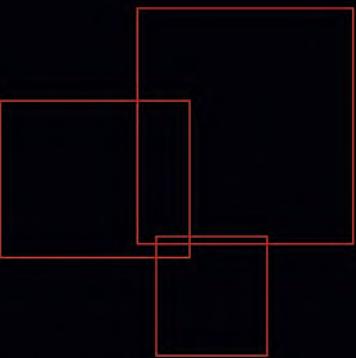


Qingguo Zhou
Zebang Shen
Binbin Yong
Rui Zhao
Peng Zhi



Theories and Practices of Self-Driving Vehicles



THEORIES AND PRACTICES OF SELF- DRIVING VEHICLES

This page intentionally left blank

THEORIES AND PRACTICES OF SELF- DRIVING VEHICLES

QINGGUO ZHOU

Professor, Lanzhou University and Deputy Director,
Engineering Research Center for Open Source Software
and Real-Time Systems, Ministry of Education, China

ZEBANG SHEN

R&D Center, Mercedes-Benz Group AG, Beijing, China

BINBIN YONG

An associate professor and masters supervisor in the
School of Information Science and Engineering, Lanzhou
University, China

RUI ZHAO

School of Information Science & Engineering, Lanzhou
University, Lanzhou, Gansu, China

PENG ZHI

School of Information Science & Engineering, Lanzhou
University, Lanzhou, Gansu, China



Elsevier
Radarweg 29, PO Box 211, 1000 AE Amsterdam, Netherlands
The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, United Kingdom
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Copyright © 2022 Huazhong University of Science and Technology Press. Published by Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-323-99448-4

For information on all Elsevier publications visit our website at
<https://www.elsevier.com/books-and-journals>

Publisher: Glyn Jones
Acquisitions Editor: Glyn Jones
Editorial Project Manager: Naomi Robertson
Production Project Manager: Prasanna Kalyanaraman
Cover Designer: Christian J. Bilbow

*Supported by the textbook construction fund of
Lanzhou University*

Typeset by TNQ Technologies



www.elsevier.com • www.bookaid.org

Working together
to grow libraries in
developing countries

Contents

Contributors

ix

1. First acquaintance with unmanned vehicles	1
Zebang Shen, Ming Lei, Peng Zhi and Rui Zhao	
1.1 What are unmanned vehicles?	2
1.2 Why do we need unmanned vehicles?	7
1.3 Basic framework of the unmanned vehicle system	9
1.4 Development environment configuration	22
References	25
2. Introduction to robot operating system	27
Zebang Shen, Xiaowei Xu, Peng Zhi and Rui Zhao	
2.1 ROS introduction	28
References	62
3. Localization for unmanned vehicle	63
Zebang Shen, Gang Huang, Peng Zhi and Rui Zhao	
3.1 Principle of achieving localization	63
3.2 ICP algorithm	65
3.3 Normal distribution transform	72
3.4 Localization system based on global positioning system (GPS) + inertial navigation system (INS)	81
3.5 SLAM-based localization system	87
References	93
4. State estimation and sensor fusion	95
Zebang Shen, Yu Sun, Peng Zhi and Rui Zhao	
4.1 Kalman filter and state estimation	96
4.2 Advanced motion modeling and EKF	115
4.3 UKF	138
References	144

5. Introduction of machine learning and neural networks	147
Zebang Shen, Binbin Yong and Peng Zhi	
5.1 Basic concepts of machine learning	148
5.2 Supervised learning	151
5.3 Fundamentals of neural network	157
5.4 Using Keras to implement the neural network	165
References	175
6. Deep learning and visual perception	177
Zebang Shen, Tingting Yu, Peng Zhi and Rui Zhao	
6.1 Deep feedforward neural networks—why is it necessary to be deep?	178
6.2 Regularization technology applied to deep neural networks	180
6.3 Actual combat—traffic sign recognition	187
6.4 Introduction to convolutional neural networks	196
6.5 Vehicle detection based on YOLO2	206
References	216
7. Transfer learning and end-to-end self-driving	217
Zebang Shen, Yunfei Che and Rui Zhao	
7.1 Transfer learning	218
7.2 End-to-end selfdriving	220
7.3 End-to-end selfdriving simulation	221
7.4 Summary of this chapter	228
References	229
8. Getting started with self-driving planning	231
Zebang Shen, Wei Wang and Rui Zhao	
8.1 A* algorithm	232
8.2 Hierarchical finite state machine (HFSM) and autonomous vehicle behavior planning	241
8.3 Autonomous vehicle route generation based on free boundary cubic spline interpolation	247
8.4 Motion planning method of the autonomous vehicle based on Frenet optimization trajectory	255
References	272

9. Vehicle model and advanced control	273
Zebang Shen and Rui Zhao	
9.1 Kinematic bicycle model and dynamic bicycle model	273
9.2 Rudiments of autonomous vehicle control	278
9.3 MPC based on kinematic model	291
9.4 Trajectory tracking	295
References	305
10. Deep reinforcement learning and application in self-driving	307
Jinqiang Wang and Rui Zhao	
10.1 Overview of reinforcement learning	308
10.2 Reinforcement learning	309
10.3 Approximate value function	314
10.4 Deep Q network algorithm	315
10.5 Policy gradient	319
10.6 Deep deterministic policy gradient and TORCS game control	319
10.7 Summary	325
References	325
Index	327

This page intentionally left blank

Contributors

Yunfei Che

Huawei Technologies Co., Ltd., Xi'an, China

Gang Huang

SATG, Intel, Shanghai, China

Ming Lei

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Xiaowei Xu

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Zebang Shen

R&D Center, Mercedes-Benz Group AG, Beijing, China

Yu Sun

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Jinqiang Wang

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Wei Wang

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Binbin Yong

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Tingting Yu

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Rui Zhao

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Peng Zhi

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Qingguo Zhou

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

This page intentionally left blank

CHAPTER 1

First acquaintance with unmanned vehicles

Zebang Shen¹, Ming Lei², Peng Zhi² and Rui Zhao²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

1.1 What are unmanned vehicles?	2
1.1.1 Classification standards for unmanned vehicles	2
1.1.2 How difficult is the implementation of unmanned vehicles?	5
1.2 Why do we need unmanned vehicles?	7
1.2.1 Improvement of road traffic safety	7
1.2.2 Alleviation of urban traffic congestion	8
1.2.3 Improvement of travel efficiency	8
1.2.4 Lowering the threshold for drivers	9
1.3 Basic framework of the unmanned vehicle system	9
1.3.1 Environmental perception	10
1.3.2 Localization	13
1.3.3 Mission planning	14
1.3.4 Behavior planning	16
1.3.5 Motion planning	17
1.3.6 Control system	18
1.3.7 Summary	21
1.4 Development environment configuration	22
1.4.1 Simple environment installation	22
1.4.2 Install the robot operating system (ROS)	23
1.4.3 Install OpenCV	24
References	25

With the rapid advancement of driverless technology in the past 2 years, major vehicle companies and driverless system solution providers, such as Baidu Apollo and Jingchi, have also been making continuous efforts to commercialize driverless technologies. Obviously, unmanned vehicle technology is no longer an out-of-reach “technology of the future.” The field of unmanned vehicle technology includes not only vehicle control, path planning, perception fusion, and other fields but also cutting-edge fields such as artificial intelligence, machine learning, deep learning, and

reinforcement learning. Unmanned vehicles are bound to set off a new technological and market revolution in the next 5–10 years.

From the perspective of engineering applications, learning and applying various basic algorithms in unmanned vehicle systems are important tasks. This chapter presents an overview of unmanned vehicle systems and introduces the concept of unmanned vehicles and unmanned vehicle systems, the significance of unmanned vehicle technology, the development history of unmanned vehicle technology, and the architecture of modern unmanned vehicle systems. Through this book, readers will have a clear and complete understanding of currently popular unmanned vehicle systems and algorithm systems.

1.1 What are unmanned vehicles?

Unmanned vehicles, which are also called autonomous driving or self-driving vehicles, are vehicles that are able to control their movements on the basis of their own perceptions and understanding of the surrounding environmental conditions. With such capabilities, they could reach the level of human-driven vehicle systems.

In addition to being an interdisciplinary subject, unmanned vehicle systems are composed of a wide range of technologies, including multi-sensor fusion technology, signal processing technology, communication technology, artificial intelligence technology, and computer technology. In sum, the technology of unmanned vehicle systems identifies the surrounding environment and vehicle state by using a variety of vehicle sensors, such as cameras, LIDAR, millimeter-wave radars, global positioning systems (GPS), and inertial sensors. The obtained environmental information, including road information, traffic information, vehicle location, and obstacle information, can be analyzed and judged independently to achieve the autonomous control of vehicle movement that is expected of unmanned vehicles.

1.1.1 Classification standards for unmanned vehicles

In the classification of vehicle intelligence, the industry currently has two sets of standards: one is set by the National Highway Traffic Safety Administration (NHTSA) under the U.S. Department of Transportation, while the other is set by the Society of Automotive Engineers (SAE) International. The L0, L1, and L2 classifications of the two institutions are the same. Meanwhile, their L4 classifications differ, with the L4 classification by

Table 1.1 Classification mechanism for unmanned vehicle systems under SAE standards.

Level	Name	Steering, acceleration, and deceleration control	Observation of the environment	Response to intense driving	Coping with working conditions
L0	Manual driving	Driver	Driver	Driver	
L1	Assisted driving	Driver + system	Driver	Driver	Partly
L2	Semiautonomous driving	System	Driver	Driver	Partly
L3	Autonomous driving	System	System	Driver	Partly
L4	Highly autonomous driving	System	System	System	Partly
L5	Fully autonomous driving	System	System	System	Fully

the NHTSA subdivided into L4 and L5 under the SAE standards. As the SAE standards are mainly adopted in China, this book uses SAE standards for the introduction. [Table 1.1](#) presents the SAE grading standard.¹

The L0 level means that the vehicle is driven by a human driver.

L1, also known as assisted driving, indicates the presence of advanced driver assistance system (ADAS) functions, including lane departure warning, forward collision warning, and blind spot detection warning lights and so on. The ADAS mainly provides early warning, and it has no active intervention functions.

L2 denotes semiautonomous driving or partially automatic driving. This type of system already has ADAS functions for intervention assistance, including adaptive cruise control (ACC), automatic emergency braking, and lane keeping assist system, etc. Vehicles of this level are equipped with functions such as autonomous acceleration on highways or autonomous braking in emergency situations. Relative to L4 vehicles, L2 vehicles can achieve simple automatic control operations.

From L2 to L3, the capabilities of an unmanned vehicle system undergo a fundamental change. For vehicles with L2 classification or lower, drivers still monitor their environment and are required to control their vehicles directly. L3 denotes autonomous driving. An unmanned vehicle system with an L3 classification is equipped with comprehensive intervention assistance functions, including automatic acceleration, automatic braking,

and automatic steering. For an L3 vehicle, it can perceive the surrounding driving environment according to its own sensors. However, it still requires a human driver to perform monitoring tasks and even intervene in the autonomous driving system in emergency situations.

L4 denotes highly autonomous driving, which means that in a limited area or limited environment, such as a fixed park, closed area, or semi-enclosed highway. The L4 vehicle can fully perceive the environment and autonomously intervene in emergencies without the aid of a human driver. At the L4 level, the vehicle can have no steering wheel, accelerator, or brake pedal, but it can only be used in special scenarios and environments. The main difference between L4 and L3 lies in the need for human intervention. L4 unmanned vehicles can solve problems on their own in emergency scenarios. By contrast, L3 unmanned vehicles require the intervention of human drivers during emergencies.

L5 denotes fully autonomous driving. An L5 vehicle requires neither a driver nor any user to intervene in the steering wheel, accelerator, brakes, etc. It is not limited to driving in specific scenarios and can be adapted to automatic driving in any scene and environment.

At present, most companies are still at the L2–L4 stages of driverless technology. Hence, existing driverless prototype vehicles can only be tested in specific and restricted areas (such as closed or semiclosed parks and some high-speed sections of a highway with good road conditions). The safety officers onboard such vehicles are required to intervene at any time. Many internet companies, such as Baidu, Waymo, Uber, Jingchi, and Xiaoma, are testing and developing L4 driverless systems, but at the time of writing this book, several practical problems still remain for L4, including technology, cost, mass production, laws, regulations, and the market. Some companies claim that they have implemented L4 driverless driving in specific parks, but such a claim is doubtful. On the one hand, these fixed parks are generally small in scale and simple in scenarios. Hence, they could not cover certain realistic complex scenes, such as those involving traffic lights, various complexities, and batches of pedestrians crossing the road. The demo that can only be used for unmanned vehicles in such a special area does not reach the L4 classification in practice. On the other hand, fixed parks are obviously open and complicated as far as unmanned areas are concerned. Unmanned vehicles on open roads are much more complicated than unmanned vehicles in a closed and semiclosed environment. There is a “long tail effect” in unmanned vehicle technology; that is, the last 5% of technical problems that need to be solved may cost more than 95%. On the

other hand, the current L4 level of unmanned vehicle technology is mainly based on high-definition maps (HD maps). However, the construction of HD maps entails high costs, and the scope of application of L4 is limited. Hence, the driverless system is almost completely inoperable when the basic map changes greatly or when no HD map is available. This feature also explains why the current cases reported by startup companies are mostly well-designed venue demonstrations rather than real practical applications of driverless vehicles on ordinary public roads. This current scenario also raises the topic of the level of difficulty of realizing driverless technology on real public roads. This topic is elaborated on herein.

1.1.2 How difficult is the implementation of unmanned vehicles?

Before discussing the difficulty of implementing a reliable unmanned vehicle system that runs on public roads, let us view a picture of the traffic condition on a public road. Fig. 1.1 shows a common road condition in India.

The road shown in the figure has no clear lane markings and is utilized by various user types, including passersby, bicycles, tricycles, animals, and even horse-drawn carriages. This example may seem extreme, but such road conditions are common in developing countries. Although China's urban public road facilities (road lines, traffic signs, traffic lights) are relatively complete, the areas such as villages and counties are made up of many types of traffic participants whose behaviors are unpredictable and are characterized by complicated situations. For complex traffic scenes, any well-trained and sober human driver can easily gain control over problems and successfully complete the driving task. However, unmanned vehicle



Figure 1.1 Traffic condition in India.

systems cannot easily deal with such scenarios, given the current technology accumulation. The complex and changing traffic conditions are one of the biggest obstacles in realizing fully automatic driving.

Another hurdle in the implementation of unmanned vehicles is attributed to human regulations and systems. Road traffic presents the various states in different countries and regions. For example, drivers in the UK drive on the left side of the road, whereas drivers in China drive on the right side. Moreover, different countries use different types and symbols in traffic signs, and their meanings vary greatly. Therefore, no universal unmanned vehicle system exists, and the road laws and customs in different countries show great variations. An unmanned vehicle system also needs to be “localized.” If it is universal, then this technology’s cost and system complexity would be quite high and unrealistic.

Another obstacle comes from people’s “high expectations” toward machines. Humans can tolerate their own mistakes, but their tolerance for machine errors is extremely low. Under complex and changeable driving scenarios, unmanned vehicles will inevitably make mistakes; for example, sensors may not identify passersby when light conditions are unfavorable, and HD maps may not reflect road sections that have just been completed and opened to traffic. The fact is that the current technology and algorithms (e.g., theory of robotics or the method of artificial intelligence) are still far behind and are unable to meet the high expectations of the public for unmanned vehicles.

The cost of unmanned vehicles presents another challenge. After more than a 100 years of development, the automotive industry has reduced the cost of traditional cars to the extreme. People can buy the cars they need with only tens of thousands of dollars. Meanwhile, unmanned vehicle systems introduce additional costs, which cover new sensor equipment, computing equipment, software, etc. Take the most widely used LIDAR for unmanned vehicles as an example. The price of LIDAR that meets the L4 classification is generally greater than US\$100,000. The hardware cost of these sensors alone far exceeds the price of most vehicles, and the reduction of costs to achieve mass production is currently an important research topic for the commercialization of driverless vehicles.

Naturally, driverless technology faces many obstacles, such as the urgent need to improve driverless regulations, high R&D investment, safety issues, perception of complex scenarios, problems in artificial intelligence technology, and mass production scale. Autonomous driving technology remains

challenging at present. In other words, we need to conduct extensive technical research in the field of unmanned vehicles. Those who want to work in this field can dedicate much time and effort to make great progress.

1.2 Why do we need unmanned vehicles?

Unmanned vehicle technology can bring about social reform because high-level unmanned vehicles can fundamentally make people's travel habits and lifestyles increasingly intelligent. Research shows that unmanned vehicle technology can improve road traffic safety and alleviate urban traffic congestion. The application of unmanned vehicle technology to various fields leads to the emergence of new industrial chains and creates a large number of employment opportunities.

1.2.1 Improvement of road traffic safety

In 2015, approximately 35,092 and 260,000 people died in car accidents across the United States² and in China, respectively. In terms of the proportion of the driving population, one fatal accident occurs for every 88 million miles driven on average. The average probability of a fatal car accident caused by a human driver is 0.011% annually and 0.88% in a lifetime. At the same time, approximately 2.6 million people are injured in road traffic accidents every year, and billions of dollars are spent on auto repair costs (limited to deductibles). Reducing accidents by 25% could greatly limit people's expenses.

The following factors are the major causes of road traffic accidents:

- Drivers being distracted and losing concentration;
- Over speeding;
- Drunk driving;
- Reckless driving.

In unmanned vehicles, these problems do not apply as unmanned vehicle systems are not disturbed by users' actions, such as eating or texting. When passengers are in a hurry, autonomous driving vehicles do not exceed speed limits, and they bring passengers to their destinations in a very smooth and reasonable manner and in strict accordance with traffic laws. Drunk driving and reckless driving on highways are also impossible in autonomous driving. Various studies have shown that high-level unmanned vehicle systems can greatly improve the safety of road traffic and reduce the occurrence of traffic accidents.³

1.2.2 Alleviation of urban traffic congestion

Traffic congestion is a problem faced by many big cities. The widespread use of unmanned vehicles will ease the problem of urban traffic congestion to a large extent because it can completely overcome human factors and increase the efficiency of urban traffic operations. Driven by big data technology, future unmanned vehicles will be able to

- Quickly understand congestion situations and make timely line adjustments;
- Avoid unreasonable congestion caused by a large number of human factors, such as the practice of jumping the line;
- Apply the dynamic segmentation of reversible lanes.

Nonetheless, the alleviation of traffic congestion still depends on specific applications in different scenarios.

For example, in the case of traffic jams, roads become increasingly congested because vehicles jump the line. Meanwhile, unmanned vehicles line up according to certain rules and order, thereby greatly reducing congestion and improving people's travel efficiency. Driverless vehicles can also automatically adjust their routes according to real-time road conditions and safely deliver passengers to their destinations in the shortest possible time. Data show that the average commute time in the United States is 50 min. When autonomous driving becomes popular, people can have extra time to deal with other things as they will not be helplessly stuck on the road.

1.2.3 Improvement of travel efficiency

For the operation of manned travel services, we often encounter unfavorable situations, such as being refused by drivers, unavailability of car service, and low-quality drivers. In the operation of unmanned travel services, no passenger is refused service because drivers are not involved. As long as there is a dispatch order, unmanned vehicles will pick up and drop off passengers as required, and they will not refuse to carry passengers because of short travel distances. In addition, unmanned vehicles may prevent the occurrence of accidents when driving at night, such as those caused by fatigue driving, drunk driving, and vehicle robbery. In terms of driverless operation, in addition to supplementing energy, vehicles will be on standby for 24 h a day. People from all walks of life, including the elderly, weak, and sick; those with disabilities; and pregnant women can enjoy the safety and convenience brought by driverless travel services.

Given the absence of drivers, unmanned travel services will correspondingly reduce labor costs and thereby amortize travel spending. Collectively, these factors can greatly improve people's travel efficiency and change people's travel methods and lifestyles.

1.2.4 Lowering the threshold for drivers

In previous years, relevant departments imposed strict requirements on a driver's age and physical condition when applying for a driving license. For example, underaged and overaged people are not allowed to apply for a driver's license, and people with physical disabilities are required to undergo multiple evaluations. The emergence of driverless cars has greatly reduced the threshold for driving applications. In the fully autonomous driving stage, applying for a driver's license will not be necessary. People need not have sufficient driving or reversing skills when using unmanned vehicles. Driverless cars will thus be significant to traffic management departments and car users.

In the current technological situation, the original driverless technology is still likely to require car owners to have the necessary driverless supervision qualifications to ensure safety in extreme situations. This work contends that with the development of technology, unmanned vehicles will become increasingly mature and perfect, and we can finally realize the ultimate goal of completely unmanned vehicles. Moreover, traditional travel will bring about a new type of service or mobile living space rather than difficulties and inconveniences.

1.3 Basic framework of the unmanned vehicle system

The core of the unmanned vehicle system can be summarized into three parts: perception, planning, and control. The interaction of these three parts and their interaction with vehicle sensor hardware and environment are shown in Fig. 1.2.

Fig. 1.2 shows that the driverless software system is actually a hierarchical structure where the perception, planning, and control modules play different roles and influence each other. The detailed descriptions of the functions of these three layers are as follows.

Perception refers to the ability of unmanned systems to collect information from the environment and extract relevant knowledge. Environmental perception refers to the semantic classification of a scene by an unmanned system on the basis of its understanding of the environment,

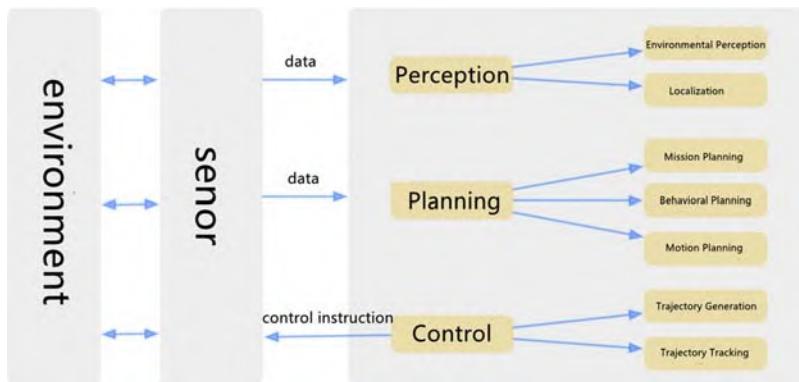


Figure 1.2 Basic framework of an unmanned vehicle system.

including the type of obstacles, road signs and markings, pedestrian–vehicle detection, traffic signals, and other data. Localization is the postprocessing of the perception results, and it helps unmanned vehicles to understand their position relative to the environment through the positioning function.

Planning refers to the process in which unmanned vehicles make decisions and plans to reach a destination. For unmanned vehicles, this process usually includes reaching the destination from the starting point, avoiding obstacles, and constantly optimizing the driving trajectory and behavior to ensure the safety and comfort of the vehicle. The planning layer is usually subdivided into three layers: mission planning, behavioral planning, and motion planning.⁴

Control refers to the ability of an unmanned vehicle to accurately perform planned actions and routes and to timely give the vehicle actuator the appropriate throttle, direction, brake signal, etc., to ensure that it can drive as expected.

1.3.1 Environmental perception

To ensure that the unmanned vehicle understands the surrounding environment, the environmental perception part of the unmanned vehicle system usually needs to obtain a large amount of environmental information, including the location, speed, and possible behavior of pedestrians and vehicles at the next moment, the driving area, and traffic rules. Unmanned vehicles usually obtain such information by fusing the data from LIDAR, cameras, millimeter-wave radars, and other sensors.⁵ In this section, we briefly illustrate the application of LIDAR and cameras in unmanned sensing systems.

Laser radar is a type of equipment that uses laser beams for detection and ranging. For example, the 64-line laser radar produced by Velodyne Company in the United States can send millions of laser pulses to the outside world every second and is internally equipped with a mechanical rotating structure. Laser radars can establish a three-dimensional map of the surrounding environment in real time. They usually rotate and scan the surrounding environment at a frequency of 10 Hz. The results of each scan are three-dimensional graphs composed of dense points, each with spatial coordinate (x, y, z) information. The scanning results are vast laser points known as point cloud data. The graph constructed by point cloud data is also called a point cloud graph. Fig. 1.3 shows a point cloud map drawn by a Velodyne VLP-32 c laser radar.

At present, laser radars are still one of the most important sensors in driverless systems because of their high reliability and accuracy. However, in real scenes, LIDAR is not always ideal. For example, in an open area, point clouds may be too sparse, or information may be lost because of the lack of feature points. For irregular objects, their feature patterns cannot be easily identified with LIDAR. In weather conditions with heavy rain and fog, the accuracy of LIDAR is also greatly affected.

Two steps are adopted to process laser point cloud data: segmentation and classification. Segmentation involves regrouping the discrete points in point cloud images by using a clustering algorithm to aggregate them into a



Figure 1.3 Point cloud map drawn by laser radar.

whole, while classification is distinguishing the categories to which these points belong, such as vehicle class or other obstacle class.

After the target segmentation of a point cloud is completed, the segmented target needs to be correctly classified. We can learn from the classification algorithms in machine learning, such as support vector machine (SVM), decision trees, k-means algorithm, and others, to classify clustering characteristics. As a result of the development of deep learning in recent years, the industry has been using convolutional neural networks (CNN) to classify three-dimensional point cloud clustering characteristics. However, regardless of whether the SVM method based on feature extraction or the CNN method based on the original point cloud is used, the low resolution of the laser radar point cloud itself reduces the reliability of point cloud-based classification of targets with sparse reflection points (e.g., pedestrians and bicycles). Therefore, in practice, we tend to integrate laser radar and camera sensors, use high-resolution cameras to classify targets, maximize the reliability of laser radars to detect and measure obstacles, and finally integrate the advantages of both sensors to complete environmental perception.

In a driverless system, we usually use visual sensors to complete road, vehicle, and traffic sign detection, recognition, and classification. Road detection includes lane detection, drivable area detection, vehicle detection, pedestrian detection, and traffic sign detection.

The detection of lane lines involves two aspects. The first is to identify the lane line. For a curved lane line, the curvature can be calculated to determine the control angle of the steering wheel. The second is to determine the deviation of the vehicle itself relative to the lane line, that is, where the unmanned vehicle itself is on the lane line. One method is to extract the features of some lanes, including edge features, by using a Sobel operator, which usually detects a lane line by calculating the gradient of the edge line, that is, the change rate of the edge pixel, and the color features of the lane line. Then, a polynomial is used to fit the pixels of the lane line. Finally, on the basis of the polynomial and the position of the current camera mounted on the vehicle, the curvature of the front lane line and the deviation position of the vehicle relative to the lane are determined.

For the detection of driving areas, the current method is to use deep learning neural networks to segment the pixels of scenes, that is, to complete the segmentation of the driving area in an image by training a deep neural network with pixel-level classification.

1.3.2 Localization

In the perception level of unmanned vehicles, the importance of localization is selfevident. Unmanned vehicles need to know their accurate position relative to the external environment. In a complex urban road driving scene, the accuracy error of positioning should not exceed 10 cm. If the localization deviation is too large, then vehicle tires can easily rub against the road teeth and scrape guardrails during the driving process. These effects will lead to safety problems, including the bursting of tires and serious traffic accidents. Although the automatic obstacle avoidance function will be used to ensure safety in the automatic driving of vehicles, it cannot guarantee successful obstacle avoidance. In some cases, due to blind spots, software failures, and other reasons, sensors cannot necessarily guarantee 100% detection of road obstacles, such as road teeth. Therefore, in the development of unmanned vehicle technology, the improvement of positioning accuracy is of great significance at the hardware and software levels.

At present, the most widely used unmanned vehicle localization methods include the fusion of GPS and inertial navigation system (INS) positioning methods. Generally, positioning accuracy may range between tens of meters and several centimeters, and the higher the accuracy is, the higher the costs of GPS and inertial navigation sensors are. The positioning method based on GPS/INS fusion cannot easily achieve high-precision positioning when GPS signals are weak or absent, such as in areas like underground parking lots and urban areas with high buildings. Therefore, the method can only be applied to unmanned positioning tasks in some scenes, such as open environments with good signals.

Map-aided localization algorithms are also widely used unmanned vehicle localization algorithms. Simultaneous localization and mapping (SLAM) is a representative of this type of algorithm. The goal of SLAM is to build a map and locate it at the same time. SLAM determines the position of the current vehicle and the position of the current observation target by using the environmental features observed by sensors, including vision sensors and laser radar. This process involves estimating the current position by using the prior probability distribution and current observation value. Common methods include the Bayesian filter, Kalman filter, extended Kalman filter, and particle filter. These methods are based on probability and statistical principles.

SLAM is a research hotspot in the field of robot localization. In the application process of low-speed scenarios, there are also many practical examples, including unmanned ferry cars in special areas, unmanned ground sweepers, and mechanical dogs made by Boston Dynamics. In fact, in the application of such special scenes, maps were not built with localization simultaneously but were constructed in advance through sensors, such as laser radar and visual cameras in the running environment. Positioning, path planning, and other operations are further performed on the basis of the constructed SLAM map.

After obtaining a part of a point cloud map, some “semantic” elements can be added to it by using programmed and manual processing methods, such as lane markings, traffic signal markings, and traffic rules of the current section. The map containing the semantic elements is an HD map that is often referred to in the field of unmanned vehicles. In actual positioning, the scanning data of 3D laser radar and HD maps constructed in advance are used to match the point cloud to determine the specific location of an unmanned vehicle on the map. This method is collectively called scan matching. The most common method of scan matching is the iterative closest point (ICP). The ICP method completes point cloud registration on the basis of the distance measurement of the current and target scans. Another common method for point cloud registration is normal distribution transform (NDT), which is an algorithm based on a point cloud feature histogram. This positioning method based on point cloud registration can also achieve positioning accuracy within 10 cm. NDT-based point cloud positioning method will be explained later in this book.

Although point cloud registration can provide high-precision global positioning of unmanned vehicles relative to the map, the method relies heavily on building HD maps in advance. In addition, the construction of HD maps is costly, thereby increasing the computational cost of point cloud matching as Velodyne-32c generates up to 1.2 million point data per second. Moreover, the real-time requirements for calculation and the control of vehicles in the process of high-speed driving increase. Therefore, in high-speed unmanned scenes, the cost of using the point cloud matching method is relatively high.

1.3.3 Mission planning

The hierarchical structure design of a driverless planning system originated from the DAPRA City Challenge held in the United States in 2007. According to the published papers of the participating teams, most of the

participating teams divided the planning module of the driverless vehicle into three layers of structure design: task planning, behavior planning, and action planning. Task planning, usually called path planning or route planning, corresponds to the relatively top-level and global path planning, such as the path selection from the starting point to the end point.

We can also simplify the road system into a directed graph network, representing the connection between roads, traffic rules, road width, and other information. In essence, it is the “semantic” part of the HD map mentioned in the discussion of positioning. This directed graph is also called the route network graph, as shown in Fig. 1.4.

As each directed edge in the road network diagram is weighted, the path planning problem of unmanned vehicles can be transformed into the process of selecting the optimal (i.e., the minimum cost) path on the basis of a certain method to make the vehicle reach a certain destination, usually from A to B, in the road network diagram. Thus, the planning problem is transformed into a directed graph search problem. Traditional algorithms, such as Dijkstra’s algorithm, A* algorithm, and D* algorithm, are mainly used to calculate the optimal path search of discrete graphs and are widely used in the scene of searching the paths with the lowest cost in road network graphs.



Figure 1.4 Simple road network diagram.

1.3.4 Behavior planning

Behavior planning is sometimes called the decision-making module, whose main task is to make decisions and actions that the next unmanned vehicle needs to perform according to the objectives of task planning and the perception of the current environment, including factors such as the location and behavior of other vehicles and pedestrians and current traffic rules. The role of this module can be understood as the decision-making system of the driver of the vehicle. According to the objectives and the current traffic situation, the driver decides whether to follow or overtake a vehicle, whether to stop or bypass pedestrians, etc.

One method of behavior planning is to use a complex finite state machine (FSM) containing a large number of action phrases. The FSM starts from a simple starting state, jumps to different action states according to different driving scenes, and transmits the action to be performed to the lower action planning layer. Fig. 1.5 demonstrates a simple FSM.

As shown in Fig. 1.5, each state is a decision-making process for vehicle action. Some judgment conditions between states and certain states, such as the trace state and waiting state, can be selfcirculated. Although the FSM is the mainstream behavior decision-making method for unmanned vehicles, it still suffers from many limitations, including the following:

- a) To achieve complex behavior decision making, a large number of effective states need to be designed manually;
- a) A vehicle may encounter a state that the FSM does not consider, in which case the expansion of the state machine also becomes a problem.

