TECHNISCHE UNIVERSITÄT DRESDEN

Facultät Verkehrswissenschaften „Friedrich List“, Institut für Wirtschaft und Verkehr
Professur für Ökonometrie und Statistik, insb. im Verkehrswesen

Abschlussarbeit

Multi-Task Real-World Autonomous Driving

Bingyu Wang

Supervising Professor: Prof. Dr. rer. pol. Ostap Okhrin
Second supervisor: Prof. Dr. rer. pol. Pascal Kerschke
Tutor: M. Sc. Dianzhao Li

Dresden, 20. August 2025

Zusammenfassung

Diese Arbeit untersucht auf Basis einer ROS-Systemarchitektur mehrere Aufgaben des autonomen Fahrens in realen Umgebungen und vergleicht zwei Steuerungsansätze: (i) eine klassische Pipeline mit PID-Regelung und (ii) End-to-End-Deep-Learning-Modelle, die Kamerabilder direkt auf Steuerkommandos abbilden. Beide Verfahren werden auf zwei Roboterfahrzeugplattformen – dem Bosch-Fahrzeug und dem Duckiebot – implementiert und evaluiert.

Auf der Deep-Learning-Seite wird ein Regressionsmodell eingesetzt, das für die Spurhaltung eine einzelne Winkelgeschwindigkeit vorhersagt; die Winkelgeschwindigkeit (ω) wird mit dem mittleren absoluten Fehler (MAE) als Verlustfunktion trainiert. Ein Klassenungleichgewicht wird durch Resampling adressiert und mit einer asymmetrischen Verlustmethode verglichen, die Kurvensegmente stärker penalisiert.

Auf Systemseite werden eine Perspektivtransformation und Vorverarbeitung der Bilder, die Spuranpassung, die Erkennung von Verkehrsschildern und Fußgängern sowie eine PID-Regelung implementiert und zu einer stabilen autonomen Fahr-Baseline integriert.

Experimente auf realen Parcours unter normaler Innenraumbeleuchtung sowie mit zusätzlicher Beleuchtung zeigen, dass End-to-End-LSTM-Modelle die zeitliche Kontinuität besser ausnutzen und dadurch eine stabilere Spurhaltung sowie ein gleichmäßigeres Lenken bei schwachen Lichtverhältnissen erreichen. Die Merkmalsextraktionsfähigkeit des Deep Learning kann die hohe Sensitivität gegenüber Beleuchtungseinflüssen bis zu einem gewissen Grad mindern und ermöglicht dadurch stabileres Fahren in realen Umgebungen mit stärker variierender Beleuchtung. Zudem trägt Resampling wesentlich zur Balancierung des Datensatzes bei und stärkt das Training auf komplexeren Segmenten.

Abstract

This thesis explores multiple tasks for real-world autonomous driving under a ROS system architecture and compares two control methods: (i) a classical pipeline and PID control; (ii) end-to-end deep models that map camera images directly to control command outputs. Both methods are deployed and evaluated on two robotic vehicle platforms, the Bosch car and the Duckiebot.

On the deep learning side, this paper adopts a regression model that produces a single angular velocity prediction for lane keeping, training the angular velocity (ω) with mean absolute error (MAE) as the loss function. Class imbalance is handled by resampling and compared against an asymmetric-loss method that increases penalties on turning segments.

On the systems side, the work implements image perspective transformation and preprocessing, lane fitting, traffic sign and pedestrian detection and PID control, establishing a stable autonomous driving baseline.

Experiments on real tracks under normal indoor lighting and with supplemental illumination indicate that end-to-end LSTM models better exploit temporal continuity, achieving more stable lane keeping and smoother steering under low-light conditions. The feature capture capability of deep learning can, to some extent, mitigate high sensitivity to lighting, enabling stable driving in real environments with more variable illumination. In addition, resampling makes a substantial contribution to balancing the dataset and strengthening training on more complex segments.

Contents

List of Figures

List of Tables

List of Abbreviations

DL	Deep Learning
ROS	Robot Operating Systems
ML	Machine Learning
RL	Reinforcement Learning
AD	Autonomous Driving
CNNs	Convolutional Neuron Networks
LSTMs	Long Short-term Memory Networks
API	Application Programming Interface
DRL	Deep Reinforcement Learning
IMU	Inertial Measurement Unit
I/O	Input/Output
ROI	Regions of Interest
HSV	Hue–Saturation-Value
FSM	finite-state Machine
MAE	Mean Absolute Error
MSE	Mean Squared Error
BCE	Binary Cross Entropy
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative

List of Symbols

$B(x, y)$	Indicates the blue channel intensity value of the image at pixel (x, y)
$G(x, y)$	Indicates the green channel intensity value of the image at pixel (x, y)
$M(x, y)$	The binary mask,when pixels satisfy the judgement conditions corresponding to the subscript
$R(x, y)$	Indicates the red channel intensity value of the image at pixel (x, y)
T_g	global threshold
Δx_{off}	The lateral offset pixel value
α	Curvature weighting coefficient
$\dot{D}eltax(t)$	Lateral offset of the lane center relative to the vehicle center
ω	Angular velocity
$\omega(t)$	Angular velocity at time t
τ	Angular velocity threshold at curves
$e(t)$	Lateral error at time t
m	The resampling multiplier
p	Empirical positive rate
v	Linear velocity
x_{lane}	The column index of the lane center line
x_{veh}	Column index of the vehicle in the image coordinates

1 Introduction

1.1 Autonomous Driving Overview

In recent years, autonomous driving technology has advanced rapidly. Vehicles are equipped with various of sensors that support decision-making in complex environments, and the underlying algorithms have matured significantly under the influence of artificial intelligence. According to statistics, 94 % of road traffic accidents are attributable to human error [1]. By providing decision-making assistance, autonomous driving systems can effectively mitigate traffic congestion, decrease casualties, reduce energy consumption, and alleviate driver stress [2].

To date, achieving fully autonomous driving remains a challenge. In complex, rapidly changing urban environments, vehicles must perform multiple tasks: perceiving scene dynamics, detecting traffic signals, classifying traffic signs, and making timely decisions in real time. Because these capabilities are critical to driving safety, the underlying algorithms must exhibit exceptional robustness.

Deep Learning (DL) has enabled significant breakthroughs in computer vision [3], and the advent of novel sensor technologies has further propelled the advancement of autonomous driving systems. Among these innovations, end-to-end driving has emerged as a prominent research trend, typically leveraging deep neural networks [4]. In this paper, the fundamental task of lane following is realized through an end-to-end approach.

1.2 Problem Description

This paper details the process of training autonomous vehicles to perform multiple tasks in real-world environments using two different small vehicles. One vehicle uses Robot Operating Systems (ROS) to achieve autonomous control, while the other loads a pretrained model to deliver an end-to-end solution. The study concentrates on four key areas:

- A ROS-based architecture enabling reliable hardware–software communication

- A traditional computer vision and PID-based lane keeping control module
- The design and training of deep learning models for decision-making
- End-to-end lane following and a hybrid multi-task driving framework that combines end-to-end proposal with conventional control methods.

1.3 Research Structure

This paper is divided into seven chapters, with the structure of each chapter organized as follows:

Chapter One, Introduction, discusses the current state of research on real world autonomous driving and deep learning, the motivation behind this paper, and the description of the research problem addressed.

Chapter Two, Research Background, primarily provides the theoretical research foundation used in this paper and reviews previous studies related to this paper.

Chapter Three, Hardware Platform and Integration, presents an overview of the two robotic vehicle hardware platforms used in this work, the ROS architecture, the mapping configuration and parameters for the real-world testbed, and the hardware–software interfaces.

Chapter Four, Software Implementation, details the software-layer implementations of both the baseline pipeline and the end-to-end model, including the underlying methodology and code-level realization.

Chapter Five, Experiments, delineates the architectural and algorithmic differences among the models, compares their post-training metrics, and analyzes the factors responsible for the observed performance gaps.

Chapter Six, Performance Evaluation. This chapter uses data collected in real-world environments to directly compare traditional pipeline methods with end-to-end methods, highlighting the advantages and limitations of each.

Chapter Seven, Conclusion and Future Work, presents the conclusions of this paper and the research directions that can still be explored.

2 Research Background

2.1 Theoretical Basis

2.1.1 Machine Learning

Machine Learning (ML) studies methods enabling computers to autonomously improve performance based on experience. As one of the fastest-growing fields in today's technology sector, it bridges the gap between computer science and statistics, providing a solid foundation for artificial intelligence and data-driven science [5].

Machine learning are generally organized into three types—supervised learning, unsupervised learning, and reinforcement learning. (Figure: 2.1) Supervised learning employs labeled training examples, whereby the model iteratively adjusts its parameters to minimize the discrepancy between its predictions and the ground truth labels. It is typically used for classification or regression tasks. For instance, in this paper, supervised learning is applied to regress the vehicle's angular velocity. In the field of autonomous driving, using annotating traffic sign images to train traffic sign recognition classifiers is also a standard supervised learning application.

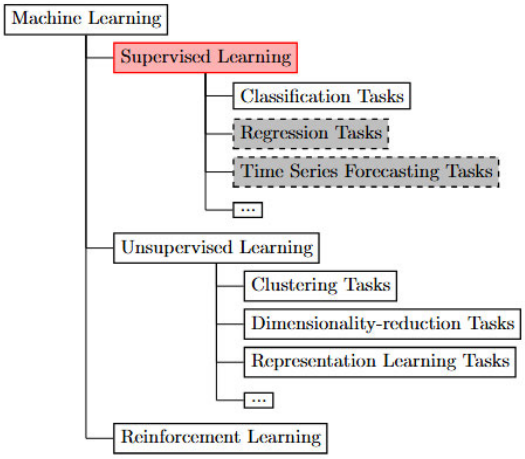


Figure 2.1: Components of machine learning

Unsupervised learning, by contrast, relies exclusively on unlabeled data, enabling the model to uncover intrinsic structures or distributional patterns within the dataset. Common applications include clustering and dimensionality reduction tasks, neither of which is addressed in the present work.

Reinforcement Learning (RL) frames the problem as an agent interacting with an environment and learning an optimal policy through reward signals provided by that environment. In Autonomous Driving (AD) research, agents typically learn driving strategies via repeated simulated interactions, using a designed reward function to reinforce desirable behaviors. Subtasks such as lane following [6], speed regulation [7], and car following [8] can all be realized through reinforcement learning based training.

2.1.2 Deep Learning

Artificial neuron networks originated at the intersection of machine learning and neuro-science [9]. These deep architectures use gradient-based optimization to adjust parameters across their multiple layers, guided by errors computed at the output [5] (shown in Figure 2.2). A gradient vector is calculated to indicate how changes in each weight affect the overall error, and the weights are then updated in the direction opposite to this gradient [10]. This training procedure is known as backpropagation. In order to efficiently minimise the loss function on large datasets, algorithms typically use a randomly selected small sample set to calculate the gradient. This is the approach called Stochastic gradient descent. ANNs are versatile tools used in both supervised and unsupervised learning. In supervised settings, they refine their parameters by reducing prediction errors on labeled data, while in unsupervised scenarios they uncover inherent data structures and distributions, supporting tasks such as clustering and dimensionality reduction.

Convolutional Neuron Networks (CNNs) are founded on four fundamental principles that exploit the characteristics of natural signals: limited receptive fields, weight sharing, pooling operations, and deep, multilayered architectures [11]. The structure of the convolutional and pooling layers draws direct inspiration from studies in visual neuroscience [12]. Convolutional layers use learnable filters to scan local regions and detect meaningful combinations of features. Pooling layers are typically inserted after convolution layers and combine similar activation values to improve the level of feature abstraction. CNNs have proven exceptionally effective for image recognition and are widely used in transportation applications to detect traffic signs, pedestrians, and other roadway objects. These applications demonstrate the central role that CNNs play in enabling end-to-end training frameworks for Autonomous driving systems.

