implement a decomposed architecture using traditional computer vision techniques to detect ego lane lines and few traffic signs [9]. Despite the existence of powerful small-scale autonomous car platforms in the hardware side [10], related studies in the software side fail to keep up with the advances in self-driving car technologies. One possible reason behind this lag could be the lack of datasets for mini cars.

1.1 Problem Definition

We seek autonomous driving solutions to a number of traffic scenarios specified by a mini autonomous car competition, OpenZeka MARC 2019 [11]. Inspired by MIT racecar platform [10], OpenZeka MARC is organized by OpenZeka and was held in February 2018, May 2018, and April 2019 with increasingly complex rules. The competition offers leagues for high schools, universities, and other hobbyists includ-

ing companies. Total of 13 teams took part in the competition in 2019. Being one of the teams which completed the race in time, we won the third place in the university league out of 7 teams.

Our requirements are mostly derived from the competition rules as follows:

- The mini car shall follow lanes at up to 0.9 m/s speed in a two-lane road as depicted in Figure 1.1.
- The mini car shall detect traffic light and signs shown in Figure 1.2.
- The mini car shall overtake a waiting car on the right lane and steer back to the right lane.
- The mini car shall reactively avoid obstacles.
- The car shall climb up and climb down the cloverleaf interchange given in Figure 1.1.
- The mini car shall choose to go straight when it encounters straight or right turn sign.
- The mini car shall stop on red light and move on green light.
- The mini car shall change to the left lane when it is close to a construction zone on the right lane as indicated by keep-left signs.
- The mini car shall be capable of passing a rough road section indicated by a loose gravel sign.
- The mini car shall wait for pedestrians on crosswalks indicated by pedestrian crossing signs.
- The mini car shall turn right after parking sign into the parking lot given in Figure 1.1.
- The mini car shall stop on any of the free parking slots.

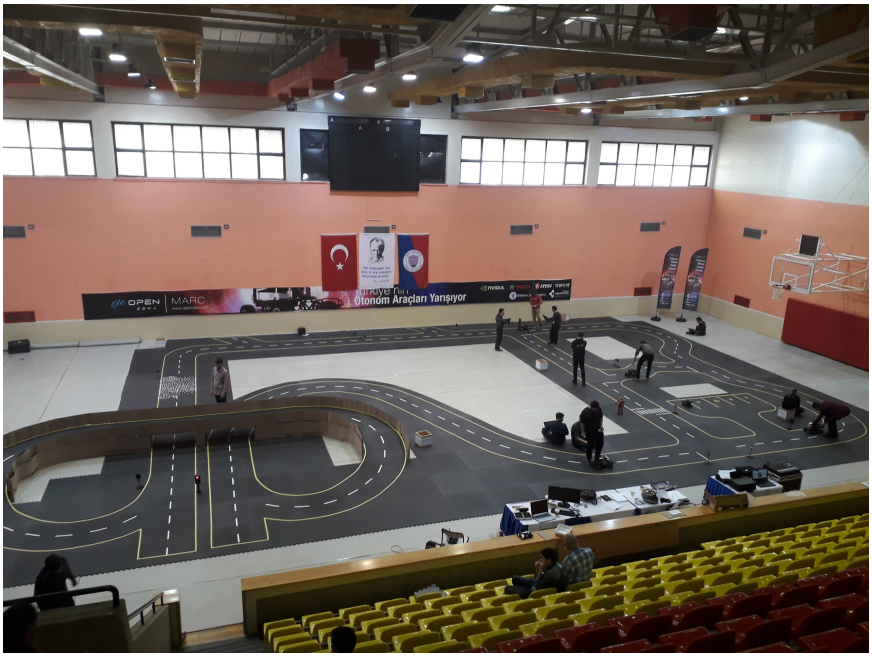


Figure 1.1: OpenZeka 2019 MARC race course.

1.2 Contributions

The main contribution of our study is as follows:

- **Pixel classification dataset for mini cars:** Existing traffic scene datasets are exclusively for real-scale cars [12, 13, 14, 15]. This poses a difficulty for on-road testing of the novel methods on mini-cars. We present a dataset with pixel-level annotations for mini cars.

- **Extended traffic sign and signal classification dataset:** We merge existing traffic sign classification datasets [16, 17, 18, 19, 20] for our scenarios and semi-automatically extend the dataset by cropping the segmented sign regions from our camera images.

- **Development of a novel lane detector:** We build our lane detector on the approach proposed by Meyer et al. [6]. This approach essentially segments the road into ego lane, parallel lane, and opposite lane. It then extracts a center line for the ego lane. For our scenarios, we skip the opposite lane and further segment the parallel lane into right and left lanes. Unlike Meyer et al. [6], we

are interested in finding center line curves for neighboring parallel lanes along with the ego lane. Our approach saves us from analyzing the parallel lane pixels for right or left categorization.

- **Local optimal trajectory planner for structured environments:** It is common that state-of-the-art autonomous vehicles rely on detailed maps to provide a reference line to their local planner for structural environments (i.e., environments with explicit drivable corridors), often by means of a global planner [21, 22, 23]. We implemented a Frenet optimal trajectory planner [24] without a map using exclusively the online computed lane centers as a source of the reference line.

- **Decomposed architecture for autonomous driving:** Autonomous vehicle architectures span a wide range of computer science research areas including but not limited to computer vision, path planning, automata theory, control theory, and distributed systems. Each of these areas comes with many problems and various alternative solutions to those problems. We put together a decomposed architecture with a selected set of solutions from different domains.

1.3 Organization

The organization of this thesis as follows:

Chapter 2 discusses various autonomous driving approaches and algorithms used in search of autonomy highlighting their advantages and disadvantages.

Chapter 3 presents our hardware configuration and software architecture at a high level. In Chapter 3, we also introduce our simulation environment that accelerated the development and verification processes.

Chapter 4 provides a detailed description of our scene interpretation module in which we implement perception and behavior planning capabilities of the car. First, we present our segmentation model which is used to parse the current scene into lanes and traffic signs. Second, we introduce our lane center line extraction method based on the parsed lane semantics. Third, we discuss our traffic sign classification approach on the

sign regions proposed by segmentation model. Next, we review our obstacle detection method. Finally, we explain our behavioral layer that implements a hierarchical finite state machine that regulates the decisions made by the car based on the perception capabilities.

Chapter 5 is dedicated to our trajectory planner and control algorithms. We start with introducing Frenet frame on a lane segment and then we explain how we plan an optimal trajectory based on the lane center lines provided by the scene interpretation module. In the second half of this chapter, we derive and tune pure pursuit control algorithm that computes steering commands to execute our optimal trajectories.

Chapter 6 presents our datasets, experiments, and results. In Chapter 6, we evaluate our segmentation and classification models. Then, we demonstrate the actions performed by our car in response to various traffic scenarios and compare them with the actions taken by a human driver.

Chapter 7 concludes our arguments by summarizing our approach to autonomous driving, highlighting its limitations and suggesting directions for future work.

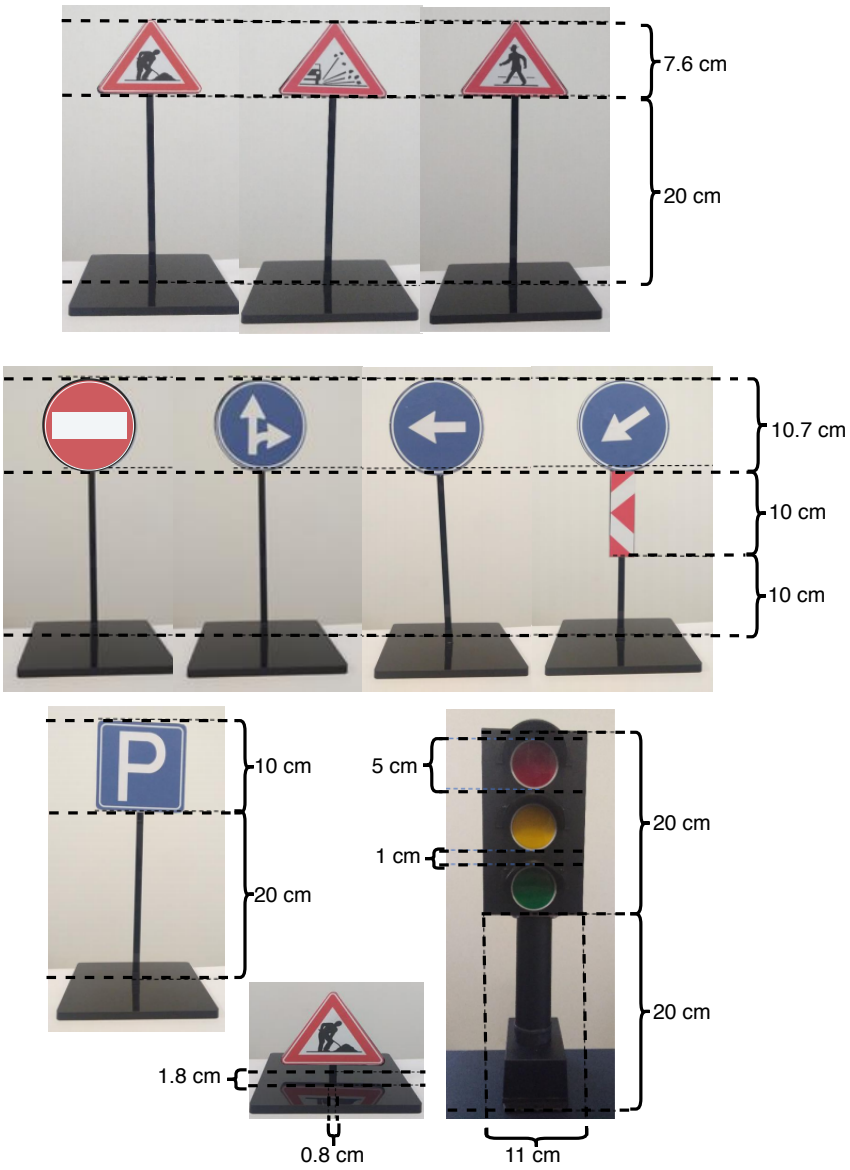


Figure 1.2: Traffic signs and light specifically designed for mini cars by OpenZeka.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we review prominent autonomous driving architectures together with the algorithms they are composed of. Along the way, we also briefly discuss the historical development of self-driving cars, particularly two most significant self-driving car competitions that paved the way for today's driverless car technologies; DARPA Grand Challenge and DARPA Urban Challenge.

In DARPA Grand Challenge 2004, none of the 15 teams saw the finishing line of the race course. In DARPA Grand Challenge 2005, Stanley, a robot car developed by Stanford Racing Team, was the first car to complete the race course. Thrun et al. [21] presents the details of the competition rules and the software design of Stanley. According to Thrun et al., a description of the race course was given to the participants in a DARPA-defined format, namely RDDF, two hours before the race. The RDDF contained a list of longitudes, latitudes, road segment widths and a list of speed limits associated with the road segments. In addition, the autonomous cars didn't need to deal with dynamic obstacles.

The authors state that Stanley's software is designed as a data processing pipeline and processing nodes communicate through a publish/subscribe mechanism. Stanley localizes itself on the RDDF by incorporating data from GPS, GPS compass, IMU, and wheel encoders with UKF at 100 Hz. It performs terrain analysis based on laser sensors and camera. For both data sources, the team automatically creates datasets through human driving and apply machine learning algorithms to classify the terrain into drivable and nondrivable regions. For the laser readings, a set of parameters such as obstacle height threshold and acceptance probability along with the Markov model parameters that capture the process and measurement noise covariances are learned

in a discriminative fashion by coordinate ascent algorithm.

As opposed to laser terrain analysis, Stanley uses generative learning algorithm for camera based terrain analysis. Drivable quadrilaterals (ahead of the vehicle, extracted from laser data) are projected onto the camera image. The pixels inside the quadrilateral are then used as training samples. From these samples, Stanley learns and maintains a database of Gaussians in RGB space that corresponds to wide variety of drivable surfaces. The range of a laser sensor is shorter than that of the camera. On the other hand, vision based terrain analysis is susceptible to color and lighting changes in the environment. Therefore, Stanley uses the laser based terrain analysis for steering control and vision based analysis for speed control so that the vision module acts as an early warning system when an obstacle is ahead but not within the range of the lasers.

Because the detailed race track is provided in RDDF, Stanley's main focus is local obstacle avoidance rather than global planning. Though there are no lanes in the course, Stanley introduces lateral offsets over the base trajectory which is a smoothed path extracted from RDDF. Stanley basically plans a trajectory to smoothly change into a lateral offset within the drivable path for obstacle avoidance similar to the lane change in highway driving. The planner finds a minimum cost trajectory by evaluating a cost function which is subject to kinematic and dynamic constraints of the vehicle, distance to obstacles, distance from the center of the road, and being on the course corridor.

Stanley uses its own steering control algorithm to track the optimal trajectory proposed by the path planner. Relying on the geometrical relation between the car pose and the trajectory, Stanley minimizes the cross-track error, which measures the lateral distance between the center of front axle and the nearest point on the target trajectory. The geometrical relation is illustrated in Figure 2.1.

Nonlinear feedback function of cross-track error is given by the equation

$$\delta(t) = \psi(t) + \arctan \frac{kx(t)}{u(t)}, \tag{2.1}$$

where k is a gain parameter, $u(t)$ is the car speed, and $\psi(t)$ denotes the orientation of

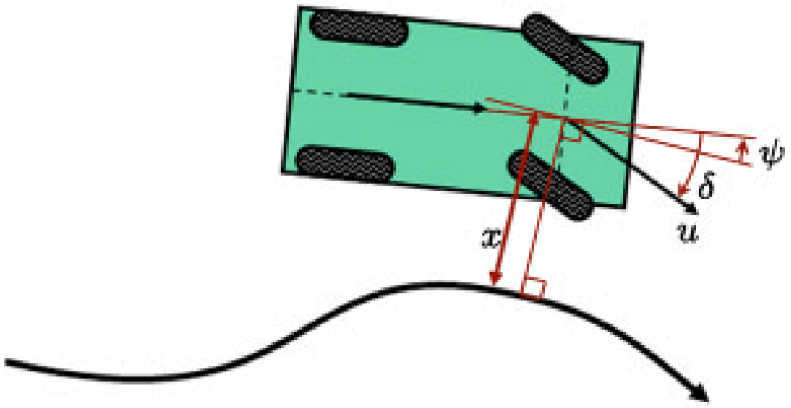


Figure 2.1: The geometrical relation between the trajectory and the car used by Stanley controller algorithm. Image from Thrun et al. [21].

the nearest trajectory segment relative to the car's orientation. The intuition behind the controller is that as the cross-track error $x(t)$ increases, the controller produces stronger steering angle $\delta(t)$ towards the trajectory. Likewise, as the speed $u(t)$ increases, the controller avoids sudden strong maneuvers. Hoffmann et al. studies the Stanley controller algorithm in greater detail [25].

In 2007, DARPA Urban Challenge took place. Montemerlo et al. [22] present the competition details and the software architecture of Junior, Stanford's another robot car and the second best car in the challenge. This time rules were more complex including overtaking parking or moving vehicles, precedence handling at intersections possibly with stop signs, merging into fast moving traffic, left turns, parking and U-turns when the road is completely blocked. Participants were provided with a road network description file, or RNDF, which contained lane information, stop signs, parking lots, and special checkpoints. In addition, the teams were also provided with a high resolution aerial image of the race course so that they can further improve the RNDF. In the competition, the vehicles were given multiple missions as a sequence of checkpoints in the RNDF.

Like Stanley, Junior's software architecture is made of sensor interfaces, perception, navigation, and drive-by-wire interfaces at the core. The design is again based on data processing endpoints communicating with publish/subscribe paradigm. Unlike Stan-

ley, Junior's modules are far more advanced. Junior's perception module segments the environment data into moving vehicles and static obstacles. Its navigation features a global path planner based on dynamic programming to find an optimum path to the mission checkpoints from the current location of the car. Moreover, the navigation module handles different driving scenarios with different planning algorithms.