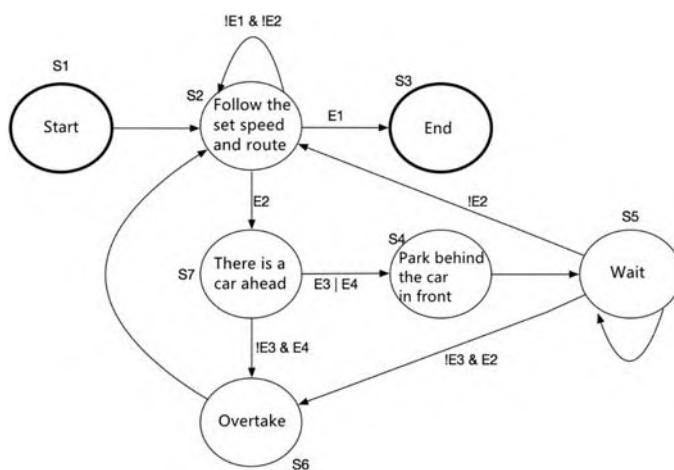


Figure 1.5 Simple finite state machine.

In addition, if the FSM does not design deadlock protection, the vehicle may even fall into a deadlock⁶ state.

1.3.5 Motion planning

The process of planning a series of executive actions to achieve a certain purpose, such as obstacle avoidance, is called action planning. Generally, two factors can be used to measure the performance of action planning algorithms: computational efficiency and completeness. The notion of computational efficiency is the computational efficiency of completing a single action plan that largely depends on the configuration space. If an action planning algorithm can return a solution in finite time under solvable conditions and not return a when it does not have a solution, then the action planning algorithm is thought to be complete.

After describing the concept of configuration space, the action planning of the unmanned vehicle faces the following scenario: with a start configuration and goal configuration with several constraints, a series of actions are found in the configuration space to reach the target configuration. The execution result of these actions under the constraints is to transfer the unmanned vehicle from the initial configuration to the target configuration. In the application scenario of an unmanned vehicle, the initial configuration is usually the current state of the unmanned vehicle (the current position, velocity, angular velocity, etc.), and the target configuration comes from the upper layer of action planning, that is, the behavior planning layer. Meanwhile, the constraint condition is the kinematic limitation of the vehicle (the maximum rotation angle, maximum acceleration, etc.).

Obviously, the calculation of action planning in the high-dimensional configuration space is huge. To ensure the integrity of the planning algorithm, we need to search almost all possible paths, and this action leads to a “dimension disaster” in continuous action planning. At present, solving this problem in action planning requires the conversion of the continuous space model into a discrete model. Specific methods can be summarized into two categories: combinatorial planning and sampling-based planning.

The combination method of motion planning, as shown in Fig. 1.6, finds the path through a continuous configuration space without using an approximation value. Given this attribute, the method can be called a precise algorithm. The combination method finds a complete solution by establishing a discrete representation of the planning problem. For example, CMU’s unmanned vehicle (BOSS) uses such action planning algorithms in

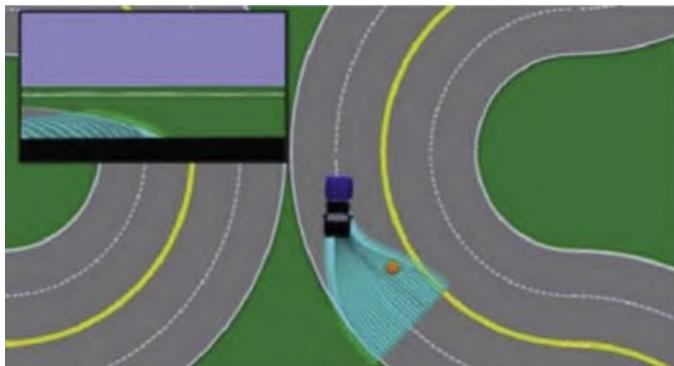


Figure 1.6 Combination method in motion planning.

the DARPA Urban Challenge. It first uses path planners to generate alternative paths and target points (Fig. 1.6) and then selects the optimal path by optimizing the algorithm. Another discretization method is the grid decomposition approach, which uses a discrete graph search algorithm, such as the A* algorithm,⁷ to find an optimization path after gridding the configuration space.⁷

Sampling-based methods are widely used because of their probability integrity. The most common algorithms are probabilistic roadmaps, rapidly exploring random trees, and fast-marching trees. In the application of unmanned vehicles, the state sampling method needs to consider the control constraints of two states, and a method that can effectively determine whether the sampling state and the parent state are achievable should be employed. In the following part, we will introduce the trajectory optimization method based on the Frenet coordinate system, which is a sampling-based motion planning algorithm.

1.3.6 Control system

As the bottom layer of an unmanned vehicle system, the control system layer aims to realize the planned actions from the vehicle control level. Thus, the evaluation index of the control module is the accuracy of the control. The control system has measurement feedback. The controller outputs the corresponding control action by comparing the measurement and the expected expectation of the vehicle. This process is called feedback control.

Feedback control is widely used in the field of automation control, and the most typical feedback controller is the proportional–integral–derivative

(PID) controller. The control principle of the PID controller is based on a simple deviation signal and consists of three components: proportion, integral, and derivative. The PID controller is simple to implement and has stable performance, and it is still the most widely used controller in the industry. However, as a pure feedback controller, the PID controller has certain problems in the control of unmanned vehicles, especially in the process of high-speed movement. The PID controller is based on the current error feedback. Thus, the control system will have very large delays in a high-speed movement scenario because of delays in the braking mechanism. However, as the PID controller has no system model, it cannot model the delay problem. To solve this problem, we introduce the control method based on model prediction.

Model predictive control (MPC) is a method that uses a vehicle motion model to predict the movement of a period of time in the future by continuously optimizing the control parameters to fit this series of movements. The time period of model prediction is usually short.⁸ MPC consists of four parts:

- Prediction model: This part is based on the current state and control input to predict the state of a future period; in an unmanned vehicle system, it usually refers to the vehicle kinematics/dynamics model.
- Feedback correction: It involves the process of applying feedback correction to the model, thus making predictive control capable of resisting disturbance and overcoming system uncertainty.
- Rolling optimization: It involves rolling optimization of the control sequence to obtain the prediction sequence that is closest to the reference trajectory.
- Reference trajectory: It refers to the set trajectory.

Fig. 1.7 shows the basic structure of MPC. As MPC is optimized on the basis of the motion model, the control delay problem in PID control can be considered when establishing the model. Thus, MPC has a high application value in unmanned vehicle control.

The other two problems are trajectory generation and trajectory tracking. Trajectory generation is to find a set of control inputs $u(t)$, which make the expected output result in the target state of the trajectory $x(t)$. The kinematics/dynamics constraints of a vehicle are the constraints of the whole trajectory generation. When a trajectory $x(t)$ does not have the corresponding control input $u(t)$ to meet the vehicle dynamics constraints, we call this trajectory unreachable. At present, the trajectory generation

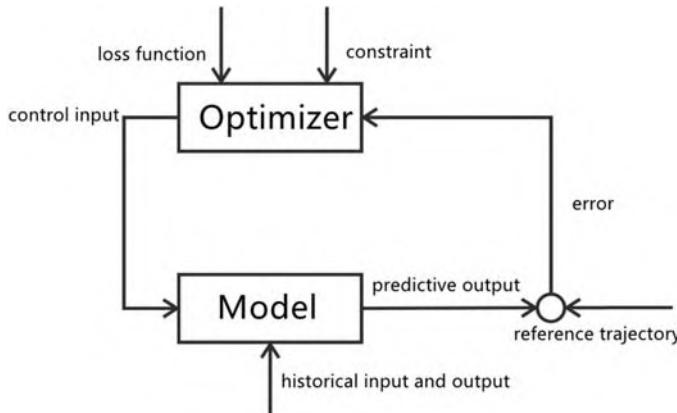


Figure 1.7 Basic structure of model predictive control.

methods used in the field of unmanned vehicles are usually based on vehicle dynamics models.

Trajectory tracking is mainly divided into two types: geometric path tracking and model-based tracking. Model-based methods usually use the kinematics and dynamics model of a vehicle. The kinematics model is effective in scenes involving low speed. For controllers based on dynamics models, they achieve a good tracking effect in high-speed scenarios, but they perform poorly in cases involving large acceleration and path curvature. Geometric path tracking uses simple geometric relations to derive steering control rules. Similar to the vector tracking method, geometric path tracking uses the look ahead distance function to measure the deviation in front of the vehicle and calculates the complexity from a simple arc with the complex geometric theorem.

The most commonly used geometric path tracking algorithms, such as the pure pursuit algorithm, are discussed herein. The pure tracking algorithm is simple and widely used in trajectory tracking. The input of the pure tracking algorithm is a series of waypoints, which make the vehicle move from the current position to the target position by calculating a curve. The key to the pure tracking algorithm is to select the target point at a certain distance (forward-looking distance) in front of the vehicle in the path so that the vehicle has the ability to track the target point. With the movement of the vehicle itself, the forward-looking target point also moves such that the vehicle drives along with a series of trajectory points. This tracking method is somewhat like the scene of human drivers driving because people

always stare at a distance in front of the road to control the direction when driving so that the vehicle reaches the position in front of the road.

1.3.7 Summary

This section outlines the basic architecture of the autonomous driving system and explains the three-tier structure of the driverless software system: perception, planning, and control. To some extent, unmanned vehicles in this hierarchical system can be seen as “manned robots” that perform a series of task planning and control movements by sensing the environment and positioning. In recent years, because of the breakthroughs in deep learning, perception technology based on visual deep learning has also played an increasingly important role in environmental perception. With the help of artificial intelligence, we are no longer limited to the perception of obstacles. We can now understand what the obstacles are, recognize scenes, and even predict the behavior of a scene and a target. The content of machine learning and deep learning will be introduced in [Chapters 5 and 6](#).

In an actual unmanned vehicle sensing system, it is usually necessary to integrate multiple measurements, such as laser radars, cameras, and millimeter-wave radars. This process involves fusion algorithms, such as the Kalman filtering algorithm, extended Kalman filtering algorithm, and coordinate conversion concepts of laser radars and cameras. These algorithms and concepts will be introduced in detail in [Chapter 4](#) of this book.

There are many localization methods for unmanned vehicles and robots. At present, one of the mainstream methods is to use GPS with the INS integration method. Another method is based on the LIDAR point cloud scan matching method. These methods will be introduced in detail in [Chapter 3](#).

The planning module is also divided into three layers: task planning (also known as path planning), behavior planning, and action planning. This book will introduce the task planning method based on the discrete path search algorithm. In behavior planning, we will focus on the application of the FSM in behavior decision making. In the action planning module, the optimization trajectory planning method based on the Frenet coordinate system will be introduced in detail in [Chapter 8](#).

As for the control module, the driverless system often uses the control method based on model prediction. To understand the MPC algorithm, we will introduce the PID controller in detail as the basis of the basic feedback control and then illustrate two simple vehicle models: a kinematics bicycle

model and a dynamics bicycle model. The details of the control module are detailed in [Chapter 9](#) of this book.

Although it is a common practice in the industry to understand an unmanned vehicle as a robot in another sense and use the thinking developed by robots to deal with the unmanned vehicle system, some cases use artificial intelligence to complete unmanned technology. In particular, end-to-end (end-to-end) unmanned vehicles based on deep learning and driving agents based on reinforcement learning are current research hot-spots. This book will introduce these two methods in [Chapter 7](#) and Chapter 10.

1.4 Development environment configuration

This book provides a large number of runnable codes for readers to understand the concept after the actual operation. The implementation of these codes will depend on a certain environment. Herein, we will guide users in the installation and operation of the codes required for a variety of environments and the recommended system for Ubuntu-16.04.

1.4.1 Simple environment installation

Ubuntu-16.04 itself contains the Python 2.7 environment. Specifically, Python 2.7 is used when dealing with a large number of codes, whereas C++ is used in cases involving a small number of codes. Pip is a modern universal Python package management tool. The installation commands are as follows:

```
$ sudo apt-get install python-pip
```

Install scipy package and numpy package:

```
$ sudo apt-get install python-scipy python-numpy
```

Use pip to install packages in this article:

```
$ sudo pip install scikit-image scikit-learn sympy jupyter numdifftools  
plotly pandas
```

Install moviepy package and then configure ffmpeg package:

```
$ sudo pip install moviepy  
$ cd into the chapter_6/3rdparty/  
$ cp ffmpeg-linux64-v3.3.1 ~/.imageio/ffmpeg/
```

Install matplotlib-2.0.2 package:

```
$ sudo pip install matplotlib==2.0.2  
$ sudo pip install seaborn
```

Install keras-2.0.0 package:

```
$ sudo pip install -U --pre keras==2.0.0
```

Install tensorflow-1.6.0 package:

```
$ sudo pip install tensorflow==1.6.0
```

The TensorFlow installation version here is the CPU version. If you want to install the GPU version of TensorFlow, refer to the following link: http://wiki.jikexueyuan.com/project/tensorflow-zh/get_started/os_setup.html

1.4.2 Install the robot operating system (ROS)

As the system is Ubuntu-16.0.4, the ROS version we want to install is Kinetic, t. The specific installation steps are as follows:

1. Install the source.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release  
-sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

If the download is slow, then consider replacing other sources. The source address can be <http://wiki.ros.org/ROS/Installation/UbuntuMirrors>.

2. Set the key.

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyserver.net:80  
--recv-key 421C365BD9FF1F717815A3895523BAEAB01FA116
```

3. Update.

```
$ sudo apt-get update
```

4. Install ROS software.

```
$ sudo apt-get install ros-kinetic-desktop
```

5. Install rosdep package.

```
$ sudo rosdep init  
$ rosdep update
```

6. Set the environment.

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc  
$ source ~/.bashrc
```

1.4.3 Install OpenCV**1. Run the installation.**

```
$ sudo apt-get install build-essential ke git libgtk2.0-dev pkg-config  
libavcodec-dev libavformat-dev libswscale-dev python-dev python-numpy  
libtbb2 libtbb-dev libjpeg-dev libpng-dev libtiff-dev libjasper-dev  
libdc1394-22-dev
```

2. Download the installation package from the download page and unzip it. The official link is as follows: <https://opencv.org/>.

3. Install from the source:

- Create a temporary area;

```
$ cd ~/opencv  
$ mkdir build  
$ cd build
```

- Configure;

```
$ cmake -D CMAKE_BUILD_TYPE=Release -D  
CMAKE_INSTALL_PREFIX=/usr/local ..
```

- Create;

```
$ make -j7
```

- Install.

```
$ sudo make install
```

References

1. *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. https://www.sae.org/standards/content/j3016_201401/.
2. *Traffic Safety Facts 2015*. National Highway Traffic Safety Administration; 2015.
3. Pan F, Qi R, Zhang X, et al. The research survey and development prospect of automated driving. *Technol Innov Appl*. 2017;(2):27–28. ISSN 2095-2945.
4. Pendleton S, Andersen H, Du X, et al. Perception, planning, control, and coordination for autonomous vehicle. *Machines*. 2017;5(1):6.
5. Wang S, Dai X, Xu N, et al. A survey of perception for autonomous vehicle. *J Changchun Univ Sci Technol*. 2017;40(1):1–6. ISSN 1672-9870.
6. Furda A, Vlacic L. Towards increased road safety: real-time decision making for driverless city vehicles. *IEEE Int Conf Syst Man Cybern*. 2009:2421–2426.
7. Urmson C, Anhalt J, Bagnell D, et al. Autonomous driving in urban environments: boss and the urban challenge. *J Field Robot*. 2008;25(8):425–466.
8. Camacho DEF, Bordons DC. *Model Predictive Control*. Springer London; 2007:575–615.

This page intentionally left blank

CHAPTER 2

Introduction to robot operating system

Zebang Shen¹, Xiaowei Xu², Peng Zhi² and Rui Zhao²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

2.1 ROS introduction	28
2.1.1 Brief introduction to ROS	28
2.1.1.1 <i>What is ROS?</i>	28
2.1.1.2 <i>History of ROS</i>	28
2.1.1.3 <i>Features of ROS</i>	29
2.1.2 Concept of ROS	29
2.1.2.1 <i>Master</i>	29
2.1.2.2 <i>Node</i>	30
2.1.2.3 <i>Topic</i>	30
2.1.2.4 <i>Message</i>	32
2.1.3 Catkin create system	32
2.1.4 Project organization structure in ROS	33
2.1.4.1 <i>CMakeLists.txt</i>	34
2.1.5 Practice based on husky simulator	35
2.1.6 Basic ROS programming	39
2.1.6.1 <i>ROS C++ client library (roscpp)</i>	39
2.1.6.2 <i>Write simple publish and subscribe code</i>	42
2.1.6.3 <i>Parameter services in ROS</i>	44
2.1.6.4 <i>Small case based on husky robot</i>	45
2.1.7 ROS services	53
2.1.8 ROS action	57
2.1.9 Common tools in ROS	57
2.1.9.1 <i>Rviz</i>	58
2.1.9.2 <i>rqt</i>	58
2.1.9.3 <i>TF coordinate conversion system</i>	59
2.1.9.4 <i>URDF and SDF</i>	61
References	62

2.1 ROS introduction

The robot operating system (ROS)¹ is the most widely used open-source robot software platform, and its birth has dramatically improved the efficiency of robot system development.² Part of the software implementation of unmanned driving systems still relies on the ROS platform. For the development of high-speed unmanned driving, the real-time performance of ROS is insufficient. Nevertheless, because of the powerful communication function at the bottom of the system and the joint maintenance by excellent engineers from all over the world, ROS is still the best choice among open-source frameworks for the implementation of underlying driverless systems. Thus, many driverless vehicle prototypes, such as Baidu Apollo, Autoware, and Udacity, are based on ROS. For example, Baidu Apollo adds a real-time framework on the basis of the ROS platform to meet the real-time control requirements of high-speed autonomous driving. At the same time, ROS2 has begun to actively explore and develop real-time system communication functions further to meet the development needs of high-speed autonomous driving. This chapter provides an introduction to ROS.

2.1.1 Brief introduction to ROS

2.1.1.1 What is ROS?

The ROS concept includes the following³:

- Communication platform: ROS provides a publish—subscribe communication framework for building distributed computing systems easily and quickly.⁴
- Tools: ROS provides massive tools that combine visualization and debugging to configure, start, selftest, debug, visualize, simulate, login, test, and terminate distributed computing systems.
- Abilities: ROS has the functions of control, planning, forecasting, positioning operation, and so on.
- Platform support: The growth of ROS relies on community support. In particular, the website wiki.ros.org focuses on ROS compatibility and supports documentation, thereby providing a one-stop solution to help users quickly search and learn thousands of ROS packages from developers worldwide.

2.1.1.2 History of ROS

ROS originated from a 2007 collaboration between the STAIR Program from the Stanford Artificial Intelligence Lab and the Personal Robots Program from the robotics company Willow Garage, which has been promoting

ROS since 2008. ROS has been used in many schools, companies, and other research institutions, and it continues to provide fast methods and standards for robot programming.

2.1.1.3 Features of ROS

- Point-to-point design: The point-to-point design of ROS, along with mechanisms such as service and node managers, enables good distributed network computing topologies that can accommodate the challenges of multiple robots.
- Distributed design: Programs can run and communicate in multiple networked computers.
- Multilanguage support: ROS now supports multiple languages, such as C++, Python, Octave, and LISP, as well as various interface implementations in other languages. Multiple languages can be freely mixed and matched by language-independent message processing.
- Light weight: ROS encourages all drivers and algorithms to be engaged into separate libraries that do not depend on ROS. The system established by ROS features modularization, and the codes in each module can be compiled separately using the CMake tool, which helps realize the concept of simplification.
- Free and open source: Most of the source codes in ROS are publicly available.

2.1.2 Concept of ROS

The core ROS concepts include the following⁵:

- Master
- Node
- Topic
- Message

2.1.2.1 Master

The ROS master is used to manage the communication between ROS nodes; it provides a registration list of nodes and enables the search for other computing resource information through the remote procedure call protocol.⁶ Without the master node, other nodes will not be able to communicate with target nodes to exchange messages or invoke services. However, the master increases the risk of the system in some aspects. For example, all nodes manage and invoke services through the master. Once the master dies, all other nodes will be affected. Therefore, a new message communication architecture, that is,

the data distribution service,⁷ is proposed in the ROS2 version, which is used to solve such problems and realize real-time transmission.

Code list 2.2.1 Run master command

```
roscore
```

2.1.2.2 Node

A node, known as a “software module,” is an independently compiled process for computing tasks. Generally, ROS makes the program modular so that it can grow in scale. A system typically consists of many nodes. We use a “node” to visualize the system when ROS is working. When many nodes are running simultaneously, we can easily plot point-to-point communication as a graph, in which processes are nodes and point-to-point connections are midline connections.

Code list 2.2.2 Run node

```
rosrun package_name node_name //start by using package and node name
```

Code list 2.2.3 Show the list of active nodes

```
rosnode list //list the currently active nodes
```

Code list 2.2.4 Retrieve node information

```
rosnode info node_name //list the information of node
rosnode kill node_name //kill the intended running node
rosnode cleanup // if a node crashes but does not run, you can run this
command to clear the node registration information. This command is useful
to clean up the node running environment when a node crashes
rosnode machine-name // list the node information running on the
specified computer, or list the machine name
```

2.1.2.3 Topic

In ROS, messages are delivered on a publish—subscribe pattern. One node can publish messages on a given topic, and another subscribes to specific

types of data for a topic. Multiple nodes may subscribe to messages on the same topic at the same time. In general, publishers and subscribers do not have to know about one another's existence. Imagine posting topics in a forum. After the post is made, each participant in the discussion is a node, and all comments are posted under the topic. If someone creates a post, a node posts a message to the topic. Someone can quote and reply to the message and write their own opinion, while the other person can receive a reply notification, which is a notification subscription for the other person. Publication and subscription are completely separate object operations.

Code list 2.2.5 View active topics

```
rostopic list -v //show the detail information of topic
```

Code list 2.2.6 Subscription and printing of topic content

```
rostopic echo /topic_name //show the live content of a topic

//Output the topic data in bagfile to the file in the specified CSV format,
and use excel drawing function to draw the track, route and so on to improve
the degree of data visualization. Especially suitable for paths, GPS tracks,
etc.

rostopic echo -b bagfile.bag -p /topic_name > data.csv
```

As shown in Fig. 2.1, the CSV file output by topic can be used to complete the complex track reproduction function with a simple tool like Excel, which is very practical.

Code list 2.2.7 View the topic type

```
rostopic info /topic_name

rostopic hz /topic_name //show the topic publishing frequency, which is
useful to check whether sensor data is being published as expected

rostopic pub /topic type args //publishing data to a given topic is useful
when debugging data
```

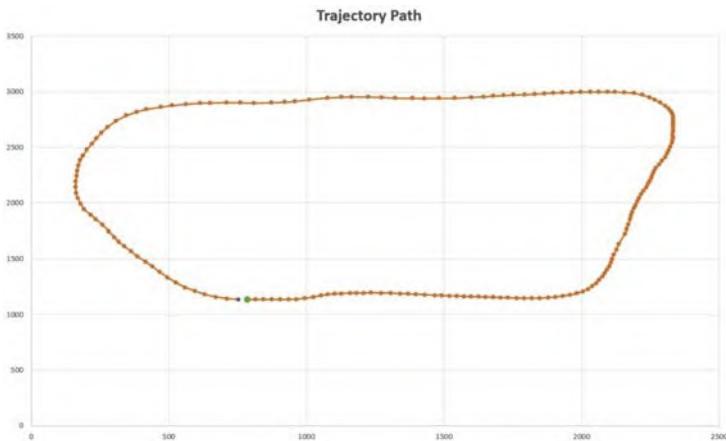


Figure 2.1 Track reproduction by using ROS topic echo.

2.1.2.4 Message

The message is a data structure used to define topic types. It supports a series of data structures, such as integers, floating points, and Boolean strings. The file format is *.msg.

Code list 2.2.8 Show the topic information

```
rostopic type /topic_name //view the type of message in topic
rosmsg info message_type //further view the message
rosmsg list //list all messages
rosmsg show message_name //show the field definitions in message
```

2.1.3 Catkin create system

Catkin is a compilation and building system for ROS that generates executables, libraries, and interfaces.

Code list 2.3.1 Enter the workspace

```
cd ~/catkin_ws
```

Code list 2.3.2 Create a package

```
catkin_create_pkg package_name depend1 depend2 depend3 //depend is the
name of dependent libraries
```

Whenever a new package is created, the source is executed in the new environment after compilation.

Code list 2.3.3 Update the environment

```
source devel/setup.bash
```

Catkin_make is a command line tool for compiling packages in the catkin environment.

The workspace created by catkin has three folders:

- Src: A folder used to store the source code, mainly for the operator to code;
- Build: Used to store cache files and intermediate files when compiling;
- Devel: The development directory where some target build files are stored before being installed.

Detailed instructions can be viewed at <http://wiki.ros.org/catkin/workspaces>.

2.1.4 Project organization structure in ROS

ROS software is organized using packages, which typically contain the following:

- /src: source code;
- /msg: defines various message files;
- /srv: defines various service files;
- /launch: contains the launch file used to launch the node;
- /config: contains the configuration file and used to load dynamic configuration parameters;
- /test: ROS test file;
- /include/_package_name_: C++, including the header file;
- /doc: contains the document file;
- package.xml: packages XML compilation, operation, copyright, and other information;
- CMakeLists.txt: CMakefile build file;

package.xml: This file defines the package properties, including the following:

- package name;
- version number;
- author;
- protocol;
- dependencies on other packages;
- ...

Code list 2.4.1 Simple package information

```
<?xml version="1.0"?>

<package format="2">

<name>ros_practice</name>

<version>0.0.1</version>

<description>The ros_practice package</description>

<maintainer email="your-email@gmail.com">Adam</maintainer>

<license>MIT</license>

<buildtool_depend>catkin</buildtool_depend>

<build_depend>roscpp</build_depend>

<build_depend>sensor_msgs</build_depend>

<build_export_depend>roscpp</build_export_depend>

<build_export_depend>sensor_msgs</build_export_depend>

<exec_depend>roscpp</exec_depend>

<exec_depend>sensor_msgs</exec_depend>

</package>
```

In this example, we use the client library of roscpp with a sensor_msgs message.

2.1.4.1 CMakeLists.txt

The CMakeLists.txt file is the input to the CMake build system. Herein, we do not explain in detail the writing of CMake given its complexity, but we provide the commonly used CMake syntax.

- *cmake_minimum_required*: minimum version of CMake required
- *project()*: package name

- *find_package()*: finds the build required by other CMake/Catkin packages
- *add_message_files() add_service_files() add_action_files()*: generates message/service/action
- *generate_messages()*: call message generation
- *catkin_package()*: specify the build information for the package
- *add_library()/add_executable()/target_link_libraries()*: library for building executable codes and target link libraries
- *install()*: installation rule

2.1.5 Practice based on husky simulator

Husky is a mid-sized mobile wheeled and outdoor unmanned ground vehicle development platform produced by Clearpath. It is equipped with a light detection and ranging scanner, a global positioning system, an inertial measurement unit, and other sensors and is fully compatible with ROS.⁸ The physical object is shown in Fig. 2.2.

We use the Husky simulator to practice ROS programming. Husky is similar to the driverless vehicle under study. This simulator is a tool based on the physical simulation environment of Gazebo.⁹ Gazebo is a ROS robot simulation platform software based on the ODE physics engine. It can simulate many physical characteristics of robots and environments. For the Husky simulator installation, please refer to http://wiki.ros.org/husky_gazebo/Tutorials/Simulating%20Husky.

Note that if ROS Kinetic has problems with the apt installation, please refer to <https://answers.ros.org/question/256756/how-to-install-husky-simulator-in-kinetic/>.



Figure 2.2 Husky in action.

Code list 2.5.1 Starts the Husky emulator using roslaunch to start an empty emulated environment

```
roslaunch husky_gazabo husky_empty_world.launch
```

Roslaunch is the mechanism for launching multiple nodes in ROS. Launch files are ROS provided files that can run multiple nodes simultaneously. They are written in a special XML format that ends with `.launch`.

We adopt `rqt_graph` to check the running node and topic, as shown in Fig. 2.3.

We find that the/gazebo node subscribes to a/CMD_vel topic.

```
rostopic info /husky_velocity_controller/cmd_vel
```

Code list 2.5.2 Use rostopic info /husky_velocity_controller/cmd_vel to obtain the information

```
Type: geometry_msgs/Twist
```

```
Publishers:
```

```
* /twist_mux (http://adam: 40181/)
```

```
Subscribers:
```

```
* /gazebo (http://adam: 35678/)
```

This control command is a `geometry_msgs/Twist` message. In ROS, we use the `geometry_msgs/Twist` message type to issue motion commands. The control command is utilized by the basic controller node. The controller node subscribes to the `/cmd_vel` topic whose full name is “command velocity” and converts the movement command (Twist message) into a motor control signal through the PID control algorithm.



Figure 2.3 Using `rqt_graph` to view running nodes and topics after running the emulator.

```
rosmsg show geometry_msgs/Twist
```

Code list 2.5.3 Message format using rosmsg show *geometry_Msgs/Twist*

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

“Linear” denotes linear velocity (unit: m/s), and “angular” represents angular velocity (unit: rad/s).

Herein, the car is driving at a speed of 4 m/s and an angular speed of 0.5 rad/s by sending a Twist message in the command line to/husky_vvvelocity_controller/cmd_vel.

Code list 2.5.4 Driver instruction

```
rostopic pub -r 10 /husky_velocity_controller/cmd_vel
geometry_msgs/Twist '{linear: {x: 4, y: 0, z: 0}, angular: {x: 0, y: 0, z:
0.5}}'
```

We can also control the car by sending a message directly to the/cmd_vel topic (rostopic info/cmd_vel to see which nodes subscribe to it).