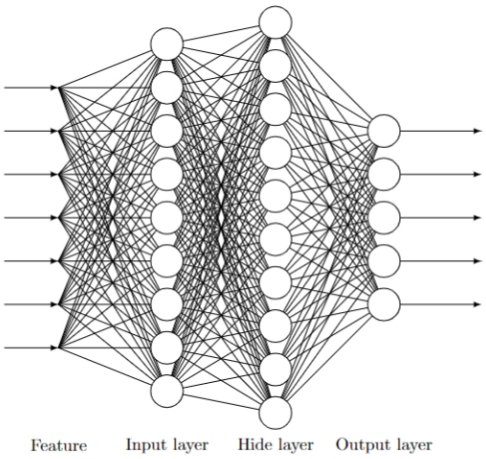


Figure 2.2: Artificial neural network architecture

Long Short-term Memory Networks (LSTMs), a form of recurrent neural network, have proven to be effective and scalable for a variety of sequential learning tasks [13]. They excel in capturing dependencies that span long time intervals [14]. Each LSTM layer consists of multiple interconnected units, which are called memory blocks(Figure: 2.3). Each memory block contains one or more cells that hold the state of the block, alongside three gates that regulate information flow. The input gate controls how much of the new input is written into the cells. The forget gate decides which parts of the existing cell state should be retained or discarded. Finally, the output gate determines how much of the cell's current state will be passed out as the output of the block. All interactions with cell states, whether writing new information, erasing old data, or reading information, are performed through these controlled structures.

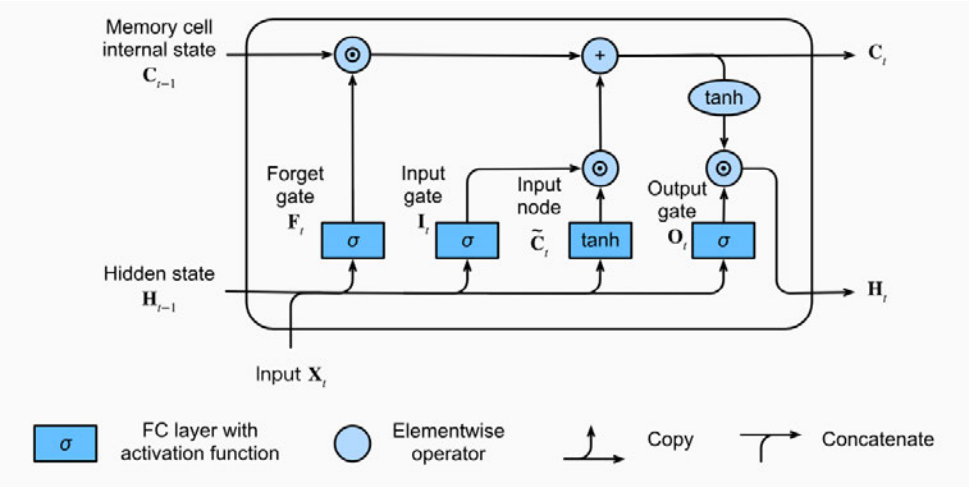


Figure 2.3: LSTM network architecture
Source:https://d2l.ai/chapter_recurrent-modern/lstm.html)

2.1.3 Robot Operation System

ROS is a software framework for developing robotic applications [15]. It provides services specifically designed for heterogeneous computing clusters and has become one of the most widely adopted robot software platforms in both academic research and industrial practice [16].

ROS represents each running process as a node in a graph, with connections between nodes called topics [17]. Nodes exchange data by publishing and subscribing to topics, call each other or provide services, and read or write data from a shared parameter server. The ROS Master enables this architecture by registering nodes, coordinating topic connections, and managing parameter server updates. Once a node registers with the master node, all messages and service traffic will be transmitted directly between nodes via peer-to-peer links, rather than being relayed through the master node(Figure: 2.4).

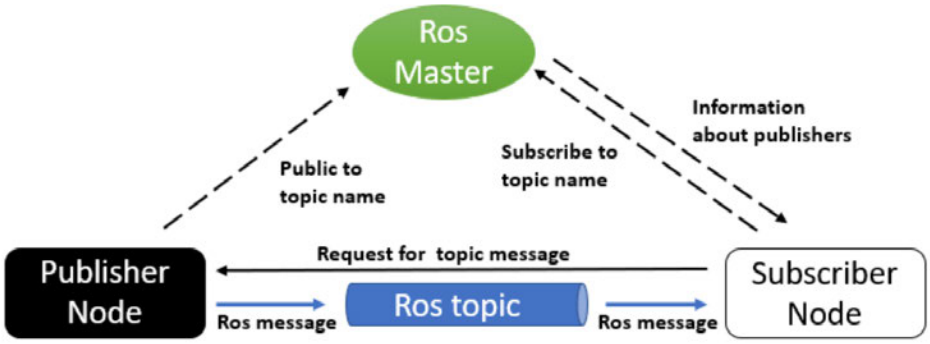


Figure 2.4: ROS communication framework (Source:https://trojrobert.github.io/hands-on-introdution-to-robot-operating-system%28ros%29/)

The ROS client libraries primarily run on Unix-like systems and offer C++ (roscpp) and Python (rospy) as their main programming interfaces. Via these Application Programming Interface (API), ROS supports the implementation of core modules such as hardware drivers, path planning, and perception algorithms. In this work, ROS manages vehicle hardware-software communication: Once the advanced algorithm calculates the target speed, the speed is encapsulated into a message and published to a topic, and the drive node subscribes to the topic to activate the motor and control the movement. In addition, open-source libraries such as OpenCV can be seamlessly integrated through ROS packages, allowing straightforward image processing and computer vision tasks within ROS nodes.

2.2 Related Work

Autonomous driving technology began to take shape in the 1990s, with the Eureka Project PROMETHEUS [18] representing one of the earliest large-scale research efforts in this field. That initiative prompted Daimler-Benz to develop the VITA II vehicle [19], which successfully demonstrated autonomous operation on highways.

In 2004, the United States launched the DARPA Grand Challenge to accelerate the development of self-driving cars [20]. By 2007, this competition had evolved into the Urban Challenge, where research teams from all over the world tested their autonomous driving systems in mock urban settings [21]. Because real city environments are complex and full-size autonomous vehicles are expensive, researchers often use small-scale model cities in the lab for testing. These test beds integrate traffic lights, road signs, and other urban infrastructure to simulate real-world environments. At the same time, they are equipped with scaled-down vehicles with hardware and software similar to full-size cars, enabling safer and more controllable algorithm verification. As a result, many platforms now use small-car testbeds in small-scale model cities to evaluate autonomous driving [22]. Platforms such as Duckietown [23] and DeepRacer [24] have seen widespread academic adoption for this purpose. The Bosch Future Mobility Challenge [25], initiated in 2017 by the Bosch Engineering Center in Cluj-Napoca, Romania, has provided an international forum for innovation in autonomous driving. In this competition, teams develop self-driving solutions on 1:10-scale vehicles. This paper employed both the Bosch Challenge vehicle and the Duckiebot to implement and evaluate autonomous driving approaches. Details of the hardware and software deployment are described in the following sections.

In autonomous driving, Yurtsever et al. [26] divide system architectures into two main categories: modular systems and end-to-end systems [4]. Modular designs break the driving task into separate subtasks, while end-to-end approaches employ deep learning to map sensor inputs directly to driving commands. Three primary end-to-end methods are recognized: direct supervised deep learning [27][28][29], neuro evolution [30], and Deep Reinforcement Learning (DRL) [31][32]. Direct supervised deep learning trains neural networks on large driving datasets; neuro evolution uses evolutionary algorithms to optimize network structure and weights; and DRL learns reward-optimal driving policies through interaction [33][34]. In my previous work, I evaluated DRL in simulation; this paper, however, adopts direct supervised deep learning for end-to-end training.

3 Hardware Platforms and Integration

3.1 Platforms Introduction

Below are the main specifications of the vehicles supplied by the Bosch Challenge and Duckietown platforms, covering three categories: propulsion system, sensors, and computing platform.

3.1.1 Bosch Challenge Car

The Bosch Challenge vehicle is propelled by a brushless DC motor paired with an electronic speed controller and steered via a dedicated servo. It carries an Inertial Measurement Unit (IMU) for attitude and acceleration feedback. Sensor Input/Output (I/O) and Pulse width modulation motor commands are handled by an ST Nucleo F401RE control board.

Originally, the primary onboard computer was a Raspberry Pi with its standard camera module (Figure: 3.1). To achieve higher performance, this was replaced by an NVIDIA Jetson Nano 4 GB (Figure: 3.2a) running Ubuntu 20.04 and ROS Noetic, paired with a front-mounted IMX219-160 camera for image acquisition (Figure: 3.2b). Table 3.1 lists the hardware specifications of the Bosch Challenge car.

3.1.2 Duckiebot

The Duckietown robot (Figure: 3.3) employs two DC motors managed by a PCA9685 PWM HAT. A front-mounted IMX219-160 camera with a 160° field of view provides visual input. An onboard 6-axis IMU, hall-effect wheel encoders, and a time-of-flight sensor enable obstacle detection. Its sole compute platform is an NVIDIA Jetson Nano 4 GB developer kit. Table 3.2

Table 3.1: Hardware specifications of the Bosch Challenge car

Hardware Module	Bosch Car
Drive motor	brushless DC motor paired + electronic speed controller
Camera	front-mounted IMX219-160 camera(8 MP, 160° FOV)
IMU	onboard 6-axis IMU
Other sensors	-
Main control board	NVIDIA Jetson Nano 4 GB developer kit
Auxiliary control board	ST Nucleo F401RE control board
operating system	Ubuntu 20.04 and ROS Noetic

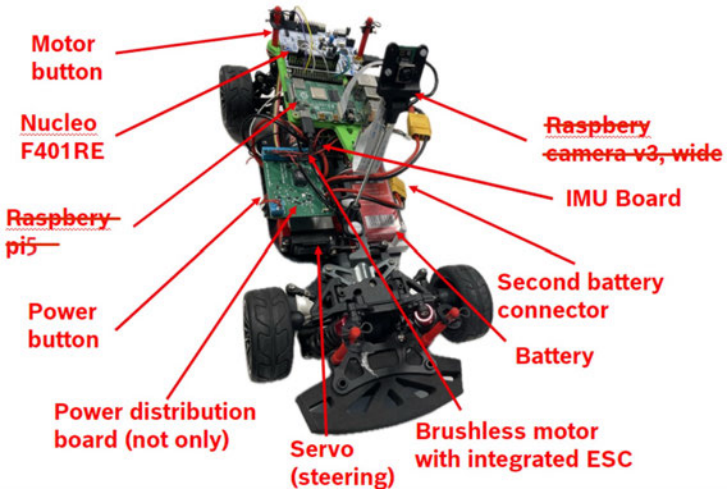


Figure 3.1: Original design for Bosch Challenge car

lists the hardware specifications of the Duckiebot.

3.2 ROS System Architecture

3.2.1 Overview of Package Structure and File functions

This section outlines the ROS package layouts and file functions for two platforms—the Bosch car and the Duckiebot. First comes the Bosch car:
Customised package: mypackage/
Build

- CMakeLists.txt
  Defines the catkin build pipeline, dependencies, and install rules.
- package.xml
  Declares package metadata and build dependencies.

(a) Bosch car's replacement onboard computer

(b) Bosch car's replacement camera

Figure 3.2: Bosch car replacement kit

Figure 3.3: Original design for Duckiebot

Launch

- launch/img_process.launch
  Integrates the camera subscription and the image-processing pipeline, subscribes to traffic-sign and pedestrian detection results, and publishes linear and angular velocity commands to the chassis motors.
- launch/motor_test.launch
  Reliability check of the motor. Publishes simple test commands and inspects motor feedback to verify correct operation.

Core control

- src/base_sub_c.cpp
  Includes pattern_new.h, computes the angular velocity, and publishes the simple velocity command to the motor.
- src/pattern_new.cpp

Table 3.2: Hardware specifications of the Duckiebot

Hardware Module	Duckiebot
Drive motor	two DC motors
Camera	front-mounted IMX219-160 camera(8 MP, 160° FOV)
IMU	onboard 6-axis IMU
Other sensors	Hall-effect wheel encoders; ToF sensor
Main control board	NVIDIA Jetson Nano 4 GB developer kit
Auxiliary control board	PCA9685 PWM HAT
operating system	Ubuntu 20.04 and ROS Noetic

Implements lane detection and the distance between the vehicle and the lane center. Also contains the PID algorithm for computing angular velocity.