It performs free-form navigation in parking lots, at U-turns or whenever the car gets stuck for extended period of time. The free-form planner is specifically developed for Junior and named hybrid A* by the Stanford Racing Team. Hybrid A* associates discrete search space of regular A* with a continuous state by performing forward simulations with different steering angles and computes a score based on the continuous state. The continuous state is represented by x-y position of the car, heading direction, and the direction, either forward or reverse. Whereas the path found by the regular A* and Field D* algorithms cannot be executed due to their discrete nature, hybrid A* can find executable paths that accounts for the nonholonomic constraints of the vehicle. Hybrid A* uses dual admissible heuristics. One heuristic is nonholonomic without obstacles, and the other is holonomic with obstacles. Once a solution is found, it is smoothed for a better driving experience. Extensive study and experiments show that hybrid A* produces near-optimal solutions [26, 27]. Figure 2.2 demonstrates the difference between regular A*, Field D*, and hybrid A* algorithms.

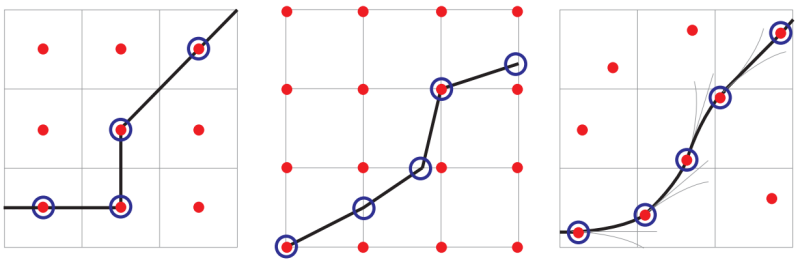


Figure 2.2: Left: Regular A* solutions pass through only the center of grids. Center: Field D* solutions can have arbitrary linear paths from cell to cell. Right: Hybrid A* associates a continuous state with each cell and computes a score for the continuous state. Image from Dolgov et al. [26].

Junior uses a different planner for normal on-road navigation. It performs internal simulations with different steering parameters. The internal simulations generate can-

didate trajectories with respect to a reference path. This reference path is essentially the smoothed center of the lane obtained from RNDF. The planner evaluates the candidate trajectories by a cost function and finally selects the best trajectory. The cost function also regulates the lane change or overtaking behavior of Junior. When the right lane is blocked, the car chooses to shift left. When overtaking is complete, it steers back to the right lane as it would be more costly to occupy the left lane.

Driving behavior of Junior is governed by a hierarchical finite state machine. The state machine decides on the U-turns, handles intersection precedence and stop signs, prevents the car from getting stuck, switches to parking navigation in a parking lot or chooses the true planner for the current scenario in general.

Werling et al. [24] report that they generated optimal trajectories in Frenet frame and tested on Junior without obstacles. They also present their obstacle avoidance experiments in simulation. In their method, they suggest integrating trajectory generation with a behavioral layer that decides on the high level as to whether the car should keep a constant velocity, follow the car in front with a constant distance, merge into traffic or stop at a point. Figure 2.3 demonstrates a velocity keeping instance in this approach.

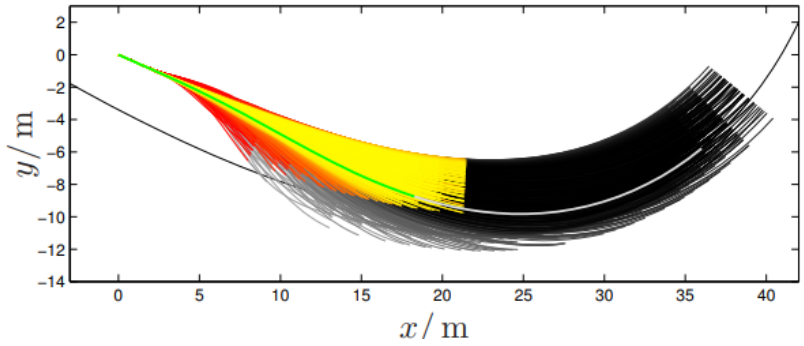


Figure 2.3: A sample optimal trajectory generated in Frenet frame in velocity keeping mode. Colors from red to yellow represent increasing lateral cost. Colors from grey to black represent increasing longitudinal cost. Green and light grey colors represent the optimal trajectory, which leads the car to the reference line and desired speed. Image from Werling et al. [24].

Yoneda et al. [28] further extend the method developed by Werling et al. [24] in-

troducing an additional adjust mode while switching from velocity keeping mode to distance keeping in an effort to eliminate strong acceleration and deceleration during the mode switching in quest of a more natural driving experience.

Fast forward to the present day, Autoware [23], being one of the modern open source self-driving car platforms is based on ROS [29]. ROS is a commonly used, extensible, component based, highly modular middleware framework with many reusable packages and visualization tools that dramatically accelerated today's robot development and prototyping processes. Unsurprisingly, publish/subscribe mechanism is at the core of ROS communication patterns. Autoware implements perception, decision-making, planning and path tracking capabilities. Figure 2.4 illustrates the Autoware architecture at a high level.

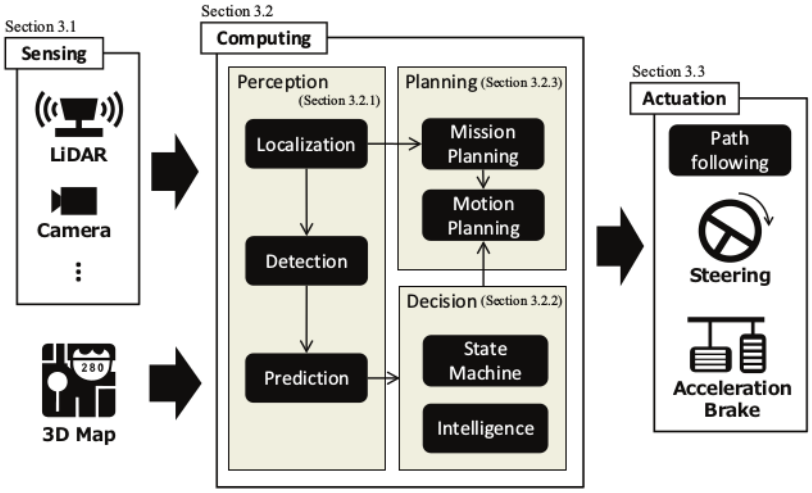


Figure 2.4: Autoware high level architecture and data flow. Image from Kato et al. [23].

Perception capabilities are made of localization, detection, and prediction modules. For the localization, Autoware relies on high definition 3D maps. It localizes itself by applying scan matching between the 3D map and LiDAR scans. Therefore, a 3D map of the environment should be created beforehand using SLAM techniques, in which scan matching is applied against previous LiDAR scans instead of a 3D map such that a transformation between the LiDAR scans are obtained and a cumulative point cloud is continually updated. Other perception modules also rely on the localization.

For example, the car is localized in the 3D map, it projects 3D map features into the front view camera images to define a ROI for traffic light detection and classification in order to eliminate full image search on every image frame. The detection module supports both deep learning and traditional image processing and machine learning techniques. It features YOLO2 [30] and SSD [31] models for detecting objects in the traffic such as traffic signals, pedestrians, and other vehicles. In the prediction module, Autoware associates detected objects with time, so that it estimates trajectories for the moving objects which are then used in the planning modules. Based on the perception modules, Autoware makes decisions in response to the environmental changes. The decision-making scheme is captured in a finite state machine similar to Junior.

Autoware features two sets of planners, a mission planner and motion planners. The mission planner is responsible for creating a rough global path from the current location to the destination in the map. Motion planners, on the other hand, generate local trajectories taking the global plan as a reference. In unstructured environments such as parking lots, hybrid A* is used similar to Junior. For well-structured environment scenarios such as navigating on the lanes, state lattice based algorithms are preferred. Pivtoraiko et al. [32] introduce space lattice based planning. The state lattice is made of motion primitives of a specific car. The motion primitives are generated offline by a precise trajectory generator respecting the mobility model of the vehicle such as steering limits and wheelbase. Then, the lattice search space could be searched by D* algorithms for optimal trajectories. This method was also successfully used in DARPA Urban Challenge by the winner vehicle, Carnegie Mellon University's Tartan Racing for navigating in unstructured environments [33]. Figure 2.5 illustrates a state lattice. McNaughton et al. [34] later extended this approach and applied it to structural environments.

Autoware uses pure pursuit controller to generate low level steering commands to execute the given trajectory from the motion planners. Kim et al. [35] studies the controller with its geometrical derivation and also give some useful pointers on tuning.

Backed by Baidu, Apollo is another open source autonomous driving platform with

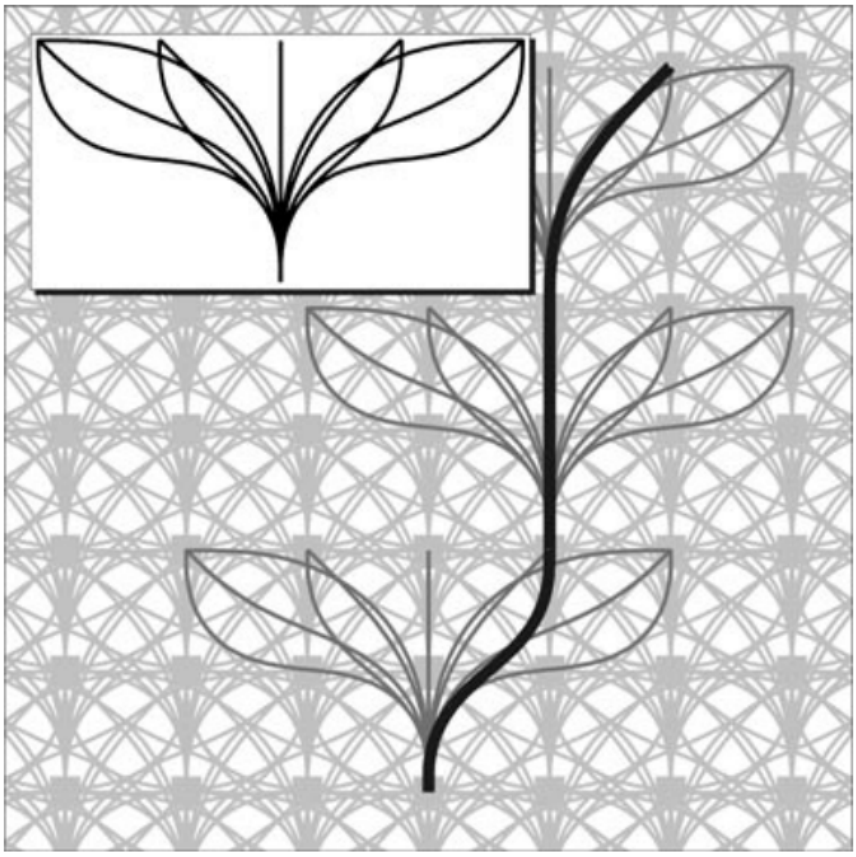


Figure 2.5: An example state lattice without reverse motions. Image from Pivtoraiko et al. [32].

its giant dataset [12]. Similar to other decomposed architectures, Apollo is also made of localization, perception, prediction, routing, motion planner, and vehicle control components as shown in Figure 2.6. Like Autoware, Apollo also relies on high definition 3D maps for location and perception [36].

The routing component finds a global plan from the current location to a destination in the map like previous architectures; however, Apollo's lane-based motion planner is quite different. Apollo does not directly use this global plan as a reference path for trajectory generation, but rather it generates multiple lane level reference lines from it taking traffic regulations (e.g., traffic signs, signals and lane markings) and safety measures into account. During lane level motion planning, Frenet frames are constructed based on the given reference lines. Lane level path and speed optimizers generate the optimal trajectories in Frenet frame for each lane. Finally, a trajectory

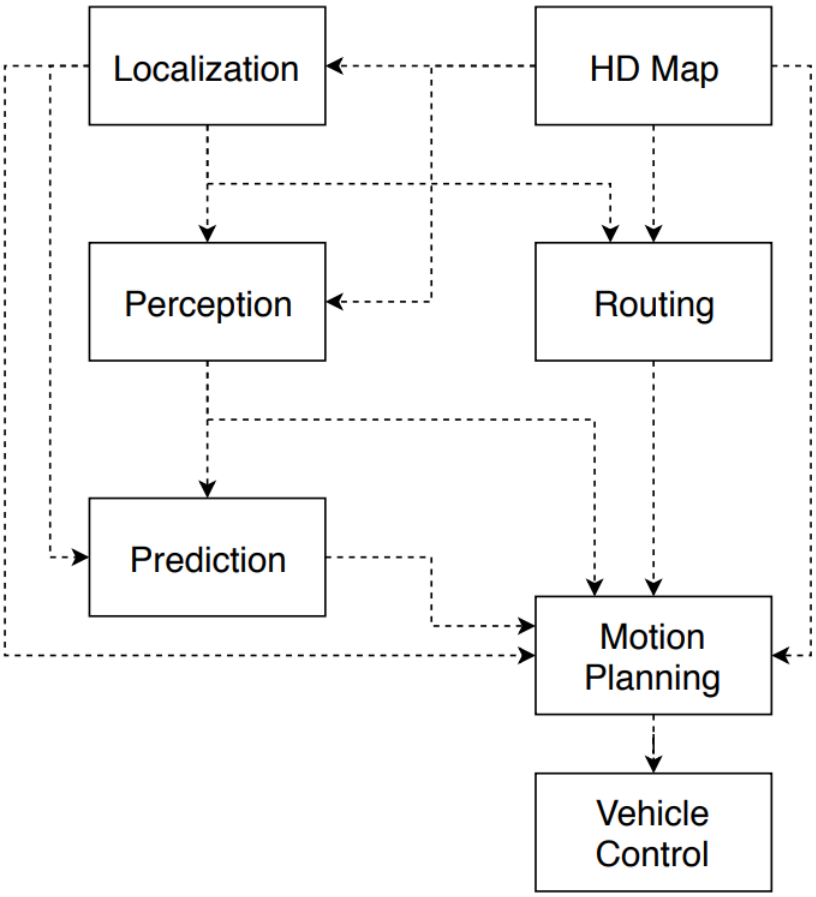
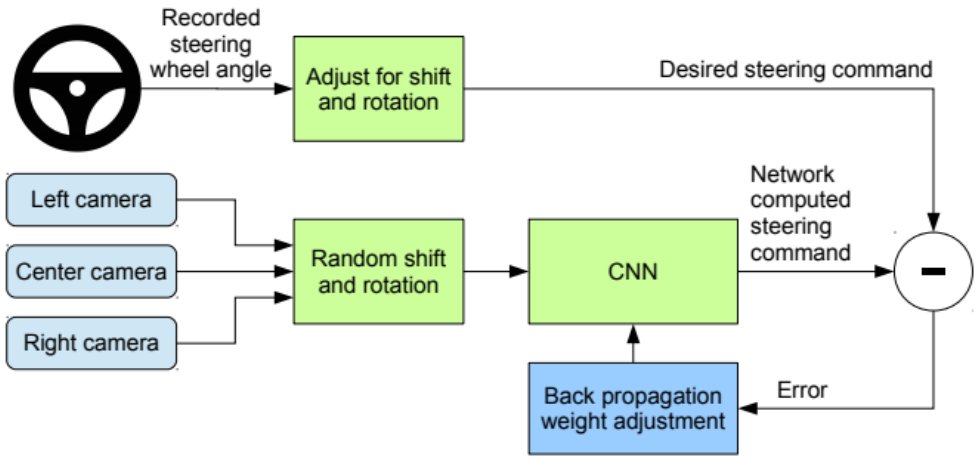


Figure 2.6: Apollo high level architecture and data flow. Image from Fan et al. [36].

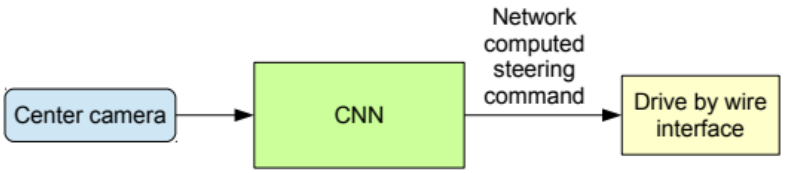
decider chooses the best trajectory for the maneuver given the cost of each trajectory, car status, traffic regulations. This approach allows for dealing with different traffic regulations that apply for different lanes of the same road [36].