Code list 2.5.5 Send messages to /cmd_vel theme

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 4, y: 0,
z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
```

Then, we use the handle function to control the Husky robot. We use the teleop_TWIST_joy package to convert the handle data into Twist instructions and publish Twist to the/cmd_vel theme.

Code list 2.5.6 Launches the launch file under *teleop_twist_joy*

```
roslaunch teleop_twist_joy teleop.launch
```

Code list 2.5.7 The following information is printed on the terminal after launching

```
PARAMETERS
* /joy_node/autorepeat_rate: 20
* /joy_node/deadzone: 0.3
* /joy_node/dev: /dev/input/js0
* /rosdistro: kinetic
* /rosversion: 1.12.12
* /teleop_twist_joy/axis_angular: 0
* /teleop_twist_joy/axis_linear: 1
* /teleop_twist_joy/enable_button: 8
* /teleop_twist_joy/enable_turbo_button: 10
* /teleop_twist_joy/scale_angular: 0.4
* /teleop_twist_joy/scale_linear: 0.7
* /teleop_twist_joy/scale_linear_turbo: 1.5

NODES
joy_node (joy/joy_node)
teleop_twist_joy (teleop_twist_joy/teleop_node)
```

The above nodes indicate that joy_node and teleop_twist_joy have been started in the joy and teleop_TWIST_joy packages, respectively. The above code is the parameter information of the launch file, with a focus on the following three parameters.

Code list 2.5.8 Launch the file with three parameters

```
* /teleop_twist_joy/axis_angular: 0
* /teleop_twist_joy/axis_linear: 1
* /teleop_twist_joy/enable_button: 8
```

The enable_button indicates that you can press this button (no. Eight button, corresponding to the back button on the handle) for control. The first two buttons, respectively, represent the angular speed and linear speed of the axis (corresponding to the left rocker on the handle).

Note: Husky, as a wheeled robot, can only control the speed in the x-direction (forward) and the angular speed in the z-direction (Yaw). You can start the rqt_graph command to view the directed graph (Fig. 2.4) of all current nodes and topics:

```
rosrun rqt_graph rqt_graph
```

The ROS operation instructions in the command line have been described. Next, we discuss the ROS programming practices.

2.1.6 Basic ROS programming

The previous section discussed the core concepts of ROS and the basic operations on the command line. This section will complete a simple ROS package through the Husky simulator, describe the syntax of CMakeList files under ROS through examples, and use Rviz to visualize the results of laser scanning.

ROS programs can usually be coded in multiple languages. As C++ is relatively close to the practice of unmanned vehicle engineering, this part starts from C++ to learn ROS programming.

2.1.6.1 ROS C++ client library (roscpp)

Here is a “Hello World” program written with the ROS C++ library:



Figure 2.4 rqt_graph at the present point.

Code list 2.6.1 “Hello World” program written using the ROS C++ library

```
#include <ros/ros.h>

int main(int argc, char** argv)
{
    ros::init(argc, argv, "hello_world"); //initialize node

    ros::NodeHandle nodeHandle; //define nodehandler

    ros::Rate loopRate(10); //ros frequency control

    unsigned int count = 0;

    while (ros::ok()) {
        ROS_INFO_STREAM("Hello World " << count); //print information

        ros::spinOnce();

        loopRate.sleep();

        count++;
    }

    return 0;
}
```

The meaning of the program is as follows:

Code list 2.6.2 Introduce ROS header files

```
#include <ros/ros.h>
```

Code list 2.6.3 Initialize the ROS node

```
ros::init(argc, argv, "hello_world");
```

This method should be called first by each ROS node, which is named hello_world.

Code list 2.6.4 Create a handle for the process node

```
ros::NodeHandle nodeHandle;
```

The first created NodeHandle initializes the node, and the last destroyed NodeHandle frees all resources occupied by the node.

Code list 2.6.5 Specify the frequency of the loop

```
ros::Rate loopRate(10);
```

ros:: Rate is a helper class for cycling at a specified frequency; it is defined as 10 Hz in this case.

Code list 2.6.6 Loops

```
while(ros::ok()){...}
```

This statement is executed cyclically at the frequency previously specified in the Rate object, where *ros:: ok()* terminates the loop by returning false when the ROS node finishes running. *ros:: ok()* returns false in the following cases:

- SIGINT is triggered (Ctrl-C);
- It is kicked out of the ROS network by another node of the same name;
- *ros: shutdown()* is called by another part of the program;
- All *ros: NodeHandles* in the node have been destroyed.

In the body of the loop, there is a *ros:: spinOnce()* statement, which means that when we subscribe to a topic, executing this statement implies that only one callback is processed. This topic is discussed further herein.

2.1.6.1.1 Node handle

Four types of syntaxes are available for defining node handles:

- Default public handle: `nodeHandle = ros::NodeHandle();`
- Private handle: `nodeHandle = ros::NodeHandle("~");`
- Namespace handle: `nodeHandle = ros::NodeHandle("adam");`
- Global handle (not recommended): `nodeHandle = ros::NodeHandle("/");`

When the nodes use these four definition handles to subscribe to a topic, they subscribe to the following topics:

- */namespace/topic*
- */namespace/node_space/topic*
- */namespace/adam/topic*
- */topic*

2.1.6.1.2 ROS logging method

The direct use of the C++ standard input—output logging syntax is generally discouraged in ROS nodes. Instead, *ROS_INFO* is widely used as it automatically sends information to the command line, log files, and /*rosout* topics.

Other logging types include *ROS_WARN* and *ROS_ERROR*. ROS also provides *printf-style* and *stream-style* methods using the following:

Code list 2.6.7 Use of printf-style and stream-style methods

```
ROS_INFO("Hello World %d", count);

ROS_INFO_STREAM("Hello World " << count);
```

If *ROS_INFO* is used, but it is not displayed, then you should generally check whether the output tag of the node in the launch file is set to *screen*, in addition to checking the code logic.

2.1.6.2 Write simple publish and subscribe code

The following code is a simple subscription node:

Code list 2.6.8 a simple subscription node code

```
#include "ros/ros.h"

#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String& msg)

{

    ROS_INFO("I heard: [%s]", msg.data.c_str());


}

int main(int argc, char **argv)

{

    ros::init(argc, argv, "listener");

    ros::NodeHandle nodeHandle;

    ros::Subscriber subscriber =


        nodeHandle.subscribe("chatter", 10, chatterCallback);

    ros::spin();

    return 0;

}
```

The node defines a subscriber by calling the *subscribe* method on the handler, which calls the *chatterCallback* function when the subscribed topic receives a message. In this case, the *spin()* function keeps the node listening on the topic, thereby making the program continuously listen rather than directly return 0.

Defining a simple publish node is similar to the process for the previous “Hello World” node.

Code list 2.6.9 Define the simple release node code

```
#include <ros/ros.h>

#include <std_msgs/String.h>

int main(int argc, char **argv) {

    ros::init(argc, argv, "talker");

    ros::NodeHandle nh;

    ros::Publisher chatterPublisher =
        nh.advertise<std_msgs::String>("chatter", 1);

    ros::Rate loopRate(10);

    unsigned int count = 0;

    while (ros::ok()) {

        std_msgs::String message;

        message.data = "hello world " + std::to_string(count);

        ROS_INFO_STREAM(message.data);

        chatterPublisher.publish(message);

        ros::spinOnce();

        loopRate.sleep();

        count++;

    }

    return 0;

}
```

The steps are as follows:

- Define a publisher.
- Create message content.
- Post messages at a certain frequency in a loop.

2.1.6.2.1 Object-oriented node coding

In the implementation, most of the code is organized in an object-oriented style. Under a package, a `_packagename__node.cpp` is used as the node entry, which consists of the node handler and instantiates a processing class.

Code list 2.6.10 Define the node handle and instantiate a processing class

```
#include <ros/ros.h>

#include "my_package/MyPackage.hpp"

int main(int argc, char** argv)

{
    ros::init(argc, argv, "my_package");

    ros::NodeHandle nodeHandle("~");

    my_package::MyPackage myPackage(nodeHandle);

    ros::spin();

    return 0;
}
```

The interfaces of ROS (publisher, subscriber) in the nodes are defined in `MyPackage.hpp` and `MyPackage.cpp`, while the specific algorithms (recognition and detection algorithm of unmanned vehicles) are directly defined in a separate algorithm class. This programming mode makes the algorithm part and ROS programming (communication between nodes) further decouple, which is more in principle with the requirements of software engineering.

2.1.6.3 Parameter services in ROS

In the program of robots and unmanned vehicles, although nodes are independent of one another, some global configuration parameters need to be stored and read during the operation. ROS provides global parameter services, which can be stored in launch files or in separate YAML files. The parameters are defined in the launch file.

Code list 2.6.11 Define parameters

```
<launch>

  <node name="name" pkg="package" type="node_type">
    <rosparam command="load"
      file="$(find package)/config/config.yaml" />
  </node>
</launch>
```

In the above code, the parameters are obtained from the parameter server through the `getParam(name, value_variable)` method on the node handler. `getPram()` returns true if the argument exists and passes the value to `value_variable`. Here is an example of using the ROS parameter in C++:

Code list 2.6.12 Use ROS parameters in C++

```
ros::NodeHandle nodeHandle("~"); //current node namespace
std::string topic;
if (!nodeHandle.getParam("topic", topic)) { //find topic parameter
variables and assign if they exist
ROS_ERROR("Could not find topic parameter!");
}
```

2.1.6.4 Small case based on husky robot

We write a package in object-oriented programming and create a package called `husky_highlevel_controller` in `~/catkin_ws/src`, which depends on `roscpp` and `sensor_msgs`.

Code list 2.6.13 Create package named `husky_highlevel_controller`

```
catkin_create_pkg husky_highlevel_controller roscpp sensor_msgs
```

Next, we create two source files under `husky_highlevel_controller/src`, namely, `husky_highlevel_controller_node.cpp` and `husky_controller_node.cpp`. At the same time, we create the header file `husky_controller.hpp` in the `include/husky_high_level_controller`/directory.

The CMakeLists.txt file is modified as follows:

Code list 2.6.14 Modified CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)

project(husky_high_level_controller)

add_definitions(-std=c++11)

find_package(catkin REQUIRED COMPONENTS

    roscpp

    sensor_msgs

)

catkin_package(
    INCLUDE_DIRS include
    # LIBRARIES ${PROJECT_NAME}
    CATKIN_DEPENDS roscpp sensor_msgs
    # DEPENDS system_lib
)

#####
## Build ##
#####

include_directories(
    include
    ${catkin_INCLUDE_DIRS}
)

add_executable(${PROJECT_NAME}_node
    src/${PROJECT_NAME}_node.cpp
    src/husky_controller.cpp
)
target_link_libraries(${PROJECT_NAME}_node
    ${catkin_LIBRARIES})
```

cmake_minimum_required specifies the catkin version while *project* specifies the package name, which can be accessed in the following statements by referring to the variable `${PROJECT_NAME}`. We need to find additional packages when building the program through *find_package*. If CMake uses the *find_Package* function to find packages, then it creates environment variables containing information that the package can be found. These environment variables describe the package's header file, source file path, libraries, and library paths that the package depends on. In general, the reason why we make CMake packages as catkin *COMPONENTS*, that is, writing the name of these packages after *COMPONENTS*, is to build a simple set of environment variables. Package environment variables found through *COMPONENTS* are added to *catkin_variables* altogether.

catkin_package is a macrostate used to produce the project. Its position must be placed before the object functions like *add_library* and *add_executable*. *catkin_package* usually includes five arguments as follows:

- INCLUDE_DIRS: header directory
- LIBRARIES: project library directory
- CATKIN_DEPENDS: other catkin projects on which this project depends
- DEPENDS: other CMake projects on which this project depends
- CFG_EXTRAS: some configuration options

The statements in the CMake code, which follows the build state, are used to specify the build target. Before specifying the build target, you need to specify the path of the header file and library file. The syntax is as follows:

- Include path: header file path required for the build; specified by “`include_directories()`”;
- Library path: the path to the library that needs to be built by the executable target; specified by “`link_directories()`”;

After the header and library paths are added, the *add_executable* function specifies the executable target that needs to be built. The first parameter of the *add_executable* function is the name of the executable target that needs to be built (specified by itself), and the following parameters are the path to the source file, separated by spaces if multiple source files are involved. Then, *add_library* is used to specify the library files needed to build the target, wherein the first argument is the name of the target to be built, and the following arguments are a list of library files that the target depends on. Finally, the library is linked to the executable target through *target_link_libraries* with the following parameters:

Code list 2.6.15 target_link_libraries parameters

```
target_link_libraries(<target name>, <lib1>, <lib2>, ... , <libN>)
```

husky_highlevel_controller_node.cpp is written after the package's build file is configured. This source file is an entry to a node that declares a handler to the node.

Code list 2.6.16 husky_highlevel_controller_node.cpp

```
#include <ros/ros.h>

#include "sensor_msgs/LaserScan.h"

#include "husky_high_level_controller/husky_controller.hpp"

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <iostream>

using namespace std;

int main(int argc, char ** argv){

    ros: : init(argc, argv, "laser_listener");

    ros: : NodeHandle node_handle;

    husky_controller: : HuskyController test(node_handle);

    return 0;

}
```

A class is instantiated, then the header and source files are completed for this class.

Code list 2.6.17 husky_controller header file

```
#include <ros/ros.h>

#include "sensor_msgs/LaserScan.h"

namespace husky_controller{

class HuskyController {

public:

    HuskyController(ros::NodeHandle &node_handle);

private:

    void LaserCallBack(const sensor_msgs::LaserScan::ConstPtr &msg);

    ros::NodeHandle &nodeHandle_;

    ros::Subscriber laserSub_;


};

} // namespace husky_controller
```

Code list 2.6.18 husky_controller source file

```
#include "husky_high_level_controller/husky_controller.hpp"

namespace husky_controller{

HuskyController::HuskyController(ros::NodeHandle &node_handle):

nodeHandle_(node_handle) {

    std::string topic;

    if(!nodeHandle_.getParam("/laser_listener/laser_topic", topic)){
        ROS_ERROR("Load the laser scan topic param fail!!");

    } else{

        HuskyController::laserSub_ = nodeHandle_.subscribe(topic, 1,
&HuskyController::LaserCallBack, this);

    }

    ros::spin();
}
```

```
    }

}

void HuskyController: :LaserCallBack(const sensor_msgs: :LaserScan: :
ConstPtr &msg) {

    unsigned long len = msg->ranges.size();

    std: : vector<float> filtered_scan;

    for (int i = 0; i < len; ++i) {

        if(std: : isnormal(msg->ranges[i])){

            filtered_scan.push_back(msg->ranges[i]);

        }

    }

    for (int j = 0; j < filtered_scan.size(); ++j) {

        ROS_INFO_STREAM(filtered_scan[j]);

    }

}

} //namespace husky_controller
```

The function of this node is very simple. It outputs nonzero and infinite points of the Husky robot laser scan data by subscribing to the /scan topic. Although the topic is not specified in the code, it usually subscribes to this topic by *getParam()*, which obtains the parameters specified in the launch file. Thus, we implement the launch file.

Code list 2.6.19 Launch file code

```
<?xml version="1.0"?>

<launch>

<include file="$(find husky_gazebo)/launch/husky_playpen.launch">

<arg name="laser_enabled" value="true"/>

</include>

<node
  pkg="husky_high_level_controller"
  type="husky_high_level_controller_node" name="laser_listener"
  output="screen">

  <param name="laser_topic" value="/scan"/>

</node>

<node pkg="Rviz" type="Rviz" name="Rviz"/>

</launch>
```

In this launch file, a parameter *laser_topic* is specified for the node *husky_high_level_controller*, and a launch file *husky_playpen.launch* is included at the same time. The launch file actually launches a gazebo simulation of a Husky robot. When we introduce the launch file, we pass *laser_enabled* to it with a value of “true,” indicating that the laser capability of the Husky robot is enabled. Finally, a Rviz node is added. Rviz is ROS’ official 3D visualization tool, which we will use later to present laser scans.

The package is compiled using catkin_make. Then, the source is set up, and the launch file is bashed and launched.

Code list 2.6.20 Launch the launch file

```
cd ~/catkin_ws
catkin_make
source devel/setup.bash
# for bash, you should replace the setup.zsh with setup.bash
# roslaunch husky_high_level_controller high_controller.launch
```

The command line continuously displays the laser scan data and opens the gazebo simulation environment and Rviz tool. We set the Fixed Frame as odom in Rviz's Global Options, as shown in Fig. 2.5.

We also add a LaserScan display, as indicated in Fig. 2.6.

We set the topic of LaserScan to “/scan” and set the size to 0.1 m. The laser scan result is presented in Fig. 2.7.

Fig. 2.8 shows robot cars in Gazebo.

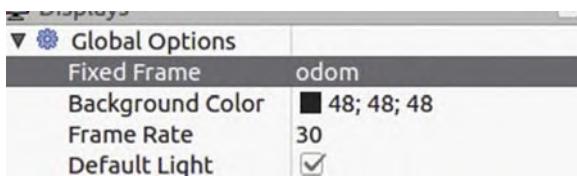


Figure 2.5 Specify fixed frame as odom.

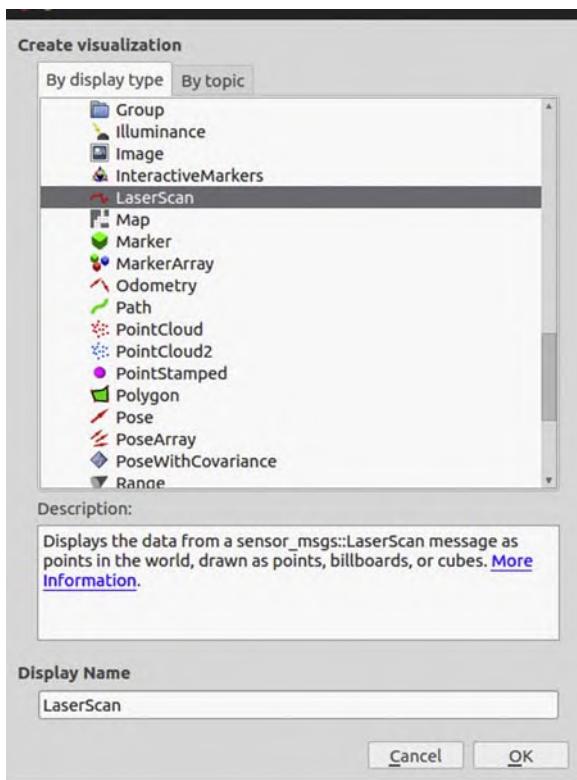


Figure 2.6 Adding a LaserScan display.

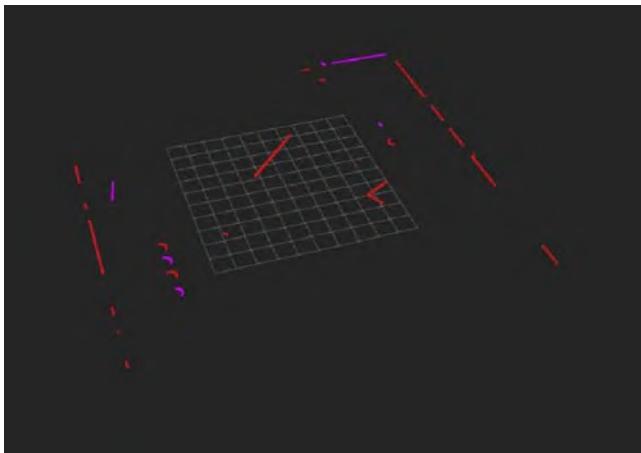


Figure 2.7 Laser scan result.



Figure 2.8 Situation in Gazebo.

2.1.7 ROS services

In addition to the publish–subscribe pattern, ROS provides a request/response mechanism, as shown in Fig. 2.9.

This communication mechanism is implemented through ROS services. Service is similar to the topic, but unlike the topic, it has feedback to the client node. The definition of a service is similar to that of a message. The file name traditionally uses the `.srv` suffix, the front of the file is the request message format, the middle is separated by “`—`,” and the back is the reply message format, as shown in Fig. 2.10.

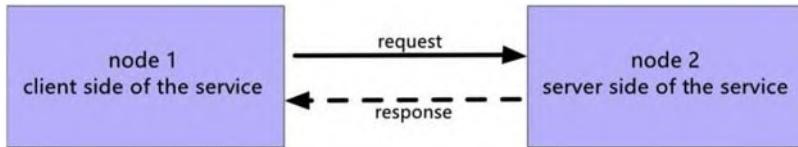


Figure 2.9 Request/response mechanism.

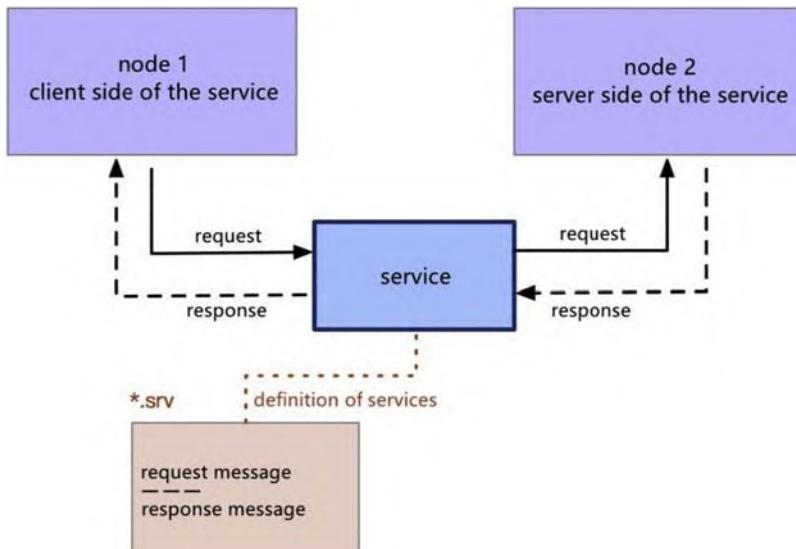


Figure 2.10 Response message format.

Similarly, services can be analyzed in the command line. Common service analysis commands are as follows:

- `rosservice list`: list all the current services
- `rosservice type`: service name: show the service type
- `rosservice call`: service name parameter: call service

The definition of a Server/Client in ROS is very similar to the publisher/subscriber pattern. For example, in the following code, the Client sends two integers to the Server while the Server sums them up and reports the sum back to the Client program. We start by defining the server code.

Code list 2.7.1 Define Server

```

#include <ros/ros.h>

#include <roscpp_tutorials/TwoInts.h>

bool add(roscpp_tutorials::TwoInts::Request &request,
         roscpp_tutorials::TwoInts::Response &response)

{
    response.sum = request.a + request.b;

    ROS_INFO("request: x=%ld, y=%ld", (long int)request.a, (long
int)request.b);

    ROS_INFO(" sending back response: [%ld]", (long int)response.sum);

    return true;
}

int main(int argc, char **argv)

{
    ros::init(argc, argv, "add_two_ints_server");

    ros::NodeHandle nh;

    ros::ServiceServer service =
        nh.advertiseService("add_two_ints", add);

    ros::spin();

    return 0;
}

```

Through the above code, the Server is instantiated by calling *nodeHandle advertiseService (service_name callback_function)*. When a request is received, the Server executes the defined callback function in ROS; thus, *ros:: spin()* should be added to keep the node running and listening for the callback. In callback functions, the response section should be attached when processing is complete. The callback function of the service usually returns a Boolean value, and if the value is true, then the callback was executed correctly. Next, we analyze the Client as follows:

Code list 2.7.2 Client

```
#include <ros/ros.h>
#include <roscpp_tutorials/TwoInts.h>
#include <cstdlib>

int main(int argc, char **argv) {
    ros::NodeHandle nh;
    ros::ServiceClient client =
        nh.serviceClient<roscpp_tutorials::TwoInts>("add_two_ints");
    roscpp_tutorials::TwoInts service;
    service.request.a = atoi(argv[1]);
    service.request.b = atoi(argv[2]);
    if (client.call(service)) {
        ROS_INFO("Sum: %ld", (long int)service.response.sum);
    } else {
        ROS_ERROR("Failed to call service add_two_ints");
    }
    return 0;
}
```

Similar to the publisher definition, we use `nodeHandle.serviceClient<service_type>(service_name)` to create a Client and instantiate a service as a message. Here we just need to fill the message in the request part of the service. On the Client, run the `client.call(service);` to send a request to the Server and get the feedback via `service.response`.

2.1.8 ROS action

ROS also has a communication mechanism called Action. Action is similar to service, but it provides more operations, including the Client canceling the Server's task and the Server feeding back information in real time during the process. Similar to services, actions are often defined in files with the suffix “*action*” in the format of goal, split line, result, split line, feedback. In fact, from the perspective of ROS internal implementation, actions in ROS are implemented through a set of topics. Fig. 2.11 shows the Action mechanism.

2.1.9 Common tools in ROS

The previous article used some ROS native tools (including `rqt_graph` and `Rviz`). In this section, we will introduce the tools commonly used in ROS, including `Rviz`, `rqt`, TF conversion system, Unified Robot Description Format (URDF), and Simulation Description Format (SDF).

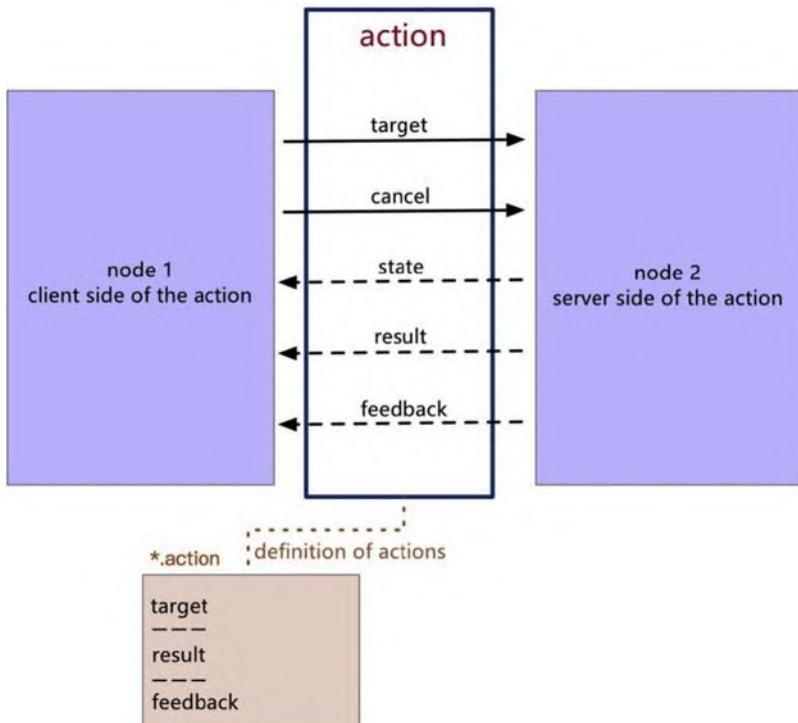


Figure 2.11 Action mechanism.

2.1.9.1 Rviz

As a 3D visualization tool in ROS, Rviz visualizes messages on current topics (such as images and point clouds) by subscribing to these topics. If the unmanned vehicle/robot contains multiple cameras and can provide multiple camera image input displays, then the information we want to display can be visualized by adding display plugins in Rviz. Rviz's built-in display plugin is shown in Fig. 2.12.

Display plugins can be added through the Add button at the bottom of the display bar or by message type in the current topic. After configuring the display plugin, you can save the Rviz configuration file so that you do not need to manually add it again the next time that you open Rviz. In addition to the native display plugin, you can customize and extend the Rviz display plugin. You can start Rviz with `rosrun Rviz` or write the node to the launch file to start it.

2.1.9.2 rqt

ROS rqt is a tool based on the Qt development framework. It contains a large number of drawing components and debugging tools. You can use “`rosrun rqt_gui rqt_gui`” to start the user interface of rqt. As you can see in the following Fig. 2.13, rqt itself already contains a number of debugging tools, but you can still customize and extend the rqt plugin.

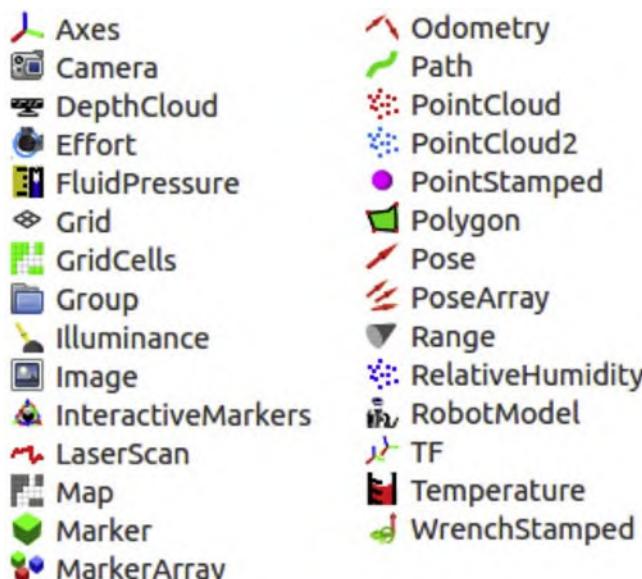


Figure 2.12 Rviz display plugin.

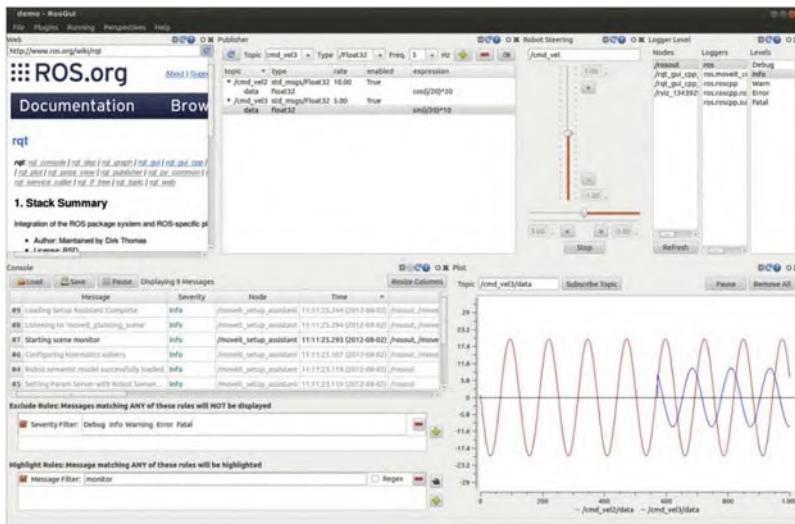


Figure 2.13 rqt plugin.

ros-kinetic-rqt	ros-kinetic-rqt-joint-trajectory-controller	ros-kinetic-rqt-reconfigure
ros-kinetic-rqt-action	ros-kinetic-rqt-joint-trajectory-plot	ros-kinetic-rqt-robot-dashboard
ros-kinetic-rqt-bag	ros-kinetic-rqt-launch	ros-kinetic-rqt-robot-monitor
ros-kinetic-rqt-bag-plugins	ros-kinetic-rqt-launchtree	ros-kinetic-rqt-robot-plugins
ros-kinetic-rqt-bhand	ros-kinetic-rqt-logger-level	ros-kinetic-rqt-robot-steering
ros-kinetic-rqt-common-plugins	ros-kinetic-rqt-moveit	ros-kinetic-rqt-runtime-monitor
ros-kinetic-rqt-console	ros-kinetic-rqt-msg	ros-kinetic-rqt-rviz
ros-kinetic-rqt-controller-manager	ros-kinetic-rqt-multiplot	ros-kinetic-rqt-service-caller
ros-kinetic-rqt-image-view	ros-kinetic-rqt-nav-view	ros-kinetic-rqt-shell
	ros-kinetic-rqt-plot	ros-kinetic-rqt-srv
	ros-kinetic-rqt-pose-view	ros-kinetic-rqt-tf
	ros-kinetic-rqt-graph	ros-kinetic-rqt-top
	ros-kinetic-rqt-orb-dashboard	ros-kinetic-rqt-topic
	ros-kinetic-rqt-gui	ros-kinetic-rqt-web
	ros-kinetic-rqt-gui-cpp	ros-kinetic-rqt-wrapper
	ros-kinetic-rqt-py-common	
	ros-kinetic-rqt-py-console	
	ros-kinetic-rqt-py-trees	

Figure 2.14 rqt toolkit.

Plugins commonly used in rqt include rqt_image_view (display image), rqt_multiplot (draw 2D plot), rqt_graph (display current ROS node graph), rqt_console (display ROS terminal debug print messages), rqt_bag (graphical playback control of rosbag file), etc. You can also view topic content and data using the rqt tool or even run Rviz inside rqt. These tools are easy to use and will not be described here. Interested readers can check out the rqt toolkit shown in Fig. 2.14.

2.1.9.3 TF coordinate conversion system

TF is a coordinate transformation toolkit that tracks multiple reference coordinates under ROS over time. With the help of TF, we can complete the coordinate transformation of the point and vector coordinates of data in two reference systems at any time. A robot system usually has many

three-dimensional reference frames that change over time, such as World Frame, Base Frame, Gripper Frame, and Head Frame. TF can also track these reference frames over time and allow users to request.

- What is the relation between the robot's head reference frame and the global reference frame 5 seconds ago?
- Where is the object which the robot picks up relative to the robot's central reference frame?
- Where is the robotics central reference system relative to the global reference system?

TF is implemented on the topics *tf* and *tf_static* via the publisher/subscriber mechanism shown in Fig. 2.15.

Generally speaking, the use of TF can be divided into the following two categories:

- a) Monitor TF transform: receive and cache all reference transforms published in the system and query the required reference transform from it.
- b) Broadcast TF transform: broadcast the coordinate transformation relationship between reference systems in the system. Different parts of the system likely perform multiple TF transformation broadcasts, and each broadcast can directly insert the reference frame transformation relationship into a tree structure (we call it the tf tree). We can use ros-run *tf view_frames* to export and visualize the TF transformation tree data in PDF form. The TF data for the Husky robot are shown in Fig. 2.16.

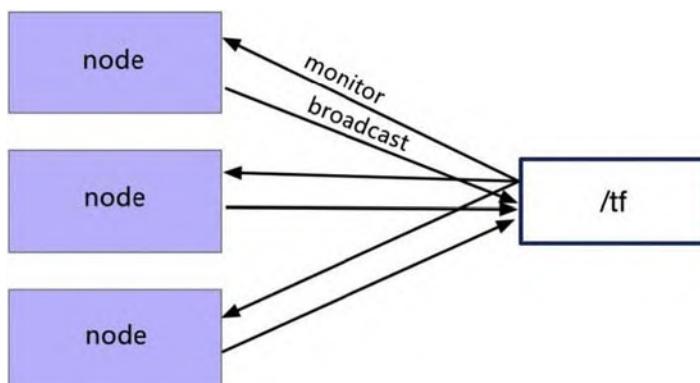


Figure 2.15 TF implementation mechanism.

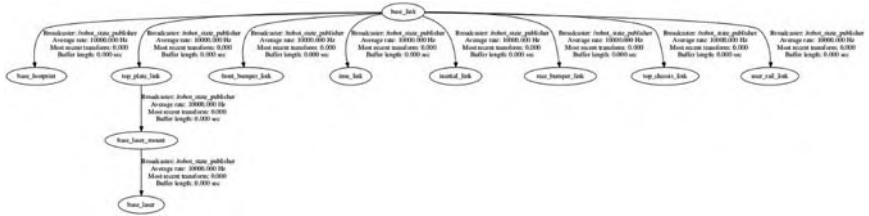


Figure 2.16 Transformation tree of Husky robot.