- include/pattern_new.h
  The corresponding header that declares the core data structures.

- src/scripts/controller.py
  Publishes velocity commands to the motors. If a traffic sign is detected, applies the corresponding speed control. Differ between the base_sub_c.cpp, this block is more comprehensive, where including obstacle and traffic sign detection.

- src/scripts/messageconverter.py
  Converts high-level velocity commands into the motor-controller board's format. The board expects a fixed protocol. This module performs the required message translation.

- src/scripts/pattern_new.py, src/scripts/pattern_pedestrain.py, src/scripts/pattern_trafficsign.py
  Python counterparts with the same logic as in the Duckiebot setup, which will mention in detail later.

functional testing

- src/scripts/base_sub.py
  Subscribe to pre-processed images, perform lane detection, and estimate the lateral offset relative to the lane center. For tuning and evaluation purposes only. This is a static accuracy check, i.e. the vehicle engine will not be started.

- src/scripts/base_pub.py
  Read raw camera images, perform camera calibration and pre-processing, and publish processed image topics.

- src/scripts/img_process.py
  An independent image processing script used to validate the processing steps offline. Works as a reliability check for pre-processing.

- src/scripts/img_sub.py
  Subscribes to the processed image topic and optionally renders output images or saves them to local workspace for offline debugging and dataset creation.

Official dependency package: OpenCV/
Auxiliary OpenCV resources required by the C++ lane detection path, which is compiled for lower latency. Linked via CMake and invoked from the C++ nodes.

```
mypackage/
|-- CMakeLists.txt
|-- package.xml
|-- launch/
    |-- img_process.launch
    |-- motor_test.launch
|-- src/
    |-- base_sub_c.cpp
    |-- pattern_new.cpp
    |-- scripts/
        |-- base_sub.py
        |-- base_pub.py
        |-- controller.py
        |-- messageconverter.py
        |-- img_process.py
        |-- img_sub.py
        |-- pattern_new.py
        |-- pattern_pedestrain.py
        |-- pattern_trafficsign.py
        |-- pattern_lane.py
|-- include/
    |-- pattern_new.h

OpenCV/
```

Here comes the duckiebot:
Customised package: mypackage/
Build

- CMakeLists.txt
  Defines the catkin build pipeline, dependencies, and install rules.
- package.xml
  Declares package metadata and build dependencies.

Launch

- launch/test.launch
  One launcher that brings up camera subscription and data logging, selects the control node, starts obstacle and traffic sign detection, runs lane detection, and publishes motor commands to the chassis.

Core control

- scripts/auto_control_node.py
  End-to-end integrated test node. The loader loads a selected model and produces angular velocity predictions.
- scripts/auto_control_node_lstm.py
  LSTM control node. Loads lstm.py with models/lstm.pth and predicts lane keeping angular velocity from temporal input.
- scripts/auto_control_node_omega.py
  CNN control node. Loads cnn.py with models/cnn.pth and predicts angular velocity for lane keeping.

Image I/O

- scripts/camera_sub.py
  Subscribes to raw images, performs images pre-processing and republishes processed images for downstream models.
- scripts/data_collect.py
  Saves images and labels to support subsequent end-to-end training.

Model definitions

- scripts/cnn.py, scripts/lstm.py, scripts/duckienet.py
  Network architectures used in the end-to-end pipeline (matching .pth files in models/*.pth).

Pattern-based classical CV method

- scripts/pattern_new.py
  Lane detection, lane center offset estimation, PID-based omega computation. Used as a callable module by higher-level nodes.
- scripts/pattern_pedestrain.py
  Pedestrian detection using classical computer vision.

- scripts/pattern_trafficsign.py
  Traffic sign detection using classical computer vision.

- scripts/twist_control_node.py
  Subscribes to processed image, invokes angular velocity prediction and publishes. Can also choose to trigger speed control or parallel parking sequence based on traffic sign recognition.

Models

- models/*.pth
  Directory containing all trained deep learning models.

Official dependency package: dt-ros-commons/
Duckietown's official common package provides camera acquisition, chassis interfaces, and other utilities.

```
mypackage/
|-- CMakeLists.txt
|-- package.xml
|-- launch/
    |-- test.launch
|-- src/
|-- scripts/
    |-- auto_control_node.py
    |-- auto_control_node_lstm.py
    |-- auto_control_node_omega.py
    |-- camera_sub.py
    |-- data_collect.py
    |-- cnn.py
    |-- lstm.py
    |-- duckienet.py
    |-- pattern_new.py
    |-- pattern_pedestrain.py
    |-- pattern_trafficsign.py
    |-- twist_control_node.py
|-- models/
    |-- cnn.pth
    |-- lstm.pth
    |-- duckienet.pth
dt-ros-commons/
```

3.2.2 Node Relationship of the Core Module

The following is the complete set of nodes and topic interfaces required to implement key AD functions on the Bosch platform (shown in Fig. 3.4).

- Node: preprocessed_image_publisher: preprocessed image publisher
  Function: Grab frames from the camera and publishes
  Publishes:/camera/raw_image_perspective: original perspective image.
  Publishes:/camera/raw_image: BGR bird's-eye image.
  Publishes:/camera/preprocessed_image: binarized, cropped, undistorted image.

- Node: traffic_sign_detector: traffic sign detection node.
  Function: Runs sign detection and outputs the detected class.
  Publishes:/pattern/trafficsign
  Subscribes:/camera/raw_image_perspective

- Node: pattern_node: pedestrian detection node
  Function: Performs pedestrian detection and outputs if it exists.
  Publishes:/pattern/pedestrian
  Subscribes:/camera/raw_image_perspective

- Node: img_subscriber: velocity computation node
  Function: Subscribes to the pre-processed image, performs lane fitting and PID computation, and combines it with a constant linear speed to produce a baseline control command.
  Publishes:/bfmc_cmd.
  Subscribes:/camera/preprocessed_image

- Node: controller_node: policy integration node
  Function: Integrate the lane keeping steering velocity and the speed control strategy through the traffic sign and road condition.
  Subscribes:/bfmc_cmd.
  Subscribes:/pattern/trafficsign
  Subscribes:/pattern/pedestrain

Below is the set of nodes and topic interfaces required to implement key AD functions using duckiebot (shows in Fig. 3.5).

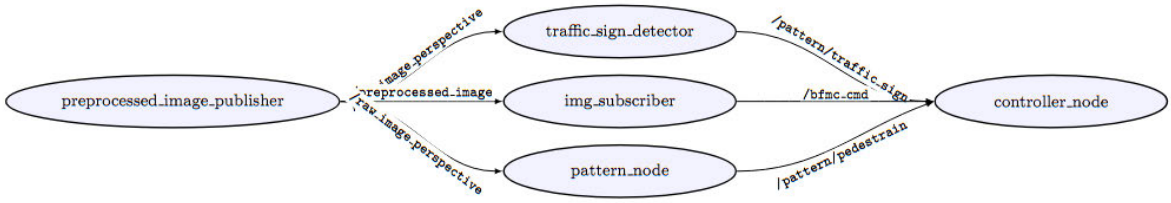


Figure 3.4: ROS node communication for Bosch vehicle

- **Node: PerspectiveProcessorNode**: preprocessed image publisher
  Function: outputs processed images for downstream nodes.
  Publishes:/camera_node/image/processed/compressed: binarized, cropped, undistorted image.
  Subscribes:/camera_node/image/compressed: original perspective compressed image.

- **Node: TrafficSignDetectorNode**: traffic sign detection node.
  Function:Detects and classifies traffic signs.
  Publishes:/pattern/trafficsign
  Subscribes:/camera_node/image/compressed

- **Node: PatternDetectorNode**: pedestrian detection node
  Function: Performs pedestrian detection and outputs if it exists.
  Publishes:/pattern/pedestrian
  Subscribes:/camera_node/image/compressed

- **Node: TwistControlNode**: policy integration node
  Function: Call the lane recognition and PID control modules to detect and track the right lane, adjust the lateral position in real time to maintain a constant speed in the right lane. At the same time, adjust the speed based on the detection results of pedestrians and traffic signs.
  Subscribes:/camera_node/image/processed/compressed.
  Subscribes:/pattern/trafficsign

Subscribes:/pattern/pedestrain

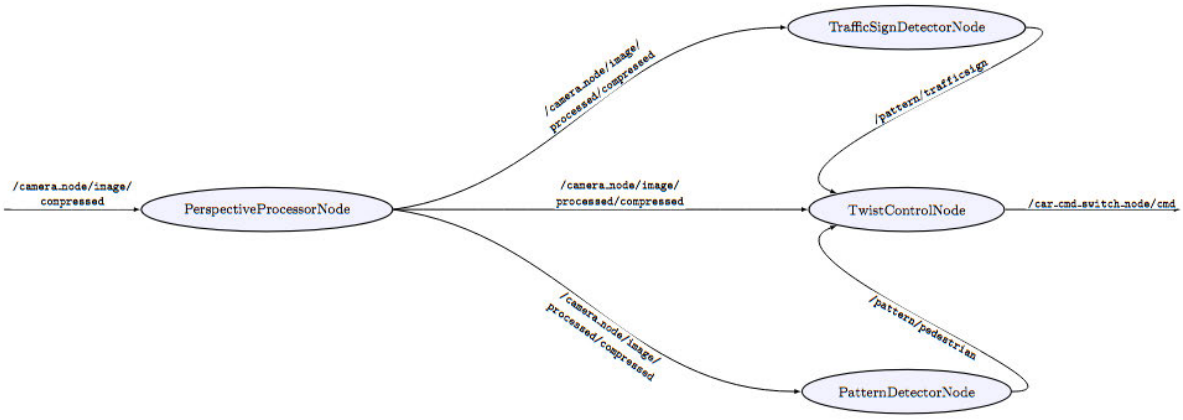


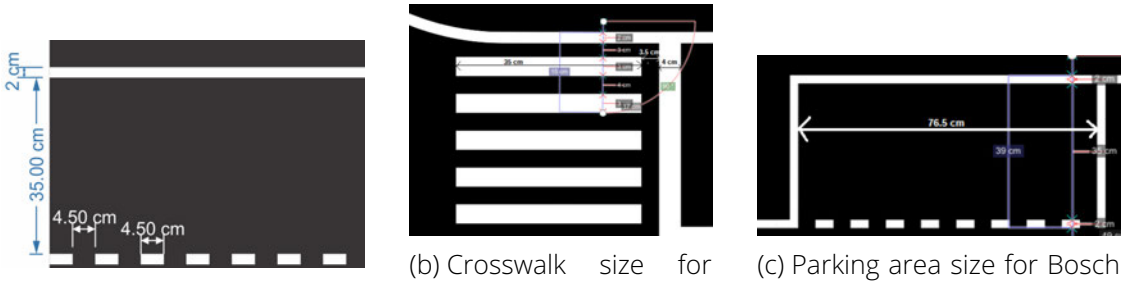
Figure 3.5: ROS node communication for Duckiebot

3.3 Real World Map Design

Given that the Bosch vehicle is slightly larger than the Duckiebot, the map sizes were adjusted to accommodate each platform. The track forms a closed loop, consisting of a half-lap single-lane section and a half-lap double-lane section. Two special zones are included: a pedestrian crosswalk and a parallel parking area. For the Bosch Challenge, the crosswalk was placed on the single-lane segment. In the Duckiebot configuration, the low fixed mounting of the camera prevented reliable detection of crosswalk markings outside the lane, so the crosswalk was omitted.

On the Bosch track, lane width (Figure: 3.6a) is 35 cm; crosswalk stripes (Figure: 3.6b) measure 35 cm in length and 3 cm in width; and parking spaces (Figure: 3.6c) are 76.5 cm long by 39 cm wide.