A different approach to autonomous cars is to learn a mapping from input images to steering angle and speed commands in an end-to-end manner. Bojarski et al. [4] were the first to apply this method to a real-sized car with the modern deep learning techniques. They collected 72 hours data with different cars in different weather and lighting conditions from various places. For the data acquisition, they installed three cameras on the car behind the windshield and recorded timestamped videos from the left, right and center cameras along with the steering commands controlled by a human driver. During training, they augmented the dataset by random shifting and rotating the images and adjusting the recorded commands accordingly. The trained

model then successfully steered the car by using the images only from the central camera. Figure 2.7 demonstrates the training and testing steps of this approach.



(a)



(b)

Figure 2.7: (a) End-to-end training scheme. (b) Steering command inference from raw camera images. Image from Bojarski et al. [4].

Bechtel et al. [7] replicated the study [4] with a small-scale, low-cost platform using a web camera and a Raspberry Pi 3 for inference. They conducted successful experiments in a specially built test course for their RC car.

Do et al. [8] also implemented a similar approach in another RC car platform using Pi camera and Raspberry Pi 3 for inference. Instead of learning steering angles in a regression model, they learned a steering angle probability for discretized steering angle space. In addition to basic lane following, they also learned to turn left or right when the corresponding traffic sign is encountered. The traffic sign diameter was 15 cm in their experiments.

There are several traffic scene segmentation datasets available. Currently, the largest and the most comprehensive one is ApolloScape [12]. It is followed by Cityscapes [13], KITTI [14], and Mapillary Vistas [15] datasets. Because these datasets are created for real-sized cars on real roads, they were not suitable for us, so we had to create our own segmentation dataset. The existing traffic sign and signal classification datasets [16, 17, 18, 19, 20], on the other hand, was useful to train an initial classifier as they are mostly independent of the scene and car size.

Sakai et al. [35] present a collection of various autonomous navigation algorithms implemented in Python Programming Language in their basic forms. The collection includes aforementioned hybrid A*, Frenet optimal trajectory planner, state lattice planner, Stanley controller, and pure pursuit controller algorithms.

Unlike Stanley, Junior, Autoware and Apollo, we don't have a detailed map of the driving course. As a result, our best option is to follow the lanes unless a traffic sign or some other condition mandates otherwise. For the same reason, we have to create our reference paths for our local trajectory generation either from the online detected lane centers or according to the traffic regulations. Meyer et al. [6] study semantic lane segmentation for mapless driving, which bears similarities to our lane detection approach. Authors' motivation is the fact that as the high definition maps quickly get out of date due to constructions an autonomous car should also be able to perform basic navigation tasks without a precise map, but possibly with a coarse map, specifically for intersections. They start with creating their own dataset by extending Cityscape [13]. Their approach is to annotate the road surface as ego lane, parallel lane, and opposite lane and learn these regions with a semantic segmentation model.

Stanley controller algorithm is weak to discontinuities along the trajectory as it directly drives towards the closest point on the trajectory. Stanley and Junior guarantees a smooth trajectory by smoothing already known center reference lines or post processing the output of hybrid A*. We cannot guarantee a smooth path as we use discontinuous predefined path in response to traffic signs and signals or due to instantaneous segmentation errors in the lane detection. Conversely, as pure pursuit controller drives along an arc it is less likely to be affected by discontinuities.

CHAPTER 3

ARCHITECTURAL OVERVIEW

In this chapter, we introduce our proposed architecture of the mini driverless car with its hardware and software components. We also present our simulation environment, which was implemented to speed up the development and verification processes.

3.1 Hardware Configuration

Autonomous cars are equipped with a powerful central computer, actuators and many sensors such as IMU, Camera, and LIDAR. Our mini driverless car also has the equivalent hardware components as shown in Figure 3.1. Details of these components are given in Table 3.1.

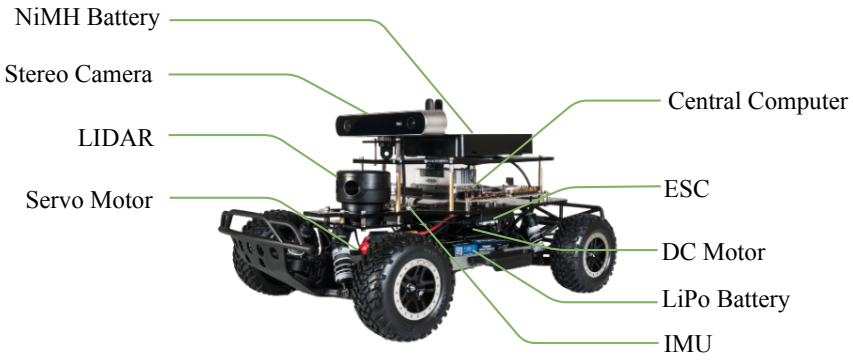


Figure 3.1: Hardware components of the mini autonomous car.

The central computer runs various sophisticated algorithms to fuse raw data from sensors to achieve environmental perception, select the best possible action accordingly, and finally send speed and steering angle commands to the ESC in order to execute the action. The ESC generates necessary electronic signals from the commands and

Table 3.1: Mini autonomous car hardware configuration details.

Component	Description
Vehicle	TRAXXAS SLASH 4X4 PLATINUM EDITION
Central Computer	NVIDIA Jetson TX2 Developer Kit
Stereo Camera	Stereolabs ZED Camera
2D LIDAR	Scanse Sweep LIDAR
ESC	Vedder Electronic Speed Controller
IMU	SparkFun 9 DoF Razor IMU M0
USB3.0 Hub	USB 3.0 7-Port Hub with 2 Charging Ports UH720
Joystick	Logitech F710 Wireless Gamepad (940-000142)
LiPo Battery	4200 mAh 7,4V 25C
NiMH Battery	MARC Power Lite 3400 mAh 16V and 12V outputs

feeds them to servo and DC motors to control steering angle and speed, respectively. We replace the stock ESC that ships with the vehicle with a different speed controller known as VESC, an open source ESC, since it is highly configurable and thus supports full control at lower speeds [37]. The hardware setup also provides a joystick control in order to acquire dataset from a human driver and take over the control during the autonomous drive in case of an emergency. We use two different batteries to power the vehicle. While NiMH battery powers the central computer and sensors through the USB 3.0 Hub, LiPo battery supplies current to the actuators as they need a more consistent power source.

3.2 Software Architecture

We use ROS [29] to implement our architecture. ROS defines notion of nodes and allows nodes to communicate through publish/subscribe and reply/request mechanisms. Similar to previous studies [21, 22, 23], we also rely on publish/subscribe mechanism for the data flow between our nodes. Nodes subscribe to the message streams from other nodes. The result of the computation of a node is then published to the other nodes. This asynchronous messaging allows the software to act as a data processing

pipeline. The architecture is roughly grouped into four modules.

- **sensor interface –** The sensor interface reads raw data from individual sensors, converts them to meaningful engineering data, then feeds them to the other modules.
- **scene interpretation –** The scene interpretation module makes sense of the environment by finding traffic lanes, detecting obstacles, and classifying the traffic signs. Once the objects of interests are detected and classified, the module also locates these objects in the real world coordinates with respect to car's body frame. Then it decides on a high-level behavior that best fits to its current perception of the environment such as keeping a lane or stopping on a red light.
- **navigation –** The navigation module first takes the kinematic constraints, traffic rules, and nearby obstacles into account and generates an optimal trajectory to realize the high-level behavior. Then it computes steering angle and speed values in order to follow the optimal trajectory as close as possible.
- **actuator –** A pre-configured firmware in the VESC that converts steering and speed commands into electronic signals to drive the motors.

Figure 3.2 illustrates the overall data flow in the software modules. Out of these modules, nodes in sensor interface and actuator modules are already provided as ROS packages. They are not implemented but configured within the scope of this thesis.

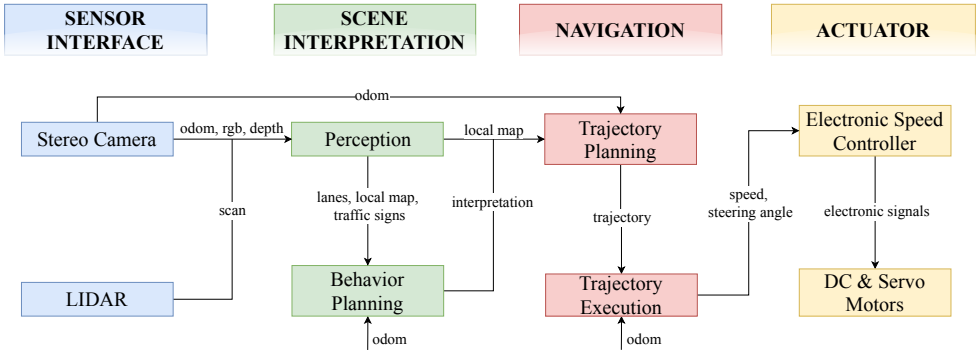


Figure 3.2: Software architecture overview.

Zed camera node is configured to publish visual odometry, RGB, and depth topics at a rate of 15 Hz. Message types that flow through these topics are well-defined

in ROS. An odometry message contains position, orientation, and linear and angular velocities with respect to the starting point. RGB image message is a rectified color image from the zed camera, which is used for image segmentation and classification. Because zed camera features stereo images, it can also publish a depth image, which is used to locate the traffic signs with respect to the body frame.

LIDAR node is configured to provide scan data to the costmap nodes in the scene interpretation module at 5 Hz rotational speed. The costmap nodes then publish local occupancy grid maps that indicate nearby objects. Scene interpretation maintains two local maps in different sizes. The small map is used in navigation module for collision avoidance. The larger map is internally used by the scene interpretation module for behavior planning to guide the trajectory planner before the navigation module observes the obstacles.

Behavior planner component decides if the car should stop or in which speed range it should move, whether it should track the lanes or follow a predefined path. At the end, the behavior planner captures the current desired behavior in an interpretation message and publishes to the trajectory planner.

Trajectory planner subscribes to the odometry, 3×3 local map, and interpretation topics so as to find a collision-free, kinematically feasible, smooth, and optimal trajectory. The trajectory message contains waypoint locations in the odometry frame and a recommended speed for each waypoint.

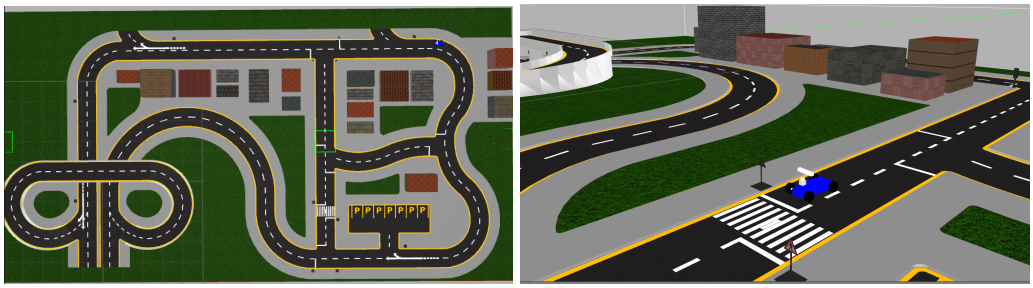
Finally, given the odometry and optimal trajectory, the trajectory execution node computes steering angles to closely follow the optimal trajectory and publishes the recommended speed and steering commands to the actuator module.

3.3 Simulation Environment

It is not always practical to try new ideas on the target platform for several reasons. First, it is too risky to run an updated version of the software in the target platform as it might crash into an obstacle. Second, the batteries have certain life time and we do not want to drain them for each immature update to the code base. Third,

deploying and testing the software on the car is time consuming. Last but not least, running the software on the car requires a large enough space with various traffic signs, lanes, a bridge and many other urban conditions, which we could barely afford a few, therefore, we had no better option than creating a simulated environment.

We used Gazebo to model our simulation world and robot car. Figure 3.3 shows the simulated urban area. It simulates every case in OpenZeka MARC 2019 except that we have to manually toggle red and green lights, and manually walk the pedestrian out of the scene.



(a) (b)

Figure 3.3: (a) Gazebo simulation environment top view. (b) The car, traffic signs, and the bridge in the simulation environment.

Simulated sensors were also carefully tuned to reflect actual sensor behaviors, but still actual camera images look blurrier. Moreover, we did not simulate the changing lighting conditions and vibrations from the actuators. Figure 3.4 gives the actual and simulated camera views for comparison.

(a) (b)

Figure 3.4: (a) Actual camera view. (b) Simulated camera view.

Despite all the peculiarities of the simulation environment, it made it possible to quickly collect datasets without needing any additional hardware, not even a joystick as it supports keyboard commands. We trained our segmentation and classification models on those datasets and tested in the simulation environment. We also developed our trajectory planning and control algorithms in the simulation, which drastically reduced the risk of damaging any equipment. In a nutshell, the simulation was not a replacement for the real world, but rather served as a flexible testbed.

CHAPTER 4

SCENE INTERPRETATION

In this chapter, we present our mini driverless car's perception capabilities and how it draws on these capabilities to interpret the current scene into high-level driving behaviors by means of a rule-based system. We first discuss our semantic segmentation model and semantic segmentation classes. Second, we explain the methods we used to locate the right and left lanes with respect to the body frame. Third, we describe our relatively small classification model that runs on top of the segmentation results to classify traffic signs and lights. Then, we demonstrate our local maps that are used for locating nearby obstacles. Finally, we introduce our behavior planner, which relates the given environmental perception with the traffic rules and picks a primitive driving pattern accordingly.

4.1 Environmental Perception

4.1.1 Semantic Segmentation

Semantic segmentation is the backbone of our visual perception component. We deploy a semantic segmentation deep learning model for which we manually annotate each camera image into ego lane, right lane, left lane, roadside and traffic sign classes using an image annotation tool, namely *labelme* [38]. Once trained, the segmentation model classifies each pixel of the input image into these classes. In order to visualize the labels, we assign a color for each class. Figure 4.1 illustrates segmentation labels blended on the camera images. We annotate traffic lights and traffic signs with green, ego lane with blue, left lane with yellow, right lane with red, roadside with pink and all other pixels are considered background and assigned black color.

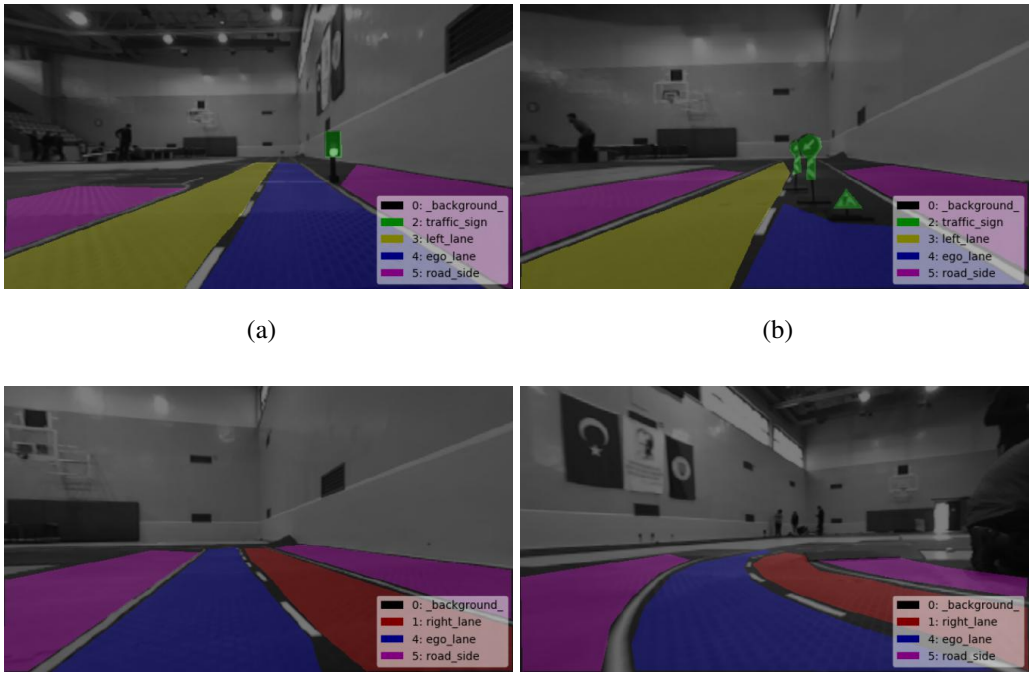


Figure 4.1: (a) Traffic light, ego lane, left lane, and roadside labels. (b) Traffic signs, ego lane, left lane, and roadside labels. (c) Ego lane, right lane, and roadside labels. (d) Labels on a curvy road segment.