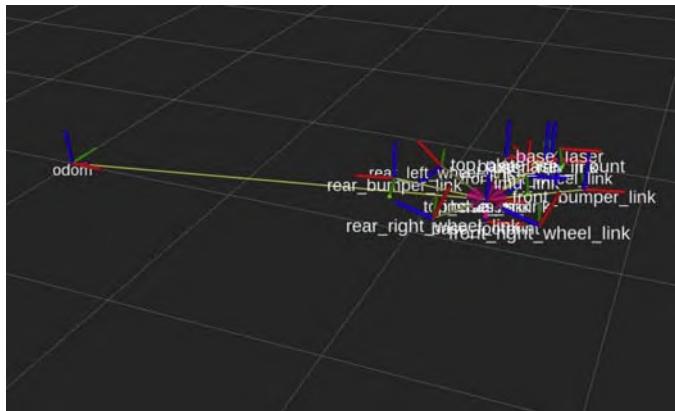


Figure 2.17 Visualization results of each reference frame of the robot under Rviz.

In addition, Rviz also provides a 3D display plugin for TF conversion. Fig. 2.17 presents a visualization of the Husky robot's various reference frames.

2.1.9.4 URDF and SDF

URDF is an XML syntax for describing a robot model. The robot model here explicitly includes the following:

- Kinematics and dynamics models
- Visualization
- Collision model

URDF generation is done using ROS' native macro language xacro, which is stored in the `/robot_description` parameter server. By looking at the launch file of the Husky robot, in the “`husky_empty_world.launch`” file, `spawn_husky.launch` is referenced, and it provides the Husky robot with three parameters (`kinect_enabled` is used to enable the kinematics model in

the URDF of the Husky robot). In the *spawn_husky.launch* startup file, we enable the robot model (URDF) with the following parameters.

Code list 2.9.1 Enable the robot model (URDF parameter)

```
<param name="robot_description" command="$(find xacro)/xacro.py
      '$(arg husky_gazebo_description)'
      laser_enabled:=$(arg laser_enabled)
      ur5_enabled:=$(arg ur5_enabled)
      kinect_enabled:=$(arg kinect_enabled)" />
```

Similarly, URDF can also be visualized by Rviz, whose display plugin is *RobotModel*. According to the actual application, the robot model can be customized. In the research and development of unmanned vehicles, the robot model is the vehicle model. These vehicle models also need to meet some kinematic and dynamic requirements and have collision characteristics.

In addition to customizing the description of the vehicle model (URDF), ROS provides an SDF. The description format is also defined by using XML syntax. These descriptions include descriptions of the environment (light, gravity, etc.), as well as static and dynamic objects, sensors, robots, etc. SDF is the standard description syntax for Gazebo emulators. In addition, Gazebo can automatically convert URDF to SDF.

References

1. ROS: <http://www.ros.org/>.
2. Wikipedia: robot operating system <https://zh.wikipedia.org/wiki/%E6%A9%9F%E5%99%A8%E4%BA%BA%E4%BD%9C%E6%A5%AD%E7%B3%BB%E7%B5%B1>.
3. About ROS: <http://www.ros.org/about-ros/>.
4. Baidupedia: robot operating system <https://baike.baidu.com/item/ros/4710560?fr=aladdin>.
5. ROS wiki: <http://wiki.ros.org/ROS>.
6. Remote procedure call: https://en.wikipedia.org/wiki/Remote_procedure_call.
7. OMG, Data Distribution Service for Real-Time Systems, Object Management Group, 1.2 formal/07-01-01 edition.
8. Husky robot: <https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/>.
9. Gazebo simulation: <http://gazebosim.org/>.

CHAPTER 3

Localization for unmanned vehicle

Zebang Shen¹, Gang Huang², Peng Zhi³ and Rui Zhao³

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²SATG, Intel, Shanghai, China; ³School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

3.1 Principle of achieving localization	63
3.2 ICP algorithm	65
3.3 Normal distribution transform	72
3.3.1 Introduction to NDT algorithm	72
3.3.2 Basic steps of NDT algorithm	73
3.3.3 Advantages of NDT algorithm	74
3.3.4 Algorithm example	75
3.4 Localization system based on global positioning system (GPS) + inertial navigation system (INS)	81
3.4.1 Localization principle	82
3.4.2 Localization fusion of different sensors	84
3.5 SLAM-based localization system	87
3.5.1 SLAM localization principle	88
3.5.2 SLAM applications	90
References	93

3.1 Principle of achieving localization

Simultaneous localization and mapping (SLAM), also known as concurrent mapping and localization, is a concurrent map building and localization method.¹ The SLAM approach can be described as a robot that moves from an unknown location in an unfamiliar environment, localizes itself on the basis of the location estimation and map during the movement, and then builds incremental maps according to its localization to achieve autonomous robot localization and navigation.

In localization based on high-precision point cloud maps, the data generated by the Light Detection and Ranging (LiDAR) system are transformed into point cloud maps by the Point Cloud Library (PCL)², and

localization is completed by matching between point clouds. A high-precision point cloud map is shown in [Fig. 3.1](#).

The point cloud data rendering after LiDAR scanning are shown in [Fig. 3.2](#).



Figure 3.1 High-precision point cloud map effect of some roads in a university.

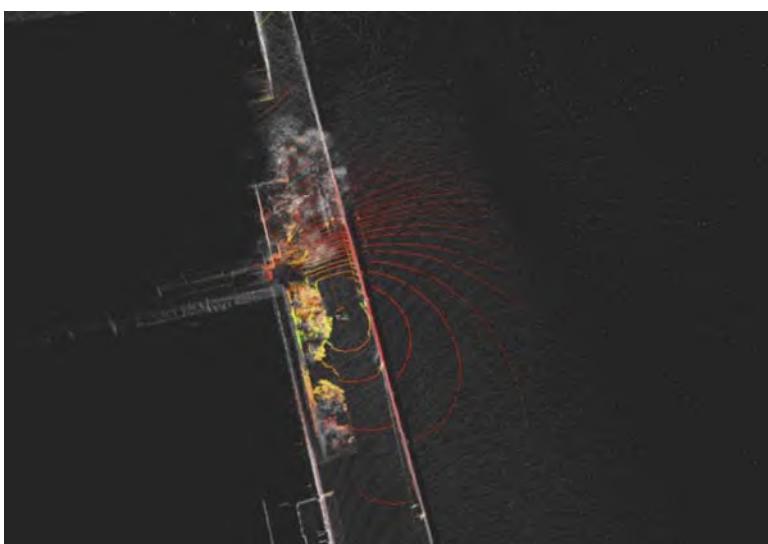


Figure 3.2 Point cloud effect obtained with LiDAR scan.

Suppose we want to know the location of an unmanned vehicle. We need to iteratively match the point cloud fragment currently scanned by LiDAR with our previously collected global point cloud map. Using the iterative closest point (ICP), we can find the closest match for each point of the first point cloud with the second point cloud and then calculate the mean squared error (MSE, a statistical concept) from all the matches. This process is referred to as finding the minimum MSE. The matching process is the process of finding the minimum MSE. The variance may come from measurement errors, or it may be caused by the scene's dynamic changes. By minimizing the error, the unmanned vehicle's position coordinates are calculated to best match the current map environment. The alignment algorithm for reducing the error is introduced herein.

3.2 ICP algorithm

The most commonly used iterative algorithm in current 3D alignment is the ICP iterative algorithm³, which needs to provide a better initial value. By contrast, the final iterative result may fall into a local optimum rather than a global optimum because of the algorithm's defects.

ICP algorithm principle: Given a reference point set P and a data point set Q (at a provided initial estimate R, T), the algorithm finds the corresponding nearest point in P for each point in Q to form a matching point pair. Then, it uses the sum of the Euclidean distances of all matching point pairs as the value of the error objective function error, utilizes singular value decomposition (SVD) to find R and T in order to minimize the error, rotates Q according to R and T, and finds the corresponding point pairs again.

Disadvantages: Noise points need to be eliminated (point pairs with too large distances or point couples containing boundary points). Point pair-based alignment does not include local shape information. Searching for the nearest point in each iteration is time-consuming. The computation may fall into a local optimum.

Specific code implementation: First, let us refer to the main application code of the ICP algorithm in the PCL.⁴

Code list 3.2.1 Main codes of ICP in PCL

```
// Create an instance class of ICP

pcl::IterativeClosestPoint< pcl::PointXYZ, pcl::PointXYZ> icp;

//Setting the input source point cloud

icp.setInputSource(cloud_sources);

//Setting the target point cloud

icp.setInputTarget(cloud_target);

/* Set the maximum corresponding point distance, 5cm, higher than this
value will be ignored */

icp.setMaxCorrespondenceDistance(0.05);

/* Set the conversion  $\epsilon$ , the error between the previous conversion and
the current conversion should be less than this value */

icp.setTransformationEpsilon(1e-10);

/* Set the Euclidean distance error threshold, the sum of all Euclidean
distance variances should be less than this value */

icp.setEuclideanFitnessEpsilon(1);

// Set the maximum number of iterations

icp.setMaximumIterations(100);

// Perform alignment operations

icp.align(final);

// Get the transformation relationship matrix

Eigen::Matrix4f transformation = icp.getFinalTransformation();
```

The following points need to be considered:

- ❑ The ICP algorithm in the PCL is based on SVD implementation,
- ❑ Several parameters need to be set before using the ICP with the PCL.
 - setMaximumIterations, the maximum number of iterations (can be set to 1 if combined with visualization and displayed one at a time).
 - setEuclideanFitnessEpsilon, which sets the convergence condition in which the MSE sum is less than a threshold value and stops the iteration.

- `setTransformationEpsilon`, which sets the difference between two change matrices (typically set to $1e-10$).
- `setMaxCorrespondenceDistance`, which sets the maximum distance between corresponding point pairs (this value significantly impacts the alignment result).

If we just run the above code and set reasonable preestimation parameters, we can realize the point cloud data's alignment calculation using the ICP. To gain more insight into the ICP calculation process, we continue to add code.

Code list 3.2.2 Example code for adding ICP

```

boost::shared_ptr< pcl::visualization::PCLVisualizer> view(new
pcl::visualization::PCLVisualizer("icp test"));

// Defining window shared pointers

int v1 ;

/* Define two windows v1, v2, window v1 to display the initial position,
v2 to display the alignment process */

int v2 ;

view->createViewPort(0.0,0.0,0.5,1.0,v1);

//The four window parameters correspond to x_min,y_min,x_max.y_max.

view->createViewPort(0.5,0.0,1.0,1.0,v2);

pcl::visualization::PointCloudColorHandlerCustom< pcl::PointXYZ>

sources_cloud_color(cloud_in,250,0,0);

//set the color of the source point cloud to red

view->addPointCloud(cloud_in,sources_cloud_color,
"sources_cloud_v1",v1);

pcl::visualization::PointCloudColorHandlerCustom< pcl::PointXYZ>

target_cloud_color (cloud_target,0,250,0);

// Target point cloud in green

view->addPointCloud(cloud_target,target_cloud_color,"target_cloud_v1
",v1);

// Adding point clouds to the v1 window

view-> setBackgroundColor(0.0,0.05,0.05,v1);

//Set the background color of the two windows

view-> setBackgroundColor(0.05,0.05,0.05,v2);

```

```
view->setPointCloudRenderingProperties(2, "sources_cloud_v1");

//set the size of the displayed points

view->

setPointCloudRenderingProperties(2,"target_cloud_v1");

pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>alignend_color(Final,255,255,255);

//set the alignment result to white

view-> addPointCloud(Final,alignend_color,
"alignend_cloud_v2",v2);

view->addPointCloud(cloud_target,target_color,"target_cloud_v2
",v2);

view->

setPointCloudRenderingProperties(2,"alignend_cloud_v2");

view->

setPointCloudRenderingProperties(2,"target_cloud_v2");

view-> registerKeyboardCallback(& keyboardEvent, (void*)NULL);

//set keyboard callback function

int iterations = 0; //number of iterations

while(!view-> wasStopped())

{
```

```
view-> spinOnce(); //run the view

if (next_iteration)

{

    icp.align(*Final); //icp calculation

    cout <<"has conveged:"<<

    icp.hasConverged()<<"score:"<<icp.getFitnessScore()<<endl;

    cout<< "matrix:\n"<< icp.getFinalTransformation()<< endl;

    cout<<"iteration = "<<++iterations;

    /*... If icp.hasConverged=1, then the alignment is

    successful and icp.getFinalTransformation() can output the transformation

    matrix ... */

    if (iterations == 1000) // set the maximum number of

iterations

        return 0;

    view->

updatePointCloud(Final,aligend_cloud_color, "aligend_cloud_v2");

}

next_iteration = false; //this iteration is over, wait for trigger

}
```

Finally, the following keyboard callback functions need to be set up to control the iterative process.

Code list 3.2.3 Example code for adding ICP

```

bool next_iteration = false; //Set the keyboard interaction function

/*... The following functions indicate that ICP calculations can only
be performed when the space bar is pressed on the keyboard ... */

void keyboardEvent(const pcl::visualization::KeyboardEvent & event, void
*nothing)

{
    if(event.getKeySym() == "space" && event.keyDown())
        next_iteration = true;
}

```

We combine the above code and add the appropriate headers.

Code list 3.2.4 Add the corresponding header file

```

#include <iostream>

#include <pcl/io/pcd_io.h>

#include <pcl/point_types.h>

#include <pcl/registration/icp.h>

#include <pcl/visualization/pcl_visualizer.h>

#include <boost/thread/thread.hpp>

#include <pcl/console/parse.h> //pcl console parsing

```

The initial purpose can be achieved by setting the appropriate parameters according to different models. The example is displayed in Fig. 3.3.

The left image shows the initial position, while the right image indicates the alignment process. The red color represents the source point cloud, and the green color is the target point cloud. The point cloud is added to the initial window, and the white color shows the alignment result. Fig. 3.3 shows how the ICP algorithm achieves the calculation process of progressive alignment.

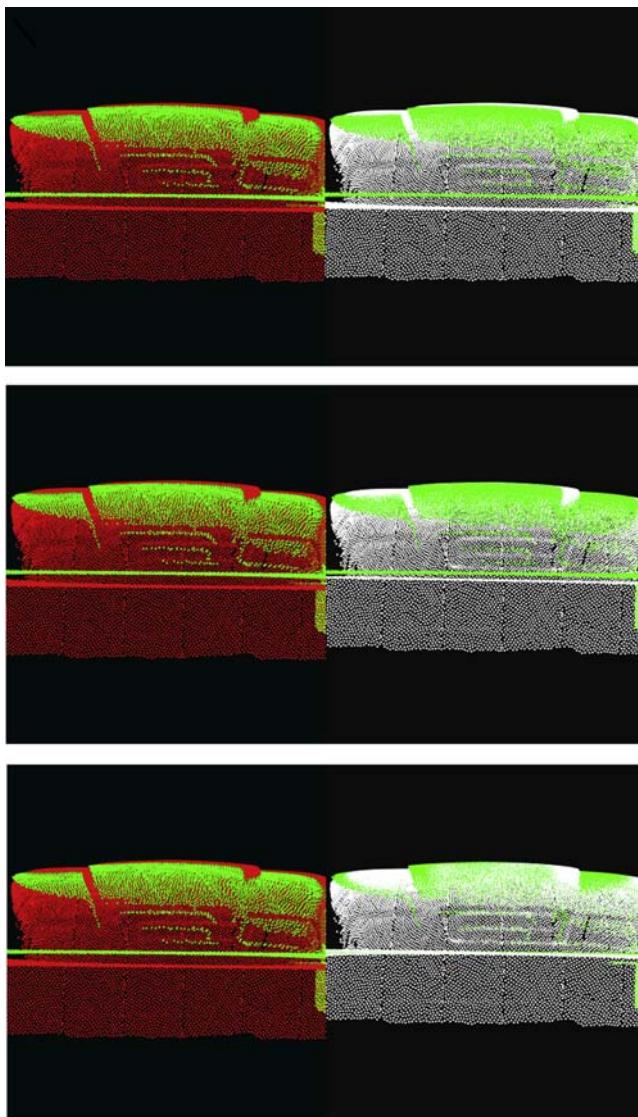


Figure 3.3 ICP application matching process.

Note that the ICP has many variants, including point-to-point and point-to-plain. The normal vector-based point-to-plain is relatively fast. It requires a reliable average vector for the input data so that you can choose the most suitable method according to your needs and the available input data.

3.3 Normal distribution transform

The normal distribution transform (NDT) algorithm is an alignment algorithm that is based on the standard normal distribution applied to a statistical model of 3D points.⁵ It uses traditional optimization techniques to determine the optimal match between two point clouds. As it does not use the corresponding points' features to calculate and match during the alignment process, the computation is faster than that of other methods.

This algorithm, which is relatively time-consuming and stable, has little relation to the given initial value, and it corrects well when the initial value error is significant. Calculating the normal distribution is a one-time job performed at initialization without consuming many costs to calculate the nearest neighbor search matching points. The probability density function (PDF) can be calculated offline at the interval of two image acquisitions. The following parts describe it in detail.

3.3.1 Introduction to NDT algorithm

According to the principles of probability and statistics⁶, if random variable X satisfies a normal distribution (i.e., $X \sim N(\mu, \sigma)$), then its PDF is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

μ indicates a normal distribution while σ^2 is the variance in the case of low dimensionality. For the multivariate normal distribution, the PDF can be expressed as

$$f(\vec{x}) = \frac{1}{(2\pi)^{\frac{D}{2}} \sqrt{|\Sigma|}} e^{-\frac{(\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu})}{2}},$$

where \vec{x} denotes the mean vector. D is the dimensionality. Σ represents the covariance matrix, whose diagonal elements indicate the corresponding elements' variance rather than the correlation of the related elements (rows and columns). Fig. 3.4 shows a probability density plot obeying a two-dimensional normal distribution.

Its PDF is a bell-shaped surface with an ellipse contour. Both marginal distributions of the standard binary distribution are one-dimensional normal distributions.

The basic idea of the NDT algorithm is to construct a normal distribution of multidimensional variables on the basis of the reference data. If

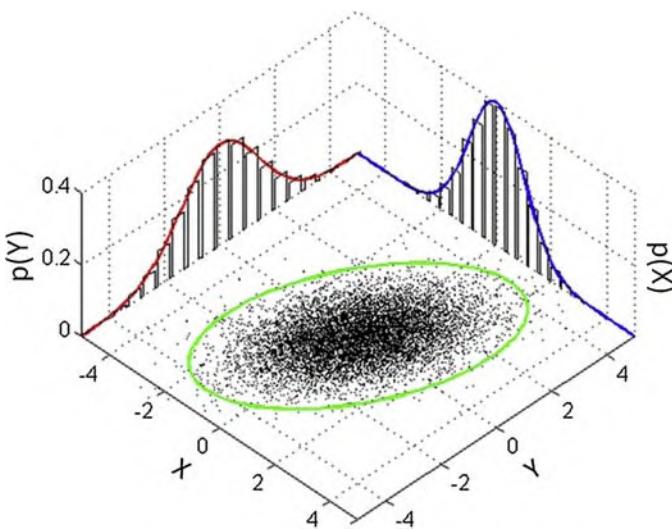


Figure 3.4 Probability density obeying a two-dimensional normal distribution and a probability density graph of two marginal distributions.

the transformation parameters can match the two LiDAR data well, the probability density of the reference system's transformation points will be significant. Therefore, an optimization method can be considered to determine the transformation parameters that make the sum of probability density the largest when the two LiDAR point cloud data are best matched.

3.3.2 Basic steps of NDT algorithm

- 1) Grid the reference point cloud and calculate the multidimensional normal distribution parameters for each grid

The space occupied by the reference point cloud is divided into a grid of a specified size. For our 3D map, we divide the entire length of scanned points into grids by using tiny cubes. Then, for each grid, the PDF is calculated on the basis of the issues within the grid. Here, $\vec{y}_k = 1, \dots, m$ denotes all the scanned points within a grid.

$$\text{Mean value: } \vec{\mu} = \frac{1}{m} \sum_{k=1}^m \vec{y}_k$$

$$\text{Covariance matrix: } \Sigma = \frac{1}{m} \sum_{k=1}^m (\vec{y}_k - \vec{\mu})(\vec{y}_k - \vec{\mu})^T$$

$$\text{Probability density function: } f(\vec{x}) = \frac{1}{(2\pi)^{\frac{3}{2}}\sqrt{|\Sigma|}} e^{-\frac{1}{2}(\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu})}$$

- 2) Initialize the transformation parameters and find the maximum likelihood

Our goal with NDT alignment is to find the current scan's pose so that the probability that the point of the recent scan lies on the reference scan surface is maximized. To perform a transformation (translation, rotation, etc.) on the current point cloud, we use the transformation parameter \vec{p} . The recent scan is a point cloud $X = \{\vec{x}_1, \dots, \vec{x}_n\}$, and the transformation parameter \vec{p} is initialized on the basis of a set of X. We use the spatial transformation function $T(\vec{p} \cdot \vec{x}_k)$ to represent the pose transformation \vec{p} used to move the points \vec{x}_k . We combine this representation with the previous set of states' density functions (one PDF for each grid). The best transformation parameter \vec{p} would be the pose transformation that maximizes the likelihood function

$$\text{Likelihood: } \theta = \prod_{k=1}^n f\left(T\left(\vec{p}, \vec{x}_k\right)\right).$$

Maximizing the likelihood is also equivalent to finding the minimum negative log-likelihood $-\log \theta$:

$$-\log \theta = - \sum_{k=1}^n \log\left(f\left(T\left(\vec{p}, \vec{x}_k\right)\right)\right).$$

- 3) Optimization parameters

The task here is to use an optimization algorithm to adjust the transformation parameters. The negative log-likelihood is minimized by using Newton's method in the NDT algorithm and will not be described here.

3.3.3 Advantages of NDT algorithm

Using a normal distribution to represent an otherwise discrete point cloud has many benefits. This representation of a smooth surface divided by a grid is continuously derivable, and each PDF can be considered an approximation of a local character. Fig. 3.5 shows a 3D point cloud and its meshing effect.

The cube used to describe the surface part of the space in Fig. 3.5 has a 1 m side, with the bright part indicating a high probability.

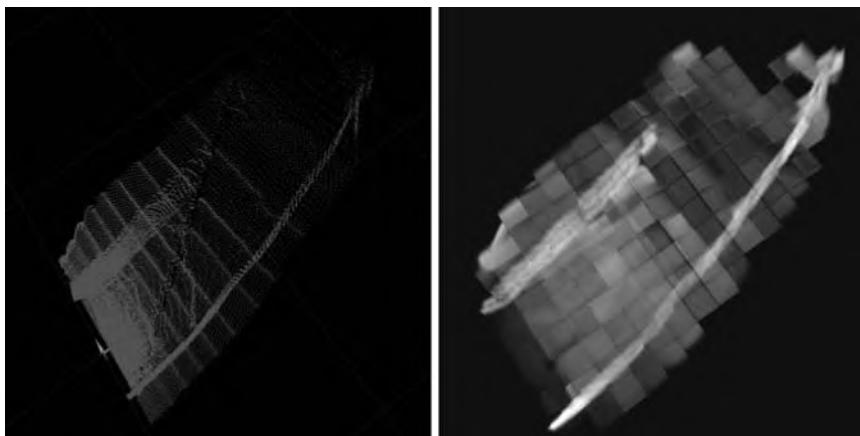


Figure 3.5 3D point cloud map and its meshing effect.

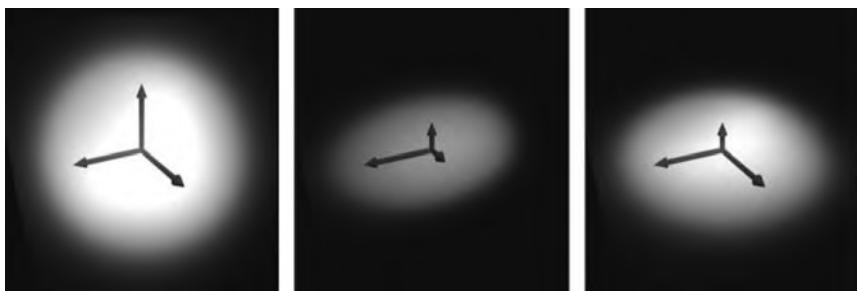


Figure 3.6 Relationship between three covariance matrix eigenvalues and surface shape.

In addition to describing this surface's location, the cube also contains information such as the orientation and smoothness of the surface. [Fig. 3.6](#) shows the relationship between the covariance matrix's eigenvalues and the shape of the character.

Take a 3D PDF as an example; if the three eigenvalues are close to one another, then this normal distribution describes a spherical surface. If one eigenvalue is much larger than the other two, then this normal distribution describes a line. If one eigenvalue is much smaller than the other two, then this normal distribution describes a plane. The PDF here does not require a normal distribution. It is acceptable with any PDF that reflects the scanned surface's structural information and is robust to abnormal scan points.

3.3.4 Algorithm example

The principles of how to use the normal distribution transform (NDT) for alignment were briefly described above. This subsection presents an example of

using the NDT algorithm to determine the rigid body transformation between two large point clouds (both with more than 100,000 points). The two point clouds are aligned using the NDT function provided by the PCL,^{7,8} and the point cloud data are stored in build/cloud1.pcd and build/cloud2.pcd files.

Step 1 Read the point cloud information from the PCD file.

Code list 3.3.1 Read point cloud information

```

pcl::PointCloud< pcl::PointXYZ>::Ptr
read_cloud_point(std::string const & file_path){

    // Loading first scan.

    pcl::PointCloud< pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<
pcl::PointXYZ>);

    if (pcl::io::loadPCDFile< pcl::PointXYZ> (file_path, *cloud) == -1)

    {

        PCL_ERROR ("Couldn't read the pcd file\n");

        return nullptr;

    }

    return cloud;
}

```

Code list 3.3.2 Read the point cloud from the main function separately

```

auto target_cloud = read_cloud_point("cloud1.pcd");

std::cout << "Loaded " << target_cloud->size () << " data points from
cloud1.pcd" << std::endl;

auto input_cloud = read_cloud_point("cloud2.pcd");

std::cout << "Loaded " << input_cloud->size () << " data points from
cloud2.pcd" << std::endl;

```

Code list 3.3.3 Number of points read from two PCD files

```
Loaded 112586 data points from cloud1.pcd
```

```
Loaded 112624 data points from cloud2.pcd
```

Step 2 Filter input point cloud.

Optimization for a large number of points is time-consuming. In this work, we use voxel_filter to filter the input point cloud. Only the input_cloud is filtered here to reduce the data volume to about 10%, while the target_cloud is not filtered.

Code list 3.3.4 Filtering the input point cloud

```

pcl::PointCloud< pcl::PointXYZ>::Ptr filtered_cloud (new
pcl::PointCloud< pcl::PointXYZ>);

pcl::ApproximateVoxelGrid< pcl::PointXYZ> approximate_voxel_filter;

approximate_voxel_filter.setLeafSize(0.2, 0.2, 0.2);

approximate_voxel_filter.setInputCloud(input_cloud);

approximate_voxel_filter.filter(*filtered_cloud);

std::cout<<"Filtered cloud contains "<< filtered_cloud->size() << "data
points from cloud2.pcd" << std::endl;

```

Code list 3.3.5 Only 10% of the original number of scan points after filtering

```
Filtered cloud contains 12433data points from cloud2.pcd
```

Code list 3.3.6 Initialize NDT and set NDT parameters

```

pcl::NormalDistributionsTransform< pcl::PointXYZ, pcl::PointXYZ> ndt;

ndt.setTransformationEpsilon(0.01);

ndt.setStepSize(0.1);

ndt.setResolution(1.0);

ndt.setMaximumIterations(35);

ndt.setInputSource(filtered_cloud);

ndt.setInputTarget(target_cloud);

```

Here, `ndt.setTransformationEpsilon()` sets the transformation ϵ (the maximum difference allowed between two consecutive transformations), which is the threshold to determine whether our optimization process has converged to the final solution. `ndt.setStepSize(0.1)` sets the maximum step size for Newton's method optimization. `ndt.setResolution(1.0)` sets the edge length of the cube when meshing. The mesh size setting is critical in NDT. The oversized setting leads to poor accuracy, and the undersized setting leads to high memory usage and can only be matched if the two point clouds are not very different. `ndt.setMaximumIterations(35)` denotes the number of iterations for optimization. In this work, we set it to 35 iterations. That is, when the number of iterations reaches 35 or converges to a threshold value, the optimization is stopped.

Step 3 Initialize transformation parameters and start optimization.

We initialize the transformation parameters \vec{p} (given an estimated value), and the initialization data for the transformation parameters are often derived from the measured data.

Code list 3.3.7 Initialize transformation parameters and start optimization

```
Eigen::AngleAxisf init_rotation(0.6931, Eigen::Vector3f::UnitZ());

Eigen::Translation3f init_translation (1.79387, 0.720047, 0);

Eigen::Matrix4f init_guess = (init_translation *
init_rotation).matrix();

pcl::PointCloud<pcl::PointXYZ>::Ptr output_cloud (new pcl::PointCloud<
pcl::PointXYZ>);

ndt.align(*output_cloud, init_guess);

std::cout << "Normal Distribution Transform has converged:" <<

ndt.hasConverged()
```

Code list 3.3.8 Save the point cloud after alignment and export it to the file cloud3.pcd

```
pcl::transformPointCloud(*input_cloud, *output_cloud,
ndt.getFinalTransformation());

pcl::io::savePCDFileASCII("... /cloud3.pcd", *output_cloud);
```

Step 4 Visualize the aligned point cloud.

We write a function to visualize the point cloud after alignment, where the target point cloud (i.e., our existing HD map) is plotted with red dots, and the input point cloud is plotted with green dots.

Code list 3.3.9 Visualize the aligned point cloud map

```
void visualizer(pcl::PointCloud< pcl::PointXYZ>::Ptr target_cloud,
pcl::PointCloud< pcl::PointXYZ>::Ptr output_cloud){

    // Initializing point cloud visualizer

    boost::shared_ptr< pcl::visualization::PCLVisualizer>
    viewer_final (new pcl::visualization::PCLVisualizer ("3D Viewer"));

    viewer_final-> setBackgroundColor (0, 0, 0);

    // Coloring and visualizing target cloud (red).

    pcl::visualization::PointCloudColorHandlerCustom< pcl::PointXYZ>
    target_color (target_cloud, 255, 0, 0);

    viewer_final-> addPointCloud< pcl::PointXYZ> (target_cloud,
target_color, "target cloud");

    viewer_final-> setPointCloudRenderingProperties
    (pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 1, "target cloud");

    // Coloring and visualizing transformed input cloud (green).

    pcl::visualization::PointCloudColorHandlerCustom< pcl::PointXYZ>
    output_color (output_cloud, 0, 255, 0);

    viewer_final-> addPointCloud< pcl::PointXYZ> (output_cloud,
output_color, "output cloud");
```

```
viewer_final-> setPointCloudRenderingProperties  
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 1, "output cloud");  
  
// Starting visualizer  
  
viewer_final-> addCoordinateSystem (1.0, "global");  
viewer_final-> initCameraParameters ();  
  
// Wait until visualizer window is closed.  
  
while (! viewer_final-> wasStopped ())  
{  
    viewer_final-> spinOnce (100);  
    boost::this_thread::sleep (boost::posix_time::microseconds  
(100000));  
}  
}
```

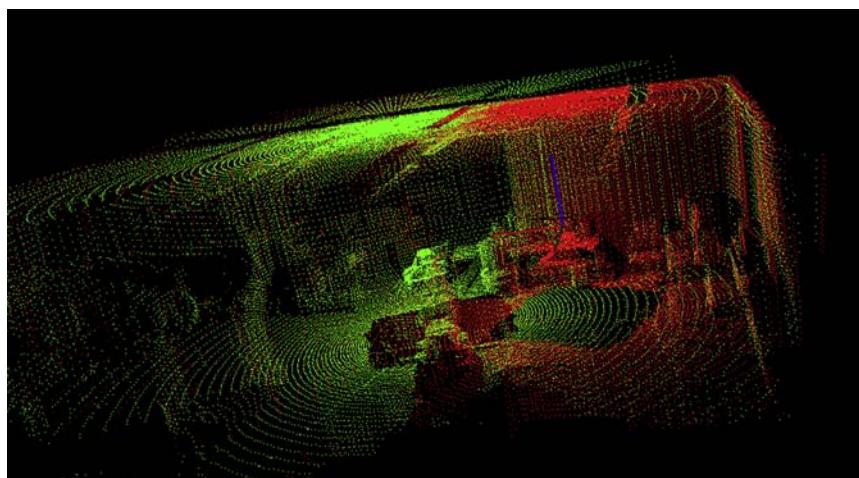


Figure 3.7 Resulting graph of sample run.