The default map for the Duckietown platform (Figure: 3.7a) uses white edge lines with a yellow centerline. In this study, all markings (Figure: 3.7b) were changed to white to more closely reflect real-world conditions, increasing the difficulty of the training. Due to the Duckiebot's



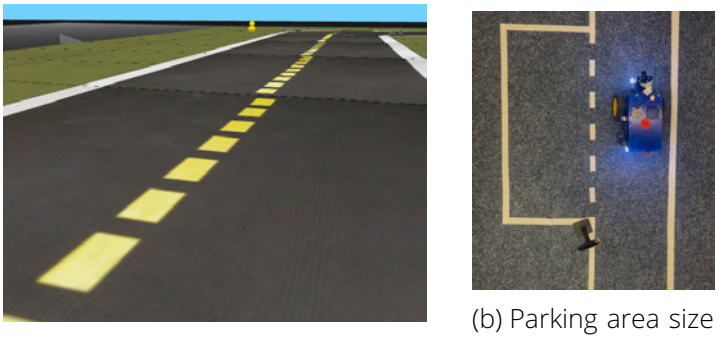
(a) Lane size for Bosch car

(b) Crosswalk size for Bosch car

(c) Parking area size for Bosch car

Figure 3.6: Map size for Bosch car

smaller size, each scenario was scaled down: lanes measure 28 cm in width, and parking spaces are 62 cm long by 32 cm wide. Figure 3.7 contrasts the standard Duckietown layout with the modified map used here.



(a) Original Duckietown map design

(b) Parking area size for Duckiebot

Figure 3.7: Map changes for Duckiebot

3.4 Hardware–Software Interface and Integration

Whether on the Bosch car platform or on the Duckietown platform, both adopt the same system architecture. Package names may differ slightly, but interfaces and functionality are identical. The entire system is organized top-down into five layers, as shown in the figure 3.8:

1. Sensing & Input

   This layer includes the drivers and I/O for the camera and IMU. Its core components are CameraHandler and IMU-listener.
   CameraHandler handles image acquisition and the generation of Regions of Interest (ROI).
   IMU-listener reads inertial measurement data.

2. Perception & Scene Understanding
   This layer comprises four detection modules and Pose fusion. The CameraHandler outputs ROIs for objects, lanes, signs, traffic lights, which are routed to the corresponding detectors.
   Object classifier detects obstacles such as pedestrians.
   Lane detection estimates lane markings.
   Sign detection recognizes traffic signs.
   Traffic light detection identifies traffic-signal states.
   Pose fusion ingests pose-related measurements from the IMU-listener.

3. Behavior and Motion Plan

   This layer centers on Decision making and the Local Trajectory Generator.
   Decision making aggregates triggers and states from Layer 2 and makes decisions regarding obstacles, signs, traffic lights and lane status.

4. Vehicle Control

   The vehicle control system comprises a series of task-specific controllers: lane following, parking, crosswalk and pedestrian detection. The decision making module selects or activates the corresponding controller via a trigger.

5. Actuation

   Actuation aggregates the controller outputs into commands (angular velocity and linear velocity), which are sent to the chassis via SerialHandler for execution.
   Talker is used for external publishing, debugging, and visualization.
   Logging subscribes to key topics for runtime monitoring, replay, and diagnostics.

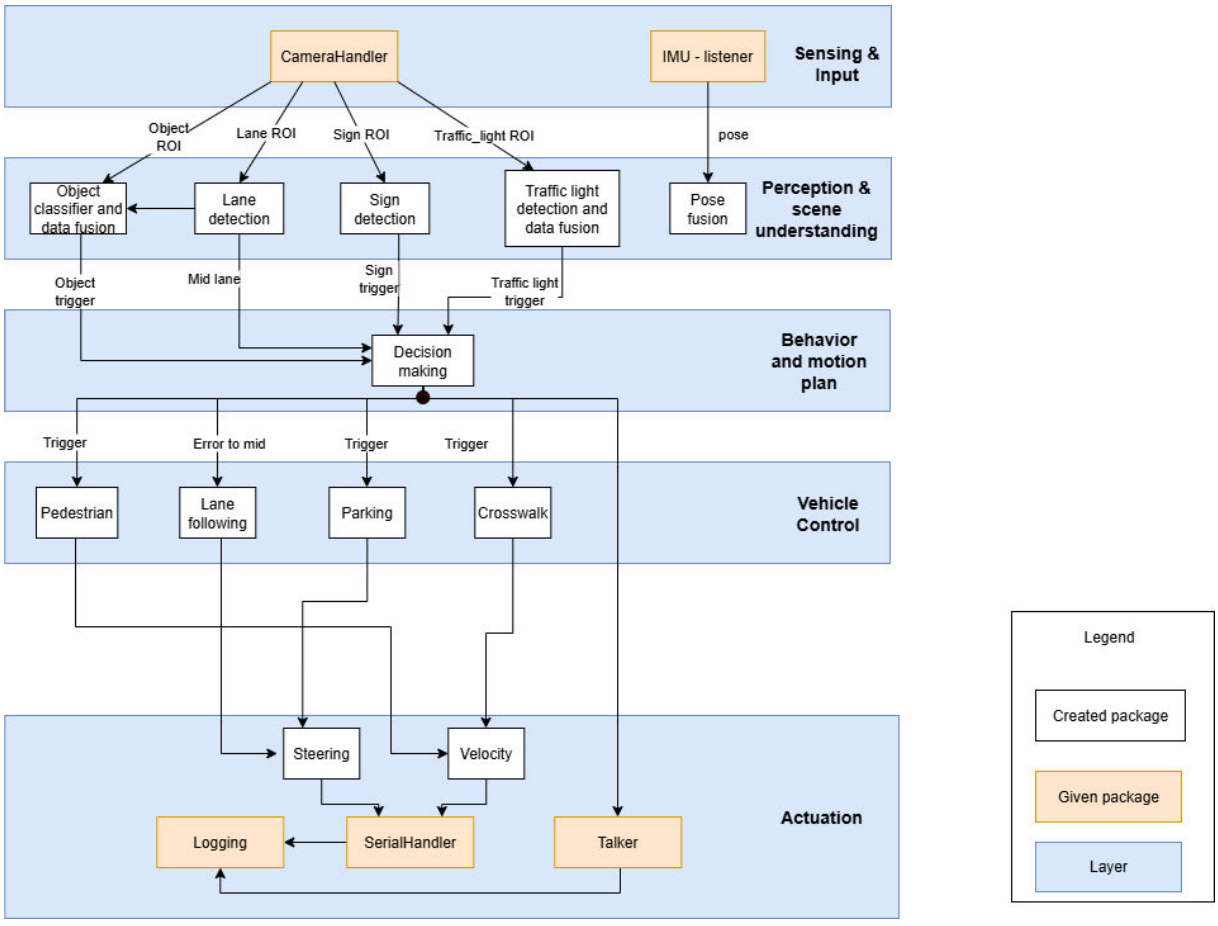


Figure 3.8: System architecture and hardware-software interfaces

4 Software Implementation

This chapter begins by presenting the baseline lane following system based on traditional computer vision and PID control, detailing each step from lane detection and error quantification to PID parameter design and tuning. Subsequently, the end-to-end learning approach is explored, covering data collection strategies, image pre-processing and augmentation, neural network architecture, and loss function selection. By progressing from the baseline to the end-to-end implementation, this chapter establishes a solid foundation for the analysis of experimental results.

4.1 Baseline Method

4.1.1 Image Pre-processing and Feature Extraction

The raw input from the front-facing camera is a full-color RGB image in which the lane markings are known to be white. To isolate these markings under varying illumination, the following pipeline is applied:

1. Convert the BGR color image to a gray-scaled image.
2. Apply a 5×5 Gaussian filter to the grayscale image to suppress high-frequency noise, thereby mitigating spurious edges in the subsequent Canny detector.
3. Perform Canny edge detection on the smoothed image. The Canny edge map identifies locations of strong gradients (e.g., lane boundaries and other high-contrast edges).
4. Extract the green channel.
    - In the BGR color model, each pixel has three intensity components: $[B(x,y), G(x,y), R(x,y)]$. The green channel is obtained by extracting the G component from the three-channel image to form a single-channel grayscale image:

$$g(x,y) = G(x,y) \in [0, 255] \tag{4.1.1}$$

- Yellow and white lane markings exhibit strong reflectance in the visible spectrum and typically provide higher luminance contrast in the green band.
- Given a global threshold T_g, applying thresholding to $M_g(x,y)$ to obtain a single-channel binary mask $M_g(x,y)$. Structures that are bright in the green channel (e.g., white lane markings) are mapped to white (255), and all others to black (0).

$$M_g(x,y) = \begin{cases} 1, & \text{if } G(x,y) \geq T_g \\ 0, & \text{otherwise.} \end{cases} \tag{4.1.2}$$

5. Fuse the binarised edge mask with the green channel threshold mask. Combining these two masks produces a mask that contains both bright areas and clear edge trajectories.

6. Convert the fused binary map back to a three-channel representation to interface with downstream modules that expect three-channel inputs.

4.1.2 Lane Model Fitting

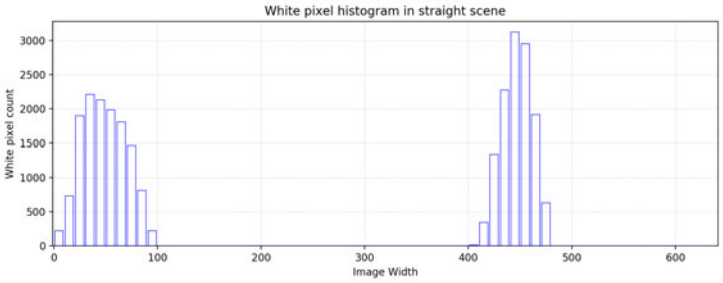
The binarized lane image is next subjected to a perspective transform, mapping the camera's pitched view into a bird's-eye view. Figure 4.1 illustrates the conversion from the original image to the binarized bird's-eye view. This transformation uses a pre-calibrated homography matrix to project the road-plane region onto a fixed-size overhead image, establishing a correspondence between pixel coordinates and real-world distances.

(a) Original RGB image (b) Binary bird-eye image

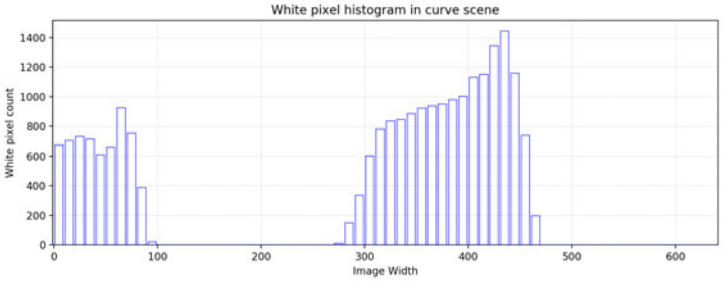
Figure 4.1: Image pre-process

In the resulting top-down binary image, a horizontal histogram is created by counting white pixels in each column. Figure 4.2 shows the white-pixel histograms on a straight road and a curved road, respectively. Local peaks in this histogram indicate the pixel positions of the road lines. To distinguish left from right, the image is split vertically at its center: the peak in the

right half corresponds to the right line.



(a) Histogram of the white pixel count in straight lane



(b) Histogram of the white pixel count in curve lane

Figure 4.2: Histogram of the white pixel count

In the bird's-eye binary image, a bottom-up sliding-window search is performed. First, the column corresponding to the peak of the right-half histogram is taken as the initial window center. At this column, a rectangular window with fixed width and height is opened to count white pixels. If white pixels are present, the mean of their column indices is computed and adopted as the new window center. The window is then moved one layer upward and reopened at the updated center, iterating layer by layer as illustrated in Figure 4.3. In this way, the right-lane column obtained by coarse localization from the histogram is refined into a dense, contiguous set of pixels. On curved road segments, this sequence naturally shifts leftward or rightward. If several consecutive layers contain no white pixels, the segment is regarded as noise or a local break in the markings, and the search is terminated early.