We adopt U-net architecture for our traffic scene segmentation task inspired by Srihari Humbarwadi [39]. U-net works well with a few training images and yields fast and precise segmentation results. [40]. As shown in Figure 4.2, the U-net architecture is in shape of U, thus the name U-net. Our slightly modified architecture inputs an RGB image of size $128\times128\times3$ and encodes it in the contraction side through applying multiple blocks of layers. Each block takes an input and applies two 3×3 convolutions, each followed by a batch normalization, ReLU activation, and a 2×2 max pooling with stride 2 for downsampling. After each downsampling step, we double the number of feature maps (i.e., the number of filters or channels). Expansive side decodes the captured features in the bottleneck section and also is made of multiple blocks. Each expansive block applies a 2×2 up-convolution for upsampling, halves the number of feature maps, concatenates the corresponding feature maps from contracting and expansive sides, and finally applies two 3×3 convolutions, each followed by a batch normalization and ReLU activation. At the output layer, we use a

1×1 convolution followed by a batch normalization and softmax to map the resulting 16 feature maps to the desired number of classes, which is 6 in our case including the background class.

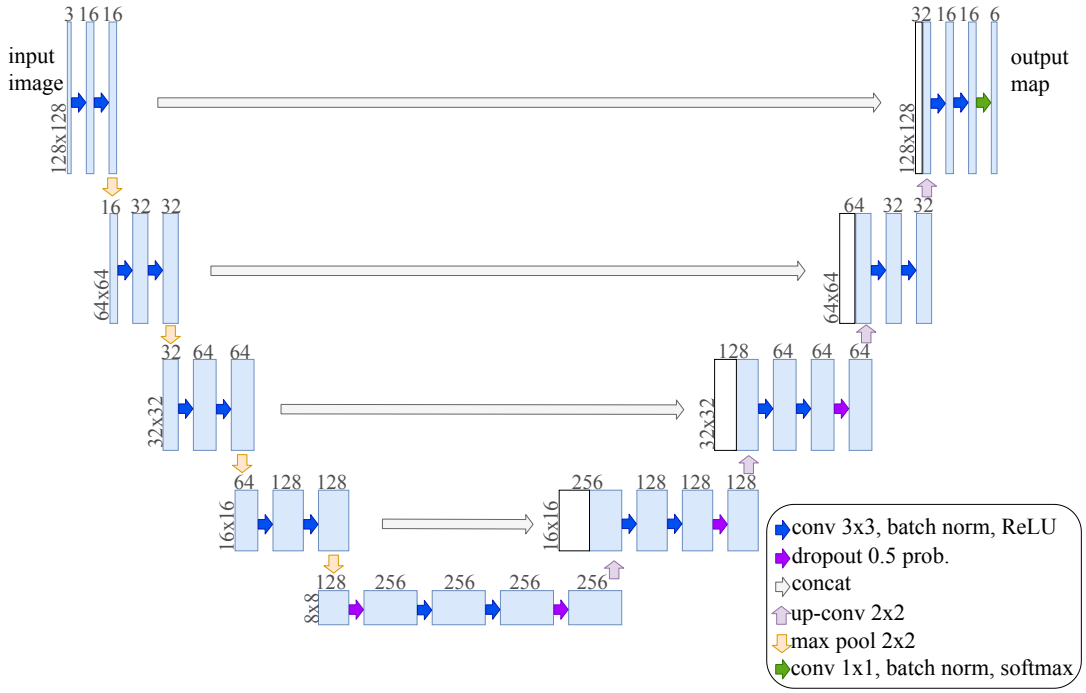


Figure 4.2: Modified U-net architecture for semantic segmentation. Each feature map is represented by a blue box. The number of channels in each feature map is given on top of its corresponding box. The width and height dimensions are given at the lower-left edge of each block of layers. White boxes denote the concatenated feature maps from the contracting side. The operations are depicted by the arrows.

In order to meet smooth driving experience requirements, the segmentation inference shall run at 10 Hz and share the limited resources on the central computer, which sets an upper bound for the size of our segmentation model. Therefore, we resize down the camera images from 672×376 to 128×128. We also lower the initial number of filters to 16, which is 64 as described in the original U-net paper. Note that as opposed to the original architecture, we use paddings for the convolutional layers so that the output of a convolution layer has the same width and height as the input. As a result, we don't crop contracting feature maps before concatenating them with the corresponding feature maps on the expanding side. Another deviation from the original architecture is that we train the model in batches (batch size of 8), so we

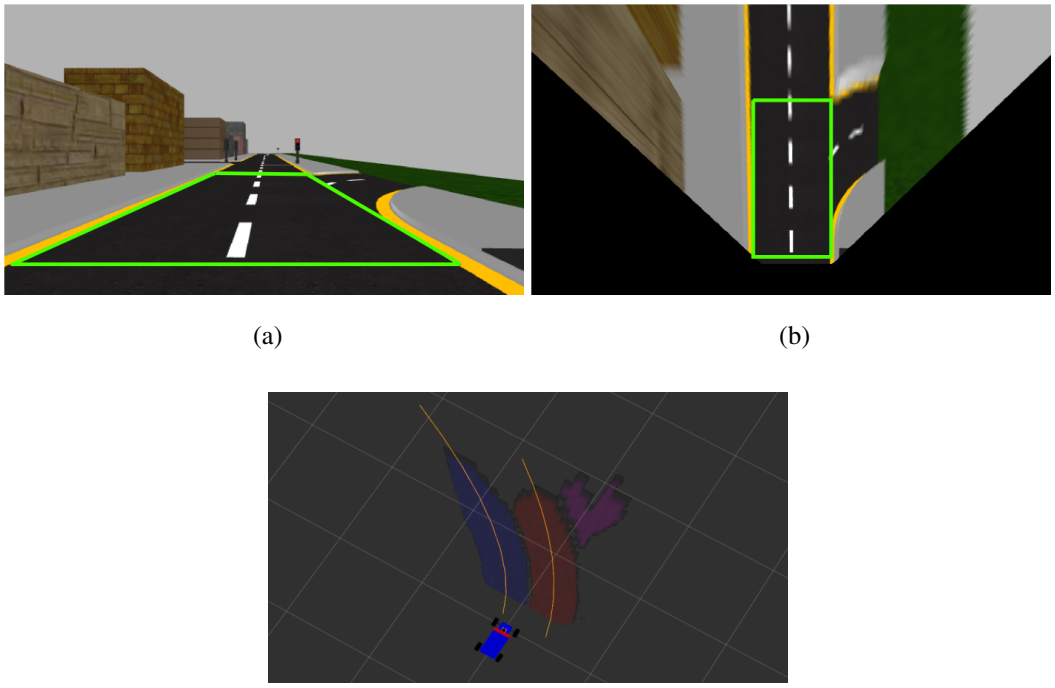
apply batch normalization following each 3×3 and 1×1 convolutional layer. We also place additional dropout layers to the bottleneck section and expansive side. We use Adam optimizer with sparse categorical cross entropy loss function from Keras which ships with TensorFlow as a high-level API [41, 42].

4.1.2 Lane Detection

Because we are trying to navigate without a prior map of the environment as in the study of Meyer et al. [6], the lane detection is used not only for adhering to the traffic rules, but also for the whole navigation task until the car encounters a traffic sign to change direction.

Our strategy for the lane detection is based on the semantic segmentation. We first run an argmax on the output segmentation map across the channel axis and assign a color to each channel as shown in Figure 4.1. Then we apply inverse perspective mapping to obtain birdseye view as in Figure 4.3. The corners of the quadrilaterals in Figure 4.3 are selected such that each pixel in the birdseye view represents a 1×1 cm^2 in the world. Finally, we create a point cloud from the birdseye view by downsampling it by a factor of 2 for faster second order polynomial fitting to the lanes. The resultant birdseye point cloud is also given in Figure 4.3.

In order to find midpoints along the lanes and make it more robust to erroneous segmentation patches, we apply sliding window and guided searches for each lane. Both search types are illustrated in Figure 4.4 for the ego lane. It is straightforward to apply the same procedure for the right and left lanes. While the sliding window search uses histogram to find starting lane locations, guided search relies on the previous polynomial fits with some margin. Every time we lose a lane, we initiate a sliding window search for that particular lane. Once the lane is found, we switch to the guided search as it is computationally less demanding. For each lane, we average the coefficients of up to last 5 polynomial fits and obtain the linearly spaced 9 points along the average polynomial curve. These 9 midpoints for each lane are the waypoints that describe the lanes on the road.



(a) (b)

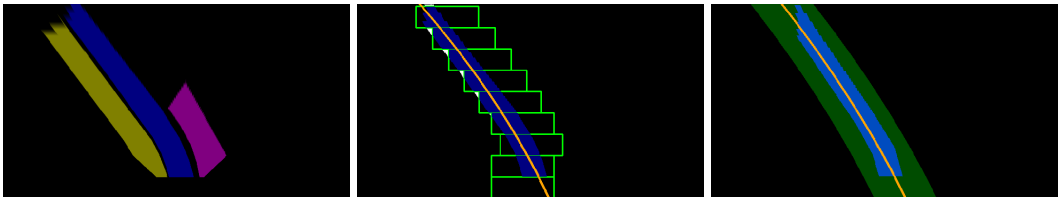
(c)

Figure 4.3: (a) Before inverse perspective mapping. (b) Birdseye view after inverse perspective mapping. (c) Birdseye view point cloud with ego and right lane curves in orange.

4.1.3 Sign Detection

Traffic signs and lights regulate the car's behavior on certain conditions and ensure the car travels the entire course with appropriate direction indications. We treat the traffic lights the same way as the traffic signs, so the methods used for the traffic signs also apply to the traffic lights.

Though there exist multiple well-established object detection deep learning architectures with different size, performance and accuracy trade-offs [43, 44], our limited resources in memory and computational power as well as the soft real-time requirements at 10 Hz challenge us to use a more resource-friendly approach. Instead of running an object detection model, we run a much simpler classifier model on the segmented traffic sign patches. This approach also saves us from labelling the whole dataset for the object detection in addition to the semantic segmentation annotations.



(a) (b) (c)

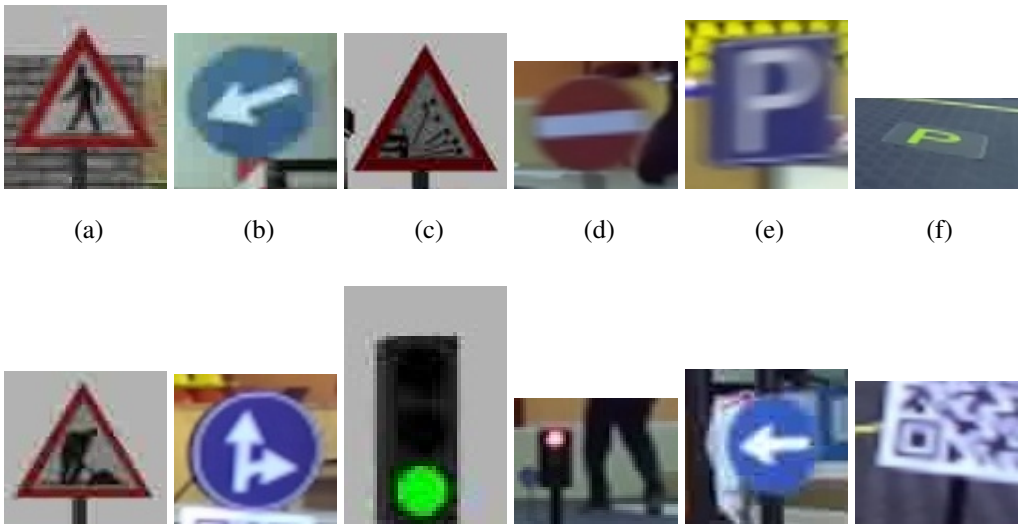
Figure 4.4: (a) Segmentation birdseye view image. (b) Sliding window search for the ego lane. (c) Guided search for the ego lane on the next image frame after the sliding window search. The green color denotes the sliding windows and guided search area. The orange color denotes the averaged polynomial curve.

On the downside, we cannot detect the signs in an end-to-end fashion without resorting to computer vision techniques for finding bounding rectangles for each sign patch in the segmentation output [45].

We first create a classification dataset selecting applicable signs from existing datasets [16, 17]. Because these are small image patches irrespective of the scene, they worked well for our miniature world. Having trained the classifier with existing datasets, we run the classifier model along with the segmentation on our training courses, which contain more traffic signs than usual. Then, we semi-automatically create a new classification dataset by cropping and saving the classified traffic sign patches. We later manually go through the saved image patches and ensure they are in the correct class folder. Whenever an inappropriate image patch is mistakenly classified as a traffic sign, we also put them into a special negative class folder. Augmenting our classification dataset in this manner significantly improves our classifier as it learns from its own mistakes.

We use *PedestrianCrossing*, *KeepLeft*, *LooseGravel*, *NoEntry*, *Parking*, *ParkingSlot*, *RoadWork*, *StraightOrRight*, *TrafficLightGreen*, *TrafficLightRed*, *TurnLeft*, and *Negative* classes. Figure 4.5 illustrates these classes with image patches automatically cropped from the training courses. The patches are converted to grayscale and resized to 32×32 before being fed into the classification model. The classification architecture is depicted in Figure 4.6. We train the model using Adam optimizer with cross entropy loss in 32 batches applying shift, rescale, shear, zoom, and brightness

augmentation methods from Keras during training [41, 42]. Algorithm 1 explains the overall classification process on a traffic sign segmentation mask.



(a) (b) (c) (d) (e) (f)

(g) (h) (i) (j) (k) (l)

Figure 4.5: Classified and saved traffic sign images from real and simulation training courses. (a) PedestrianCrossing, (b) KeepLeft, (c) LooseGravel, (d) NoEntry, (e) Parking, (f) ParkingSlot, (g) RoadWork, (h) StraightOrRight, (i) TrafficLightGreen, (j) TrafficLightRed, (k) TurnLeft, and (l) Negative.

4.1.4 Obstacle Detection

We maintain two different local occupancy grids in size and resolution in the perception component. Each occupancy grid is driven by the same laser scan message. While 3×3 grid with 0.1 meter/cell resolution is passed to the trajectory planner, 4×4 grid with 0.2 meter/cell resolution is used to analyze obstacle in the neighborhood that could change the current driving behavior such as speed profile, which is also a parameter to the trajectory planner. Figure 4.7 demonstrates the local occupancy grids in action. The car is always centered on the grids; therefore, the grids also move as the car moves.

We generate two events from the large occupancy grid. The first event, *ObstacleAhead* indicates there is an obstacle ahead of the car within a range of 2 meters. The

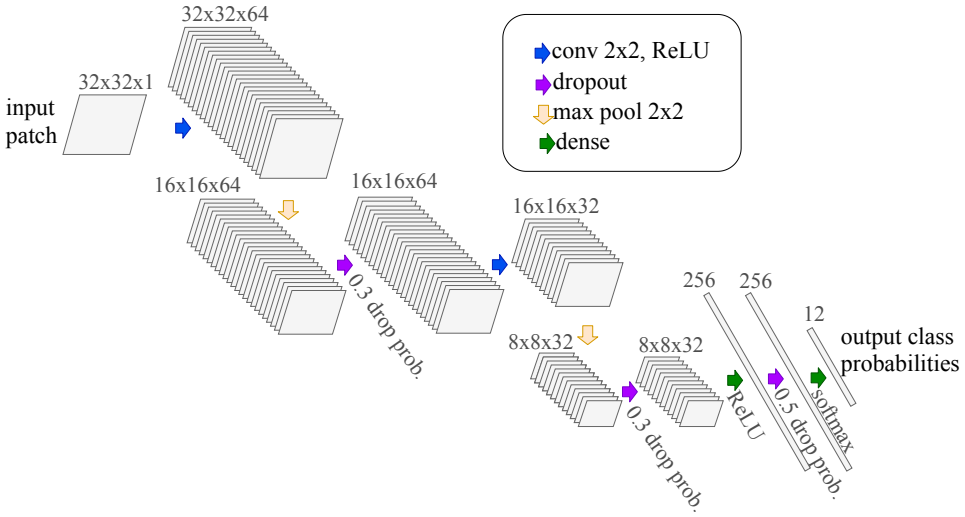


Figure 4.6: Classification architecture. Layer shapes are given on the corresponding layers.

other event, *DangerousObstacle* alerts that there is an obstacle dangerously too close to the car within 0.7 meters. Algorithm 2 gives the steps for generating these events.