The final result of the experiment is shown in Fig. 3.7, where the red dots indicate the target point cloud (i.e., our existing high-precision map),

the green dots indicate the input point cloud, and the red and green superimposed parts are the current matching effect.

3.4 Localization system based on global positioning system (GPS) + inertial navigation system (INS)

The INS is an autonomous navigation system based on the combination of signals from gyroscopes and accelerometers. It can operate in a variety of environments, such as water, land, and air. The basic working principle of inertial guidance, based on Newton's laws of mechanics, measures the acceleration and angular velocity information of the carrier in the inertial reference system, integrates these measurements over time, transforms them into the coordinate navigation system, and finally obtains the information of velocity v , yaw angle yaw , and position x/y in the coordinate navigation system. It is usually combined with the GPS and fused with latitude and longitude information to provide accurate position information.

The INS has the following advantages:

1. As it does not depend on any external information and does not radiate energy to the outside, it is well concealed and not affected by external electromagnetic interference.
2. It can work in the air, on the Earth's surface, and even underwater all day and all the time.
3. It is capable of providing position, velocity, heading, and attitude angle data and results in good continuity of navigation information and low noise.
4. It has a high data update rate, short-term accuracy, and stability.

The disadvantages are as follows:

1. As the navigation information is generated by the integral, the localization error increases with time, the long-term accuracy is poor, and problems such as temperature drift and zero drift arise quickly.
2. A long initial alignment time is required before each use.
3. The equipment is expensive.

The Global Navigation Satellite System (GNSS), a navigation system based on GPS signals, is essential in unmanned vehicle localization systems. Ordinary GNSS devices can generally achieve localization accuracy within 10 m in the standard positioning service (SPS) localization mode. The localization accuracy may fluctuate within a specific range because of

weather, ionosphere, cloud cover, and solar activity changes. In the urban environment, especially in densely populated megacities and dense high-rise environments, the localization accuracy can reach the 10–100 m range. In addition to the above SPS, other localization services include the precise localization service for military assistance, satellite-based augmentation system that uses satellites as reference stations, ground-based augmentation system that uses ground-based base stations as reference stations, and differential global localization system that uses the differential correction of the reference station to improve the localization accuracy.

At present, many unmanned vehicle companies, including Baidu, Jingchi, and Pony, generally use real-time kinematic (RTK) GPS + INS and other localization methods, such as the RTK method, real-time dynamic differential method, and carrier phase differential technology. The RTK method is a new standard GPS measurement approach. The previous static, fast static, and dynamic measurements need to be solved afterward to obtain centimeter-level accuracy. At the same time, RTK is a measurement method that can achieve centimeter-level localization accuracy in the field in real time. It uses the carrier phase dynamic real-time differential method. RTK technology is built on the basis of the real-time processing of the carrier phase of two measurement stations. It can provide real-time three-dimensional coordinates of observation points with centimeter-level accuracy. However, in the urban environment with dense and tall buildings, localization errors can still reach 10–50 m. The error mainly results from the blockage of RTK GPS signals by structures, reflection and diffraction, and dense and tall buildings. The worse the localization signal is, the larger the localization error will be.

3.4.1 Localization principle

The principle of GPS localization is easy to understand, mainly using the triangulation method. The principle is shown in Fig. 3.8.

Three satellites form a triangle, and another satellite provides timing calibration. With these satellites, a vehicle's current satellite coordinate position can be calculated by computing the geometric data of the three satellite positions and fusing the results of the synchronization calculations.

According to these descriptions, the unmanned vehicle's position can be calculated by obtaining the parameters of the satellite geometric plane and the radio propagation time. However, in practical engineering applications, the process is much more complicated than such theoretical calculations. In

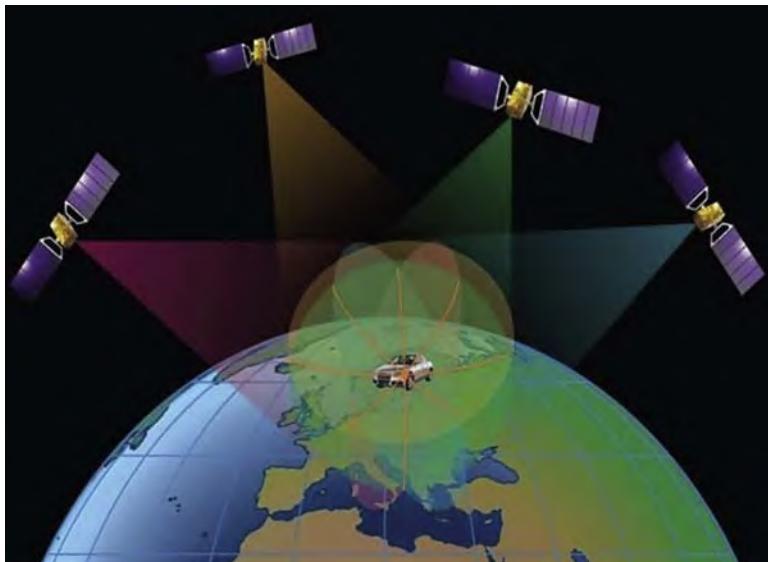


Figure 3.8 Triangulation principle.

actual engineering applications, the satellite signal propagation is also affected by the reflection of the atmospheric ionosphere, cloud reflection and refraction, and the reflection and refraction of signals by trees and tall buildings. All these factors affect GPS signal propagation and the accuracy of ranging information.

To reduce the influence of weather and clouds on GPS signals, other GPS technologies, such as differential GPS (DGPS), have emerged. This technology improves localization accuracy by installing a GPS monitoring receiver at a precisely known location (reference station), calculating the distance between the reference station and the GPS satellite, and then correcting the result according to the error.

DGPS is divided into two major categories: position differential and distance differential. Distance differential is divided into two categories: pseudo-range differential and carrier phase differential. RTK technology is a differential method for the real-time processing of carrier phase observations from two stations. The carrier phase collected by the reference station is sent to the user receiver, and the difference is solved for the coordinates. The carrier phase differential system can make the localization accuracy reach centimeter levels. Hence, many unmanned vehicle companies adopt RTK technology for positioning. As a result of the extremely

high cost of hardware equipment, the technology of RTK localization is not highly feasible for mass production and commercial use at present.

Thus far, we have come to understand the principle of GPS positioning. The update frequency of GPS device signals is generally low at the 1–20 Hz level. In some cases, the signal update frequency could be more than 50–100 Hz. Most of these frequencies are based on the interpolation algorithm, and the data are made up in the time gap and do not provide a real sense of signal generation data. For a vehicle moving at high speed, if the rate is 100 km/h, then the distance the car travels per second is about 28 m. If the update frequency of the GPS is 1 Hz, then the next GPS signal will be received after the vehicle has traveled 28 m. Such a low update frequency can lead to safety problems in unmanned vehicle driving. Although the update frequency of these INS devices is high, usually around 50–200 Hz, we can obtain the speed, acceleration, heading angle, and other vehicle information through INS devices and then obtain the position information through the integral operation of time. For a vehicle running at 100 km/h, if the frequency of the GPS localization signal is increased to the level of 100 Hz and the localization result is output once every 10 ms, then the position information needs to be updated about once every 28 cm to ensure the minimum basic safety requirements for automatic driving. Therefore, the introduction of INS equipment has a critical role in the improvement and facilitation of localization frequency.

3.4.2 Localization fusion of different sensors

This section introduces how GPS can be combined with INS to achieve positioning and some methodological techniques for data fusion.

At this point, we now understand the basic GPS operation principle. Generally, the signal output from GPS devices mainly includes longitude, latitude, altitude, and some control information indicating time synchronization, signs, etc. Through the latitude and longitude coordinates, the longitude and latitude information can be converted to projected plane coordinates x , y , z , and other position information according to the needs of the scene through coordinate systems, such as the ECEF (geocentric ground solid coordinate system) and UTM (horizontal Mercator). Thus, the plane position information can be obtained.

The INS device's primary data output contains acceleration in the x , y , and z directions, as well as the corresponding rotational angular velocity information. Through the time integration operation of these measured

values, we can obtain the velocity information in the x , y , and z directions and the relative position and angular information, such as roll, pitch, and yaw.

With x , y , z (two sets, GPS and INS outputs); vx , vy , vz ; and roll, pitch, yaw, we have at least nine state vectors to represent a vehicle's attitude and position information at this time. With these state vectors, we can do the data fusion of the state vectors according to the principle of the extended Kalman algorithm. The principle of the extended Kalman algorithm is not repeated here. The basic principle, based on the continuous input measurements and Bayesian probability, is to update the calculation of the current target's attitude position and other information. For the x , y , and z (two groups) variables mentioned earlier, we know that these two data groups are more reliable than GPS because the inertial measurement unit (IMU) has a significant and large drift as time accumulates and the IMU heats up. Hence, it has a more substantial impact on localization accuracy (except for the IMU with very high precision). We can use the x , y , and z of the GPS output as the primary position state vector reference and give lower weights to the x , y , and z of the IMU as the secondary position state vector reference. Instead of using the x , y , and z measurements of the IMU directly, they can be, respectively, calculated from dx/dt , dy/dt , and dz/dt for the reasons described earlier because we know that their relative rate changes are highly accurate during this period. By establishing the extended Kalman equation and solving the data output time synchronization of different devices, we can obtain the data fusion results between two other machines and calculate a more accurate state vector fusion result, that is $[x, y, z, vx, vy, vz, \text{roll}, \text{pitch}, \text{yaw}]$, thus obtaining the vehicle's current new multisensor fusion-based position and attitude information. This feature indicates the essence of the multisensor localization fusion principle.

Similarly, we can also add other sensor devices, such as cameras, ultrasonic devices, odometers, and wheel speed meters, because all these sensors have their advantages and disadvantages. They can provide measurements of different dimensional measures. By fusing the sizes of different dimensions of multiple sensors, we can calculate a reliable set of vehicle attitude data. This capability explains why multisensor fusion can improve localization accuracy.

A brief flow diagram of the system state vector fusion is shown in Fig. 3.9.

As shown in Fig. 3.10, the above method can also be extended to the fusion of other sensors⁹, each of which can provide measurements in various dimensions and ultimately calculate the final estimated pose and position of the vehicle through the fusion logic described in Fig. 3.9.

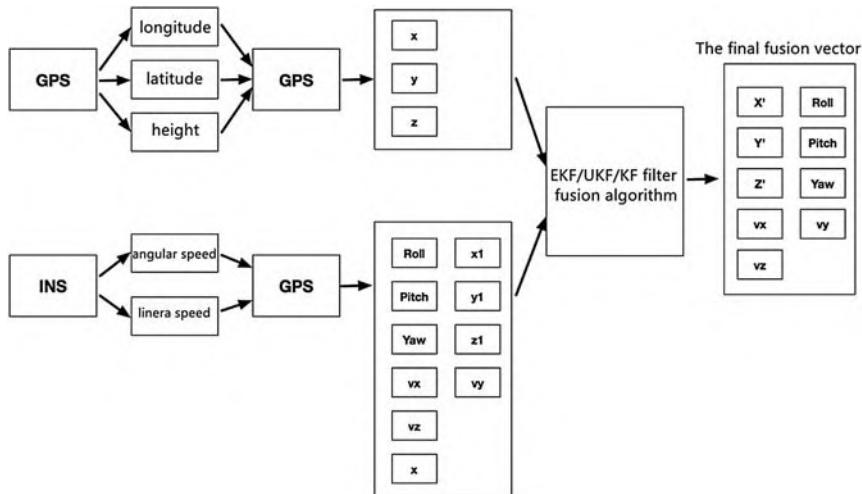


Figure 3.9 Multi-sensor localization fusion block diagram.

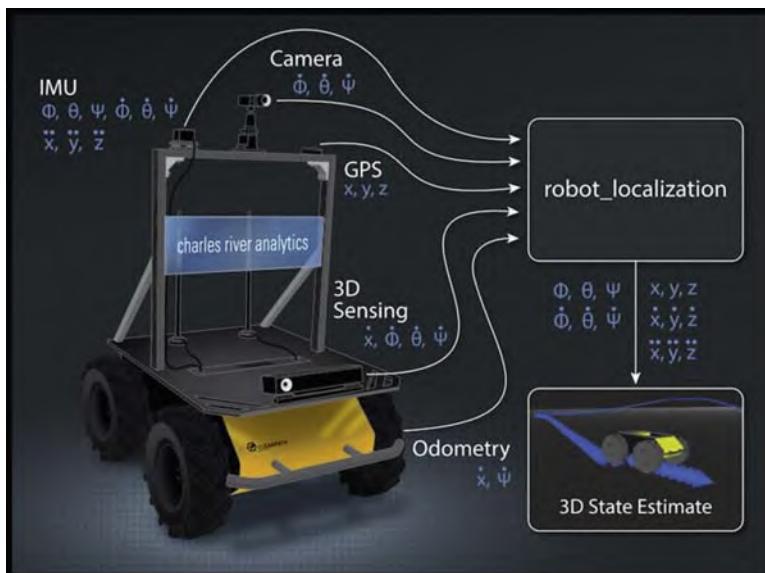


Figure 3.10 Multisensor fusion localization example.

3.5 SLAM-based localization system

SLAM is a complex system-level concept and does not refer to a specific algorithm. It includes image matching processing modules, filtering processing, closed-loop detection, graph optimization theory, and matrix operations. It is a complex engineering system (Fig. 3.11), and this book only provides a preliminary introduction to the basic principles and applications of SLAM.

The SLAM map schematic is shown in Fig. 3.12. The black border is the edge of the obstacle detected by LiDAR, indicating that this road is not accessible. The gray and white area is the free area for driving. The radial-like lines suggest the presence of windows or doors. Some of the LiDAR points are scattered out. By scanning the entire environmental space, a 2D map of the LiDAR view can be formed. By matching and comparing the environments, the robot or vehicle uses the result to determine where it is currently located on the map. The green line is a representation of the route planned and driven by the robot and vehicle.

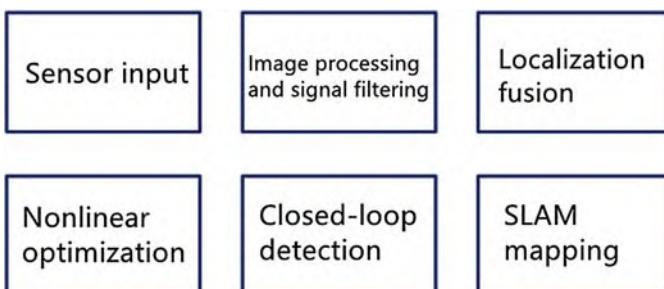


Figure 3.11 SLAM system module.



Figure 3.12 SLAM diagram.

SLAM is mainly used in robotics, where a map is constructed by scanning environmental data points with a LiDAR or vision camera, and a vehicle localizes itself on the basis of map matching. It is common in autonomous driving applications in low-speed scenarios, such as unmanned cleaning vehicles, unmanned ferries in low-speed parks, and unmanned delivery vehicles. For high-speed autonomous driving, SLAM is not suitable for application to large areas and high-speed autonomous driving scenarios because of the substantial computational overhead, time delay, data storage, and other issues based on a grid for calculation, as well as the high-performance requirements of unmanned vehicles for real-time control and safety. High-speed autonomous driving uses HD maps in map positioning. This section and the following are brief descriptions of SLAM's application in the field of robotics and low-speed autonomous driving scenarios.

3.5.1 SLAM localization principle

At present, the mainstream SLAM technology mainly adopts two technical routes. One is SLAM based on LiDAR point cloud data and the more well-known algorithm frameworks publicly available, including G mapping and Hector SLAM. The advantage of this SLAM technology is the high accuracy of building map measurement. However, the cost of LiDAR is too high, and the feasibility of mass production commercialization is low. The other technology is vision camera-based SLAM that is developing in parallel. The well-known algorithm frameworks publicly available are ORB-SLAM2, MonoSLAM, PTAM, LSD-SLAM, DSO, and so on. This SLAM technology requires low sensor cost, but its accuracy in map building is slightly low, and its interference by light and environment is high.

Visual SLAM is divided into monocular and binocular SLAM according to the number of cameras used. Monocular SLAM is low cost but has low accuracy because of the inability to measure depth and scale and other issues. Binocular SLAM can calculate depth information after system calibration. Therefore, in terms of robustness and reliability, binocular SLAM is somewhat better than monocular SLAM. Generally, visual SLAM is used in combination with IMU sensors to improve map building accuracy and pose estimation accuracy to a great extent.

With the development of deep learning and artificial intelligence (AI) technology in recent years, some SLAM techniques have combined AI, deep learning, target detection, semantic segmentation, and other processes

in the field of SLAM, such as semantic SLAM. These methods can obtain rich semantic information from images. This semantic information can help infer geometric information. For example, the size of a known object is an essential geometric cue.

The basic principle of SLAM positioning is described. Typically, robots have drift problems because of movement uncertainty. A sound SLAM system can handle the environment's delay and the fate of the robot's trajectory.

As shown in Fig. 3.13, suppose a robot's motion body moves 10 m forward in the X-axis direction from the origin x_0 (0,0). Theoretically, its position should be (10,0). However, in engineering practice, the robot does not arrive at (10,0) accurately because of inaccuracies in the measurement equipment, such as inaccurate gyroscope measurements, wheel slippage, and zero drift of the measurement equipment. The robot arrives at positions (9.8,0) or (10.1,0) because of the inaccuracy of the motion of the moving body. In fact, in terms of mathematical theory, the position variable of X_1 fits a Gaussian distribution centered at (10,0) with a variance δ that could be 0.1 or 0.2. If the robot continues to move forward without correction, then the error will increase until the system fails.

We want to make the position of x_1 as accurate as possible when the initial value x_0 is given with position (0,0). It is a calculation of how to maximize the likelihood estimate of the x_1 work. By introducing a new reference, the reliability of the reference position can be increased, as shown in Fig. 3.14. For an analogous description, consider a person walking in a desert in the middle of nowhere; with a big tree as a reference, also known as a landmark, this person can know their current location and how far they have walked. By constantly observing the milestone and updating the current position in real time, the present position error can be corrected. However, as the landmark position is lost, the localization error gradually becomes extensive. Thus, we need to maximize the likelihood estimate of the current situation by continuously iterating the new landmark observations.

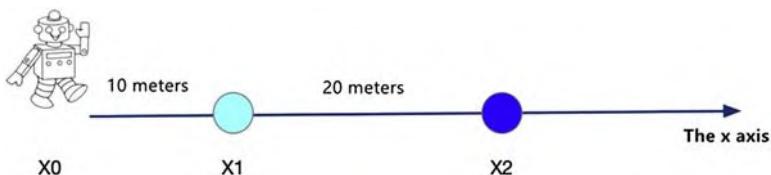


Figure 3.13 Schematic of SLAM principle 1.

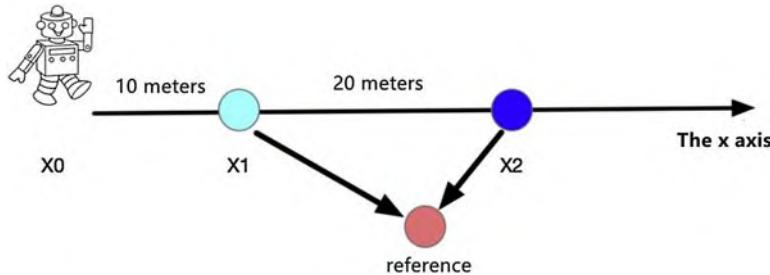


Figure 3.14 Schematic of SLAM principle 2.

On the basis of the previous description, the constraint relationship equation is established as follows:

Initial position constraint equation: $x_0 = 0$.

Equation of motion constraint: $x_2 - x_1 = 10$

Observation constraint equation: $x_1 - L_0 = 20$

We compose the above constraint equation conditions in the order of each state vector space into the state vector—matrix expression and then update the expression according to the new measurement at each step to express the localization relationship. This process is known as the principle of SLAM localization. Therefore, SLAM relies on three constraints:

Initial position constraint: the constraint on the initial position of x_0

Motion state constraints: constraints based on the initial position conditions and position expressions x_1 , x_2 , x_3 , etc.

Observation constraints: landmark location expressions

SLAM is based on the above three constraints and can be combined with algorithms such as the extended Kalman filter and example filter for localization. The initial position conditions of all the road signs and robots are first obtained as preconditions. Then, the position matrix equation is used to store the individual position coordinates of each waypoint and robot. Finally, the position update is performed by continuously iterating the update to reduce the position error.

3.5.2 SLAM applications

Many open-source visual SLAM (VSLAM) schemes exist and are classified according to their characteristics.

- Sparse method SLAM: ORB-SLAM, PTAM, MonoSLAM
- Semidense method SLAM: LSD-SLAM, DSO, SVO
- Dense method SLAM: DTAM, DVO, RGBD-SLAM, RTAB-MAP, etc.

These SLAM schemes work well in the experimental or demo stage. However, once they are applied in actual engineering, various problems, such as robustness problems and accuracy problems, arise. It is essential to understand these SLAM schemes to remedy the defects further and improve the algorithms.

According to applications' characteristics, the current mainstream VSLAM is divided into two main categories: direct method and feature point method. According to the number and situation of hardware devices, it is further divided into three categories: monocular, binocular, and RGBD-SLAM.

In the following, we will further understand SLAM by analyzing where the common problems of VSLAM lie.

1. Speed issues: Regardless of the type of VSLAM, as long as it relies on the camera sensor, the transmission of video frame data must be considered. When the movement speed is too fast, the problem of blurred images can occur. Image matching is the core of VSLAM in establishing data association. If there are a large number of mismatches, then the localization problem cannot be solved. Therefore, once there are multiple critical frames that are blurred and mismatched, the whole system will fail. As a result of a large amount of vision processing data, hardware devices' processing capacity must be considered. This requirement also poses a challenge to the engineering application of hardware computing units.
2. Environmental issues: The operation of VSLAM also depends on the external environment to some extent. For example, suppose the external environment is a weak or no-texture situation (e.g., a white wall) and a repeated-texture condition (e.g., similar buildings); it is also tricky for the SLAM algorithm to continue working this time. The problem posed by soft textures is that there are few apparent features for VSLAM to extract. Thus, its feature values cannot be extracted, and the algorithm cannot work correctly. The problem introduced by repeated textures is that the similarity of the environment can cause a large number of mismatches and, therefore, wrong location associations, which can also make the localization invalid. For visual SLAM, another problem is that the environment visible to the camera must be primarily static. In other words, VSLAM is not sufficiently adaptive and robust to dynamic environments. In practical engineering applications, if most of the scene seen by the camera is moving, the camera will think that it is driving itself. Hence, once relative motions are abundant, VSLAM will believe that it is moving. At this time, significant problems will emerge with the system's robustness, and they will result in inaccurate localization.

3. Illumination problems: The issues currently faced by vision sensors all exist in VSLAM. For example, in alternating light and dark environments, VSLAM can easily cause matching error problems because of camera exposure such that imaging speed cannot keep up.

In the following, we briefly introduce the advantages and disadvantages of each SLAM so that the reader can have a deep understanding of the current SLAM system.

- i. Advantages and disadvantages of monocular SLAM: The advantage of monocular SLAM is that only one camera is needed. There is no need to consider significant errors in the calibration parameters caused by binocular cameras during long-term use or collision situations. The disadvantage is the uncertainty of scale, that is, the inability to measure depth.
- ii. Advantages and disadvantages of binocular SLAM: The advantage of binocular SLAM is that it easily obtains scene depth information without considering the problem of monocular scale uncertainty. The disadvantage is that it imposes requirements for the device's volume and that it necessitates baseline correction between two cameras.
- iii. Advantages and disadvantages of RGBD-VSLAM: The edge of RGBD-VSLAM is that the scene depth information can be obtained directly. When working online, the corresponding depth calculation work is reduced; that is, the computational workload is reduced. RGBD depth cameras are also considered to be the most promising vision sensors for future VSLAM. The disadvantage is the slightly high relative cost.
- iv. Advantages and disadvantages of the direct method and feature point method: The advantage of the feature point method is that it is more robust to fast motion than the direct method. The disadvantage of the feature point method is that its real-time performance is worse than that of the direct method because the corner point detection, description, and matching are time-consuming. The advantage of the direct approach is the fast processing speed and simple explanation and matching because the features are often gradient points and edges, and the build map features are denser than those in the feature point method. The disadvantage of the direct approach is the insufficient robustness for fast motion. Moreover, it is only suitable for short baseline matching. When optimizing iterative poses, the camera motion that is too fast can easily lead to excessively long baselines of adjacent frames, which lead to considerably large pixel deviations of adjacent structures and cause the pose solution to fall into an erroneous local optimum solution.

References

1. Dissanayake G, Durrant-Whyte H, Bailey T. A computationally efficient solution to the simultaneous localisation and map building (SLAM) problem. *IEEE Trans Robot Autom.* 2013;17(3):229–241.
2. PCL(Point Cloud Library): <http://pointclouds.org>.
3. Zhou C, Li Y, et al. The research on improvement of ICP algorithm for 3D point cloud. *Comp Technol Dev.* 2011;21(8):75–77. ISSN 1673-629X.
4. Iterative Closest Point (Point Cloud Library): http://pointclouds.org/documentation/tutorials/interactive_icp.php.
5. Magnusson M, Lilienthal A, Duckett T. Scan registration for autonomous mining vehicles using 3d-ndt. *J Field Robot.* 2007;24(10):803–827.
6. Normal Distribution: https://en.wikipedia.org/wiki/Normal_distribution.
7. Normal Distribution Transform (Point Cloud Library): <http://wwwpclcn.org/study/shownews.php?lang=cn&id=80>.
8. Takeuchi E, Tsubouchi T. *A 3-D scan matching using improved 3-D normal distributions transform for mobile robotic mapping.* 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems; 2006:3068–3073. <https://doi.org/10.1109/IROS.2006.282246>.
9. ROS robot_localization package: http://docs.ros.org/lunar/api/robot_localization/html/index.html.

This page intentionally left blank

CHAPTER 4

State estimation and sensor fusion

Zebang Shen¹, Yu Sun², Peng Zhi² and Rui Zhao²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

4.1	Kalman filter and state estimation	96
4.1.1	What is the Kalman filter?	96
4.1.2	Kalman filter	96
4.1.2.1	<i>Status forecast</i>	97
4.1.2.2	<i>Calculation of forecast error</i>	98
4.1.2.3	<i>Measurement error</i>	98
4.1.2.4	<i>Calculation of Kalman gain</i>	99
4.1.2.5	<i>Calculation of optimal estimate</i>	99
4.1.2.6	<i>Calculation of the error of the optimal estimate</i>	100
4.1.3	Kalman filter in autonomous vehicle sensing module	102
4.1.3.1	<i>Sensors for autonomous vehicle perception module</i>	102
4.1.3.2	<i>Kalman filter-based pedestrian localization estimation</i>	102
4.1.3.3	<i>Kalman filter pedestrian state estimation in Python</i>	105
4.2	Advanced motion modeling and EKF	115
4.2.1	Advanced motion models for vehicle tracking	115
4.2.2	EKF	118
4.2.2.1	<i>Jacobi matrix</i>	118
4.2.2.2	<i>Process noise</i>	121
4.2.2.3	<i>Measurement</i>	124
4.2.2.4	<i>Python implementation</i>	126
4.3	UKF	138
4.3.1	Movement model	139
4.3.2	Nonlinear processing/measurement models	140
4.3.3	Lossless transformation	140
4.3.4	Projections	141
4.3.4.1	<i>Prediction of sigma point</i>	142
4.3.4.2	<i>Predicted mean and variance</i>	142
4.3.5	Measurement updates	143
4.3.5.1	<i>Update status</i>	144
4.3.6	Summary	144
	References	144

4.1 Kalman filter and state estimation

4.1.1 What is the Kalman filter?

Applying object tracking and prediction is usually necessary to estimate and predict some attractive targets. Why do we make estimates? In an actual scene, we typically need to continuously observe and predict the movement and development of the target to derive a suitable decision for the current state. In estimating a feature of a target, such as the distance, the most direct approach is to use sensor measurement. However, given the measurement errors and noise, the measured values are completely reliable. We can use the method based on probability and statistics to analyze statistics and estimate state quantities. The Kalman filter is a state estimation algorithm that combines prediction (prior distribution) and measurement update (likelihood estimation).¹

A fundamental probability theory is introduced first as a priori knowledge under the Kalman filter algorithm. If the reader already has an understanding of this area, these essential contents can be skipped.

- A priori probability $P(X)$ refers to the probability that is deduced from existing empirical knowledge by relying only on subjective experience. For example, the probability of a coin appearing with heads and tails is taken to be half for each, that is, 50%.
- Posterior probability $P(X|Z)$ is the probability under relevant evidence or given conditions. An example is a probability that a car accident will occur under the situation in which the driver is drunk.
- Likelihood estimation $P(Z|X)$ refers to the probability of using an existing consequence to infer an inherent property. An example is the probability of a driver in a car accident being drunk.

Bayesian formula:

$$P(A|B) = \frac{P(A) \times P(B|A)}{P(B)}$$

$P(A|B)$ is the posterior probability, while $P(A)$ is the prior probability. The likelihood estimate $P(B|A)$ is estimated from empirical values. From the above equation, we can see that the posterior probability distribution is proportional to the prior distribution multiplied by the likelihood estimate.

4.1.2 Kalman filter

The Kalman filter is a recursive algorithm. Each recursion consists of two main steps. First, a prediction is calculated. Second, the predicted and

measured values are weighted and summed to obtain the optimal estimate. The determination of the weights is completed by three other steps. In sum, the Kalman filter consists of five steps.²

- Calculate the predicted value x'_k at the moment k on the basis of the optimal estimate \hat{x}_{k-1} at the moment $k - 1$.
- Calculate the error of the predicted value p'_k at the moment k from the error of the optimal estimate p_{k-1} at the moment $k - 1$.
- Calculate the Kalman gain K_k at the moment k according to the error of the predicted value p'_k and the measured value's error r at the moment k .
- Calculate the optimal estimate \hat{x}_k at the moment k combined with the measured value z_k and the Kalman gain K_k at the moment k on the basis of the predicted value x'_k .
- Calculate the optimal estimate \hat{x}_k at the moment k of the error p_k according to the error of the predicted value p'_k and the Kalman gain K_k at the moment k .

These steps correspond to the five formulas of the Kalman filter. Next, a specific example is used to illustrate the principle of the Kalman filter algorithm. Suppose a rocket recovery is to be performed. Before the rocket lands on the ground, we need to know the rocket's current height from the ground; otherwise, it will crash easily because of improper control. In this case, the concerned state variable is the aircraft's altitude from the ground. Hence, the aircraft's altitude from the ground is what needs to be estimated. Why do we need to estimate the state variable? It is not possible to accurately measure the vehicle's actual height above the ground in reality. The Kalman filter is used to solve this problem by going through the following steps.

4.1.2.1 Status forecast

First, a certain model is necessary to predict the height at the current moment. Assuming that with a fixed time interval sampling, the height at the moment k will become 95% of the height at the moment $k - 1$, then the following relationship can be obtained:

$$\text{Height}^{(k)} = 0.95 \times \text{Height}^{(k-1)}.$$

$\text{Height}^{(k-1)}$ denotes the true height at the moment $k - 1$, and $\text{Height}^{(k)}$ denotes the true height at the moment k . In reality, the true height at the moment $k - 1$ is not available, and the assumed model is not always

completely correct. Thus, the calculated Height^(k) is not the true value and is instead a predicted value with error. Height^(k-1) can be replaced by the optimal estimate \hat{x}_{k-1} at time $k - 1$. Then, the above expression can be abstracted into the following form:

$$x'_k = a\hat{x}_{k-1}.$$

This expression is the process model, which is an equation based on a physical model (e.g., some physical motion model, Newtonian mechanics). The process model reflects the motion state of the target object. The expression x'_k represents the predicted value at time k and is a constant coefficient equal to 0.95 in this example. That is, the optimal estimate at the moment $k - 1$ is used to calculate the predicted value at the moment k .

Consider a simple ratio (constant) that is used to describe the movement rule of the “recycling” rocket. What problems will occur?