After obtaining this contiguous point set, a quadratic polynomial is fitted by least squares to produce a smooth and continuous right-lane boundary. The column with the maximum histogram response is then used as the stable right-lane index for the current frame. The coordinate of the centerline of the lane (x_{lane}) is computed by subtracting half of the lane width from the position of the right lane. And the vehicle's lateral position (x_{veh}) is taken as the image midline column. The difference between these two quantities yields the lateral offset (Δx_{off}).

$$x_{\text{lane}} = x_R - \frac{W_{\text{lane}}}{2} \tag{4.1.3}$$

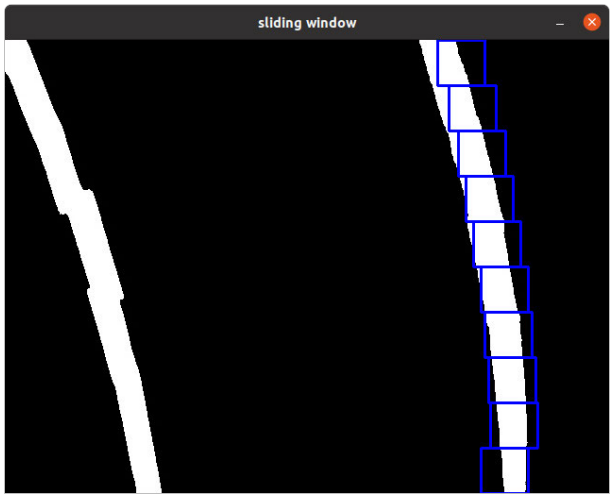


Figure 4.3: Sliding window search

$$x_{\text{veh}} = \left\lfloor \frac{W}{2} \right\rfloor \tag{4.1.4}$$

$$\Delta x_{\text{off}} = x_{\text{lane}} - x_{\text{veh}} \tag{4.1.5}$$

This pipeline operates in the top-down domain, which greatly reduces perspective distortion. By combining histogram-peak initialization, sliding-window tracking, and quadratic fitting, it achieves efficient and robust lane localization. The resulting lateral error serves as a high-quality input to the PID controller.

4.1.3 PID Control

Below the lateral error $e(t)$ is defined. At time t, the error equals the previously computed $Deltax(t)$. In this work I adopt a minimalist PID policy. A pure proportional controller is used to compute the steering command from the error:

$$\begin{aligned} e(t) &= \Delta x(t) \\ \omega(t) &= -K_p \cdot e(t) \end{aligned} \tag{4.1.6}$$

The negative sign in the control law ensures that, when the vehicle drifts left (i.e. $e(t)<0$), the steering command $\omega(t)$ is positive (i.e. a right turn), and vice versa.

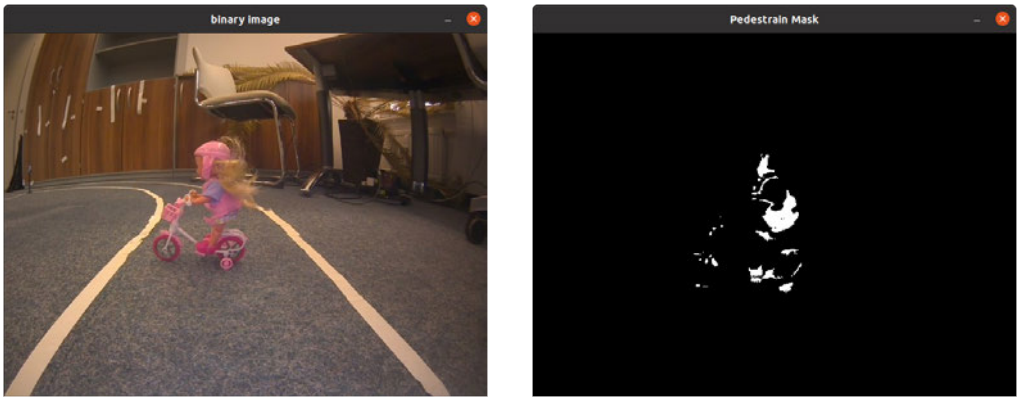
4.1.4 Extended Autonomous Driving Functions

After achieving basic lane keeping, the system must also respond to pedestrians, crosswalks, and parking signs to ensure safety and support multitask operation. The functions presented

in this section are implemented using OpenCV color-thresholding.

1. Color-Based Object Detection

   - Pedestrian Detection
     In this study, a pedestrian obstacle placed on the road (Figure 4.4a) was used, enabling rapid detection by isolating its distinctive pink color. Pink pixels are segmented via Hue–Saturation-Value (HSV) thresholding (4.1.7). When the number of separated pink pixels exceeds the preset threshold, it is determined that a pedestrian has been detected. Figure 4.4b shows the pink-pixel mask, which makes it possible to directly evaluate the selected HSV boundaries and check the detection results.



(a) Pedestrian (b) Pedestrian mask

Figure 4.4: Pedestrian detection

$$M_{\text{pink}}(x,y) = \begin{cases} 1, & \text{if } (H_{\text{cv}}, S_{\text{cv}}, V_{\text{cv}}) \in [\boldsymbol{\ell}_{\text{pink}}, \boldsymbol{u}_{\text{pink}}], \\ 0, & \text{otherwise.} \end{cases} \tag{4.1.7}$$

   - Crosswalk Detection
     Crosswalk sign (shows in Figure 4.5a) is only located in the right part of the map in this study. Blue pixels are extracted via HSV thresholding (4.1.8), and When the count of blue pixels in the right half of the image exceeds a predefined threshold, the vehicle is considered to have entered the crosswalk zone. Figure 4.5b shows the crosswalk sign mask.

$$M_{\text{c}}(x,y) = \begin{cases} 1, & \text{if } (H_{\text{cv}}, S_{\text{cv}}, V_{\text{cv}}) \in [\boldsymbol{\ell}_{\text{blue}}, \boldsymbol{u}_{\text{blue}}], \\ 0, & \text{otherwise.} \end{cases} \tag{4.1.8}$$

(a) Crosswalk Sign

(b) Crosswalk Sign Mask

Figure 4.5: Crosswalk sign detection

- Parking Sign Detection
  Parking signs are represented by a blue symbol located on the left side of the roadway (shown in Figure 4.6a). The left half of the image is segmented using the same HSV blue thresholds (4.1.9). When the count of blue pixels in the left half of the image exceeds a predefined threshold, the side-parking trigger is initiated. Figure 4.6b shows the parking sign mask.

(a) Parking Sign

(b) Parking Sign mask

Figure 4.6: Parking sign detection

$$M_{\text{pk}}(x,y) = \begin{cases} 1, & \text{if } (H_{\text{cv}}, S_{\text{cv}}, V_{\text{cv}}) \in [\boldsymbol{\ell}_{\text{blue}}, \boldsymbol{u}_{\text{blue}}], \\ 0, & \text{otherwise.} \end{cases} \tag{4.1.9}$$

2. Speed Control Strategy

Pedestrian Stop
Upon pedestrian detection, the vehicle's linear velocity is set to 0 m/s and held until the pedestrian leaves the detection area.
Crosswalk Deceleration
When a crosswalk sign is detected, the linear velocity is reduced to a preset deceleration speed (0.4 m/s) and then restored to the normal driving speed after the crosswalk has been passed.
Parking Sign Response
Upon entering the parking-sign zone, the vehicle similarly decelerates to 0.4 m/s. Once the vehicle reaches the trigger region, the side parking maneuver sequence is initiated.

3. Parallel Parking Sequence
   Side parking is executed by a simple finite-state Machine (FSM). When the color-based detector reports that the count of deep-blue pixels within the left ROI exceeds a threshold, the system declares entry into a parking zone. Upon trigger, the vehicle decelerates and, after a delay of approximately 6 seconds, transitions to the parking control sequence, issuing the preset (v,ω) commands in order to complete the maneuver. The parking sequence is summarized in Table 4.1. After the sequence ends, omega is again computed by the PID controller, and lane keeping restores.

Table 4.1: Parking sequence

Step	v	ω	Duration	Purpose
1	0.3	ω from PID	7.0	Straighten while approaching
2	0.0	0.0	2.0	Stop
3	-0.3	-4.3	1.8	Back left to enter the area
4	-0.3	4.0	1.5	Back right to correct heading
5	-0.3	0.0	0.1	Back straight to align
6	0.0	0.0	2.0	Stop
7	0.3	-3.5	1.7	Forward right to leave

4.2 End-to-end Method

4.2.1 Data Collection and Data Distribution

To build the end-to-end lane-following model, data were collected exclusively under PID-based lane keeping, without executing side-parking maneuvers or pedestrian stops. The collection procedure and key points are as follows:

1. PID-driven scenario coverage

   - The vehicle traversed three typical road scenarios (single-lane, double-lane, and side-parking zone) at a constant forward speed. A PID controller issued angular velocity commands based on the lateral error, ensuring stable tracking of the lane center.
   - Image acquisition was limited to the lane-keeping phase; frames captured during side-parking or pedestrian interactions were excluded.
   - Because the double-lane segment exhibited low ambient illumination, supplemental lighting is provided during data capture (see Fig. 4.7a). With supplemental lighting, a top-down binarized view can accurately extract lane markings (Fig. 4.7c). Figure 4.7b shows the situation without light, this segment exhibited incomplete line detection, which severely degraded the PID controller's performance. The added illumination thus ensured reliable detection after perspective transformation and binarization, stabilizing PID control.

(a) Double-lane segment extra light installed

(b) Lane detection without extra light

(c) Lane detection with extra light

Figure 4.7: Low ambient lighting double-lane segment

2. Sample distribution and data-balancing strategies

   - The camera recorded at 10 FPS, yielding approximately 120000 frames for subsequent supervised learning.
   - Frames were classified by the recorded angular velocity ω. Straight-road samples are defined by condition (4.2.1). Curved-road samples are defined by condition (4.2.2).

$$|\omega| < \tau \tag{4.2.1}$$

$$|\omega| > \tau \tag{4.2.2}$$

Table 4.2 showed straight-road samples accounted for about 60 %, curved-road samples about 40 %, indicating a clear class imbalance.

Table 4.2: Sample distribution of lane types

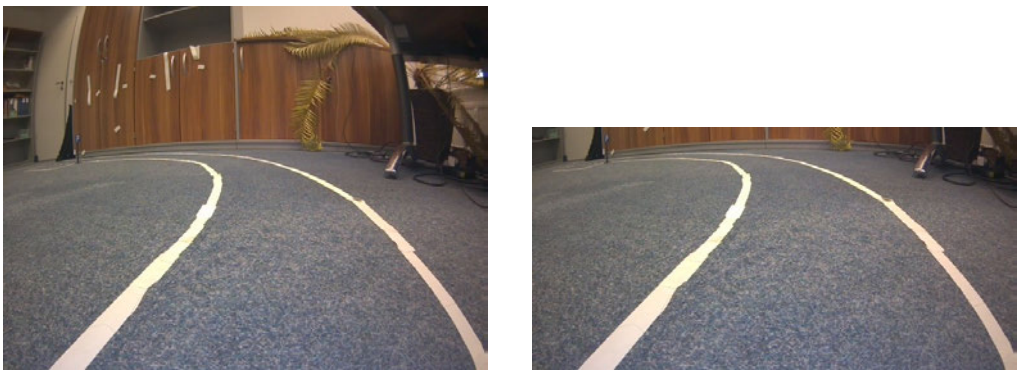
	Straight-road	Curved-road
Count	73947	44943
Proportion	62.2%	37.8%

- To address the scarcity of curved-road samples, two strategies were applied: resampling of curved-road frames and weighted loss function for assigning higher weight to underrepresented classes. Both methods yielded different performance profiles in subsequent experiments.

By applying these balancing steps, a dataset with approximately equal class representation and diverse driving scenarios was obtained, providing a robust foundation for the end-to-end supervised learning stage.

4.2.2 Image Pre-processing

In the original 640×480 frames, the upper one-third region typically contains no lane information and includes substantial irrelevant noise. Therefore, each frame is cropped to retain only the lower two-thirds of the original image, yielding an input size of 640×320 (shown in Fig. 4.8). Prior to training the neural network, the following data augmentation techniques are applied to enhance model robustness:



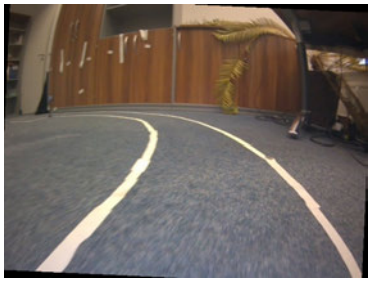
(a) Original size frame (b) Cropped frame

Figure 4.8: Noise cancellation

1. Small-angle rotations to simulate camera tilt caused by vehicle motion and road unevenness. Figure 4.9 illustrates the rotation augmentation, showing example frames with

slight left and right camera rotations.