4.2 Behavior Planning

Behavior planner consumes the events generated by the perception component. It transitions the car into appropriate states depending on the hierarchical finite state machine given in Figure 4.8 as the events arrive. When the car is in WAIT state, if there is no obstacle ahead and red light is not observed, transition 1 is executed and the car starts driving itself. While the car is moving, if it observes a pedestrian crossing sign or a red light, it performs transition 2 and starts waiting until transition 1 conditions are satisfied. In DRIVE state, the default behavior is to follow the lanes at normal speed, 0.9 m/s. When the car comes across any of straight-or-turn-right, turn-left, or park signs, it stops lane following by sourcing predefined waypoints for these signs to the trajectory planner instead of the waypoints extracted from the lanes. In the case that the last predefined waypoint is reached or a manual intervention through a joystick for an emergency while following a predefined path occurs, it executes transition 4 to put the car back into the lane following state. If any of LooseGravel, KeepLeft, RoadWork or DangerousObstacle events is reported while driving at the

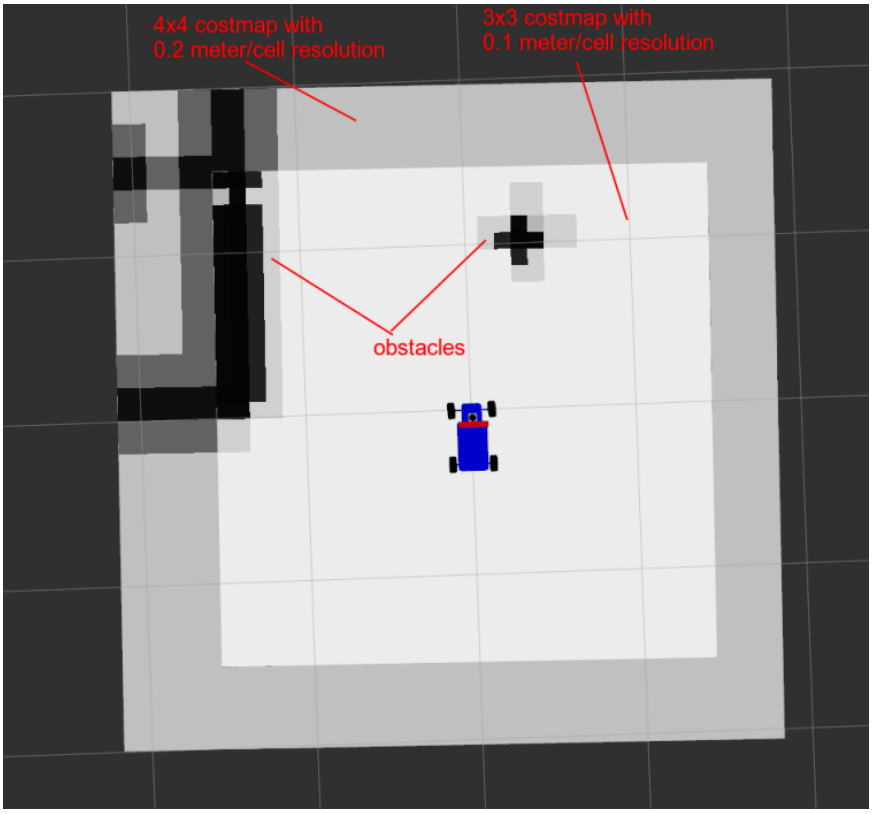


Figure 4.7: The small and large local occupancy grids.

normal speed, the car goes into SLOW SPEED state through transition 5 and starts driving at 0.5 m/s. When none of these events are reported, the car switches back to the NORMAL SPEED state. Finally, when the car sees a parking slot frame on the ground, it performs transition 7 into the PARK state, in which the car moves into the parking slot and stops.

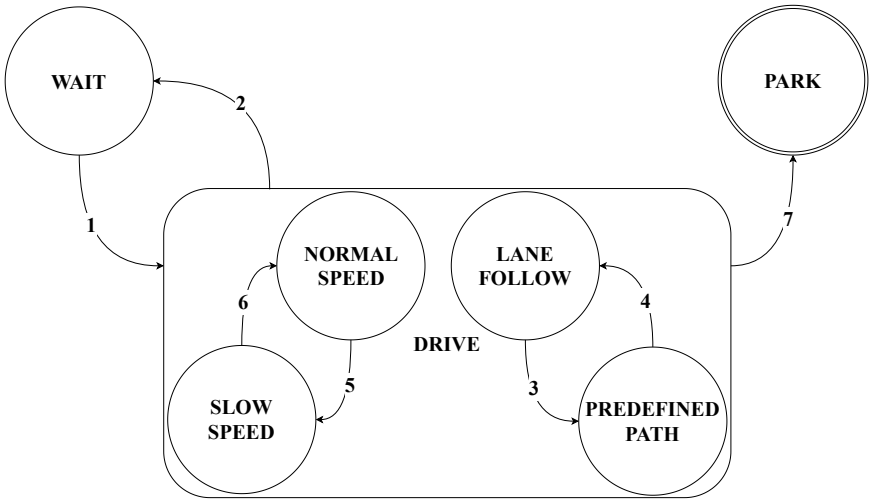


Figure 4.8: Hierarchical finite state machine that governs the high-level behavior of the car. The numbered arrows 1 to 7 denote the transitions between states. DRIVE state has two sub-states to control speed profile and waypoint source to the navigation module. PARK is the terminal state.

Algorithm 1 Classification of segmented traffic signs.

procedure PARSE-TRAFFIC-SIGNS($mask, rgb, depth, intrinsic, record, classes$)
    $cx, cy, fx, fy \leftarrow intrinsic$ ▷ Unpack camera intrinsic parameters
    $rects \leftarrow list()$
    $k \leftarrow 8$ ▷ Some pixel margin to the sign patch
    $contours \leftarrow cv2.findContours(mask)$ ▷ Use OpenCV to get traffic sign contours
    for $contour$ **in** $contours$ **do**
        $rect \leftarrow cv2.boundingRect(contour)$ ▷ Use OpenCV to get a bounding rectangle of each contour
        if $distance(rect, rects[i]) < 48$ **then**
            $rects[i] \leftarrow merge(rect, rects[i])$ ▷ Merge rectangles if they are too close
        end if
        $rects.append(rect)$
    end for
    $t \leftarrow currentTime()$
    for $class$ **in** $classes$ **do**
        $events[class] \leftarrow$ **False**
        if $t - last(detections[class]).t > 2$ **then**
            del $detections[class]$ ▷ Remove more than 2 second old detections
        end if
    end for

Algorithm 1 Classification of segmented traffic signs (continued)

for u, v, w, h **in** $rects$ **do**
  $sign \leftarrow rgb[v - k : v + h + k, u - k : u + w + k]$
  $class, confidance \leftarrow classify(sign)$ ▷ Run inference
  if $confidance > 0.8$ **then**
    $z \leftarrow mean(depth[v : v + h, u : u + w])$
    $x \leftarrow z\frac{u - c_x}{f_x}$
    $y \leftarrow z\frac{v - c_y}{f_y}$
    $detections[class].append(x, y, z, t)$
    if $len(detections[class]) > 3$ **and** $last(detections[class]).x < 2.$ **then** ▷ We see the sign more than 3 times with enough confidence within a range of 2 meters, so we can safely report that we are tracking it
      $objects[class] \leftarrow (x, y, z, t)$
      $events[class] \leftarrow$ **True**
      if record **is True then**
        $name \leftarrow unique()$
        $save(sign, class, name)$ ▷ Record the sign to further enrich the dataset
      end if
    end if
  end if
end for
return $detections, objects, events$
end procedure

Algorithm 2 Obstacle events.

Require: Pose x, y, θ extracted from the latest odometry message. θ is the yaw angle in radian.

procedure PARSE-OBSTACLES($grid, x, y, \theta$)

  $nonzeroy, nonzerox \leftarrow nonzero(grid)$ ▷ Get nonzero cells (i.e., the obstacle cells)

  $obstaclex \leftarrow nonzerox * grid.resolution + grid.origin.x$

  $obstacley \leftarrow nonzeroy * grid.resolution + grid.origin.y$

  $diffx, diffy \leftarrow obstaclex - x, obstacley - y$

  $angles \leftarrow arctan(\frac{diffy}{diffx})$

  $angles \leftarrow rad2deg(\theta - angles)$

  $mask = (angles < 10)\&(angles > -10)$

  $events[ObstacleAhead] \leftarrow count\_nonzero(mask) > 0$

  $distances \leftarrow \sqrt{diffx^2 + diffy^2}$

  $mask = (angles < 10)\&(angles > -10)\&(distances < 0.7)$

  $events[DangerousObstacle] \leftarrow count\_nonzero(mask) > 0$

  return $events$

end procedure

CHAPTER 5

NAVIGATION

In this chapter, we focus on our trajectory planning and control algorithms. The trajectory planner receives a primitive behavior mode, for our case, either stop or cruise mode, along with a set of waypoints as a reference path and delivers a safe, feasible trajectory consisting of another set of waypoints each with a recommended speed to the control block. The control block executes the trajectory by adjusting the steering angle in order to closely follow the waypoints at recommended speeds.

5.1 Trajectory Planning

We build cubic spline curves from reference waypoints to represent reference paths. Because the map of the environment is not known a priori for our scenarios, the reference waypoints are either extracted from the lanes or predefined for a specific traffic sign. In order to generate trajectories along a reference path, we use Frenet frame [24, 28] defined by normal and tangential vectors $\vec{n}_r$, $\vec{t}_r$ on the reference path.

We define normal trajectory $d(t)$ and tangential trajectory $s(t)$ to the reference path $\vec{r}(s)$ as shown in Figure 5.1. To use the same convention as Yoneda et al. [28], we also let $d(t)$ and $s(t)$ be offset pattern and distance pattern, respectively. While the offset pattern corresponds to lateral displacement, distance pattern corresponds to longitudinal acceleration and deceleration. In order to obtain a trajectory $\vec{x}$ in Cartesian coordinates, we combine the two patterns by the equation

$$\vec{x}(s(t), d(t)) = \vec{r}(s(t)) + d(t)\vec{n}_r(s(t)). \tag{5.1}$$

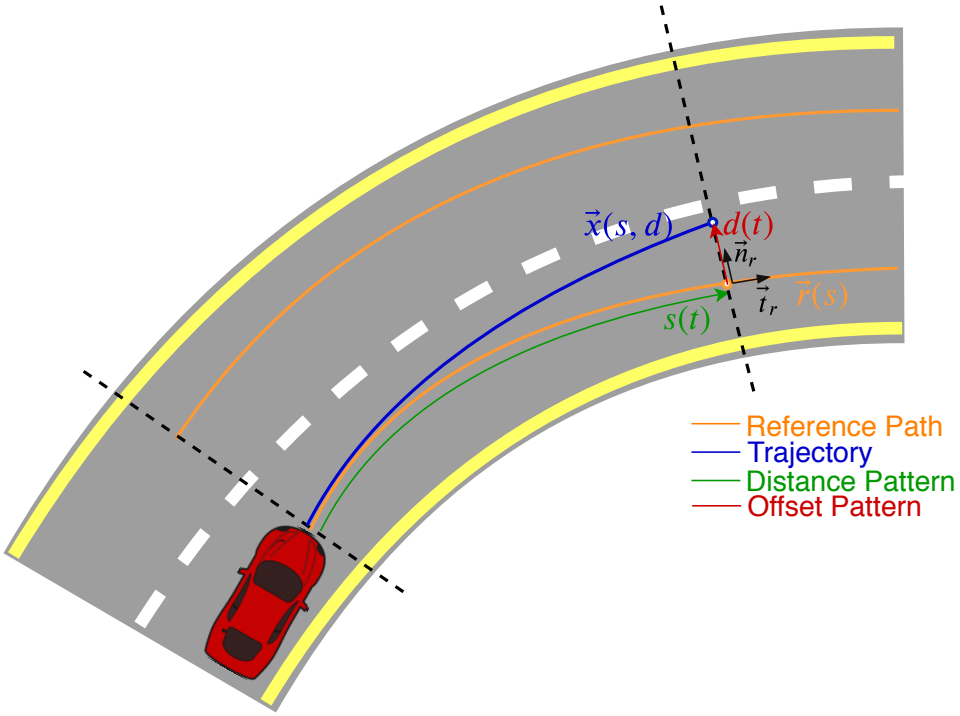


Figure 5.1: Frenet frame based on the right reference path extracted from the right lane.

Offset and distance patterns are represented by 5-degree polynomials as given by the equations

$$d(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5, \tag{5.2}$$

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5. \tag{5.3}$$

We generate a set of trajectories forward simulating various terminal conditions for offset and distance patterns. a_i and b_i are calculated using initial conditions $[d_0, \dot{d}_0, \ddot{d}_0]$, $[s_0, \dot{s}_0, \ddot{s}_0]$ at t_0 and terminal conditions $[d_1, \dot{d}_1, \ddot{d}_1]$, $[s_1, \dot{s}_1, \ddot{s}_1]$ at $t_1 = t_0 + \Delta T$, where ΔT is a preview time in seconds. We use $4 < \Delta T < 6$ with 0.2 second increments for our offset and distance patterns. The offset pattern is defined as a quintic function to control the lateral position. The distance pattern is defined using quartic and quin-

tic functions for cruise mode and stop mode, respectively. In cruise mode, we don't pay attention to the terminal conditions at longitudinal positions, but rather we focus on keeping the speed constant around the speed profile set by the behavior planner; therefore, we set $b_5 = 0$. On the other hand, we specify a longitudinal position as a terminal condition for the stop mode expecting the car to comfortably slow down and eventually stop at the specified position.

For the offset pattern, minimizing the lateral speed and acceleration leads to a more comfortable driving experience, so we choose the terminal conditions $[\Delta d, 0, 0]$ for the candidate trajectories, where $\Delta d = \{-0.1, 0, 1.0\}$ m. For cruise mode, we choose $[\dot{s}_1 + \Delta\dot{s}_1, \ddot{s}_1]$, where $\Delta\dot{s}_1 = \{-0.1, 0, 0.1\}$ m/s and $\dot{s}_1$ is the target speed set by the behavior planner. For stop mode, we choose $[s_1, 0, 0]$ terminal conditions, where s_1 is the stop position.

For all candidate trajectories, we apply a sanity check to see if the candidate is actually drivable. If there is an overlap between an obstacle grid of the occupancy grid map and the car's bounding circle along a trajectory, we conclude that the trajectory is not drivable. Non-drivable trajectories are eliminated from the trajectory set. If no drivable trajectory is left, we set the speed to zero until a valid trajectory is found again. Otherwise, the trajectory with the minimum cost is selected among the surviving trajectories as the best trajectory and forwarded to the controller for execution. The cost function C is defined by the equations

$$C = k_{path} C^i_{path} + k_d C_d + k_s C_s, \tag{5.4}$$

$$C_d = k_j \int_{t_0}^{t_1} \dddot{d}(t)dt + k_x \Delta T + k_y \Delta d^2, \tag{5.5}$$

$$C_s = \begin{cases} k_j \int_{t_0}^{t_1} \dddot{s}(t)dt + k_x \Delta T + k_y \Delta s^2, & \text{if stop mode;} \\ k_j \int_{t_0}^{t_1} \dddot{s}(t)dt + k_x \Delta T + k_y \Delta \dot{s}^2, & \text{if cruise mode,} \end{cases} \tag{5.6}$$

where C_d and C_s are the costs from offset and distance patterns, respectively. Both terms include integral of jerks, preview times, and displacement of terminal condi-

tions. In multi-lane settings, C^i_{path} is the lane cost for *i*th lane. For our scenarios, we have only two lanes; therefore, we assign a higher cost to the left lane so as to prevent the car from occupying the left lane when the right lane is free as shown in Figure 5.2. In this way, the car automatically shifts to the left lane when the right lane is blocked. Similarly, the car chooses the right lane when it is available as it would be less costly than driving on the left lane. We use $k_j = k_x = 0.1$ and $k_d = k_s = k_{path} = 1.0$ for our cost evaluations.