Obviously, the height above the ground of a real rocket does not scale down as much as the simple process model described above. For the sake of simplicity, let us assume that this simple process model works and can roughly be described by the motion of this “magic rocket,” with only occasional deviations (e.g., influenced by air turbulence). Noise is then added to describe the difference between the process model and the actual motion when computing x'_k . This noise is called process noise, which is represented as w_k . Thus, the computational equation of x'_k becomes

$$x'_k = a\hat{x}_{k-1} + w_k.$$

For simplicity, the analysis that follows first ignores the treatment of noise, but it is reconsidered in the sensor fusion part.

4.1.2.2 Calculation of forecast error

The predicted values are subject to the error and are assumed to be p'_k obtained from the following equation:

$$p'_k = ap_{k-1}a^T.$$

The expression p_{k-1} is the error of the optimal estimate at the moment $k - 1$, and a^T denotes the transpose of a .

4.1.2.3 Measurement error

The measured value comes from the sensor measurement (e.g., GPS, barometer), and the result always comes with an error called noise, which is

caused by the precision of the sensor itself. The measured value is expressed by the following formula:

$$z_k = x_k + \nu_k.$$

In the formula, z_k denotes the measured value at the moment k . x_k denotes the true value at time k , and ν_k is the measurement noise. Generally, this noise is satisfied with a Gaussian distribution, which obeys the mean value of r and a normal distribution with variance δ . Although the measurement noise is not a known ν_k value, the mean value of the measurement noise can be obtained from measurement experiments or directly from the sensor manufacturer r .

4.1.2.4 Calculation of Kalman gain

The equation to calculate the Kalman gain is

$$K_k = p'_k / (p'_k + r),$$

where K_k denotes the Kalman gain. According to this expression, K_k takes values in the range $[0,1]$. The actual meaning of the Kalman gain will be given later in the analysis. p'_k is the error of the predicted value at the moment k , and r is the mean value of the measurement noise.

4.1.2.5 Calculation of optimal estimate

In reality, the real height of a rocket cannot be known, and only an estimate can be made. In the Kalman filter, the formula to calculate the optimal estimate is

$$\hat{x}_k = x'_k + K_k(z_k - x'_k),$$

where \hat{x}_k is the optimal estimate at the moment k . x'_k is the predicted value at the moment k calculated using the process model. z_k is the measured value at the moment k . K_k is the Kalman gain. This equation is transformed slightly as follows:

$$\hat{x}_k = (1 - K_k)x'_k + K_k z_k.$$

From this formula, the Kalman gain is actually a weight used to measure whether the measured value or the predicted value is important. For ease of understanding, two extreme examples are given. If $K_k = 0$, that is, the gain is 0, then at this point, the formula degenerates as follows:

$$\hat{x}_k = x'_k.$$

The result indicates that the current measurement is very untrustworthy. Thus, the predicted value given by the process model is used directly as the optimal estimate of the current state. If $K_k = 1$, that is, the gain is 1, then the formula would be as follows:

$$\hat{x}_k = z_k.$$

The result indicates that the current measurement is very plausible and is thus used directly as the optimal estimate of the current state. If K_k is between 0 and 1, then the predicted and measured values have some reliability.

4.1.2.6 Calculation of the error of the optimal estimate

The error of the optimal estimate is calculated by the formula

$$p_k = (1 - K_k)p'_k.$$

p_k denotes the error of the optimal estimate. K_k denotes the Kalman gain. p'_k denotes the error of the predicted value computed by the process model at the moment k . Given the estimated moment k , why do we need to calculate the error of the optimal estimate at the moment k p_k ? The answer is mentioned earlier, that is, the “Kalman filter is a recursive algorithm.” In $k + 1$, the calculation of the moment k requires the use of p_k to calculate p'_{k+1} . The following formula can be referred to:

$$p'_k = a p_{k-1} a^T.$$

This equation at the $k + 1$ moment becomes

$$p'_{k+1} = a p_k a^T.$$

In the equation, p_k is the error of the optimal estimate at the moment k calculated above.

These steps are necessary to solve the problem in the example with the Kalman filter. For the prediction, only a fixed process model and process noise are theoretically considered. However, as we are now estimating the state of the mechanical control, we need to model the mechanical control itself in the prediction process. Hence, we add another control signal that is denoted by $b u_k$ in the prediction part. The complete Kalman filter prediction and update process can be achieved by combining the aforementioned solutions.

The first is the prediction of states.

$$\begin{aligned}x'_k &= a\hat{x}_{k-1} + bu_k \\p'_k &= ap_{k-1}a^T\end{aligned}$$

The process of the Kalman filter update is as follows:

$$\begin{aligned}K_k &= p'_k / (p'_k + r), \\ \hat{x}_k &= x'_k + K_k(z_k - x'_k), \\ p_k &= (1 - K_k)p'_k.\end{aligned}$$

Using a linear algebraic approach to represent predictions and updates, the process of prediction then becomes as follows:

$$x'_k = A\hat{x}_{k-1} + Bu_k, \quad (4.1)$$

$$P'_k = AP_{k-1}A^T. \quad (4.2)$$

The process of updating is as follows:

$$G_k = P'_k C^T (C P'_k C^T + R)^{-1}, \quad (4.3)$$

$$\hat{x}_k = x'_k + G_k(z_k - Cx'_k), \quad (4.4)$$

$$P_k = (I - G_k C)P'_k. \quad (4.5)$$

The symbol C appears in Eqs. (4.3) and (4.4) because the matrices x'_k and matrix z_k may not have the same size. For example, x'_k contains the velocity v and position y . However, in reality, there is only the sensor measuring velocity v . The position y cannot be measured by the sensor; thus, z_k only contains the velocity v . At this moment, the sizes of x'_k and z_k are not the same so that the addition and subtraction operations cannot be performed. Hence, a matrix C should be imported to ensure the same values for z_k and Cx'_k and for x'_k and $G_k(z_k - Cx'_k)$. Thus, Eq. (4.4) can be established. I in Eq. (4.5) represents the unit matrix.

Why is the Kalman filter algorithm called a filter algorithm? Take the one-dimensional Kalman filter as an example. If one simply believes the measured signal, then the signal contains very hairy noise. However, when running the Kalman filter algorithm for estimation, the estimated signal will be smooth and seem to filter out the effect of noise. This characteristic is inherent in a filter algorithm. Kalman filter has been proved to achieve optimal estimation in linear problems.

4.1.3 Kalman filter in autonomous vehicle sensing module

4.1.3.1 Sensors for autonomous vehicle perception module

If autonomous vehicles were to be operated safely on the road, they need to have “ears to hear and eyes to see.” Thus, what are the ears and eyes of an autonomous vehicle? These parts correspond to the various sensors installed in the autonomous vehicle. The sensors on an autonomous vehicle can be deployed by the dozens, and they can be of different types, such as the following:

- Stereo Camera
- Traffic Sign Camera
- Radio Detection and Ranging (RADAR)
- Light Detection and Ranging (LIDAR)
- Inertial Measurement Unit (IMU)

Stereo cameras are often used to obtain image and distance information. Traffic sign cameras can be used to identify traffic signs on the basis of vision. RADARs are generally installed inside the front and rear bumpers of vehicles to measure moving objects relative to the vehicle coordinate system and locate and measure distance, speed, etc. However, it is easily disturbed by strong reflective objects. Thus, it is usually not used for the detection of stationary objects. LIDAR, generally installed on top of a vehicle, uses infrared laser beams to obtain the distance and position of an object. The advantage of LIDAR is its high spatial resolution and accurate measurement. The disadvantage is that the equipment is bulky and easily affected by heavy rain and foggy weather.

Various sensors have their own advantages and disadvantages. In an actual autonomous vehicle, data from multiple sensors are often combined to sense the vehicle’s surroundings and thereby provide reliable and stable environmental sensing information. This process of combining measurement data from various sensors to estimate a state is called sensor fusion. The later sections will give a detailed introduction to the application of the extended Kalman filter (EKF) and lossless Kalman filter in sensor fusion. In this section, the Kalman filter algorithm is mainly considered, particularly in the estimation of pedestrian and vehicle localization based on single sensor data.

4.1.3.2 Kalman filter-based pedestrian localization estimation

Although simple, the Kalman filter is a very important part of the technological system of autonomous vehicles. Of course, the techniques used in real autonomous vehicle projects are increasingly complex, but the basic

principles are still the same as those presented in this book. In driverless vehicles, the Kalman filter is mainly used for state estimation, for example, for pedestrians, bicycles, and other vehicles in the vicinity. In the following, we present the algorithm for pedestrian state estimation as an example.

Consider a scenario in which an unmanned vehicle is moving forward normally, and then a pedestrian suddenly appears in front of it. The autonomous vehicle, in this case, needs to make a prediction and judgment on the behavioral action of the pedestrian in front of it in order to adopt an appropriate obstacle avoidance or parking strategy. If we want to estimate the motion state of the pedestrian, then we first need to establish the expression of the state equation of the object being estimated. The state of a person can be expressed by the mathematical equation as $x = (p, v)$, where p is the current position of the pedestrian and v is the current velocity of the pedestrian. A state is represented by a vector as follows:

$$x = \begin{pmatrix} p_x, p_y, v_x, v_y \end{pmatrix}^T.$$

The above equation represents the position components of the pedestrian in the x - and y -directions p_x, p_y and the velocity components v_x, v_y . After determining the state of the object to be estimated, a process model for generating an estimate of the current state should also be established. In this work, the application of the Kalman filter algorithm is explained in terms of one of the simplest process models, that is, the constant velocity model (pedestrian velocity is assumed to be uniform first), assuming that the process model is

$$x_{k+1} = Ax_k + v,$$

which is extended as follows:

$$x_{k+1} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}_k + v.$$

It is called a constant velocity model because expanding this determinant above yields the following:

$$p_x^{k+1} = p_x^k + v_x^k \Delta t + v$$

$$\begin{aligned} p_y^{k+1} &= p_y^k + v_y^k \Delta t + \nu \\ v_x^{k+1} &= v_x^k + \nu \\ v_y^{k+1} &= v_y^k + \nu \end{aligned}$$

The constant velocity process model assumes that the movement of the predicted target has a constant velocity. In the problem of pedestrian state prediction, pedestrians do not necessarily move at a constant velocity. Thus, the process model also contains a certain amount of process noise, with ν also considered as the process noise of the model in this problem. The process noise in pedestrian state estimation is the sudden acceleration and deceleration of a pedestrian. Given the acceleration factor of the pedestrian, the original model expression becomes

$$x_{k+1} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}_k + \begin{bmatrix} \frac{1}{2} a_x \Delta t^2 \\ \frac{1}{2} a_y \Delta t^2 \\ a_x \Delta t \\ a_y \Delta t \end{bmatrix}_k.$$

Then, according to Eq. (4.2), the second step of the prediction becomes

$$A' = APA^T + Q.$$

Q is the covariance matrix of the process noise. This process is the state A update process, which is essentially the covariance matrix of the estimated state probability distribution. As process noise is brought in randomly, the process noise ν is essentially a Gaussian distribution: $\nu \sim N(0, Q)$. Q is the covariance matrix of the process noise, and the expansion of Q takes the following form:

$$Q = \begin{bmatrix} \sigma_{p_x}^2 & \sigma_{p_x p_y} & \sigma_{p_x v_x} & \sigma_{p_x v_y} \\ \sigma_{p_y p_x} & \sigma_{p_y}^2 & \sigma_{p_y v_x} & \sigma_{p_y v_y} \\ \sigma_{v_x p_x} & \sigma_{v_x p_y} & \sigma_{v_x}^2 & \sigma_{v_x v_y} \\ \sigma_{v_y p_x} & \sigma_{v_y p_y} & \sigma_{v_y v_x} & \sigma_{v_y}^2 \end{bmatrix}.$$

If we define $G = [0.5\Delta t^2, 0.5\Delta t^2 \cdot \Delta t, \Delta t]^T$, the expression of the formula is simplified as follows:

$$Q = G \cdot G^T \cdot \sigma_v^2.$$

σ_v^2 : For a pedestrian, assume that his velocity is approximately 0.5 m/s^2 .

In the measurement step, the speed of the pedestrian can be measured directly using v_x and v_y acquired by the sensor. Thus, according to the state expression, the measurement matrix C can be expressed as follows:

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The covariance matrix of the measurement noise R is

$$R = \begin{bmatrix} \sigma_{v_x}^2 & 0 \\ 0 & \sigma_{v_y}^2 \end{bmatrix}.$$

In the above formula, $\sigma_{v_x}^2$ and $\sigma_{v_y}^2$ describe how “poor” the sensor’s measurements can be. This characteristic is natural for the sensor and is therefore often provided by the sensor manufacturer. Finally, P_k is calculated as follows:

$$P_k \leftarrow (I - G_k C) P_k.$$

At this point, the whole process of the constant velocity model-based process modeling shown in Fig. 4.1 and the Kalman filter-based pedestrian state estimation is complete. The process is implemented in Python.

4.1.3.3 Kalman filter pedestrian state estimation in Python

We summarize the whole process of the Kalman filter. In some papers and materials, the state transfer matrix is usually represented with F (A in previous paper). The observation matrix is usually expressed using H (C in the previous paper), and the Kalman gain is usually denoted by K (G in the previous paper). The following shows the essence of the Kalman filter process in the literature.

Note that the formula also contains Bu_k , which means that the internal control of an object is considered when tracking its state. However, Bu_k in the problem of estimating the state of pedestrians, bicycles, and other cars is not measurable. Thus, Bu_k is simplified as 0 in this problem.

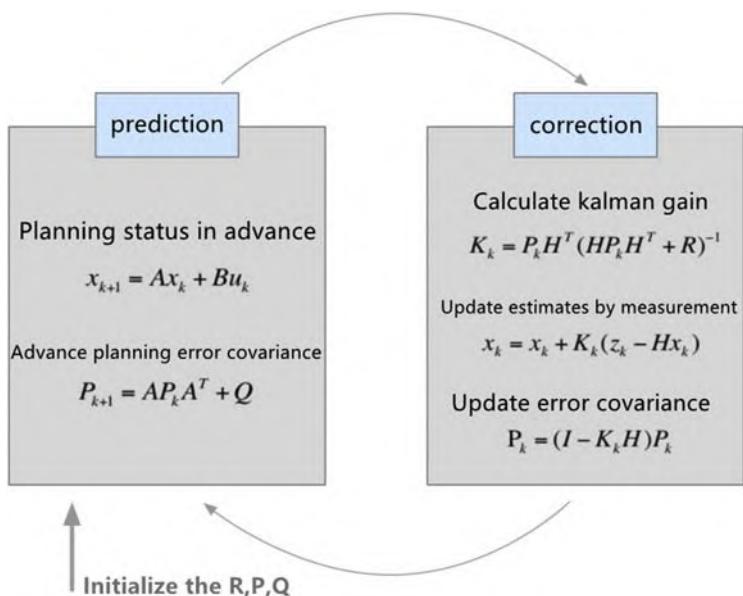


Figure 4.1 Kalman filter process.

The following code is executed in Jupyter notebook, a Python interactive editor that can directly run and display edited codes. This editor features instant operation and display. The necessary libraries are first loaded as follows:

Code list 4.1.1 Load the necessary libraries

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from scipy.stats import norm
```

Next, the pedestrian state x is initialized (from the previous section, the pedestrian state includes state variables, such as position and velocity in the x - and y -directions), along with the pedestrian uncertainty factor (the a priori estimated covariance matrix). The measured time interval is denoted as dt ; the state transfer matrix as F ; and the observation matrix as H . The code implementation is as follows.

Code list 4.1.2 Process matrix F and measurement matrix H

```

x = np.matrix([[0.0, 0.0, 0.0, 0.0]]).T
print(x, x.shape)

P = np.diag([1000.0, 1000.0, 1000.0, 1000.0])
print(P, P.shape)

dt = 0.1 # Time Step between Filter Steps

F = np.matrix([[1.0, 0.0, dt, 0.0],
               [0.0, 1.0, 0.0, dt],
               [0.0, 0.0, 1.0, 0.0],
               [0.0, 0.0, 0.0, 0.0, 1.0]])

print(F, F.shape)

H = np.matrix([[0.0, 0.0, 1.0, 0.0],
               [0.0, 0.0, 0.0, 0.0, 1.0]])

print(H, H.shape)

ra = 10.0**2

R = np.matrix([[ra, 0.0],
               [0.0, ra]])
print(R, R.shape)

```

The covariance matrices of the measurement noise R and process noise Q are calculated as follows.

Code list 4.1.3 Process matrix Q and measurement matrix R

```

ra = 0.09

R = np.matrix([[ra, 0.0],
               [0.0, ra]])

print(R, R.shape)

sv = 0.5

G = np.matrix([[0.5*dt**2],
               [0.5*dt**2],
               [dt],
               [dt]]))

Q = G*G.T*sv**2

from sympy import Symbol, Matrix

from sympy.interactive import printing

printing.init_printing()

dts = Symbol('dt')

Qs = Matrix([[0.5*dts**2], [0.5*dts**2], [dts], [dts]]))

Qs*Qs.T

```

Code list 4.1.4 Define a unit matrix

```

I = np.eye(4)

print(I, I.shape)

```

Some random measurements are generated.

Code list 4.1.5 Generate some random measurement data array

```
m = 200 # Measurements

vx= 20 # in X

vy= 10 # in Y

mx = np.array(vx+np.random.randn(m))

my = np.array(vy+np.random.randn(m))

measurements = np.vstack((mx,my))

print(measurements.shape)

print('Standard Deviation of Acceleration Measurements=% .2f' %

np.std(mx))

print('You assumed %.2f in R.' % R[0,0])

fig = plt.figure(figsize=(16,5))

plt.step(range(m),mx, label='$\dot{x}$')

plt.step(range(m),my, label='$\dot{y}$')

plt.ylabel(r'Velocity $m/s$')

plt.title('Measurements')

plt.legend(loc='best',prop={'size':18})
```

Here are some process values for the display of results.

Code list 4.1.6 Process value

```
xt = []
yt = []
dxt= []
dyt= []
zx = []
zy = []
px = []
py = []
Pdx= []
Pdy= []
Rdx= []
Rdy= []
Kx = []
Ky = []
Kdx= []
Kdy= []

def savestates(x, z, p, r, k):
    xt.append(float(x[0]))
    yt.append(float(x[1]))
    dxt.append(float(x[2]))
    dyt.append(float(x[3]))
    zx.append(float(z[0]))
```

```
Zy.append(float(Z[1]))  
  
Px.append(float(P[0,0]))  
  
Py.append(float(P[1,1]))  
  
Pdx.append(float(P[2,2]))  
  
Pdy.append(float(P[3,3]))  
  
Rdx.append(float(R[0,0]))  
  
Rdy.append(float(R[1,1]))  
  
Kx.append(float(K[0,0]))  
  
Ky.append(float(K[1,0]))  
  
Kdx.append(float(K[2,0]))  
  
Kdy.append(float(K[3,0]))
```

Code list 4.1.7 Kalman filter

```
for n in range(len(measurements[0])):

    # Time Update (Prediction)
    # =====
    # Project the state ahead

    x = F*x

    # Project the error covariance ahead

    P = F*P*F.T + Q


    # Measurement Update (Correction)
    # =====
    # Compute the Kalman Gain

    S = H*P*H.T + R

    K = (P*H.T) * np.linalg.pinv(S)

    # Update the estimate via z

    Z = measurements[:,n].reshape(2,1)

    y = Z - (H*x)                      # Innovation or Residual

    x = x + (K*y)

    # Update the error covariance

    P = (I - (K*H)) * P

    # Save states (for Plotting)

    savestates(x, Z, P, R, K)
```

In the above code listing, the variable x is shared by x'_k , \hat{x}_{k-1} and \hat{x}_k in the five formulas of the Kalman filter while the variable P is shared by P'_k , P_k and P_{k-1} of the five Kalman filter formulas. This setup saves storage space without affecting the calculation results.

Code list 4.1.8 Display the estimated results relative to the speed

```
def plot_x():

    fig = plt.figure(figsize=(16,9))

    plt.step(range(len(measurements[0])),dxt, label='$estimateVx$')

    plt.step(range(len(measurements[0])),dyt, label='$estimateVy$')

    plt.step(range(len(measurements[0])),measurements[0],
label='$measurementVx$')

    plt.step(range(len(measurements[0])),measurements[1],
label='$measurementVy$')

    plt.axhline(vx, color='#999999', label='$trueVx$')

    plt.axhline(vy, color='#999999', label='$trueVy$')

    plt.xlabel('Filter Step')

    plt.title('Estimate (Elements from State Vector $x$)')

    plt.legend(loc='best',prop={'size':11})

    plt.ylim([0, 30])

    plt.ylabel('Velocity')

plot_x()
```

The predicted results are shown in Fig. 4.2.

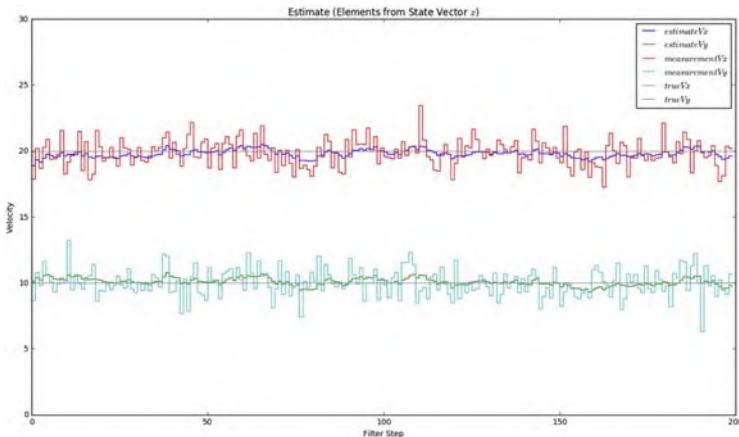


Figure 4.2 Predicted results.

Code list 4.1.9 Estimation results of localization

```
def plot_xy():
    fig = plt.figure(figsize=(16,16))

    plt.scatter(xt,yt, s=20, label='State', c='k')

    plt.scatter(xt[0],yt[0], s=100, label='Start', c='g')

    plt.scatter(xt[-1],yt[-1], s=100, label='Goal', c='r')

    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Position')
    plt.legend(loc='best')
    plt.axis('equal')

plot_xy()
```

A simple Kalman filter is implemented. However, the basic principle is similar to the one introduced in the autonomous vehicle sensory fusion

module. In this section, we only explain the basic form of the Kalman filter and not the form used in the actual autonomous vehicle project.

Although this example uses pedestrian detection in driverless applications as an application point to explain the Kalman filter algorithm, autonomous vehicles in practice do not use the original Kalman filter alone for pedestrian state estimation because the Kalman filter has a very significant limitation. It can only provide accurate estimation for linear systems. Thus, the following section introduces the EKF, which is a Kalman filter algorithm that can be applied to nonlinear systems, to help illustrate how to solve the above problem.

4.2 Advanced motion modeling and EKF

This section focuses on the **EKF** algorithm, which is widely used in nonlinear systems and is usually applied to practical vehicle state estimation (or vehicle tracking). The actual vehicle tracking motion model cannot be modeled using a simple constant velocity model. Several additional advanced motion models applied to vehicle tracking are instead presented in this section. The constant turn rate and velocity (CTRV) model is used to explain the application of the EKF algorithm. Finally, the code sample in the following part presents how to perform multisensor fusion using the EKF algorithm.

4.2.1 Advanced motion models for vehicle tracking

The first idea to clarify is that regardless of the motion model, it is essentially designed to help simplify a problem. Hence, commonly used motion models can be classified according to their complexity (frequency).

Linear motion model

- Constant velocity (CV) model
- Constant acceleration model

These linear motion models are based on the premise that the target is assumed to move in a straight line without considering the turning of the object.

Nonlinear motion model

- CTRV model
- Constant turn rate and acceleration model

The CTRV model is currently used in airborne tracking systems (aircraft). Most of these nonlinear motion models assume that the velocity v

and angular yaw rate ω are independent of each other. Therefore, in such motion models, the actual measured angular velocity can change slightly even if the vehicle is not moving because of the perturbation (instability) of the yaw velocity measurement.

Under the assumption that the steering angle Φ is constant, the relation between velocity v and yaw rate ω can be established by importing the constant steering angle and velocity model. In addition, the speed can be assumed to vary linearly, resulting in the constant curvature and acceleration (CCA) model. These motion models are interconvertible, and their relationships are shown in Fig. 4.3.³

The state transfer formula is a specific mathematical representation of the motion model. Except for CCA, the above motion models are very popular. This section focuses on the state transfer formula of the CV and CTRV models. The state transfer formula is the formula for the process model to calculate the prior distribution of the next state from the estimation of the previous state. It can be understood as the motion formula summarized on the basis of certain prior knowledge. We need to first define the state space in the model, that is, the number of states examined by the model. The state space of the CV model can be expressed as follows:

$$\vec{x}(t) = (x, y, v_x, v_y)^T.$$

(x, y) denotes the position of the target, and (v_x, v_y) denotes the velocity on the direction of the target relative to the (x, y) coordinate. As the CV model already assumes constant velocity, the acceleration is not needed

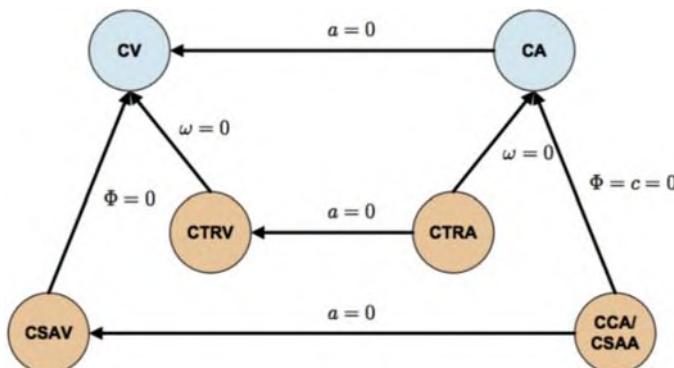


Figure 4.3 Operational model relationship diagram.

in the state space, from which the state transfer function of the CV model can be deduced as follows:

$$\vec{x}(t + \Delta t) = \begin{pmatrix} x(t) + \Delta t v_x \\ y(t) + \Delta t v_y \\ v_x \\ v_y \end{pmatrix}.$$

The constant rate of rotation and velocity model is also known as the constant angular velocity–velocity model, which, as the name implies, also assumes that the velocity and angular velocity are constant. In the CTRV model, the state quantities of the target are as follows:

$$\vec{x}(t) = (x, y, v, \theta, \omega)^T.$$

In the above formula, θ is the yaw angle, which is the angle in the range $[0, 2\pi)$ between the tracked target vehicle and the x-axis in the current vehicle coordinate system that is positive in the counterclockwise direction. ω is the yaw angular velocity. The state transfer function of the CTRV model is as follows:

$$\vec{x}(t + \Delta t) = \begin{pmatrix} \frac{v}{\omega} \sin(\omega \Delta t + \theta) - \frac{v}{\omega} \sin(\theta) + x(t) \\ -\frac{v}{\omega} \cos(\omega \Delta t + \theta) + \frac{v}{\omega} \cos(\theta) + y(t) \\ v \\ \omega \Delta t + \theta \\ \omega \end{pmatrix}.$$

The following part of this section uses the CTRV model as the motion model. One problem with the CTRV state transfer equation is that when $\omega = 0$, the denominator in the above state transfer equation is 0. To solve this problem, imagine $\omega = 0$, which actually represents that the tracked vehicle is actually traveling in a straight line at this time. Hence, the above equation is simplified as follows:

$$\begin{aligned} x(t + \Delta t) &= v \cos(\theta) \Delta t + x(t), \\ y(t + \Delta t) &= v \sin(\theta) \Delta t + y(t). \end{aligned}$$

Another problem with using these complex motion models is that the Kalman filter is only used to deal with linear problems. Clearly, the process model is now nonlinear, which means that the Kalman filter cannot be simply used for prediction and updating. The first step in the prediction equation becomes the following nonlinear functional expression:

$$x_k = g(x_{k-1}, u),$$

where the function $g()$ denotes the state transfer function of the CTRV motion model, and u denotes the control input. To solve the problem under the nonlinear system, we introduced the EKF algorithm.

4.2.2 EKF

4.2.2.1 Jacobi matrix

The essence of the EKF is to use a linear transformation to approximate a nonlinear transformation. Specifically, the EKF algorithm is linearized using a first-order Taylor expansion according to the following Taylor expansion formula:

$$h(x) \approx h(u) + \frac{\partial h(u)}{\partial x}(x - u).$$

In mathematics, Taylor's formula describes the values of a function taken in its neighborhood in terms of information about it at a point. If the function is smooth enough, that is, the values of the derivatives of each order of the function at a point are known, then Taylor's formula can construct a polynomial using these derivative values as coefficients to approximate the value of the function about that point. Taylor's formula also gives the deviation between this polynomial and the actual value of the function.

Returning to the process model mentioned earlier, the new expression for the state transfer function is as follows:

$$\vec{x}(t + \Delta t) = g(x(t)) = \begin{pmatrix} \frac{v}{\omega} \sin(\omega \Delta t + \theta) - \frac{v}{\omega} \sin(\theta) + x(t) \\ -\frac{v}{\omega} \cos(\omega \Delta t + \theta) + \frac{v}{\omega} \cos(\theta) + y(t) \\ v \\ \omega \Delta t + \theta \\ \omega \end{pmatrix}, \omega \neq 0,$$

$$\vec{x}(t + \Delta t) = g(x(t)) = \begin{pmatrix} v \cos(\theta)\Delta t + x(t) \\ v \sin(\theta)\Delta t + y(t) \\ v \\ \omega\Delta t + \theta \\ \omega \end{pmatrix}, \omega = 0.$$

Using a multivariate Taylor series, this multivariate function is expanded as follows:

$$T(x) = f(u) + (x - u)Df(u) + \frac{1}{2!}(x - u)^2 D^2f(u) + \dots$$

$Df(u)$ is called the Jacobi matrix, which is a matrix composed of the first-order partial derivatives of each dependent variable in the multivariate function with respect to each independent variable.

$$J = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \dots & \frac{\partial f}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

In vector calculus, a Jacobi matrix is a matrix composed of first-order partial derivatives in a certain way, and its determinant is called the Jacobi determinant. The importance of the Jacobi matrix is that it represents the optimal linear approximation of a differentiable equation to a given point. Thus, the Jacobi matrix is similar to the derivative of a multivariate function.

In the EKF, as $(x - u)$ itself is small enough, the value of $(x - u)^2$ is extremely small. Thus, higher-order levels are neglected in this problem, and only linearized approximations using first-order Jacobi matrices are considered.

The next step is to solve the Jacobi matrix, which can be obtained by taking the partial derivatives of each element in the CTRV model ($\omega \neq 0$).

$$J_A = \begin{bmatrix} 1 & 0 & \frac{1}{\omega}(-\sin(\theta) + \sin(\Delta t \omega + \theta)) & \frac{\nu}{\omega}(-\cos(\theta) + \cos(\Delta t \omega + \theta)) & \frac{\Delta t \nu}{\omega} \cos(\Delta t \omega + \theta) - \frac{\nu}{\omega^2}(-\sin(\theta) + \sin(\Delta t \omega + \theta)) \\ 0 & 1 & \frac{\nu}{\omega}(-\sin(\theta) + \sin(\Delta t \omega + \theta)) & \frac{1}{\omega}(\cos(\theta) - \cos(\Delta t \omega + \theta)) & \frac{\Delta t \nu}{\omega} \sin(\Delta t \omega + \theta) - \frac{\nu}{\omega^2}(\cos(\theta) - \cos(\Delta t \omega + \theta)) \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

When $\omega = 0$, the Jacobi matrix is simplified as follows:

$$J_A = \begin{bmatrix} 1 & 0 & \Delta t \cos(\theta) & -\Delta t v \sin(\theta) & 0 \\ 0 & 1 & \Delta t \sin(\theta) & \Delta t v \cos(\theta) & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

In the Python implementation that follows, the numdifftools library is used to compute the Jacobi matrix directly without rewriting the code. After obtaining the Jacobi matrix for the CTRV model, the new process model can be written as follows:

$$\begin{aligned} x_k &= g(x_{k-1}, u), \\ P_k &= J_A P_{k-1} J_A^T + Q. \end{aligned}$$

4.2.2.2 Process noise

The process noise simulates the perturbations in the motion model. The original intention for introducing the motion model is to simplify the motion problem to be handled. The simplification is based on several assumptions (in CTRV, these assumptions are constant yaw angular velocity and speed), but in real problems, these assumptions are subject to a certain amount of error, with the process noise describing how much error the system may face when it runs after a specified period operation because of model simplification. The noise in the CTRV model is introduced mainly from two sources: linear acceleration and yaw angle acceleration. Suppose that Gaussian distributions with variances of σ_a^2 and σ_ω^2 have accelerations with mean values of 0. As the mean value is 0 in the state transfer equation, we can disregard the control input and only examine the uncertainty caused by the noise Q by setting $b_u = 0$. The linear acceleration and yaw angle acceleration affect the state quantities $(x, y, v, \theta, \omega)$. The effect of these two acceleration quantities on the state is as follows:

$$\text{noise}_{\text{term}} = \begin{bmatrix} \frac{1}{2}\Delta t^2 \mu_a \cos(\theta) \\ \frac{1}{2}\Delta t^2 \mu_a \sin(\theta) \\ \Delta t \mu_a \\ \frac{1}{2}\Delta t^2 \mu_\omega \\ \Delta t \mu_\omega \end{bmatrix}$$

μ_a and μ_ω are the accelerations on the line and the corner (in this model, they are treated as process noise). This matrix is decomposed as follows:

$$\text{noise}_{\text{term}} = \begin{bmatrix} \frac{1}{2}\Delta t^2 \cos(\theta) & 0 \\ \frac{1}{2}\Delta t^2 \sin(\theta) & 0 \\ \Delta t & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ 0 & \Delta t \end{bmatrix} \cdot \begin{bmatrix} \mu_a \\ \mu_\omega \end{bmatrix} = G \cdot \mu$$

From the previous section, Q is known as the covariance matrix of the process noise, and its expression is

$$Q = E[\text{noise_term} \cdot \text{noise_term}^T] = E[G\mu\mu^T G^T] = G \cdot E[\mu\mu^T] \cdot G^T.$$

Among them,

$$E[\mu\mu^T] = \begin{pmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_\omega^2 \end{pmatrix}.$$

Therefore, the covariance matrix of the process noise in the CTRV model Q is calculated by the following formula:

$$Q = \begin{bmatrix} \left(\frac{1}{2}\Delta t^2 \sigma_a^2 \cos(\theta)\right)^2 & \frac{1}{4}\Delta t^4 \sigma_a^2 \sin(\theta) \cos(\theta) & \frac{1}{2}\Delta t^3 \sigma_a^2 \cos(\theta) & 0 & 0 \\ \frac{1}{4}\Delta t^4 \sigma_a^2 \sin(\theta) \cos(\theta) & \left(\frac{1}{2}\Delta t^2 \sigma_a^2 \sin(\theta)\right)^2 & \frac{1}{2}\Delta t^3 \sigma_a^2 \sin(\theta) & 0 & 0 \\ \frac{1}{2}\Delta t^3 \sigma_a^2 \cos(\theta) & \frac{1}{2}\Delta t^3 \sigma_a^2 \sin(\theta) & \Delta t^2 \sigma_a^2 & 0 & 0 \\ 0 & 0 & 0 & \left(\frac{1}{2}\Delta t^2 \sigma_\omega^2\right)^2 & \frac{1}{2}\Delta t^3 \sigma_\omega^2 \\ 0 & 0 & 0 & \frac{1}{2}\Delta t^3 \sigma_\omega^2 & \Delta t^2 \sigma_\omega^2 \end{bmatrix}.$$

4.2.2.3 Measurement

Suppose two sensors, LIDAR and RADAR, measure the following data at a certain frequency.

- LIDAR measures the coordinates of the target vehicle (x, y) . Here, x, y is relative to the vehicle coordinate system; that is, the vehicle is the origin of the coordinate system. The vehicle's forward direction is the x-axis, and the left direction is the y-axis in accordance with the right-hand rule.
- RADAR measures the distance of the target vehicle in polar coordinates from the vehicle under the vehicle coordinate system. ρ is the angle between the target vehicle and the x-axis ψ while $\dot{\rho}$ is the rate of change of the relative distance between the target vehicle and our vehicle (essentially the component of the actual velocity of the target vehicle on the line between this vehicle and the target vehicle).

In the previous Kalman filter, an observation matrix H is used to map the predicted results to the measurement space because the mapping itself is linear. Now, we have two cases in which RADAR and LIDAR are used to measure the target vehicle (a process called sensor fusion).

- (a)** The measurement model of LIDAR remains linear. Its measurement matrix is as follows:

$$H_L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

The predictions are mapped to the LIDAR measurement space through the formula $H_L \vec{x} = (x, y)^T$.

- (b)** The predicted mapping of the radar to the measurement space is nonlinear, and the expression is

$$\begin{pmatrix} \rho \\ \psi \\ \dot{\rho} \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ a \tan 2(y, x) \\ \frac{vx + vy}{\sqrt{x^2 + y^2}} \end{pmatrix}.$$

In this case, if $h(x)$ is used to represent such a nonlinear mapping, then this nonlinear process is also linearized using Taylor's formula when solving for the Kalman gain. With reference to the prediction process, the Jacobi matrix of $h(x)$ is the only matrix that needs to be solved.⁴

$$J_H = \begin{bmatrix} \frac{x}{\sqrt{x^2 + y^2}} & \frac{y}{\sqrt{x^2 + y^2}} & 0 & 0 \\ -\frac{y}{x^2 + y^2} & \frac{x}{x^2 + y^2} & 0 & 0 \\ \frac{\nu}{\sqrt{x^2 + y^2}} - \frac{x(\nu x + \nu y)}{(x^2 + y^2)^{\frac{3}{2}}} & \frac{\nu}{\sqrt{x^2 + y^2}} - \frac{y(\nu x + \nu y)}{(x^2 + y^2)^{\frac{3}{2}}} & \frac{x + y}{\sqrt{x^2 + y^2}} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Although this Jacobi matrix seems to be very complex, the complete expression of this Jacobi matrix need not be derived in the subsequent programming. In this work, the library numdifftools is used to solve the Jacobi matrix.⁴

In summary, the entire process of the EKF is shown in Fig. 4.4.

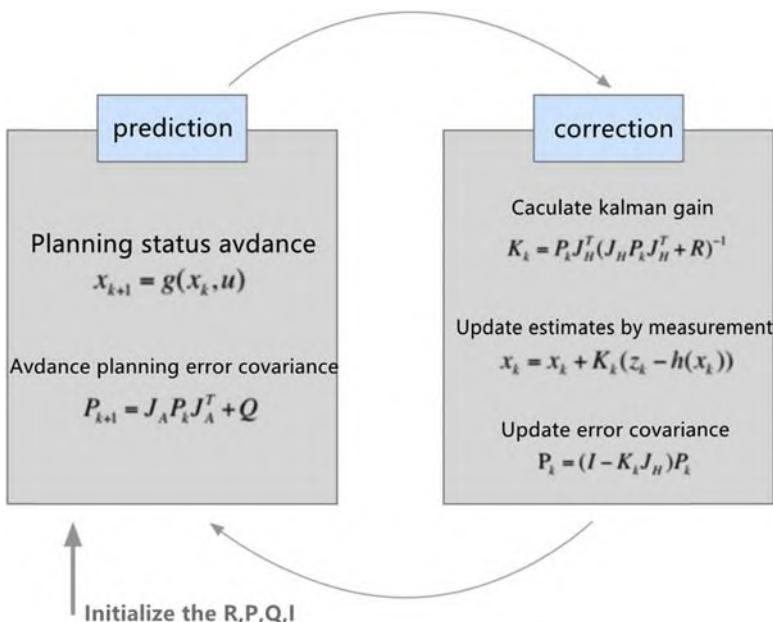


Figure 4.4 Extended Kalman filter process.

4.2.2.4 Python implementation

Python is still used to implement the interactive code, but the actual autonomous car project can also be implemented in C++ as needed. One can easily and quickly rewrite the following sample code in C++:

Import the relevant libraries as follows:

Code list 4.2.1 Introduction of related libraries