(a) Original frame



(b) Frame after right rotation

(c) Frame after left rotation

Figure 4.9: Rotation

2. Gaussian blur is used to simulate image blur caused by motion. Figures present examples from the collected image set, including (i) sharp frames acquired with a stable camera (Fig. 4.10a) and (ii) a small small amount of data exhibiting sudden motion blur (Fig. 4.10b). In this data augmentation example (Fig. 4.10c), Gaussian blur is applied to the sharp frames to simulate motion-blurred situation.

(a) Original normal frame

(b) Frame under motion blur

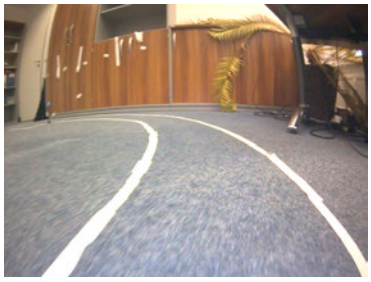
(c) Frame after gaussian blur

Figure 4.10: Gaussian blur

3. Random brightness (Fig. 4.11) adjustments to reflect varying ambient illumination conditions.

(a) Original frame

(b) Low brightness frame



(c) High brightness frame

Figure 4.11: Brightness

These augmentations enrich sample diversity and effectively expand the dataset, thereby sup-

porting more stable and robust network training.

4.2.3 Network Architecture and Parameters

Two end-to-end regression models are evaluated: a pure CNN regressor and a CNN+LSTM regressor. The former accepts a single cropped RGB frame as input, while the latter processes a sequence of cropped RGB frames. Both models output an angular velocity prediction used to generate driving control commands from real-time images.

The pure CNN regressor comprises three repeated "convolution → activation → pooling" blocks for spatial feature extraction. Global average pooling is then performed to compress each feature map into a single value for each channel, and the final angular velocity regression is performed through two fully connected layers. Detailed layer configurations are listed in Table 4.3.

Table 4.3: CNN regression structure

Layer	Operation
Conv1 + ReLU	Conv2d(3→32, 3×3, pad=1) + MaxPool2d(2)
Conv2 + ReLU	Conv2d(32→64, 3×3, pad=1) + MaxPool2d(2)
Conv3 + ReLU	Conv2d(64→128, 3×3, pad=1) + MaxPool2d(2)
GAP	AdaptiveAvgPool2d(1×1)
FC1 + ReLU	Flatten + Linear(128→64) + ReLU
FC2	Linear(64→1)

The CNN+LSTM regressor extends the pure CNN backbone with temporal modeling: the pooled features are arranged into a T sequence and fed into a single-layer LSTM, whose hidden state at the final time step is then passed through a fully connected layer to produce the angular velocity output. Detailed layer configurations are listed in Table 4.4.

Table 4.4: CNN+LSTM regression structure

Layer	Operation
Conv1 + ReLU	Conv2d(3→32, 3×3, pad=1) + MaxPool2d(2)
Conv2 + ReLU	Conv2d(32→64, 3×3, pad=1) + MaxPool2d(2)
Conv3 + ReLU	Conv2d(64→128, 3×3, pad=1) + MaxPool2d(2)
GAP	AdaptiveAvgPool2d(1×1)
LSTM	LSTM(input size=128, hidden size=128, num layers=1)
FC	Linear(128→1)

Both models are trained with the Mean Absolute Error (MAE) objective. The angular velocity

lies in the range [–3, 3]. Because the variation per sample is small and the evaluation prioritizes minimizing absolute angular velocity error, MAE is preferred over Mean Squared Error (MSE). The loss function is as follows:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{B}\sum_{i=1}^{B}|\hat{\omega}_i - \omega_i| \tag{4.2.3}$$

$$\mathcal{L} = S \cdot \mathcal{L}_{\text{MAE}} \tag{4.2.4}$$

where $S > 0$ is a fixed scaling factor introduced to stabilize optimization and improve the clarity of the numerical signal without changing the optimal solution of the optimizer.

During validation, the model often exhibited conservative steering characteristics on bends, resulting in under-steering. To address this, curvature weighting (4.2.5) and asymmetric (4.2.6) weighting were used in some experiments. Here, α increases the weight of high-curvature (turning) samples, and $\beta_{\text{under}} > \beta_{\text{over}}$ imposes a stronger penalty on under-steering to counteract conservative turns. The resulting weighted loss shows in equation 4.2.7.

$$w_{\text{curve},i} = 1 + \alpha\,|\omega_i| \tag{4.2.5}$$

$$w_{\text{asym},i} = \beta_{\text{under}}\,\mathbf{1}\{\hat{\omega}_i < \omega_i\} + \beta_{\text{over}}\,\mathbf{1}\{\hat{\omega}_i \geq \omega_i\} \tag{4.2.6}$$

$$\mathcal{L} = \frac{1}{B}\sum_{i=1}^{B} w_{\text{curve},i}\, w_{\text{asym},i}\,|\hat{\omega}_i - \omega_i| \tag{4.2.7}$$

To mitigate class imbalance (many straight segments, fewer curves), resampling is applied in comparative experiments: with a threshold τ, samples satisfying $|\omega| > \tau$ are labeled as "curved". Let $\mathfrak{T} = i;\ |\omega_i| > \tau$. The number of curved samples will be doubled while preserving the temporal continuity within each sequence. This both balances the class distribution and enlarges the effective sample count for the more complex road segments, strengthening the learning performance on the curves.

Table 4.5 detailed the training hyperparameters of the CNN and CNN+LSTM models.

Beyond the single-output regression model, this paper further train a multi-output hybrid model (structure shown in 4.6). It adopts the best-performing CNN+LSTM architecture, and uses the same inputs and dataset setup as the single-task regression.

Table 4.5: Training hyperparameters

Hyperparameters	CNN	CNN+LSTM
sequence length (T)		5
Batch Size	128	64
Epochs	20	20
Learning Rate	1×10^{-3}	1×10^{-3}
Scaling Factor	10	10
τ	1.2	1.2
α	3	3
β_{under}	1	1
β_{over}	2	2

The model outputs a four-dimensional vector. Under the policy in baseline, the vehicle slows down whenever a crosswalk or parking sign is detected, otherwise it maintains its nominal speed. Consequently, the linear speed only takes values in $[0.40, 0.45]$ and is naturally treated as a binary class. Accordingly, the hybrid model comprises one regression head and one classification head: the regression head predicts the angular velocity, while the classification head produces three independent sigmoid outputs on shared features.

Even though the control policy derives linear speed from traffic sign detection, learning an additional speed classification head offers auxiliary supervision in ambiguous cases, leading to more robust behavior.

Table 4.6: CNN+LSTM hybrid multi-task structure

Layer	Operation
Conv1 + ReLU	Conv2d(3→32, 3×3, pad=1) + MaxPool2d(2)
Conv2 + ReLU	Conv2d(32→64, 3×3, pad=1) + MaxPool2d(2)
Conv3 + ReLU	Conv2d(64→128, 3×3, pad=1) + MaxPool2d(2)
Flatten	per-frame reshape to $128 \times 16 \times 16$
LSTM	LSTM(input size=$128 \times 16 \times 16$, hidden size=128, num layers=1)
Shared feature	take last-timestep hidden state (dim=128)
Head (regression)	Linear(128→1) [outputs $\hat{\omega}$]
Head (classification)	Linear(128→3) [logits for v, crosswalk, parking]

Because of the class imbalance, where crosswalk appears in roughly 2% of images, parking signs in 5%, and the low-speed state in about 20% of frames. The much higher rate of low-speed is because once a slowdown is triggered it persists for several frames. This paper rebalance the data via inverse-frequency resampling by label. For the empirical positive rate p, the resampling multiplier m is set to:

$$m = \frac{1-p}{p} + 0.5 \tag{4.2.8}$$

The term $\frac{1-p}{p}$ balances positives and negatives by oversampling rare samples. The small +0.5 is a tempering offset that prevents weights from collapsing to near zero and provides a floor that stabilizes the mini-batch composition.

The loss is composed of regression and classification: the regression loss is the same as in the single-output model, and the classification loss uses Binary Cross Entropy (BCE):

$$\mathcal{L} = \mathcal{L}_{\text{MAE}} + \mathcal{L}_{\text{BCE}} \tag{4.2.9}$$

Let $\mathcal{C} := \{ v, \text{ crosswalk}, \text{ park} \}$, where v denotes the linear speed class, crosswalk indicates the presence of a crosswalk, and park indicates the presence of a parking sign. For sample i, let $y_{i,c} \in \{0, 1\}$ be the ground-truth label and $p_{i,c} \in (0, 1)$the predicted probability for channel. Let $z_{i,c} \in \mathbb{R}$ be the corresponding logit with $p_{i,c} = \sigma(z_{i,c})$.

$$\mathcal{L}_{\text{BCE}}^{(i)} = -\sum_{c \in \mathcal{C}} \Big(y_{i,c} \log p_{i,c} + (1 - y_{i,c}) \log(1 - p_{i,c}) \Big). \tag{4.2.10}$$

4.3 Summary

The framework follows a three-layer “perception–decision–control” architecture, centered on dual pathways to estimate angular velocity ω and multitask behavior management. Two alternative routes are provided for ω estimation: (i) a traditional vision + PID pipeline (see Algorithm 2), which pre-processes the RGB image, performs a perspective - bird’s-eye transform, applies thresholding and a bottom-up sliding-window search to extract the right lane line, uses linear or quadratic curve fitting of midpoints to infer the center line of the lane, computes lateral error, and outputs ω via a proportional control law. (ii) an end-to-end supervised approach that loads a trained convolutional temporal model to regress ω directly from cropped and normalized image sequences. At run time, the system selects one of the two routes to produce ω.

For multitask behaviors (see Algorithm 1), OpenCV-based color segmentation and counting detect key events: pedestrians (stop), crosswalks (decelerate), and parking signs (decelerate and trigger parallel parking). An FSM coordinate behavior scheduling and priority: pedestrian stop has the highest priority, followed by parking-sign and crosswalk deceleration, each constraining the current speed target. When the parking trigger is satisfied, a predefined parking sequence is executed and the system then returns to lane following. After each cycle, linear and angular velocity commands are published as ROS topics, enabling stable lane keeping and robust handling of diverse scenarios.

Algorithm 1 Autonomous Driving Control Loop

Require: Control mode $\in$ {E2E, PID}; model f_θ (if E2E); PID gain K_p; cruise speed v_{cruise}; slow speed v_{slow}; sampling period T_s

while system is running **do**
  $I_t \leftarrow$ ReadCamera() ▷ RGB frame
  $I'_t \leftarrow$ PreprocessImage(I_t) ▷ crop/resize/normalize or binary ROI
  if mode = E2E
    $\omega \leftarrow f_\theta(I'_t)$ ▷ end-to-end inference
  else ▷ traditional vision + PID
    $\omega \leftarrow$ TraditionalVisionPID(I_t)
  end if
  $v \leftarrow v_{\text{cruise}}$ ▷ default speed
  if DetectPedestrian(I_t)
    $v \leftarrow 0$
    PublishCmd(v, ω)
    continue ▷ hold stop until cleared
  end if
  if CountBlueRight(I_t) > $\tau_{\text{crosswalk}}$
    $v \leftarrow v_{\text{slow}}$
  end if
  if CountBlueLeft(I_t) > τ_{park}
    $v \leftarrow v_{\text{slow}}$
    if InParkingTriggerZone()
      ParallelParkingSequence()
      ResumePIDorE2E() ▷ return to lane-following
      continue
    end if
  end if
  $\omega \leftarrow$ Clamp(ω, $-\omega_{\max}$, $\omega_{\max}$)
  $v \leftarrow$ Clamp(v, 0, $v_{\max}$)
  PublishCmd(v, ω)
  Wait(T_s)
end while

Algorithm 2 TraditionalVisionPID

Require: RGB frame I_t; homography $\mathbf{H}$; lane width (pixels) W_p; window-count threshold τ_{win}; P-gain K_p; steering limit $\omega_{\max}$

Ensure: Steering rate ω

$I'_t \leftarrow \text{PreprocessImage}(I_t)$ ▷ crop/denoise/normalize or ROI

$B_t \leftarrow \text{WarpPerspective}(I'_t,\ \mathbf{H})$ ▷ top-down view

$M_t \leftarrow \text{WhiteMask}(B_t)$ ▷ binary mask of lane markings

$x_{\text{seed}} \leftarrow \text{BottomBandPeak}(M_t)$ ▷ seed column from the bottom band histogram