5.2 Trajectory Execution

We use pure pursuit control algorithm to track the given optimal trajectory. The algorithm relies on the nonholonomic constraints of the car. Let car configuration be $\mathbf{q} = [x\ y\ \theta]^T$, where (x, y) is the position and θ is the orientation of the car. Nonholonomic constraints satisfy the equation

$$\dot{x}\sin(\theta) - \dot{y}\cos(\theta) = 0 \tag{5.7}$$

such that longitudinal and lateral forces applied on the car tires are always smaller than the maximum friction between the tires and the ground [35]. In other words we assume the car does not slip. With these constraints, we estimate the steering angle in terms of turning radius R and the wheelbase of the car L_b from Figure 5.3 by the equation

$$\delta = \tan^{-1}(\frac{L_b}{R}). \tag{5.8}$$

Given an optimal trajectory, we first find a point ahead of the car on the trajectory such that the distance between the point and the car is the smallest distance greater than a lookahead distance L. This is illustrated in Figure 5.4.

From the figure, we write down the following equations:

$$L^2 = x_L^2 + y_L^2, \tag{5.9}$$

$$R^2 = a^2 + x_L^2, \tag{5.10}$$

$$R = a + y_L, \tag{5.11}$$

$$\sin\alpha = \frac{y_L}{L}. \tag{5.12}$$

Combining (5.9), (5.10), and (5.11), we obtain

$$R = \frac{L^2}{2y_L}. \tag{5.13}$$

Substituting (5.12) in (5.13), we obtain turning radius R in terms of feedback angle α and lookahead distance L as given by the equation

$$R = \frac{L}{2\sin\alpha}. \tag{5.14}$$

Feedback angle α is computed using the car configuration $\mathbf{q} = [x\ y\ \theta]^T$ and the position of the lookahead point (x_p, y_p) in the world coordinates by the equation

$$\alpha = \tan^{-1}(\frac{y_p - y}{x_p - x}) - \theta. \tag{5.15}$$

Note that we take the midpoint of the front axle as the car position in the world coordinates.

Finally, using (5.8) and (5.14), we derive steering angle δ in terms of feedback angle $alpha$, constant wheelbase L_b, and lookahead distance L given by

$$\delta = \tan^{-1}(\frac{2L_b\sin\alpha}{L}). \tag{5.16}$$

In order to tune the controller, we need to choose a lookahead distance L. Kim et al. suggests $L = 2R_{min}$, where $R_{min} = \frac{L_b}{\tan(\delta_{max})}$ [35]. For our car, the max steering

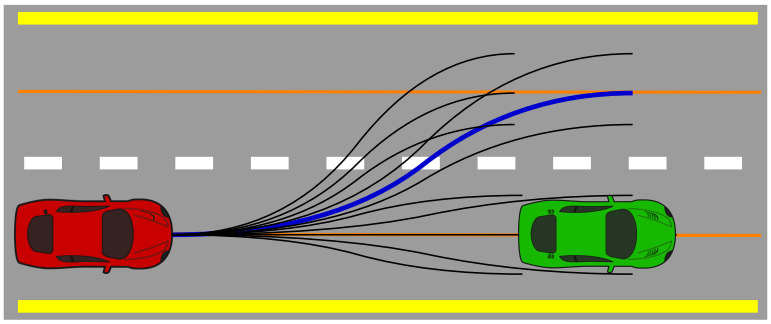
angle $\delta_{max} = 0.558$ radians and the wheelbase $L_b = 0.325$ meters. Therefore, we use $L = 1.04$ meters. Figure 5.5 illustrates the relation between R_{min} and L.

One problem with (5.16) is that it does not account for the speed. As the speed increases, steering angle δ becomes more sensitive to the feedback angle α [35]. We address this problem by adding a speed factor to lookahead distance L in the equation

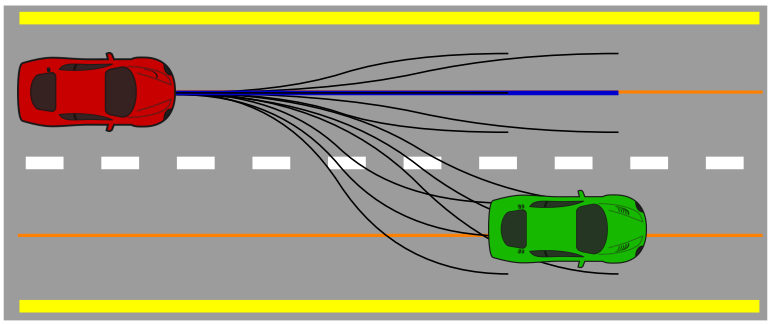
$$\delta = \tan^{-1}(\frac{2L_b \sin\alpha}{L + kv}), \tag{5.17}$$

where v is the recommended speed by the trajectory planner, k is the look forward gain. We use $k = 1$.

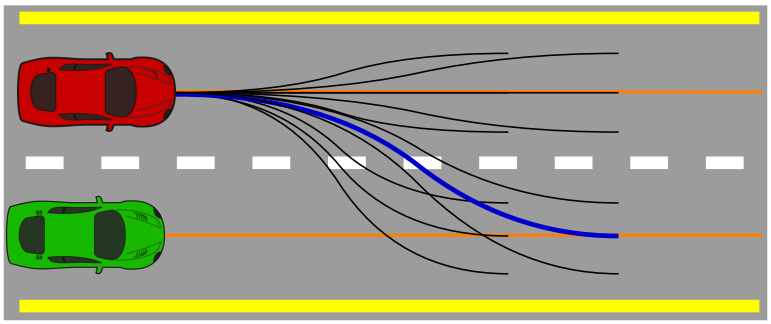
We compute a steering angle δ every time we receive an odometry message, which is published at 15 Hz and contains the current car configuration information.



(a)



(b)



(c)

Figure 5.2: Example multi-lane driving scenarios. The thick blue trajectory is the optimal trajectory for different cases. (a) The right lane is blocked, so the car attempts to change lane to the left for overtaking. (b) The right lane is still blocked, so the car keeps occupying the left lane. (c) Because the right lane is less costly when both lanes are available, the car shifts back to right lane.

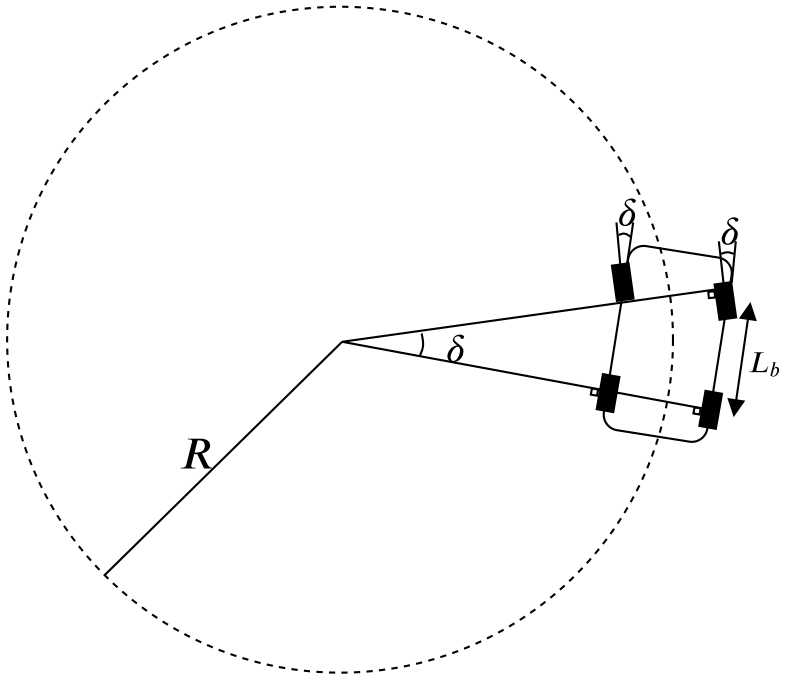


Figure 5.3: Estimation of steering angle in terms of turning radius R and wheelbase L_b.

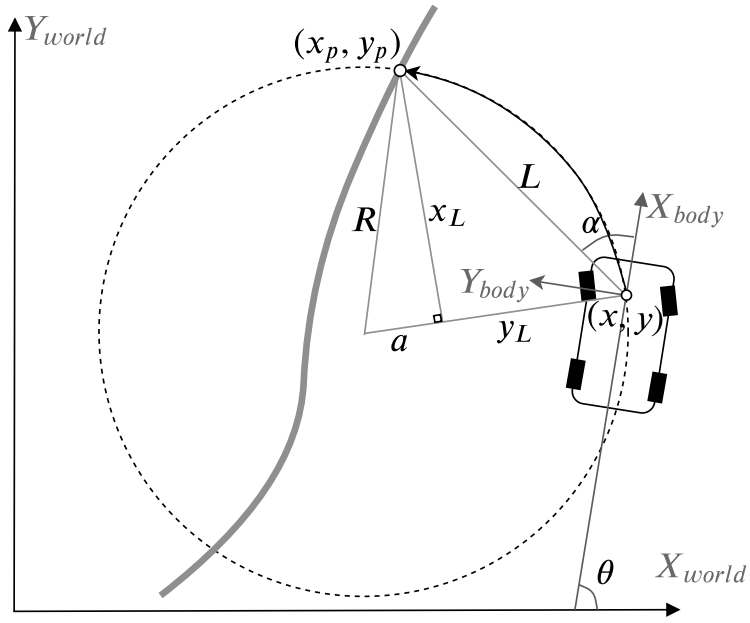


Figure 5.4: Illustration of lookahead distance L and feedback angle α.

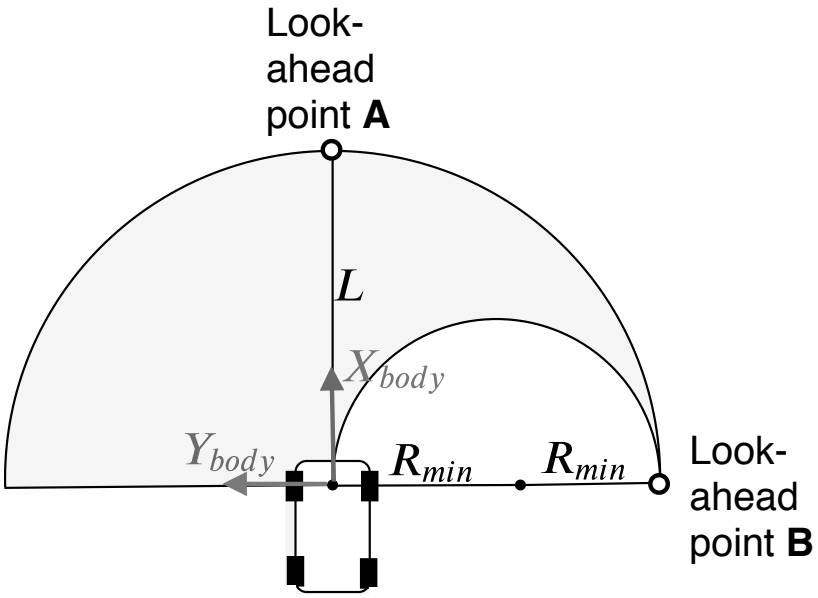


Figure 5.5: Illustration of lookahead distance L and minimum turning radius R_{min} for tuning the pure pursuit controller.

CHAPTER 6

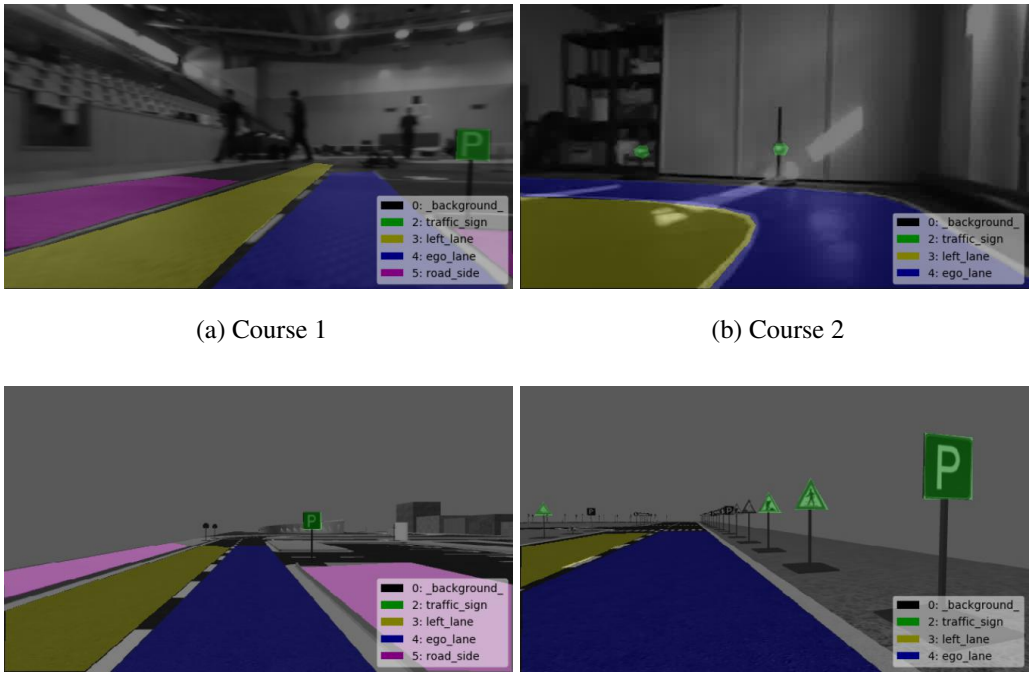
EXPERIMENTS AND RESULTS

We conducted various experiments for different scenarios in the simulation and real miniature courses. In this chapter, we introduce our courses and datasets collected from the courses. We evaluate our semantic segmentation and classification deep learning models based on these datasets. We then demonstrate the behavior of our car on different traffic scenarios.

6.1 Race Courses and Datasets

Our experiments are based on four different courses. The first course is the course provided by OpenZeka two days before the competition, so it was not available for use during the development. The second course is the one we constructed in our laboratory, which is spatially no larger than the bridge area of the actual competition course. We used it to collect simple images and test our hardware setup to ensure the car is functioning properly. The third one was developed in Gazebo simulation environment similar to the competition course. We developed the fourth course in the simulation to semi-automatically collect extra traffic sign images. Figure 6.1 illustrates the courses with semantic segmentation annotations.

We split semantic segmentation dataset into training and testing sets as shown in Table 6.1. For classification dataset, we started with a collection of relevant signs from existing datasets [16, 17, 18, 19, 20]. Then, we expand the dataset by automatically cropping the sign patches from the images collected from all our four courses as the car drives itself. The new sign patches are manually arranged and merged into the existing classification dataset. Details of the final classification dataset is presented in



(a) Course 1

(b) Course 2

(c) Course 3

(d) Course 4

Figure 6.1: Example scenes from the courses. Course 1 is the official competition course. Course 2 is a small track constructed in the laboratory. Course 3 is a simulation of Course 1. Course 4 is used to semi-automatically collect extra traffic sign images. It is not used for training or testing the semantic segmentation model.

Table 6.2.

6.2 Perception Evaluation

We evaluate our semantic segmentation model using IoU, Precision, Recall, and F1 scores for each class given in Table 6.3. Despite the fact that we used our own dataset, we compare our lane segmentation results to the results reported by Barnes et al. [46] and Meyer et al. [6] as they are mostly relevant. For ego lane, which is the main enabler for driving, Barnes et al. [46] achieve up to 85% IoU and Meyer et al. [6] achieve 80% IoU. We achieved 88% IoU on our dataset. However, note that our lanes are more obvious compared to real traffic scenes. In real scenes, lane lines are often obscured or worn-out. For the same reason, our results for neighboring lanes (i.e.,

Table 6.1: Traffic scene semantic segmentation dataset.