```
from __future__ import print_function
import numpy as np
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
from scipy.stats import norm
from sympy import Symbol, symbols, Matrix, sin, cos, sqrt, atan2
from sympy import init_printing
init_printing(use_latex=True)
import numdifftools as nd
import math
```

Then, the output dataset of the sensor containing the LIDAR and RADAR measurements of the tracked target, as well as the time points of the measurements and the real coordinates of the tracked target, needs to be read to verify the accuracy of the tracked target (Fig. 4.5).

The first column (L and R) indicates whether the measurement data are from LIDAR or RADAR. If the first column is L , then columns 2 and 3 indicate the target of the measurement (x, y) . Column 4 indicates the time

R	2.447230e+01	1.090121e+00	-2.241685e+00	1477010450050000	1.7202737e+01
L	1.168143e+01	2.086511e+01	1477010450900000	1.178496e+01	2.111700e+01
R	2.367080e+01	1.062204e+00	-2.414893e+00	1477010450950000	1.154179e+01
L	1.136299e+01	2.106885e+01	1477010451000000	1.129823e+01	2.113353e+01
R	2.310852e+01	1.096410e+00	-2.608439e+00	1477010451050000	1.105441e+01
L	1.089027e+01	2.092695e+01	1477010451100000	1.081049e+01	2.112589e+01
R	2.332606e+01	1.160452e+00	-2.318841e+00	1477010451150000	1.056661e+01
L	1.018897e+01	2.120896e+01	1477010451200000	1.032291e+01	2.109432e+01
R	2.351575e+01	1.132521e+00	-2.697811e+00	1477010451250000	1.007952e+01
L	9.996342e+00	2.097967e+01	1477010451300000	9.836580e+00	2.103913e+01
R	2.310199e+01	1.152590e+00	-2.349513e+00	1477010451350000	9.594224e+00
L	9.539206e+00	2.087620e+01	1477010451400000	9.352578e+00	2.096067e+01
R	2.307927e+01	1.140184e+00	-2.849064e+00	1477010451450000	9.111770e+00
L	8.703310e+00	2.082208e+01	1477010451500000	8.871920e+00	2.085938e+01
R	2.230044e+01	1.113492e+00	-2.997998e+00	1477010451550000	8.633149e+00

Figure 4.5 Test data.

point of the measurement with the device as a reference while columns 5, 6, 7, and 8 indicate the real (x, y, v_x, v_y) . If the first column is R , then the first three columns are (ρ, ψ, ρ) , and the meaning of the data in the remaining columns is the same as that when the first column is L .

The entire dataset is read as follows.

Code list 4.2.2 Read data

```
dataset = []

with open('data_synthetic.txt', 'rb') as f:

    lines = f.readlines()

    for line in lines:

        line = line.strip('\n')

        line = line.strip()

        numbers = line.split()

        result = []

        for i, item in enumerate(numbers):

            item.strip()

            if i == 0:

                if item == 'L':

                    result.append(0.0)

                else:

                    result.append(1.0)

            else:

                result.append(float(item))

    dataset.append(result)

f.close()
```

P , the measurement matrix of LIDAR (linear) H_L , measurement noise R , the standard deviation of the linear acceleration term in the process noise σ_a , and the standard deviation of the angular acceleration term σ_ω are initialized as follows.

Code list 4.2.3 Initialize data and perform processing

```
P = np.diag([1.0, 1.0, 1.0, 1.0, 1.0, 1.0])
print(P, P.shape)

H_lidar = np.array([[ 1., 0., 0., 0., 0., 0.,
                     [ 0., 1., 0., 0., 0., 0.]])
print(H_lidar, H_lidar.shape)

R_lidar = np.array([[0.0225, 0.], [0., 0.0225]])
R_radar = np.array([[0.09, 0., 0.], [0., 0.0009, 0.], [0., 0., 0.09]])
print(R_lidar, R_lidar.shape)
print(R_radar, R_radar.shape)

# process noise standard deviation for a
std_noise_a = 2.0
# process noise standard deviation for yaw acceleration
std_noise_yaw_dd = 0.3
```

In the prediction and measurement update process, all angle measurement values should be controlled to $[-\pi, \pi]$. As the angle remains constant whenever 2π is added or subtracted, the angles are adjusted with the following function:

Code list 4.2.4 Adjust the angle

```
def control_psi(psi):
    while (psi > np.pi or psi < -np.pi):
        if psi > np.pi:
            psi = psi - 2 * np.pi
        if psi < -np.pi:
            psi = psi + 2 * np.pi
    return psi
```

If the object state is initialized using the first radar measurement data (or LIDAR), for the LIDAR data, the measured target's (x, y) coordinates are the initial coordinates while the rest of the state terms are initialized to 0. For RADAR data, the decomposed target coordinates (x, y) can be obtained using ρ and ψ measured by the following formula:

$$\begin{aligned}x &= \rho \cos(\psi), \\y &= \rho \sin(\psi).\end{aligned}$$

The specific state initialization code is as follows:

Code list 4.2.5 Initialization code

```
state = np.zeros(5)

init_measurement = dataset[0]

current_time = 0.0

if init_measurement[0] == 0.0:
    print('Initialize with LIDAR measurement!')

    current_time = init_measurement[3]

    state[0] = init_measurement[1]
    state[1] = init_measurement[2]

else:
    print('Initialize with RADAR measurement!')

    current_time = init_measurement[4]

    init_rho = init_measurement[1]

    init_psi = init_measurement[2]

    init_psi = control_psi(init_psi)

    state[0] = init_rho * np.cos(init_psi)
    state[1] = init_rho * np.sin(init_psi)

print(state, state.shape)
```

An auxiliary function is written for saving values as follows:

Code list 4.2.6 Auxiliary functions

```
# Preallocation for Saving

px = []
py = []
vx = []
vy = []

gpx = []
gpy = []
gvx = []
gvy = []

mx = []
my = []

def savestates(ss, gx, gy, gv1, gv2, m1, m2):
    px.append(ss[0])
    py.append(ss[1])
    vx.append(np.cos(ss[3]) * ss[2])
    vy.append(np.sin(ss[3]) * ss[2])

    gpx.append(gx)
    gpy.append(gy)
    gvx.append(gv1)
    gvy.append(gv2)

    mx.append(m1)
    my.append(m2)
```

In the following code, we define the state transfer function and the measurement function and use the numdifftools library to calculate their

corresponding Jacobi matrices assuming $\Delta t = 0.05$. For example, when running the EKF, we calculate the time difference between the two measurements to replace Δt .

Code list 4.2.7 Auxiliary functions

```

measurement_step = len(dataset)

state = state.reshape([5, 1])

dt = 0.05

I = np.eye(5)

transition_function = lambda y: np.vstack((
    y[0] + (y[2] / y[4]) * (np.sin(y[3] + y[4] * dt) - np.sin(y[3])),
    y[1] + (y[2] / y[4]) * (-np.cos(y[3] + y[4] * dt) + np.cos(y[3])),
    y[2],
    y[3] + y[4] * dt,
    y[4]))

# when omega is 0

transition_function_1 = lambda m: np.vstack((m[0] + m[2] * np.cos(m[3])
    * dt, m[1] + m[2] * np.sin(m[3]) * dt, m[2], m[3] + m[4] * dt, m[4]))

J_A = nd.Jacobian(transition_function)

J_A_1 = nd.Jacobian(transition_function_1)

# print(J_A([1., 2., 3., 4., 5.]))

measurement_function = lambda k: np.vstack((np.sqrt(k[0] * k[0] + k[1]
    * k[1]), math.atan2(k[1], k[0]), (k[0] * k[2] * np.cos(k[3])) + k[1] * k[2]
    * np.sin(k[3])) / np.sqrt(k[0] * k[0] + k[1] * k[1])))

J_H = nd.Jacobian(measurement_function)

# J_H([1., 2., 3., 4., 5.])

```

Code list 4.2.8 EKF process code

```
for step in range(1, measurement_step):  
  
    # Prediction  
  
    # ======  
  
    t_measurement = dataset[step]  
  
    if t_measurement[0] == 0.0:  
  
        m_x = t_measurement[1]  
  
        m_y = t_measurement[2]  
  
        z = np.array([[m_x], [m_y]])  
  
  
        dt = (t_measurement[3] - current_time) / 1000000.0  
        current_time = t_measurement[3]  
  
  
        # true position  
  
        g_x = t_measurement[4]  
  
        g_y = t_measurement[5]  
  
        g_v_x = t_measurement[6]  
  
        g_v_y = t_measurement[7]  
  
  
    else:  
  
        m_rho = t_measurement[1]  
  
        m_psi = t_measurement[2]  
  
        m_dot_rho = t_measurement[3]  
  
        z = np.array([[m_rho], [m_psi], [m_dot_rho]])
```

```
dt = (t_measurement[4] - current_time) / 1000000.0

current_time = t_measurement[4]

# true position
g_x = t_measurement[5]
g_y = t_measurement[6]
g_v_x = t_measurement[7]
g_v_y = t_measurement[8]

if np.abs(state[4, 0]) < 0.0001: # omega is 0, Driving straight
    state = transition_function_1(state.ravel().tolist())
state[3, 0] = control_psi(state[3, 0])
JA = J_A_1(state.ravel().tolist())
else: # otherwise
    state = transition_function(state.ravel().tolist())
state[3, 0] = control_psi(state[3, 0])
JA = J_A(state.ravel().tolist())

G = np.zeros([5, 2])
G[0, 0] = 0.5 * dt * dt * np.cos(state[3, 0])
G[1, 0] = 0.5 * dt * dt * np.sin(state[3, 0])
G[2, 0] = dt
G[3, 1] = 0.5 * dt * dt
G[4, 1] = dt
```

```

Q_v = np.diag([std_noise_a*std_noise_a,
               std_noise_yaw_dd*std_noise_yaw_dd])

Q = np.dot(np.dot(G, Q_v), G.T)

# Project the error covariance ahead

P = np.dot(np.dot(JA, P), JA.T) + Q

# Measurement Update (Correction)

# =====

if t_measurement[0] == 0.0:

    # Lidar

    S = np.dot(np.dot(H_lidar, P), H_lidar.T) + R_lidar

    K = np.dot(np.dot(P, H_lidar.T), np.linalg.inv(S))

    y = z - np.dot(H_lidar, state)

    y[1, 0] = control_psi(y[1, 0])

    state = state + np.dot(K, y)

    state[3, 0] = control_psi(state[3, 0])

    # Update the error covariance

    P = np.dot((I - np.dot(K, H_lidar)), P)

    # Save states for Plotting

    savestates(state.ravel().tolist(), g_x, g_y, g_v_x, g_v_y, m_x, m_y)

else:

    # Radar

    JH = J_H(state.ravel().tolist())

```

```

S = np.dot(np.dot(JH, P), JH.T) + R_radar

K = np.dot(np.dot(P, JH.T), np.linalg.inv(S))

map_pred = measurement_function(state.ravel().tolist())

if np.abs(map_pred[0, 0]) < 0.0001:

    # if rho is 0

map_pred[2, 0] = 0


y = z - map_pred

y[1, 0] = control_psi(y[1, 0])


state = state + np.dot(K, y)

state[3, 0] = control_psi(state[3, 0])

# Update the error covariance

P = np.dot((I - np.dot(K, JH)), P)

savestates(state.ravel().tolist(), g_x, g_y, g_v_x, g_v_y, m_rho *
np.cos(m_psi), m_rho * np.sin(m_psi))

```

The primary point that needs to be emphasized is the need to consider clearly where the divisor is likely to be 0. For example, if $\omega = 0$ and $\rho = 0$, the case with code exception handling protection is needed.

After processing, the root mean square error of the prediction estimate is output, and the various types of data are saved to visualize the effect of the EKF.

Code list 4.2.9 Mean square error of output estimation

```

def rmse(estimate, actual):
    result = np.sqrt(np.mean((estimate-actual)**2))

    return result

print(rmse(np.array(px), np.array(gpx)),
      rmse(np.array(py), np.array(gpy)),
      rmse(np.array(vx), np.array(gvx)),
      rmse(np.array(vy), np.array(gvy)))

# write to the output file
stack = [px, py, vx, vy, mx, my, gpx, gpy, gvx, gvy]
stack = np.array(stack)
stack = stack.
np.savetxt('output.csv', stack, '%.6f')

```

Finally, let us look at the mean square error of the EKF in tracking the target.