$\mathcal{P} \leftarrow [\,]$; hits $\leftarrow 0$

$(x_c, y_c) \leftarrow (x_{\text{seed}},\ \text{BottomRow}(M_t))$

while $\text{WithinImage}(y_c)$ **do**

  $\mathcal{W} \leftarrow \text{MakeWindow}(M_t,\ x_c,\ y_c)$ ▷ rectangular window centered at (x_c, y_c)

  $\mathcal{S} \leftarrow \text{WhitePixelsIn}(\mathcal{W})$ ▷ white pixels inside the window

  if $|\mathcal{S}| = 0$ ▷ no support at this level

    break

  end if

  $x_{\text{med}} \leftarrow \text{MedianX}(\mathcal{S})$ ▷ median of x within the window

  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(x_{\text{med}},\ \text{WindowMidY}(\mathcal{W}))\}$

  hits $\leftarrow$ hits + 1

  $x_c \leftarrow x_{\text{med}}$ ▷ recenter next window on the median

  $y_c \leftarrow \text{ShiftUp}(\mathcal{W})$ ▷ move the window upward by one step

end while

if hits $< \tau_{\text{win}}$ ▷ insufficient windows ⇒ fit deemed unreliable

  return 0

end if

$\hat{\ell}(y) \leftarrow \text{PolyFit}(\mathcal{P})$ ▷ fit the right-lane curve from collected medians

$y^\star \leftarrow \text{RefRow}(M_t)$ ▷ reference row near the image bottom

$x_{\text{right}} \leftarrow \hat{\ell}(y^\star)$

$x_{\text{mid}} \leftarrow x_{\text{right}} - \dfrac{W_p}{2}$ ▷ centerline from right-lane position

$W \leftarrow \text{ImageWidth}(M_t)$; $x_{\text{veh}} \leftarrow \dfrac{W}{2}$ ▷ vehicle center column

$e \leftarrow x_{\text{mid}} - x_{\text{veh}}$ ▷ lateral error (pixels)

$\omega \leftarrow -K_p \cdot e$ ▷ proportional steering

$\omega \leftarrow \text{Clamp}(\omega,\ -\omega_{\max},\ \omega_{\max})$

return ω

5 Experiments

This chapter presents the changes in loss during training for the single-output regression model and the multi-task hybrid model, respectively. Four models for 20 epochs each are trained for the regression model. One model for the hybrid model is trained. After applying standard image augmentations, the training set comprised approximately 200000 images.

5.1 Single-output Regression Rodel

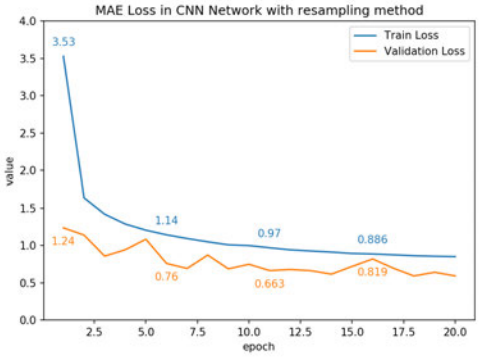
5.1.1 Resampling vs. Weighted Loss

Models trained with resampling (shown in Fig. 5.1c and 5.1a) achieved training and validation loss below 1, with the two curves gradually converge in the later stages, this indicates stable convergence and consistent generalization capabilities. In contrast, models trained with a weighted loss (shown in Fig. 5.1b and 5.1d) converged to a loss of approximately 4. During training, the train and validation curves occasionally crossed, yet toward the end they approached similar values.

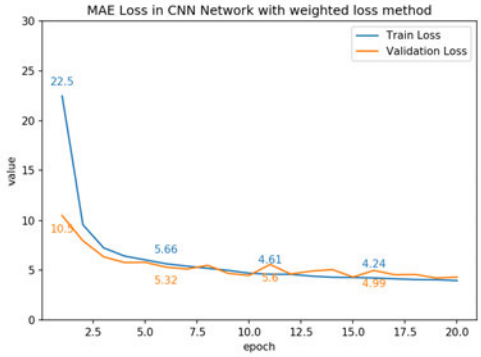
Incorporating asymmetric penalties increases the contribution of turning segments to the objective. Consequently, when reporting MAE, the curves tend to start at a higher level and terminate at a higher absolute value. Despite crossovers under the weighted loss, the final train and validation loss converge to comparable levels. Considering the overall learning curve, there is no evidence of overfitting in this setting.

5.1.2 CNN vs. LSTM

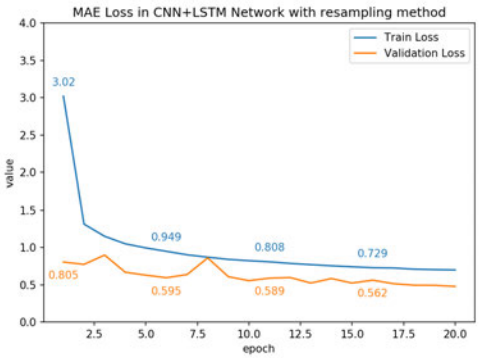
Regardless of whether resampling or a weighted loss method is used, the LSTM model (shown in Fig. 5.1c and 5.1d) consistently attains a lower overall loss than CNN models (shown in Fig. 5.1a and 5.1b). This indicates that, for angular velocity prediction, explicit temporal modeling better exploits inter-frame statistical dependencies and kinematic continuity, thereby improv-



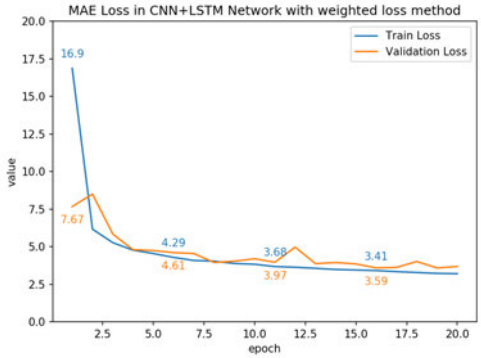
(a) MAE Loss in CNN Network with resampling method



(b) MAE Loss in CNN Network with weighted loss method



(c) MAE Loss in CNN+LSTM Network with resampling method



(d) MAE Loss in CNN+LSTM Network with weighted loss method

Figure 5.1: Global overview of loss decline

ing both fit quality and robustness.

Angular velocity sequences typically exhibit marked autocorrelation and smoothness, especially in turning segments where ω tends to maintain a consistent sign and evolve gradually. By virtue of its gated memory, the LSTM aggregates evidence over time, smoothing short-term noise and capturing turn trends. Consequently, in right-hand-traffic environments with commonly counterclockwise bends, temporal modeling yields particularly pronounced error reductions on turns, reflected in lower MAE and smoother loss trajectories.

5.2 Multi-task Hybrid Model

The Figure 5.2 report the total loss of the hybrid model. Both training and validation losses decrease and intersect around epoch 5. After the intersection, the validation curve continues to fall, but with a shallower slope than the training curve.

This behavior does not indicate overfitting. After about epoch 5 the validation curve remains

above the training curve but it continues to decline. Overfitting is indicated when the validation loss decreases to a minimum and then begins to rise, even as the training loss continues to fall. This situation is not observed. The checkpoint with the lowest validation loss is selected, rather than the final-epoch weights.

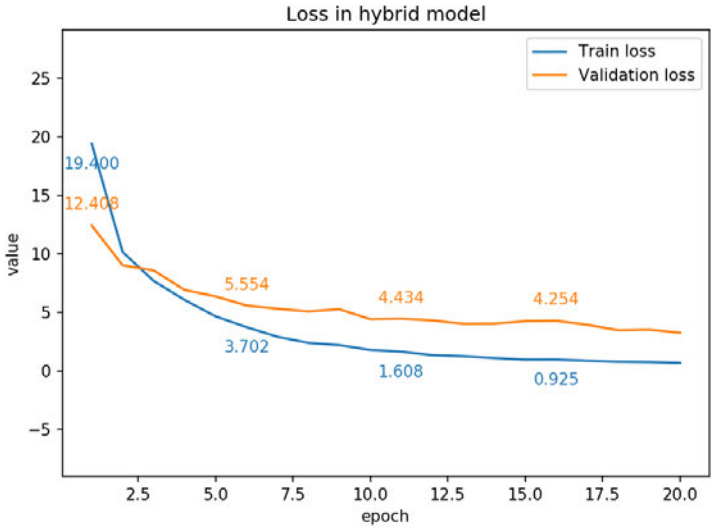


Figure 5.2: Loss in hybrid model

6 Performance Evaluation

This chapter evaluates the single-output regression model and the multi-task hybrid model. A test dataset, disjoint from the training and validation dataset, was constructed. It contains roughly 10000 untrained images, captured as consecutive frames under PID control. Both model are first evaluated offline on static images to assess whether their test losses differ substantially from those observed during training. The regression model is then evaluated in a real, closed loop setting to assess lane keeping performance. Because the hybrid model incurred substantial inference latency in real-world tests. The steering predictions could not update the vehicle pose in time, leading to unstable operation. Therefore, hybrid model only tests statically and report the performance of its regression and classification heads separately.

6.1 Single-output Regression Model

The table 6.1 compares each single-output regression model's loss on the static test set with its best validation loss. Across models, the test loss is slightly higher than the best validation loss, but the gap is small, indicating stable generalization under the same pre-processing and evaluation metric. In real-world dynamic tests, however, only the LSTM achieves stable lane keeping. The other models without explicit temporal modeling perform well on straight segments but frequently under-steer or over-steer in curves, failing to complete turns. The LSTM that performs best in dynamic tests also attains the lowest loss. Although absolute loss differences are small, the vehicle's steering range is narrow. Thus, small errors in ω can destabilize cornering. Moreover, with a low-mounted camera, both overly aggressive and overly conservative commands can cause the lane detection failure within roughly two seconds.

Next, the best-performing LSTM model is compared with the PID-control baseline. Both models are run at the same time and under comparable lighting. The angular velocity trajectory over a full circle is recorded. As shown in the figure 6.1b, LSTM model predicted ω fluctuates less. At curve exit, left-turn overshoot is often observed under PID control, followed by a brief corrective right turn (negative ω) to realign. LSTM outputs remain more stable, yielding more precise control of angular velocity.

Table 6.1: Performance of single-output regression model

	Best validation loss	test loss	real world performance
cnn with resampling	0.5914	0.6492	turn conservatively
cnn with weighted loss	4.2315	4.8759	turn aggressively
lstm with resampling	0.4781	0.5115	turn smoothly
lstm with weighted loss	3.5843	3.8941	turn aggressively

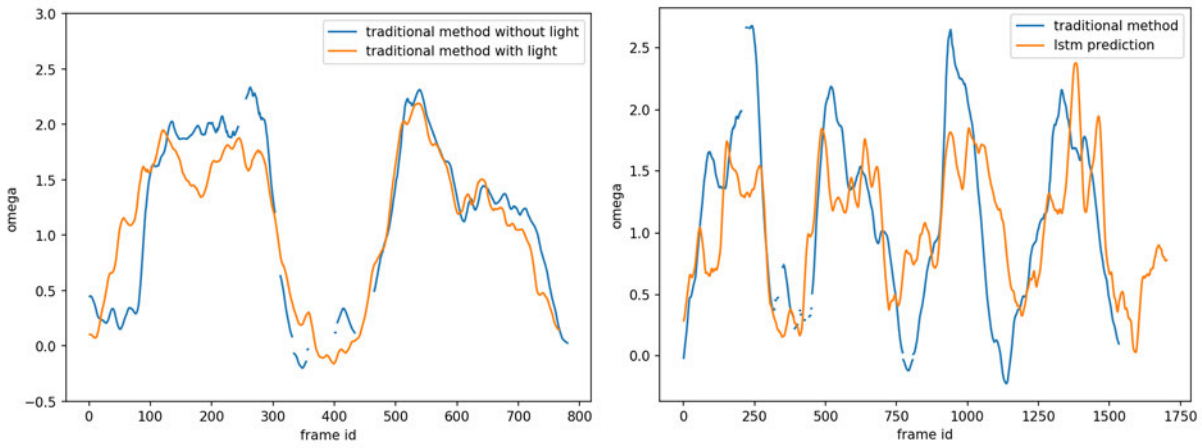
Figure 6.1a compares the changes of the angular velocity along a double-lane segment under two lighting conditions: ambient indoor lighting versus ambient plus supplemental illumination. Under indoor lighting, the classical vision-based lane detector intermittently fails. In the line plot, time steps with detection failure are encoded as $\omega = NaN$ and appear as gaps, which coincide with intervals where the lane cannot be fitted. The controller's fallback in these intervals is to output $\omega = 0$ (i.e., hold the current heading). Because failures occur primarily on straight segments and are not contiguous over the entire road, the vehicle exhibits sporadic lane-keeping failures. Even when it nominally remains in the right lane, the trajectory shows a noticeable snake-like weave, indicating that the traditional computer vision have stability issues.