Course	**Training**	**Testing**
Course 1	2249	212
Course 2	169	10
Course 3	230	29
Total	2648	251

right and left lanes) are better than the corresponding results presented by Meyer et al. [6] for opposite and parallel lanes. Our ego lane precision, recall, and F1 scores are also comparable to the results provided by these studies.

One can notice from Table 6.3 that the right lane IoU is considerably smaller than that of the other lanes and road side. This is because right lane is underrepresented in the dataset as we most of the time drive the car on the right lane, effectively using it as the ego lane. The same reason is also applicable for traffic signs. Traffic signs are small, rare, and often located towards the edge of the images.

For traffic sign detection, we run a classification model on top of the semantic segmentation that proposes regions to the classification network. Table 6.4 presents the detailed performance metrics of our 97.45% accurate classifier.

We find adding an extra negative class helpful to improve overall classification performance. Nonetheless, *Negative* class also has false negatives lowering its recall as given in Table 6.4. In other words, a non-traffic sign region can still be classified as a traffic sign as shown in Figure 6.4.

In order to evaluate our traffic sign detection performance we use mAP measure defined in PASCAL VOC 2012 competition [47]. Detected signs are first sorted by decreasing classification confidence and matched up with ground truth signs. If the matched pair achieves IoU ≥ 0.5 and has the same class label, the match is considered to be a true positive. Then using this information, we build a precision/recall curve with monotonically decreasing precision. Next, AP for the class label is computed by numerically integrating the area under the curve. Finally, we compute mAP as the

Table 6.2: Traffic sign classification dataset.

Class	Training	Testing	Auto-cropped
PedestrianCrossing	704	89	470
KeepLeft	844	83	594
LooseGravel	533	60	593
NoEntry	502	83	306
Parking	497	90	237
ParkingSlot	1730	60	1790
RoadWork	1741	86	237
StraightOrRight	1106	89	848
TrafficLightGreen	422	76	192
TrafficLightRed	1147	72	1095
TurnLeft	817	93	502
Negative	984	60	1044
Total	11027	961	7908

mean of all APs [48]. Figure 6.2 shows APs for all classes and their false positive rates excluding *Negative* class.

Figure 6.3 and Figure 6.4 visualize segmentation and sign detection results on the test images taken from real courses, Course 1 and Course 2.

Figure 6.5 presents plots to compare autonomous driving with a manual drive on Course 3. When the car sees a red light it decelerates and stops at time $t = 20s$. Sudden speed jump during the deceleration is due to an instantaneous lose of the red light. When it gets closer to the red light, it detects it back and finally stops. At $t = 40s$ the car detects road work and transitions to its slow speed state and performs a left lane change. At $t = 60s$, it reaches back to the right lane and starts to take a left turn. At $t = 70s$, it once more transitions to slow speed due to a loose gravel sign detection. Between $85s$ and $95s$, the car overtakes a static obstacle blocking the left lane and it is manually stopped when it is back to the right lane.

Figures 6.6, 6.7, 6.8, and 6.9 further illustrate various driving scenarios by providing

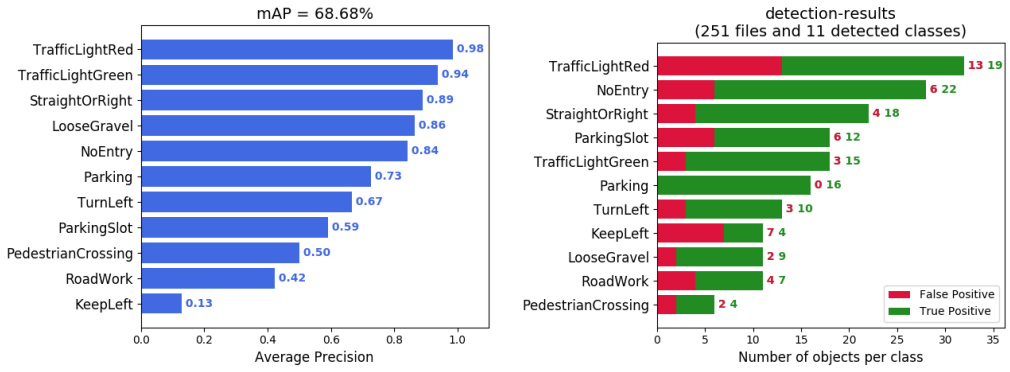
Table 6.3: Semantic segmentation test results.

Class	**IoU**	**Precision**	**Recall**	**F1**
Background	94.43%	97.81%	96.43%	97.12%
Ego Lane	88.48%	92.29%	93.18%	92.73%
Left Lane	83.52%	87.83%	94.49%	91.04%
Road Side	71.97%	78.49%	89.58%	83.67%
Right Lane	51.32%	55.60%	90.19%	68.79%
Traffic Sign	58.12%	79.52%	71.81%	75.47%
All	74.64%	81.92%	89.28%	84.80%

3D point cloud of traffic scenes and processed front view camera images.

Table 6.4: Traffic sign classification test results.

Class	Precision	Recall	F1-score
KeepLeft	100.00%	100.00%	100.00%
LooseGravel	100.00%	100.00%	100.00%
NoEntry	95.40%	100.00%	97.65%
Parking	98.86%	96.67%	97.75%
ParkingSlot	84.51%	100.00%	91.60%
PedestrianCrossing	97.72%	69.63%	97.17%
RoadWork	95.50%	98.84%	97.14%
StraightOrRight	100.00%	97.75%	98.86%
TrafficLightGreen	98.68%	97.37%	98.01%
TrafficLightRed	100.00%	97.22%	98.60%
TurnLeft	98.91%	97.85%	98.37%
Negative	100.00%	85.00%	91.90%



(a) (b)

Figure 6.2: (a) Average precision for traffic sign detection. (b) True and false prediction rates for each sign class.

Figure 6.3: Sample true sign detections on real courses.

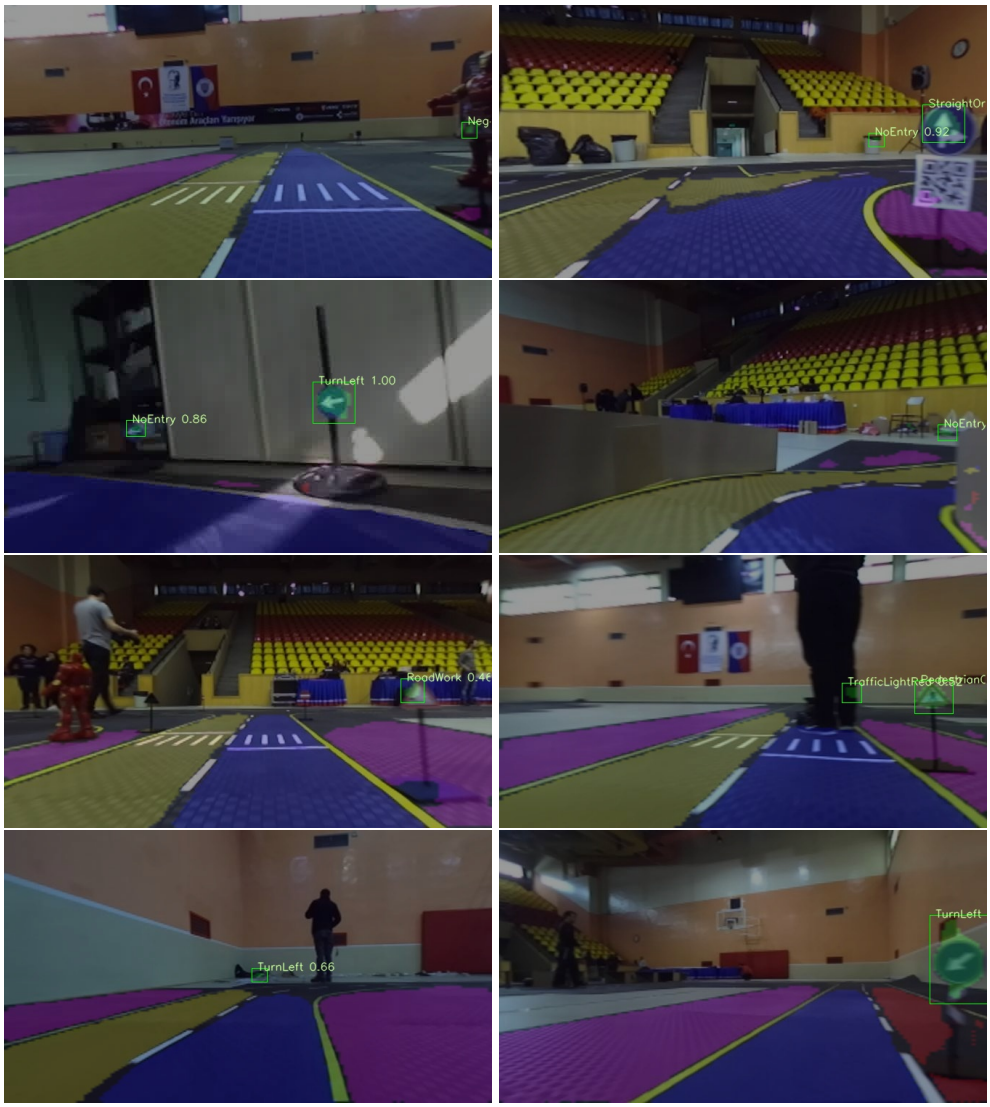
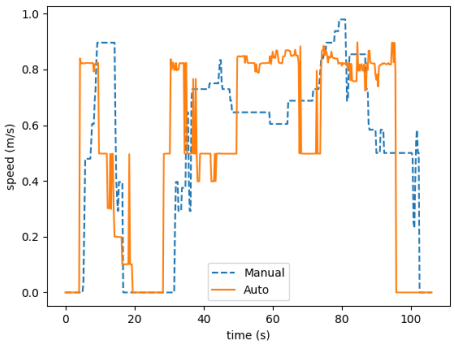
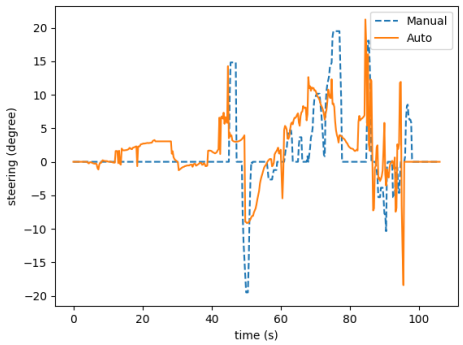


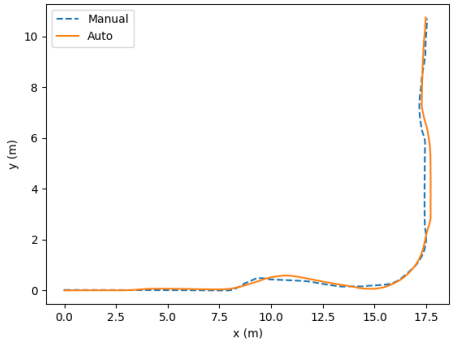
Figure 6.4: Sample false sign detections on real courses.



(a) Speed vs. time plot.



(b) Steering vs. time plot.



(c) Position x-y plot.

Figure 6.5: Comparison between autonomous driving and a human driver.

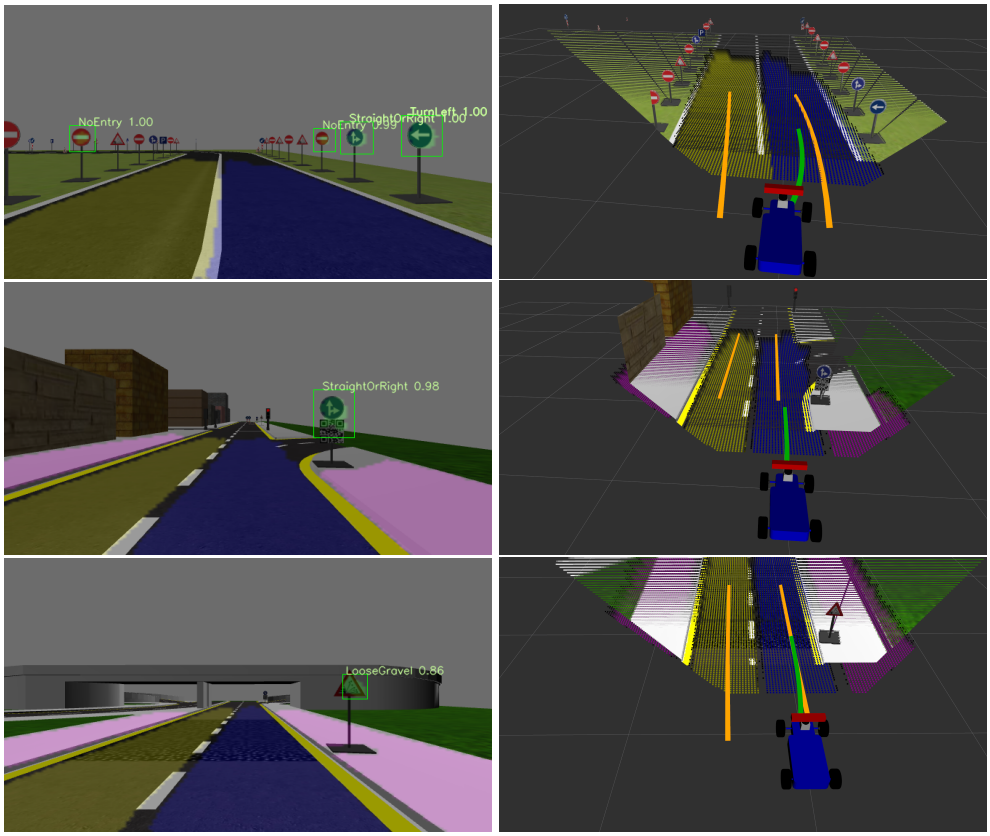


Figure 6.6: Straight road driving scenarios. For the first row, the car is configured to follow lanes and detect signs without taking any action for detections. Note that the perception component is not trained to drive on this course, but it still manages to drive.

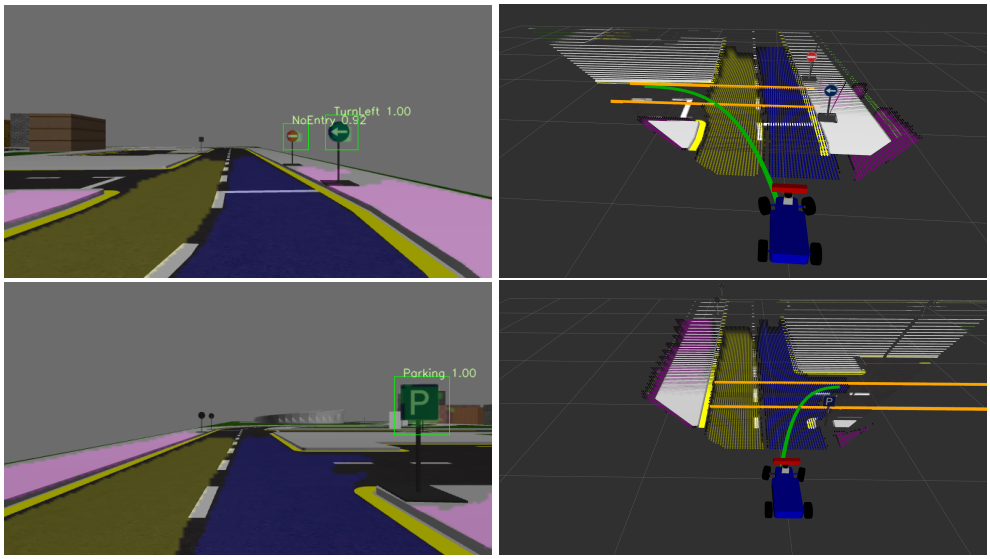


Figure 6.7: Sharp turning scenarios.

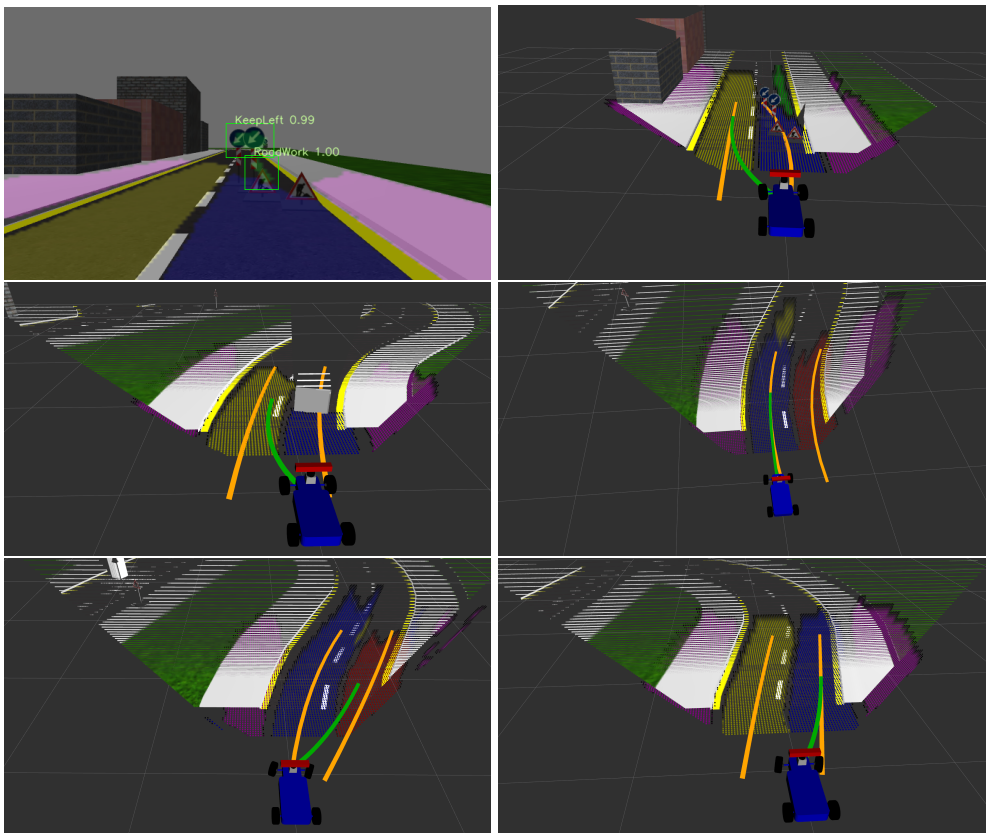


Figure 6.8: Lane change scenarios.

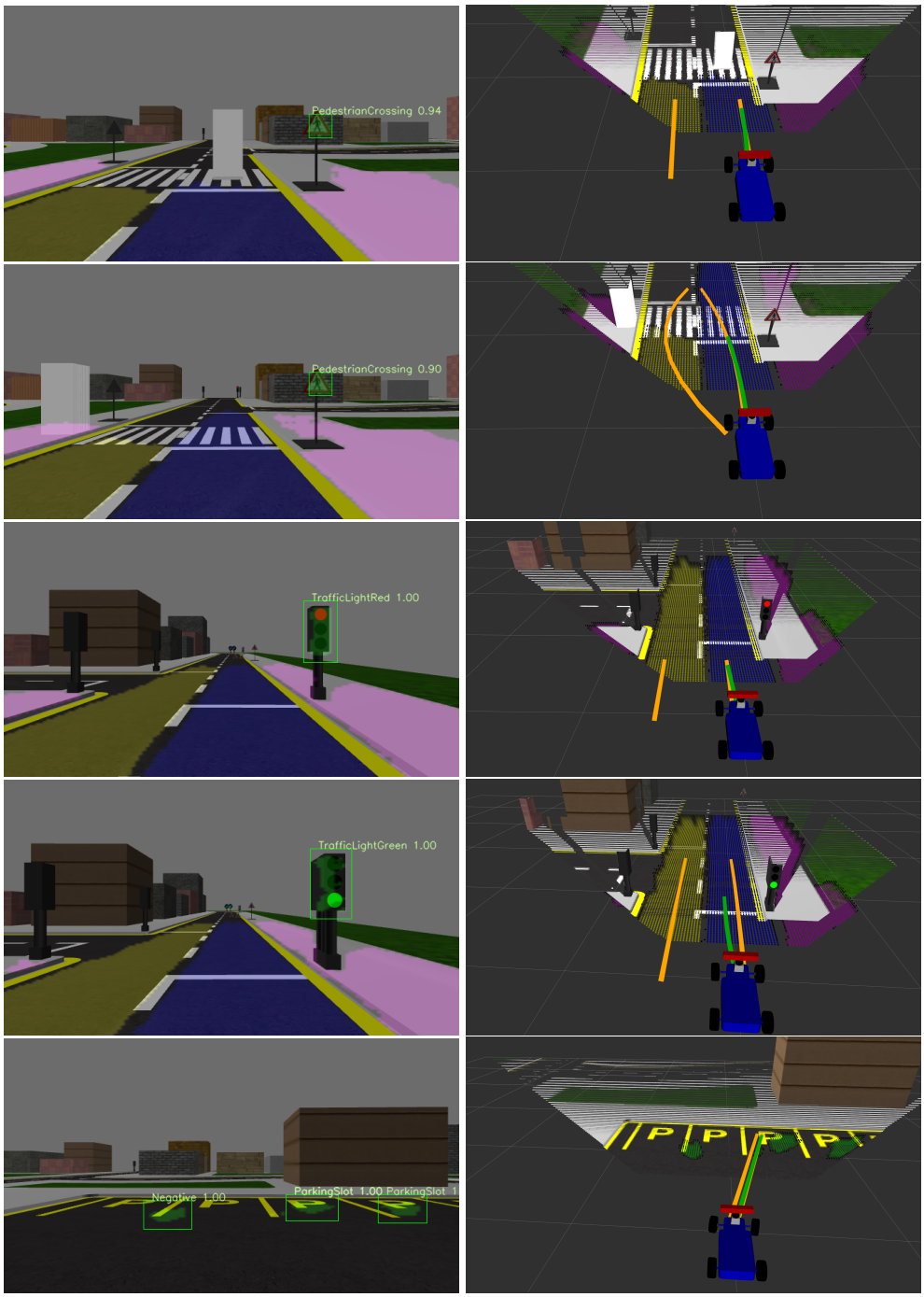


Figure 6.9: Stop scenarios.

CHAPTER 7

CONCLUSION

This thesis aimed to investigate autonomous car software architectures for urban driving scenarios. Based on the existing studies, decomposed architectures prove to be more successful in city traffic compared to end-to-end solutions. As a budget-friendly alternative, we developed a mini autonomous car with a decomposed architecture for urban scenarios including seven different traffic signs, traffic signals, bridge, overtaking a stationary car, and parking with no predefined map. These scenarios together makes the problem too complicated to be addressed with the existing mini autonomous car solutions.

Due to lack of traffic scene segmentation datasets for mini cars, we started with creating a dataset. Then we trained a U-net based model to learn ego lane, right lane, left lane, road side, and traffic signs semantics from the camera images. Learning lane semantics for right and left lanes along the with ego lane enabled the trajectory planner to implement a lane change policy such that it assigns more cost to occupying left lane when the right lane is available. We showed that we achieved 88% IoU for the ego lane, which is comparable to the related work.

For the traffic sign and signal classification, we first implemented a region proposal algorithm relying on precision of the traffic sign segmentation. Then we merged existing relevant datasets for our scenarios and augmented it by cropping the sign region proposals. Finally, we trained another relatively small deep learning model with the final dataset and achieved 97% accuracy.

We implemented a finite state machine for invoking different behaviors of our car. This behavior planner interprets environmental perception for the trajectory planner.

The trajectory planner finds an optimum trajectory for desired behavior such as target speed or target point to stop, for example, due to a red light.

Our work can be improved in a number of ways. For the perception side, the sign detection clearly has room for improvement. There are many different approaches to sign detection with varying resource demands. These approaches can be evaluated on the target platform, by modifying the approaches if necessary. An interesting approach would be to train a multi-task model for learn lane semantics as well as detecting and classifying traffic signs. For the planning side, although our trajectory planner works well for structural environments, it is not suitable for unstructured environments such as parking lots. Algorithms such as hybrid A* and state lattice planner would be better choices for parking lot navigation. In addition, we only deal with static obstacles in the current study. For responding to dynamic obstacles, we need an additional prediction component that estimates the trajectories of moving objects. The trajectory planner should also take these trajectories into account. Moreover, the trajectory planner should be extended with distance keeping mode so that the behavior planner can choose to follow moving vehicles with a safety margin. Last, we only follow lanes and directions provided by traffic signs with no sense of global direction. Integration of a coarse map would enable the car to make global plans between current and destination locations so that it could evaluate multiple directions to find the shortest path to the destination.

REFERENCES

[1] T. Holstein, G. Dodig-Crnkovic, and P. Pelliccione, "Ethical and social aspects of self-driving cars," *ArXiv*, vol. abs/1802.04103, 2018. 1

[2] M. Buehler, K. Iagnemma, and S. Singh, "The 2005 darpa grand challenge: The great robot race," 2007. 1

[3] M. Buehler, K. Iagnemma, and S. Singh, "The darpa urban challenge: Autonomous vehicles in city traffic, george air force base, victorville, california, usa," in *The DARPA Urban Challenge*, 2009. 1

[4] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *ArXiv*, vol. abs/1604.07316, 2016. 1, 17, 18

[5] C. Chen, A. Seff, A. L. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2722–2730, 2015. 1, 2

[6] A. Meyer, N. O. Salscheider, P. F. Orzechowski, and C. Stiller, "Deep semantic lane segmentation for mapless driving," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 869–875, 2018. 2, 4, 19, 30, 52, 53

[7] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun, "Deeppicar: A low-cost deep neural network-based autonomous car," *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 11–21, 2017. 2, 18

[8] T.-D. Do, M.-T. Duong, Q.-V. Dang, and M.-H. Le, "Real-time self-driving car navigation using deep neural network," *2018 4th International Conference on Green Technology and Sustainable Development (GTSD)*, pp. 7–12, 2018. 2, 18

[9] B.-C.-Z. Blaga, M.-A. Deac, R. W. Y. Al-Doori, M. Negru, and R. Danescu, "Miniature autonomous vehicle development on raspberry pi," *2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pp. 229–236, 2018. 2

[10] S. Karaman, A. Anders, M. T. Boulet, J. R. Connor, K. Gregson, W. Guerra, O. Guldner, M. Mohamoud, B. Plancher, R. T. Shin, and J. Vivilecchia, "Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at mit," *2017 IEEE Integrated STEM Education Conference (ISEC)*, pp. 195–203, 2017. 2

[11] "Openzeka marc." `https://openzeka.com/marc/`. Accessed: 2019-07-31. 2

[12] X. Huang, X. Cheng, Q. Geng, B. Cao, D. Zhou, P. Wang, Y. Lin, and R. Yang, "The apolloscape dataset for autonomous driving," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1067–10676, 2018. 4, 16, 19

[13] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3213–3223, 2016. 4, 19

[14] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361, 2012. 4, 19

[15] G. Neuhold, T. Ollmann, S. R. Bulò, and P. Kontschieder, "The mapillary vistas dataset for semantic understanding of street scenes," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 5000–5009, 2017. 4, 19

[16] R. Timofte, K. Zimmermann, and L. V. Gool, "Multi-view traffic sign detection, recognition, and 3d localisation," *2009 Workshop on Applications of Computer Vision (WACV)*, pp. 1–8, 2009. 4, 19, 32, 51

[17] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural net-*

works : the official journal of the International Neural Network Society, vol. 32, pp. 323–32, 2012. 4, 19, 32, 51

[18] V. Shakhuro and A. Konushin, "Russian traffic sign images dataset," *Computer Optics*, vol. 40, pp. 294–300, 2016. 4, 19, 51

[19] C. G. Serna and Y. Ruichek, "Classification of traffic signs: The european dataset," *IEEE Access*, vol. 6, pp. 78136–78148, 2018. 4, 19, 51

[20] S. Maldonado-Bascón, S. Lafuente-Arroyo, P. Gil-Jiménez, H. Gómez-Moreno, and F. López-Ferreras, "Road-sign detection and recognition based on support vector machines," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, pp. 264–278, 2007. 4, 19, 51

[21] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. M. Oakley, M. Palatucci, V. R. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. R. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. V. Nefian, and P. Mahoney, "Stanley: The robot that won the darpa grand challenge," *J. Field Robotics*, vol. 23, pp. 661–692, 2006. 5, 9, 11, 22

[22] M. Montemerlo, J. U. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Hähnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, A. Petrovskaya, M. Pflueger, G. Stanek, D. Stavens, A. Vogt, and S. Thrun, "Junior: The stanford entry in the urban challenge," in *The DARPA Urban Challenge*, 2009. 5, 11, 22

[23] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018. 5, 14, 22

[24] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a frenét frame," *2010 IEEE International Conference on Robotics and Automation*, pp. 987–993, 2010. 5, 13, 41

[25] G. M. Hoffmann, C. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing," *2007 American Control Conference*, pp. 2296–2301, 2007. 11

[26] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Path planning for autonomous vehicles in unknown semi-structured environments," *I. J. Robotics Res.*, vol. 29, pp. 485–501, 2010. 12

[27] J. Petereit, T. Emter, C. W. Frey, T. Kopfstedt, and A. Beutel, "Application of hybrid a* to an autonomous mobile robot for path planning in unstructured outdoor environments," in *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on*, pp. 1–6, 2012. 12

[28] K. Yoneda, T. Iida, T. Kim, R. Yanase, M. A. Aldibaja, and N. Suganuma, "Trajectory optimization and state selection for urban automated driving," *Artificial Life and Robotics*, vol. 23, pp. 474–480, 2018. 13, 41

[29] M. Quigley, B. P. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," 2009. 14, 22

[30] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 15

[31] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *ECCV*, 2016. 15

[32] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *J. Field Robotics*, vol. 26, pp. 308–333, 2009. 15, 16

[33] C. Urmson, J. Anhalt, D. Bagnell, C. R. Baker, R. A. Bittner, J. M. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, A. Kelly, D. Kohanbash, M. Likhachev, N. W. Miller, K. M. Peterson, R. Y. Rajkumar, P. E. Rybski, B. Salesky, S. Scherer, Y. Woo-Seo, R. G. Simmons, S. Singh, J. M. Snider, A. Stentz, R. Whittaker, J. Ziglar, H. Bae, B. Litkouhi, J. Nickolaou, V. Sadekar, S. Zeng, J. Struble, and M. D.

Taylor, "Tartan racing: A multi-modal approach to the darpa urban challenge," 2007. 15

[34] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," *2011 IEEE International Conference on Robotics and Automation*, 2011. 15

[35] D. Kim, C.-S. Han, and J. Y. Lee, "Sensor-based motion planning for path tracking and obstacle avoidance of robotic vehicles with nonholonomic constraints," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 2013. 15, 19, 44, 45, 46

[36] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong, "Baidu apollo em motion planner," *ArXiv*, vol. abs/1807.08048, 2018. 16, 17

[37] "Vesc – open source esc." `http://vedder.se/2015/01/vesc-open-source-esc/`. Accessed: 2019-07-01. 22

[38] K. Wada, "labelme: Image Polygonal Annotation with Python." `https://github.com/wkentaro/labelme`, 2016. 27

[39] S. Humbarwadi, "Street Scene Parsing Using Semantic Segmentation." `https://github.com/srihari-humbarwadi/street-scene-parsing-using-semantic-segmentation`, 2018. 28

[40] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *ArXiv*, vol. abs/1505.04597, 2015. 28

[41] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org. 30, 33

[42] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015. 30, 33

[43] A. Gupta, R. Puri, M. K. Verma, S. Gunjyal, and A. Kumar, "Performance comparison of object detection algorithms with different feature extractors," *2019 6th International Conference on Signal Processing and Integrated Networks (SPIN)*, pp. 472–477, 2019. 31

[44] S. H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. D. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," *ArXiv*, vol. abs/1902.09630, 2019. 31

[45] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000. 32

[46] D. Barnes, W. P. Maddern, and I. Posner, "Find your own way: Weakly-supervised segmentation of path proposals for urban autonomy," *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 203–210, 2016. 52

[47] M. Everingham, L. V. Gool, C. K. I. Williams, J. M. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, pp. 303–338, 2010. 53

[48] K. Wada, "Mean Average Precision." `https://github.com/Cartucho/mAP`, 2019. 54