With supplemental illumination, lane marking contrast is markedly improved. Detection and geometric fitting proceed reliably. Thus the PID controller produces ω that is smooth, continuous, and stable, with virtually no breaks in the plot. This demonstrates that lane keeping based on traditional computer vision is highly sensitive to illumination: when lighting is poor, detection reliability degrades first and the effect propagates to control.

Note that the images used to train the end-to-end model were RGB frames captured with supplemental lighting. Consequently, the end-to-end model is trained predominantly on better-lit samples, whereas the classical pipeline must operate at inference time in darker scenes, placing it at an inherent disadvantage.

Figure 6.1b reports a full-lap comparison under nighttime low-light conditions with only indoor lighting, over the same time, between classical vision + PID control and end-to-end + LSTM control. The classical pipeline still does not acquire the lane markings stably in specific segments, producing intermittent ω output, while the end-to-end model maintains continuous and smooth predictions under low illumination and completes the lap successfully.

These results indirectly suggest that temporal deep models can extract richer features than methods relying solely on color or edge thresholding, leading to greater robustness to illumination variation and local under-exposure.



(a) ω fluctuation under different lighting condition (b) ω fluctuation over time: PID VS. End-to-End

Figure 6.1: The superiority of End-to-End model omega prediction

6.2 Multi-task Hybrid Model

Table 6.2 compares the hybrid model's loss on the static test set with its best validation loss. The test loss is slightly lower than the best validation loss, indicating that, at least in the static evaluation, the model generalizes well and the loss shows no large fluctuations.

Table 6.2: Performance of multi-task hybrid model

	Best validation loss	test loss
hybrid model	3.2224	2.9114

The Table 6.3 reports the confusion matrix for linear speed on the static test set. The test set contains 9758 samples. Negatives account for 7582/9758 $\approx$ 78% and positives for 2176/9758 $\approx$ 22%, indicating some class imbalance. True Positive (TP) and True Negative (TN) make up the vast majority, yielding an accuracy of about 99.1%. This suggests that segments requiring a slowdown are well identified. False Negative (FN) and False Positive (FP) are rare (likely occurring near speed-transition boundaries) and their overall impact on the speed control task is limited.

The Table 6.4 reports the confusion matrix for the crosswalk sign on the static test set. The test set contains 9,758 samples. Negatives account for 9560/9758 $\approx$ 98% and Positives for 198/9758 $\approx$ 2%, indicating severe class imbalance. TP and TN make up the vast majority, yielding an accuracy of about 99.7%. FN and FP account for a very small proportion, implying that crosswalk miss-detections are rare. In the original dataset, a frame is labeled positive for

Table 6.3: Confusion matrix of the linear velocity

	Predicted	
Actual	Positive	Negative
Positive	2143	33
Negative	54	7528

crosswalk only when the vehicle is sufficiently close to the sign, and some labeling noise can occur near this boundary. Thus, a small number of misclassifications is acceptable.

Table 6.4: Confusion matrix of the crosswalk sign

	Predicted	
Actual	Positive	Negative
Positive	192	6
Negative	27	9533

The Table 6.5 reports the confusion matrix for the parking sign on the static test set. The test set contains 9,758 samples. Negatives account for 9202/9758 $\approx$ 94% and positives for 556/9758 $\approx$ 6%, indicating pronounced class imbalance. TP and TN constitute the vast majority, yielding an accuracy of about 99.7%. Similar to the crosswalk sign, the parking-sign label is set to positive only when the vehicle is sufficiently close, rather than whenever the sign is simply visible. Consequently, FN and FP for parking are both low, indicating that parking sign detection is already performing very well.

Table 6.5: Confusion matrix of the parking sign

	Predicted	
Actual	Positive	Negative
Positive	541	15
Negative	17	9185

For linear speed prediction, Precision = 0.98, Recall = 0.98, and F1 = 0.98, indicating that slowdown triggering is highly reliable: when the model predicts a slowdown, it is almost always guaranteed, and the vast majority of slowdown frames are correctly detected.

For crosswalk detection, Precision = 0.88, Recall = 0.97, F1 = 0.92. The markedly higher recall means crosswalks are rarely missed, although with some false positives, which is an acceptable trade-off under a safety-first policy.

For parking-sign detection, Precision = 0.97, Recall = 0.97, and F1 = 0.97. Both false positives and false negatives are rare, indicating stable and reliable recognition.

Overall, the weighted averages of all metrics are very high, indicating stable performance. The model delivers strong recognition and classification quality on images in the static test setting.

Table 6.6: Classification report

Label	Precision	Recall	F1-score	Support
v	0.98	0.98	0.98	2176
crosswalk	0.88	0.97	0.92	198
parking	0.97	0.97	0.97	556
weighted avg	0.97	0.98	0.97	2930

7 Conclusion and Future Work

7.1 Summary

This work builds and validates autonomous driving software systems on two platforms, the Bosch car and the Duckiebot, establishing a baseline based on traditional computer vision and PID control. On this basis, explores the feasibility of simple and hybrid models. The main findings are as follows:

1. Cross-platform system implementation and latency differences
   On the Bosch car, the core control unit is compiled and executed in C++, yielding low latency. This paper successfully implemented the right lane keeping function and achieved pedestrian detection through simple color recognition technology. This fully demonstrates the practicality of traditional computer vision and PID controllers on these platforms.

   On the Duckiebot, while most of the Bosch logic was retained. The restrictions imposed by the official Duckietown system require the core control unit to be implemented in Python, which significantly increases overall latency. Under these conditions, the baseline system still achieved lane detection, right-lane keeping, pedestrian and traffic-sign recognition, vehicle speed control, and parallel parking.

2. Performance and limitations of end-to-end and hybrid models
   Using images collected by the baseline system, this paper trained an end-to-end model that directly predicts angular velocity and enabled end-to-end right-lane keeping on the Duckiebot. This paper also trained a hybrid model that performed well on static test sets. However, due to the higher latency of the Python stack, timely control outputs could not be guaranteed during model loading, which limited on vehicle performance and prevented stable lane keeping.

3. Robustness comparison
   Under challenging conditions such as lighting variations, the end-to-end model is insensitive to illumination changes and can continuously and stably output reasonable angular velocities. In contrast, traditional computer vision methods are sensitive to lighting quality and are prone to lane detection failures under poor lighting conditions, resulting in unstable control. This indicates that, in the experiments of this paper, end-to-end learning exhibits stronger robustness and generalization.

7.2 Limitations and Future Work

1. Limited scope of end-to-end
   At present, the end-to-end pipeline covers only right-lane keeping. Speed control and parallel parking still depend on traditional computer vision and are sensitive to illumination. Given sufficient time and data, collecting enough parallel-parking images could enable stable end-to-end parallel parking and further advance a fully end-to-end multi-task autonomous driving system.

2. Hybrid model not deployed on vehicle
   The hybrid model that performed well in testing has not yet been applied in real-world operation. If hardware permits, a lower-latency C++ stack should be built for validation, which is an essential step toward fully end-to-end autonomous driving.

3. Instability due to hardware constraints
   The Duckiebot camera is mounted too low, making it difficult to capture more than two lane markings simultaneously. Once the heading deviates, the system easily "gets lost" and functionality becomes unstable. Adjusting the hardware (raising the camera height) could substantially improve driving robustness.

4. Limited generalization of supervised learning
   Training data produced via PID only consists of "correct driving" images. When a vehicle deviates from its normal trajectory, the input data generated exceeds the distribution range, causing a decline in model performance. In contrast, simulation-based trial-and-error RL is more conducive to generalization, especially in recovering from incorrect postures to correct trajectories.

Bibliography

[1] Santokh Singh. Critical reasons for crashes investigated in the national motor vehicle crash causation survey. Technical report, 2015.

[2] Michele Bertoncello and Dominik Wee. Ten ways autonomous driving could redefine the automotive world. *McKinsey & Company*, 6:1–5, 2015.

[3] Junyi Chai, Hao Zeng, Anming Li, and Eric WT Ngai. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6:100134, 2021.

[4] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020.

[5] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[6] Ahmad El Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. End-to-end deep reinforcement learning for lane keeping assist. *arXiv preprint arXiv:1612.04340*, 2016.

[7] Yuchuan Du, Jing Chen, Cong Zhao, Chenglong Liu, Feixiong Liao, and Ching-Yao Chan. Comfortable and energy-efficient speed control of autonomous vehicles on rough pavements using deep reinforcement learning. *Transportation Research Part C: Emerging Technologies*, 134:103489, 2022.

[8] Wenli Li, Yousong Zhang, Xiaohui Shi, and Fanke Qiu. A decision-making strategy for car following based on naturalist driving data via deep reinforcement learning. *Sensors*, 22(20):8055, 2022.

[9] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[10] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, 1985.

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[12] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106, 1962.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[14] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[15] Anis Koubâa et al. *Robot Operating System (ROS).*, volume 1. Springer, 2017.

[16] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science robotics*, 7(66):eabm6074, 2022.

[17] ROS.org. Open robotics, 2025.

[18] EUREKA Network. Programme for a european traffic system with highest efficiency and unprecedented safety, 2013.

[19] Berthold Ulmer. Vita ii-active collision avoidance in real traffic. In *Proceedings of the Intelligent Vehicles' 94 Symposium*, pages 1–6. IEEE, 1994.

[20] John Fulton and Joanne Pransky. Darpa grand challenge–a pioneering event for autonomous robotic ground vehicles. *Industrial Robot: An International Journal*, 31(5):414–422, 2004.

[21] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The DARPA urban challenge: autonomous vehicles in city traffic*, volume 56. Springer Science & Business Media, 2009.

[22] Dianzhao Li, Paul Auerbach, and Ostap Okhrin. Autonomous driving small-scale cars: A survey of recent development. *IEEE Transactions on Intelligent Transportation Systems*, 2025.

[23] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017.

[24] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, et al. Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning. In *2020 IEEE international conference on robotics and automation (ICRA)*, pages 2746–2754. IEEE, 2020.

[25] Bosch Engineering Center. Bosch future mobility challenge, 2025.

[26] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8:58443–58469, 2020.

[27] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[28] Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrell. End-to-end learning of driving models from large-scale video datasets. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2174–2182, 2017.

[29] Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, and Urs Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911*, 2017.

[30] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2013.

[31] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *arXiv preprint arXiv:1704.02532*, 2017.

[32] ALEX KENDALL, JEFFREY HAWKE, DAVID JANZ, PRZEMYSLAW MAZUR, DANIELE REDA, JOHN-MARK ALLEN, VINH-DIEU LAM, ALEX BEWLEY, and AMAR SHAH. Learning to drive in a day. arxiv e-prints. *arXiv preprint arXiv:1807.00412*, 2018.

[33] Dianzhao Li and Ostap Okhrin. A platform-agnostic deep reinforcement learning framework for effective sim2real transfer towards autonomous driving. *Communications Engineering*, 3(1):147, 2024.

[34] Dianzhao Li and Ostap Okhrin. Vision-based drl autonomous driving agent with sim2real transfer. In *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*, pages 866–873. IEEE, 2023.

Statement of authorship

I hereby declare on my honor that I have prepared this paper independently. The thoughts taken directly or indirectly from external sources are marked as such. The work has not been submitted to any other examination authority and has not been published. I am aware that an untrue statement will have legal consequences.

Dresden, August 20, 2025

_____

Bingyu Wang