```
0.0736336090893 0.0804598933194 0.229165985264 0.309993887661
```

Tip: Using Excel, you can visualize the EKF estimation results of the trajectory calculation conveniently. As shown in the figure, the orange dots are the measured values of LIDAR and RADAR, and the green curves are the true value of the corresponding target localization. The blue dots are the estimated values of the EKF.

Fig. 4.6 indicates that the vehicle travels on a figure-of-eight path.

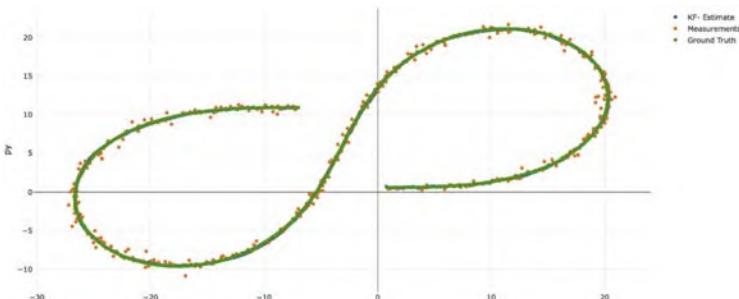


Figure 4.6 Estimation results using EKF.

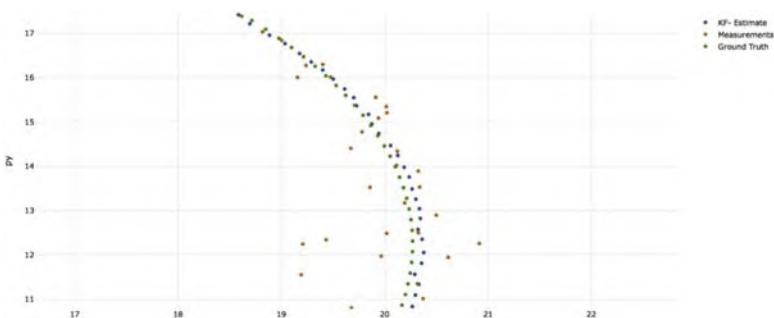


Figure 4.7 Local zoom.

A partial zoom is shown in Fig. 4.7.

Obviously, the orange dots confirm that the measured data are inaccurate because the distribution of the dots has large jumps and intervals. Thus, the figure proves the existence of noise. However, on the basis of the fusion of the measurement data of the two sensors and by combining the EKF algorithm and CTRV motion model, we can obtain an estimate that is very close to the true state of the target. This capability is a strong advantage of the Kalman filter algorithm.

Nevertheless, the EKF offers other great benefits. Using the EKF, one can accurately estimate not only the state of the target but also the quantities that cannot be measured by the sensor (e.g., in this case, v , ψ , and $\dot{\psi}$ represent the velocity, angle, and angular velocity, respectively), as reflected in the example.

Careful readers may also notice that the EKF algorithm is more computationally intensive than the Kalman filter algorithm. In fact, one of the biggest problems with the EKF is that it is more computationally intensive when solving Jacobi matrices, but it can be reduced using

computational optimization techniques and other means to obtain performance improvements.

The next section describes another widely used Kalman filter algorithm, that is, the unscented Kalman filter (UKF).

4.3 UKF

Previously, we learned about the Kalman filter and the application of the EKF in target state estimation and implemented an EKF prototype for vehicle position tracking in Python. When the problem becomes complex (nonlinear model), the computational effort becomes very unmanageable. To solve this problem, we introduce another improvement of the Kalman filter in this section, that is, the UKF.^{5,6}

As described in the previous article, the Kalman filter is mainly applicable to linear systems and not to nonlinear systems. To deal with the latter, the EKF that approximates nonlinear systems using linear functions by first-order Taylor expansions is introduced. However, the problem of this method is that for specific problems, the corresponding first-order partial derivatives (Jacobi matrices) need to be solved. As the calculation of Jacobi matrices is too complicated, let us learn another relatively simple state estimation algorithm: the UKF.

The UKF uses a statistical linearization technique called nondestructive transformation. This technique linearizes the nonlinear function of a random variable mainly through a linear regression of n points (also called sigma points) collected in the prior distribution. As the expansion of the random variable is considered, the linearization is more accurate than the Taylor series linearization (the strategy used by the EKF).

In other words, when the prediction and update models are highly nonlinear, the EKF predictions also perform poorly because the variance of the system is still propagated through the linearized nonlinear model. The UKF, a technique based on statistical sampling by selecting a set of sampling points near the place of the mean value, constructs a nonlinear function model by estimating the mean and variance of the distribution formed by these sampling point estimations to construct a nonlinear function model. Previous methods calculate the probability of sampling points on the basis of a Gaussian distribution. Conversely, the UKF estimates the mean and variance in line with the type of Gaussian distribution through a set of sampling points. By adopting this method, we can obtain accurate results and avoid the situation in which the variance is still propagated through the

linear model. However, the amount of computation is large. The UKF, like the EKF, is also mainly divided into prediction and update processes.

4.3.1 Movement model

The CTRV motion model continues to be used here. It takes the following form:

$$\vec{x}(t + \Delta t) = g(x(t)) = \begin{pmatrix} \frac{v}{\omega} \sin(\omega \Delta t + \theta) - \frac{v}{\omega} \sin(\theta) + x(t) \\ -\frac{v}{\omega} \cos(\omega \Delta t + \theta) + \frac{v}{\omega} \sin(\theta) + y(t) \\ v \\ \omega \Delta t + \theta \\ \omega \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \Delta t^2 \mu_a \cos(\theta) \\ \frac{1}{2} \Delta t^2 \mu_a \sin(\theta) \\ \Delta t \mu_a \\ \frac{1}{2} \Delta t^2 \mu_\omega \\ \Delta t \mu_\omega \end{pmatrix}, \omega \neq 0$$

When $\omega = 0$, the equation degenerates as follows:

$$\vec{x}(t + \Delta t) = g(x(t)) = \begin{pmatrix} v \cos(\theta) \Delta t + x(t) \\ v \sin(\theta) \Delta t + y(t) \\ v \\ \omega \Delta t + \theta \\ \omega \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \Delta t^2 \mu_a \cos(\theta) \\ \frac{1}{2} \Delta t^2 \mu_a \sin(\theta) \\ \Delta t \mu_a \\ \frac{1}{2} \Delta t^2 \mu_\omega \\ \Delta t \mu_\omega \end{pmatrix}, \omega = 0$$

In the EKF, the effects of linear acceleration and yaw angle acceleration are treated as process noise. Suppose they obey a Gaussian distribution with a mean value of 0 and the variance is σ_a and σ_ω . In this case, the effects of noise are directly considered in the state transfer function. As for the uncertainty terms in the functions σ_a and σ_ω , they will be analyzed later.

4.3.2 Nonlinear processing/measurement models

As shown in the previous section, the main problem in applying the Kalman filter is the treatment of nonlinear process models (e.g., CTRV) and nonlinear measurement models (RADAR measurements). This problem is described below in terms of probability distributions.

For the state being estimated, at the moment k , the state obeys a Gaussian distribution with the mean μ_k and variance σ_k . This is the posterior (if we consider the whole Kalman filter process iteratively) at k . Now, using this posterior as a starting point, we combine certain prior knowledge (e.g., CTRV motion model) to estimate the mean and variance of the state at $k + 1$. This process is the prediction of the Kalman filter. If the transformation is linear, then the predicted result still conforms to a Gaussian distribution. However, the reality is that the process model and the measurement model are nonlinear. Thus, the result is an irregular distribution. The premise at which the Kalman filter can be used is that in which the state being processed satisfies linear variations. To solve this problem, the EKF finds a linear function model to approximate this nonlinear function model, while the UKF finds a Gaussian distribution that approximates the true distribution.

The basic idea of the UKF is that finding a probability distribution that approximates a nonlinear function is easier than approximating a nonlinear function. The UKF approximates the true distribution by finding a Gaussian distribution with the same mean and covariance as the true distribution. As for determining the mean and covariance, it uses lossless transformation.

4.3.3 Lossless transformation

By using a certain method to generate a set of sigma point sets that can represent the current distribution, the points are transformed to a new space (prediction space) by a nonlinear function (process model). Finally, a new Gaussian distribution (computed with weights) is computed on the basis of these new sigma points. The whole process is shown in Fig. 4.8.

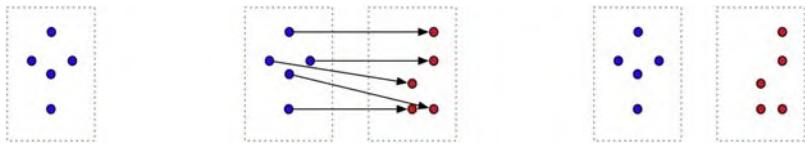


Figure 4.8 Nondestructive transformation.

4.3.4 Projections

A sigma point set is generated from a Gaussian distribution. Assuming that the number of states is $2n + 1$, sigma sampling points are generated here, and the first point is the mean value of the current state μ . The mean value of the set of sigma points is calculated as follows:

$$\begin{aligned}\chi^{[1]} &= \mu \\ \chi^{[i]} &= \mu + (\sqrt{(n + \lambda)P})_i \text{ for } i = 2, \dots, n + 1 \\ \chi^{[i]} &= \mu - (\sqrt{(n + \lambda)P})_{i-n} \text{ for } i = n + 2, \dots, 2n + 1\end{aligned}$$

λ is a hyperparameter. According to the equation above, the larger the value of λ is, the farther the sigma point is from the mean of the distribution, while the smaller the value of λ is, the closer the sigma point is to the mean of the distribution. We should note that in the CTRV model, the number of states n includes not only five states but also process noise μ_a and μ_ω because these process noises also have a nonlinear effect on the model. After adding the effect of process noise, the uncertainty matrix P becomes as follows:

$$P = \begin{pmatrix} P' & 0 \\ 0 & Q \end{pmatrix}$$

where P' is the original uncertainty matrix (the size is a 5×5 matrix), and Q is the covariance matrix of the process noise in the CTRV model. Consider the following forms of linear acceleration and Q :

$$Q = \begin{bmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_\omega^2 \end{bmatrix}.$$

σ_a^2 and σ_ω^2 are the same as those described previously. A persistent problem in the formula relates to the calculation of the open square root of the matrix. In the same case, it can be calculated as follows:

$$A = \sqrt{P}$$

$$AA^T = P$$

Solving A in the above equation is a relatively complicated process. If P is a diagonal matrix, then this solution is simplified. P denotes the uncertainty about the estimated states (covariance matrix). P is basically a diagonal matrix (the correlation between state quantities is almost zero). Thus, we can first perform a Cholesky decomposition of P . Then, the lower triangular matrix of the decomposed matrix is the required A .

4.3.4.1 Prediction of sigma point

Now that we have the sigma point set, the next step is to use the nonlinear function $g()$ to make predictions.

$$\chi_{k+1|k} = g(\chi_{k|k}, \mu_k)$$

The size of input matrix $\chi_{k|k}$ is (7, 15) because two noise quantities are considered, but the size of output matrix $\chi_{k+1|k}$ is (5, 15) because it is the result of the prediction, which is essentially based on the prior of the motion model. The mean value in the prior should not contain uncertainties such as a , ω .

4.3.4.2 Predicted mean and variance

The weights of each sigma point should be calculated, and then the weights are calculated by the formula

$$w^{[i]} = \frac{\lambda}{\lambda + n}, i = 1,$$

$$w^{[i]} = \frac{1}{2(\lambda + n)}, i = 2, \dots, 2n + 1.$$

The weights of each sigma point are then used to solve for the mean and variance of the new distribution based on the weights.⁶

$$\mu' = \sum_{i=1}^{2n+1} w^{[i]} \chi_{k+1}^{[i]}$$

$$P' = \sum_{i=1}^{2n+1} w^{[i]} (\chi_{k+1}^{[i]} - \mu') (\chi_{k+1}^{[i]} - \mu')^T,$$

where μ' is the mean value of the target state prior distribution predicted on the basis of the CTRV model $x_{k+1|k}$, which is the weighted sum of the individual state quantities at each point in the sigma point set. P' is the covariance (uncertainty) of the prior distribution. $P_{k+1|k}$ is the weighted sum of the variances of each sigma point obtained from the weighted sum of the variances of each sigma point. Therefore, the prediction part is finished, and the following part enters the measurement update part of the UKF.

4.3.5 Measurement updates

- Predictive measurement (map the prior to the measurement space and then calculate the mean and variance)

Here, we continue to use the measurement experiment data from the previous EKF. We already know that the measurement update is divided into two parts, LIDAR measurement and RADAR measurement, with the LIDAR measurement model itself being linear. Thus, we focus on the processing of the RADAR measurement model. The measurement mapping function of RADAR is

$$Z_{k+1|k} = \begin{pmatrix} \rho \\ \psi \\ \dot{\rho} \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ a \tan 2(y, x) \\ \frac{v \cos(\theta)x + v \sin(\theta)y}{\sqrt{x^2 + y^2}} \end{pmatrix}.$$

A nonlinear function represented by $h()$ subsequently is transformed using a lossless transformation without generating a sigma point. The predicted set of sigma points can be used directly, and the process noise part can be ignored. Then, the a priori nonlinear mapping can be expressed as the following sigma point prediction (i.e., the predicted mean and covariance after the nonlinear transformation).

$$z_{k+1|k} = \sum_{i=1}^{2n+1} w^{[i]} Z_{k+1|k}^{[i]}$$

$$S_{k+1|k} = \sum_{i=1}^{2n+1} w^{[i]} \left(Z_{k+1|k}^{[i]} - z_{k+1|k} \right) \left(Z_{k+1|k}^{[i]} - z_{k+1|k} \right)^T + R$$

As in the previous section, R is also measurement noise. The covariance of the measurement noise is directly added to the measurement covariance because that noise has no nonlinear effect. In this case, when the RADAR measurement is taken as an example, the measurement noise R is

$$R = E[ww^T] = \begin{pmatrix} \sigma_\rho^2 & 0 & 0 \\ 0 & \sigma_\psi^2 & 0 \\ 0 & 0 & \sigma_{\dot{\rho}}^2 \end{pmatrix}.$$

4.3.5.1 Update status

The mutual correlation function of the sigma point set in the state space and measurement space is calculated as follows:

$$T_{k+1|k} = \sum_{i=1}^{2n+1} w^{[i]} \left(X_{k+1|k}^{[i]} - x_{k+1|k} \right) \left(Z_{k+1|k}^{[i]} - z_{k+1|k} \right)^T.$$

The latter is simply calculated by following the update steps of the Kalman filter in its entirety, starting with the calculation of the Kalman gain.

$$K_{k+1|k} = T_{k+1|k} \cdot S_{k+1|k}^{-1}.$$

The status is then updated (i.e., make a final status estimate).

$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1|k} (z_{k+1} - z_{k+1|k}),$$

where z_{k+1} is the newly obtained measurement and $z_{k+1|k}$ is the measurement calculated on the basis of a priori in the measurement space. The state covariance matrix is updated as follows:

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1|k} S_{k+1|k} K_{k+1|k}^T.$$

4.3.6 Summary

We have presented the core algorithm content of the UKF. The UKF's prediction contains three main parts:

- Generated sigma point sets
 - Predicted sigma point sets based on the CTRV model
 - Calculated new mean and variance
- The UKF measurement update is a three-step process:
- Predictive LIDAR measurements
 - Predictive RADAR measurements
 - Status update

References

1. Welch G, Bishop G. An introduction to the kalman filter. *Course Notes 8 of ACM SIGGRAPH 2001*. 1995;8(7):127–132.
2. How a Kalman filter works, in pictures. <http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/>.

3. Schubert R, Richter E, Wanielik G. Comparison and evaluation of advanced motion models for vehicle tracking. In: *International Conference on Information Fusion*. IEEE; 2008:1–6.
4. Terejanu GA. Extended kalman filter tutorial. In: *Technical Report*. Terejanu GA: University of Buffalo; 2008.
5. Julier S, Uhlmann JK. Unscented filtering and nonlinear estimation. *Proc IEEE*. 2004;92:401–422.
6. Terejanu GA. *Unscented Kalman Filter Tutorial*. Department of Computer Science & Engineering; 2009.

This page intentionally left blank

CHAPTER 5

Introduction of machine learning and neural networks

Zebang Shen¹, Binbin Yong² and Peng Zhi²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

5.1 Basic concepts of machine learning	148
5.2 Supervised learning	151
5.2.1 Empirical risk minimization	151
5.2.2 Overfitting and underfitting	152
5.2.3 "Certain algorithm"—gradient descent algorithm	155
5.2.4 Summary	156
5.3 Fundamentals of neural network	157
5.3.1 Basic structure of the neural network	158
5.3.2 Unlimited capacity—fitting arbitrary functions	160
5.3.3 Forward transmission	162
5.3.4 Stochastic gradient descent	164
5.4 Using Keras to implement the neural network	165
5.4.1 Data preparation	165
5.4.2 A small change in three-layer neural network—deep feedforward neural network	171
5.4.3 Summary	175
References	175

Environmental perception is currently one of the most significant challenges in unmanned vehicle systems. In the environmental perception of unmanned vehicles, we need to identify the obstacles in the vehicle environment. Although both pedestrians and vehicles can be regarded as obstacles, they are extraordinary obstacles. Hence, their behavior and motion models are also different from those of ordinary obstacles. To make the driving behavior of unmanned vehicles similar to that of human drivers, the target obstacles need to be identified. This identification process is a typical pattern recognition process. Although point cloud data based on LiDAR can achieve target point cloud clustering, and some algorithms support pattern recognition in point clouds,¹ the resolution of point cloud

data is still low regardless of whether 64-line or 128-line lidar is used. In some cases, even human beings cannot judge what the object is based on its point cloud data. By contrast, when the resolution of the image is high, the information contained in the image is sufficient for pattern recognition. Therefore, this chapter mainly discusses image-based pattern recognition algorithms used in the environmental perception of unmanned vehicles.

What exactly is pattern recognition? Pattern recognition is the process of identifying pattern types using computer models. “Pattern” refers to the environment and object that we aim to study. For human beings, the recognition of optical information (obtained by the visual organs) and acoustic information (obtained by the auditory organs), which are two important aspects of pattern recognition, is particularly important. The representative products visible in the market are optical character and speech recognition systems. In the field of unmanned vehicle perception, particularly in the field of visual perception, the detection and recognition of images and lidar point clouds are the main methods.

Traditional computer vision often requires different artificially designed features for different tasks. For example, features need to be designed for lane and pedestrian detection. However, artificially designed features often have omissions. For unmanned vehicles with high safety requirements, software program design that ignores some details is bound to have defects, which may lead to serious safety consequences.

This section will introduce some basic concepts of machine learning, and the subsequent sections will elaborate on deep learning, including the application of machine learning and deep learning in the field of unmanned vehicles.

5.1 Basic concepts of machine learning

Machine learning is important in unmanned vehicle systems, among which deep learning has become a hot research topic in recent years. Mastering the basic theory of machine learning, such as reinforcement learning control and environmental perception based on deep learning, is the first step in the research on end-to-end unmanned vehicles. This section focuses on explaining the basic concepts of machine learning and describing the process of machine learning tasks for the reader.

We take the Modified National Institute of Standards and Technology (MNIST)² handwritten digit dataset as an example to introduce the basic concepts of machine learning. The handwritten digits are shown in Fig. 5.1.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

Figure 5.1 Modified National Institute of Standards and Technology (MNIST) handwritten digit dataset.

For a human, recognizing these handwritten digits is simple. However, for a computer, each picture is a matrix, as shown in Fig. 5.2, which leads to a difficult task that needs to be completed by designing a program.

Even if all known handwritten digit images are stored in a database, once a completely new handwritten digit appears (which has not been stored in the database), a fixed program would have a difficult time recognizing that digit.

How does machine learning solve this kind of problem?

Machine learning is a kind of algorithm used to automatically enhance performance based on the data or previous experience.³ Machine learning, which might seem difficult to understand, can be defined as follows:

- First, for the handwritten digit recognition task, the data or previous experience is the collected handwritten digits. We want the program to learn a certain capability or intelligence from these data. By learning



The number 5 in human eyes

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The number 5 in the "eyes" of a computer

Figure 5.2 In the eyes of a computer, an image is a matrix of numbers.

this capability, the program can recognize handwritten digits like a human.

- Performance criteria are metrics that measure the capability of our programs. In recognition tasks, this indicator is the accuracy of recognition. Given 100 handwritten digits, 99 of them are correctly recognized by our “smart program.” Thus, the accuracy of our program is 99%.
- Optimization based on previous experience or data helps make our “smart programs” even smarter than humans.

Machine learning algorithms can learn and improve from experiences. In many cases, we describe these experiences as data. Therefore, experience is the data, the collected data are called datasets, and each dataset is called a sample. The previously mentioned dataset is called the MNIST handwritten digit dataset. The process of learning the existing datasets is called training. Therefore, the dataset is also called the training set. We not only assess the performance of the machine learning algorithm on the training set but also hope that the algorithm can correctly recognize the handwritten digits that it has never seen before. The performance on the new samples (datasets) is called generalization capability. For a task, the stronger the generalization capability, the more successful the machine learning algorithm is.

According to the usage of datasets, machine learning can be divided into the following three categories⁴:

- Supervised learning: The dataset contains both samples (handwritten digit images) and corresponding labels (each number corresponds to each type of handwritten digit image).
- Unsupervised learning: The dataset contains only the samples and does not contain the labels corresponding to the samples. The machine learning algorithm needs to determine the category of the samples by itself.
- Reinforcement learning: This is a kind of semisupervised learning that emphasizes how to act based on the environment to maximize the expected benefits. We will focus on reinforcement learning in the subsequent chapters.

Currently, neural networks and deep learning are mostly categorized under supervised learning. With the advent of the era of big data and the improvement in computing power brought about by the graphics processing unit, supervised learning has made breakthrough progress in a large number of fields, such as image recognition, target detection and tracking, machine translation, speech recognition, and natural language processing.

However, currently, no breakthrough in the field of unsupervised learning has been made. Because the main machine learning technique applied in the field of unmanned driving is still supervised learning, we will focus on content related to supervised learning. This chapter will also introduce the research on reinforcement learning in the field of unmanned driving.

In this section, to facilitate the reader's understanding, we use handwritten digit recognition to describe the processing tasks. Many other tasks, such as classification, regression, machine translation, anomaly detection, synthesis and sampling, and missing value filling, can be handled by machine learning algorithms. These tasks are difficult to accomplish by a deterministic program designed artificially. However, patterns can be learned from a large amount of data, and these learned patterns can be used to predict a new dataset.

5.2 Supervised learning

Supervised learning is currently the most widely used machine learning method. In this section, we provide a detailed introduction of the working mode and process of supervised learning from the perspectives of empirical risk minimization (ERM) strategy, model, and optimization algorithm.

5.2.1 Empirical risk minimization

Supervised learning is essentially learning the mapping function of element x in X to element y in Y based on a given set (X, Y) , which can be expressed as follows:

$$y=f(x).$$

In MNIST handwritten digit recognition, X denotes the collection of all handwritten digital pictures, Y denotes the real digit label corresponding to these pictures, x and y denote the value of a specific element in the dataset, and function f denote the mapping relationship between the input (i.e., a handwritten digit picture) and the output (i.e., the picture represented by the value).

In such a mapping relationship, x has a large range of values (even infinite possible values). Thus, we can understand the existing sample set (X, Y) as sampling in the form of independent identically distributed random variables from a large or even infinite matrix according to some

unknown probability distribution p . Moreover, we can assume that there is a loss function L , which can be expressed as follows:

$$L(f(x), y).$$

This loss function describes the distance between the output of the function $f(x)$ and the real value y corresponding to the sample x . Notably, the smaller the loss, the closer the learned function f is to the real mapping g . Based on the loss function, we define the risk of function f as the expectation value of the loss function.

Taking handwritten character classification as an example, the probability distribution p of each sample is discrete. We can define the risk of function f using the following formula:

$$R(f) = \sum_i L(f(x_i), g(x_i))p(x_i).$$

If the risk of function f is continuous, then it can be expressed using the definite integral and probability density function. x_i denotes all possible values of the entire sample space. Therefore, the goal is to find the function f among many possible functions to minimize the risk $R(f)$. However, the real risk is based on the consideration of the entire sample space, but we cannot obtain the entire sample space. What we have is a subset (X, Y) randomly sampled from the sample space of the task we want to solve. Using this subset, we can determine the approximate value of the real distribution, such as empirical risk, as follows:

$$\bar{R}(f) = \frac{1}{n} \sum_{i=1}^n L(f(x_i), y_i).$$

(x_i, y_i) is the sample of the existing dataset. Thus, we select the function f , which can minimize the empirical risk. Such a strategy is called the ERM principle.⁵

When the training set is sufficiently large, the ERM principle can ensure a good learning effect, which is an important factor that helps ensure the success of the deep neural network. We call the size of the existing dataset the “sample size.” Regardless of the application field, a good large dataset means that the machine learning task has been half successful.

5.2.2 Overfitting and underfitting

In the process of learning the objective function f , we need a carrier to express various functions. Thus, we can select the optimal function f by

adjusting the carrier, which can minimize the empirical risk. This carrier is the model in machine learning.

We take the artificial neural network (ANN) as an example. In the field of machine learning and cognitive science, ANN is a mathematical or calculation model that simulates the structure and function of a biological neural network (i.e., the central nervous system of animals, particularly the brain). The neural network is composed of a large number of artificial neurons, and its internal structure changes under the stimulation of external information. The neural network is an adaptive system, and Fig. 5.3 shows a neural network model structure.

The neural network model structure looks complicated, but it can be simplified, as shown in Fig. 5.4.

We depict this model as a black box, which has many parameters, that is, $(w_1, w_2, w_3, \dots, w_n)$. We use W to describe the parameters in this black box, which are called model parameters. Even if the structure of the model is unchanged, the model exhibits differentiability in describing different functions when the model parameters are modified. Specifically, for the handwritten digit recognition task, we adjust the model parameters of the neural network using the ERM strategy on the handwritten digit dataset so that the neural network can fit function f and the neural network that we trained can be used for handwritten word recognition. For vehicle detection, we can also train a black box (neural network) using the ERM strategy, as shown in Fig. 5.5.

According to the basic concepts of machine learning, we determine that the key to measuring the performance of a machine learning model is its generalization capability. An important metric to measure the generalization capability is the training and test errors of the model.

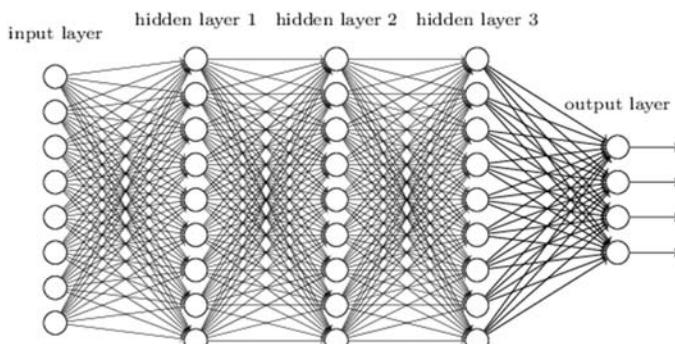


Figure 5.3 Example of a neural network.



Figure 5.4 Neural network depicted as a *black box*.

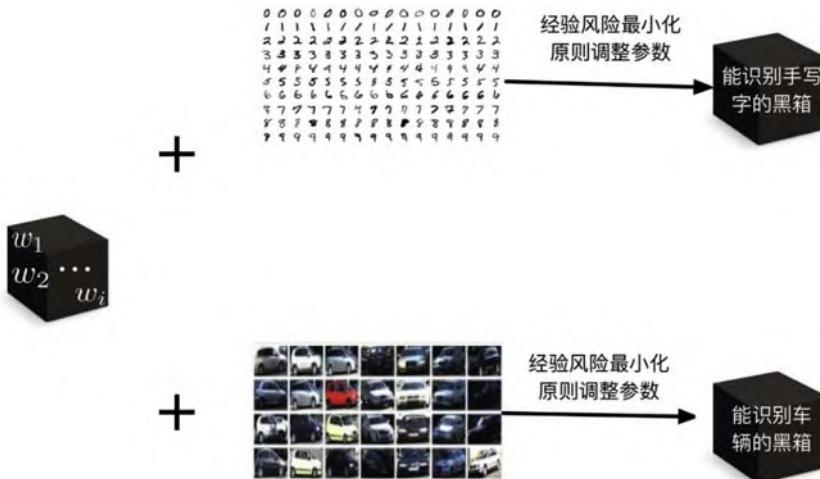


Figure 5.5 Model showing different capabilities when the parameters are adjusted.

- Training error is the prediction error of the model in the training set.
- Test error is the prediction error of the model on the test set that it has never seen.

These two errors correspond to two problems that need to be solved in machine learning tasks, that is, underfitting and overfitting. When the training error is high, the function learned by the model does not satisfy the

ERM. Specifically, when the recognition accuracy of the model in the training set is low, we call it underfitting. When the training error is low but the test error is high, that is, the gap between training and test errors is large, we call it overfitting. At this time, the model has learned some redundant rules on the training set, which is shown as high prediction accuracy on the training set and low prediction accuracy on the test set, which cannot be used to train the model parameters.

Model capacity determines whether the model tends to be overfit or underfit. Model capacity refers to the capability of the model to fit various functions. In general, the more complex the model, the more complex the function (or the rule or pattern) can be expressed. Thus, for a specific task (such as handwriting recognition), how do we select an appropriate model capacity to fit the corresponding function? According to Occam's razor, we should select the simplest hypothesis that can explain the observed phenomenon.

This principle can be considered a minimalist design principle. When dealing with a task, we should use the simplest possible model structure. How do we select the model structure? Specific problems require specific analysis, which we will discuss subsequently.

5.2.3 “Certain algorithm”—gradient descent algorithm

As mentioned previously, the parameters of the neural network can be adjusted by “certain algorithms.” Here, we introduce the gradient descent algorithm, an algorithm that adjusts the model parameters to minimize the empirical risk.

The gradient descent algorithm is a first-order optimization algorithm. To use gradient descent to find the local minimum of a function, an iterative search must be performed within a certain distance of the opposite direction of the corresponding gradient (or approximate gradient) to the current point of the function. Returning to our previous example, minimizing the empirical risk $\bar{R}(f)$ is equivalent to minimizing the loss function, which in machine learning can be written as the sum of the losses for each sample, as follows:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n L(x_i, y_i, \theta),$$

where θ denotes all parameters in the model, the derivative of L with respect to θ is denoted as $L'(\theta)$ or $\frac{dL}{d\theta}$, and the derivative of $L'(\theta)$ denotes

the slope of the function $L(\theta)$ at θ . The input–output correlation of the function can be described by the slope, as follows:

$$L(\theta + \alpha) \approx L(\theta) + \alpha L'(\theta),$$

where α is the variation. We can use the derivative to gradually make L small using the following formula, specifically by making the sign of α and the sign of the derivative opposite, that is,

$$\text{sign}(\alpha) = -\text{sign}(L'(\theta)).$$

In this manner, $L(\theta + \alpha)$ will be smaller than the original $L(\theta)$ and can be expressed as follows:

$$L(\theta + \alpha) = L(\theta) - |\alpha L'(\theta)|$$

The method for minimizing the objective function (the loss function) by moving a small step in the opposite direction of the derivative is called gradient descent. The neural network is a complex model that contains many parameters. Thus, θ here is a set of parameters or a parameter vector, and the derivative becomes a vector $\nabla_{\theta} L(\theta)$, which contains the partial derivative of all parameters. α here can be considered the step length of updating the gradient in one step during the process of gradient descent. Generally, the step length is called the learning rate, which describes the rate of gradient descent.

5.2.4 Summary

In this section, we briefly discuss the important components and structures of machine learning tasks.

First, machine learning is used to accomplish specific tasks, for example, handwriting recognition, pedestrian detection, and housing price prediction. This task must have certain performance measures, such as detection accuracy and prediction error.

Then, to handle this task, we need to design models that can learn data patterns from the data based on certain strategies (such as ERM) and certain algorithms (such as the gradient descent algorithm).

Finally, the model must be able to handle the cases that do not appear in the training set so that the machine learning task can be accomplished.

The subsequent sections will introduce the specific tasks, models, and algorithms. In general, machine learning, particularly deep learning, plays

an important role in the study of unmanned vehicles. We will gradually learn the machine learning algorithms for unmanned vehicles.

5.3 Fundamentals of neural network

In the previous section, we covered the basics of machine learning, particularly the basic components of supervised learning, namely, data, model, strategy, and algorithm. In this section, we will specifically study a supervised learning algorithm called a neural network. Deep learning models are mostly deep ANNs. Therefore, before further exploring the application of deep learning in unmanned vehicles, we consider the theoretical basis and code implementation of neural networks.

Now that we are talking about neural networks and deep learning, let us examine the relationships and categories of various concepts, as shown in Fig. 5.6.

The concept of artificial intelligence (AI) is extensive, where machine learning is a kind of method based on statistical learning in AI and neural network is one of the supervised learning algorithms in machine learning, whereas deep learning is a representation learning algorithm built by increasing the number of layers of the neural network and trained by a large amount of data.

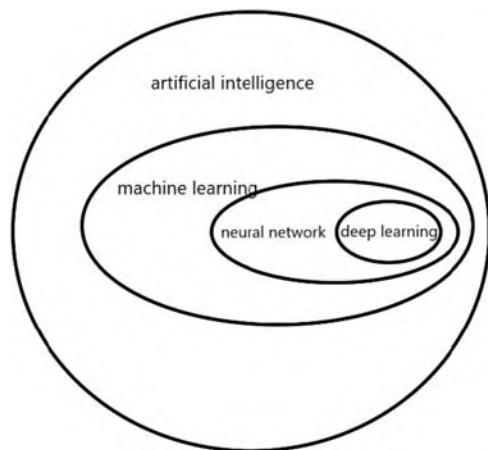


Figure 5.6 Conceptual relationships and hierarchies.

We introduced several factors of supervised learning (i.e., model, strategy, and algorithm) in the previous section. We will introduce these factors in the neural network algorithm one by one. We still use the handwritten digit recognition task introduced in the previous section as an example and the MNIST handwritten digit dataset as the experimental dataset.

5.3.1 Basic structure of the neural network

The original design of the neural network is based on the structure of biological neurons. We use a mathematical model to describe the structure of human neurons and the processes of being stimulated, being activated, and transmitting stimuli. This mathematical model is called a perceptron.⁶ Fig. 5.7 shows the comparison between biological neurons and the perceptron.

As shown in Fig. 5.7, the perceptron imitates the design of neurons. A perceptron can accept multiple inputs. We use the vector $x = (x_0, x_1, \dots, x_n)$ to represent the input value that is multiplied by a weight vector $w = (w_0, w_1, \dots, w_n)$. Inside the perceptron, these weighted inputs are summed and a small bias b is added. Thus, the value inside the perceptron can be expressed as follows:

$$h = \sum_i w_i x_i + b,$$

where w and b are the parameters that the perceptron needs to learn and h is inputted into step function f , which can be expressed as follows:

$$f(h) = \begin{cases} 1, & h > 0 \\ 0, & h \leq 0 \end{cases}.$$

This step function is a nonlinear transformation, which is a kind of activation function. The step function is not usually used as the activation function of neurons because of its rough simulation. The most commonly

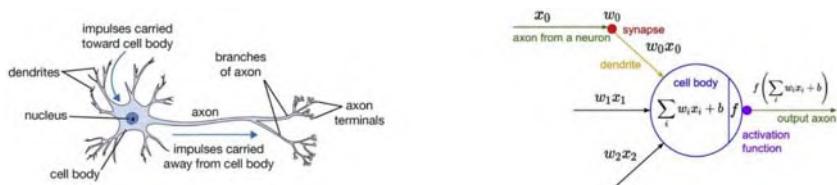


Figure 5.7 Biological and artificial neurons.

Name	Function graph	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic(a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArCTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
ReLU		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
PReLU		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
PReLU		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 5.8 Various activation functions.

used activation functions include the sigmoid, tanh, and rectified linear unit activation functions and their variants. For specific activation function expressions, please refer to Fig. 5.8. The activation expression of the perceptron can be written as follows:

$$\text{output} = f(h) = f\left(\sum_i w_i x_i + b\right)$$

The perceptron is an important model in the history of the development of the neural network. However, the fatal problem in using a single perceptron to complete machine learning tasks is that a single perceptron has been proven unable to learn the exclusive OR (XOR) relationship, which directly affects the development of neural networks in history and makes the research on neural network stagnant. Thus, how can we solve the XOR problem? The answer is to use multilayer perceptron nesting, which is a modern neural network structure. The neural network structure is a hierarchical connection structure, and its basic form is shown in Fig. 5.9.

The tail (output) of the preceding neurons of the neural network is connected to the input of the subsequent neurons, thus forming a

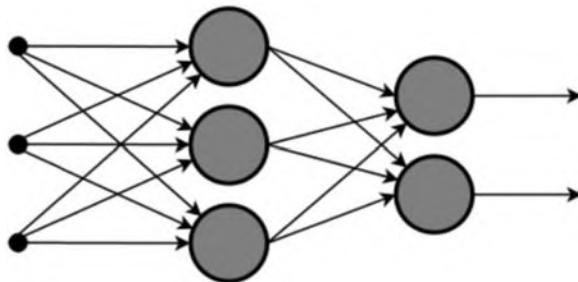


Figure 5.9 Basic structure of the neural network.

hierarchical connection structure. The first layer of the entire neural network is the input layer, whose input values are the data characteristics, and the last layer is the output layer. The middle layers are the hidden layer. Now let us focus on the meaning of the structure of a neural network.

5.3.2 Unlimited capacity—fitting arbitrary functions

In the previous section, we mentioned that the model should have a certain capacity, which is expressed as the capability to fit the function of the task to be solved. In other words, the model should have the capability to fit the function that we want. The power of the neural network lies in the fact that a three-layer neural network can fit any function with a sufficient number of hidden layer neuron nodes. In other words, given a sufficient number of hidden nodes, the neural network with more than three layers has unlimited capacity. To illustrate this, we use a neural network to simulate an XOR relationship or a NOT-AND (NAND) gate.

The NAND gate is a logic gate that realizes two-input logic in digital logic. If all of the inputs have a value of 1, then the output has a value of 0. If at least one of the inputs has a value of 0, then the output has a value of 1. The NAND gate represents logic, as shown in Fig. 5.10.

The truth table for the NAND gate is shown in Fig. 5.11.

The NAND gate has functional integrity, that is, any logical function can be established through the NAND gate. This conclusion will not be



$$Q = \text{NOT}(A \text{ AND } B)$$

Figure 5.10 Logical representation of a NOT-AND (NAND) gate.

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Figure 5.11 Truth table for the NAND gate.

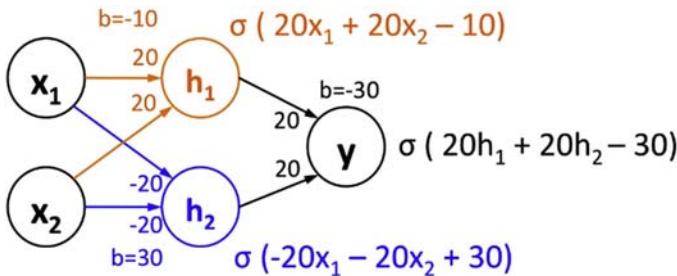


Figure 5.12 A neural network that can represent a NAND gate.

proven here because it is unrelated to the subject of unmanned vehicles. Fig. 5.12 shows a simple neural network that can describe a NAND gate, where (x_1, x_2) is the input level and σ is the activation function.

A complete and complex neural network can be equivalent to the combination of many NAND gates. By combining these gate circuits, our network can be used to express various functions. Then, how do we adjust the network to express the specified function? From the example of perception, we determine that network functions can be adjusted by adjusting parameters in the network, namely, weight and bias. As described in the previous section, we used the gradient descent algorithm to adjust the weights and biases in the neural network to fit functions. In the process of adjusting the parameters, we want a small adjustment of the parameters to be reflected in the output, as shown in Fig. 5.13.

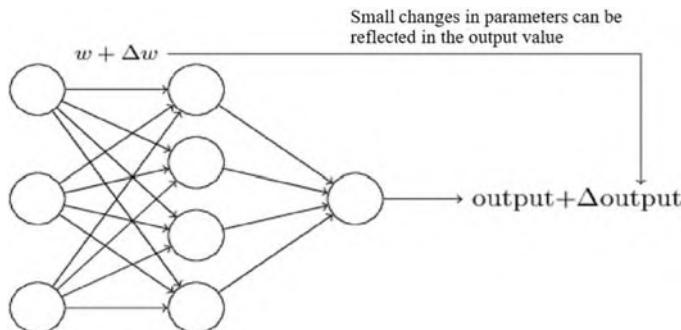


Figure 5.13 Changes in the parameters can be reflected in the output of the neural network.

5.3.3 Forward transmission

The unlimited capacity of the neural network can be used to handle handwritten digit recognition. The handwritten digit pictures are all gray images. The pixels are 28×28 ; thus, the length of the input feature X is 784. The handwritten digit pictures have 10 categories, that is, from 0 to 9. Assume that the three-layer neural network shown in Fig. 5.14 is used to process this task.

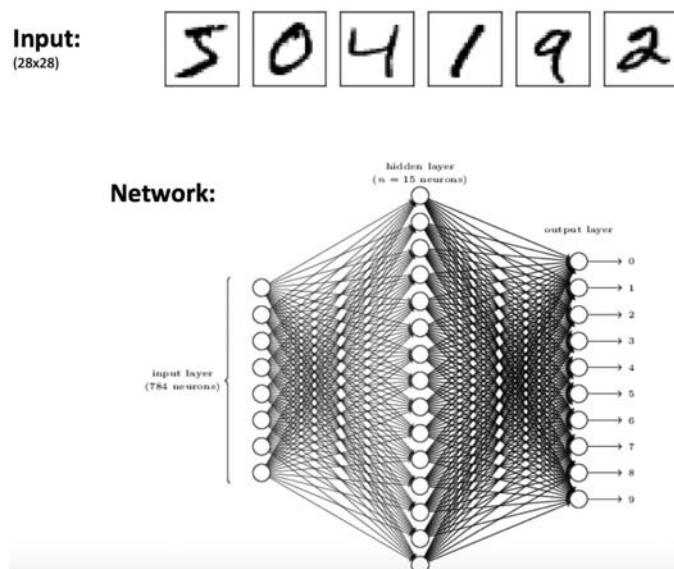


Figure 5.14 A neural network for handling the MNIST handwritten digit recognition problem.

The output of each neuron in the output layer corresponds to a category, and the output layer has 10 neurons. The single-point activation encoding method is called one-hot encoding, that is, the position of the corresponding category is 1, whereas the other positions are 0. This encoding method is effective when the number of categories is small.⁷

The output of the neural network is the score of each point, which is called logit. What we want to determine is the probability that each neuron in the output layer is activated, not the score. Therefore, we add a softmax function in the output layer of the neural network, whose function is to convert the score of the neuron into the probability, which is easy to understand. The sum of all 10 probabilities is 1. The higher the output probability is, the higher the probability of predicting a certain number. The calculation formula of the softmax function can be expressed as follows:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad j = 1, \dots, K.$$

More intuitively, we use Python to implement the softmax function, as follows:

Code list 5.3.1 Using Python to implement the softmax function

```
import numpy as np

def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x)
    return e_x / e_x.sum()

scores = [3.0, 1.0, 0.2]

print(softmax(scores))
```

The output of the previously presented code is [0.836 0.18 8 0.113 142 84 0.050 83 6]. For specific tasks, the corresponding loss function needs to be defined. For pattern recognition multiclassification tasks, the most commonly used loss function is cross-entropy, which can be expressed as follows:

$$L(\theta) = -\frac{1}{n} \sum_n [y \ln a + (1 - y) \ln(1 - a)],$$

where y is the real label and a is the output of the neuron. Such a loss function has the following satisfactory properties:

- Nonnegative: We can minimize the loss function.
- When the real value is close to the output value of the model, the value of the loss function tends to 0.

After the loss function is calculated, we obtain the output of the neural network (for handwritten digit recognition, it is the prediction result of recognition) and the distance between the output and the real value. We call such a process “forward propagation.” The premise is that forward propagation can achieve correct classification because the neural network already has the appropriate parameters. Thus, how do we adjust the appropriate parameters? The neural network makes the information of the loss function flow forward along the network through the backpropagation algorithm to calculate the gradient and update the weights. The backpropagation algorithm only updates the parameters once in an epoch. Usually, multiple epochs need to be iterated to make the network converge to the appropriate model.

5.3.4 Stochastic gradient descent

Stochastic gradient descent (SGD) is a learning algorithm adopted by nearly all deep learning models. In deep learning, a good model often needs a large amount of data, and a large amount of data will consume a large amount of computation time. As discussed in the previous section, the loss function in the machine learning algorithm is obtained through the calculation of all samples. Thus, the loss of training data can be written as follows:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n L(x_i, y_i, \theta).$$

To perform gradient descent, we need to define the loss function as follows:

$$\nabla_{\theta} L(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(x_i, y_i, \theta).$$

In other words, the gradient descent algorithm uses the entire sample set to update the gradient for each step, which is inefficient for large datasets (millions or tens of millions of samples). In SGD, instead of using the entire sample set to calculate the gradient, we randomly select a minibatch of samples

each time. The sample size of this minibatch is usually less than a 1000 (the sample size of the minibatch is usually an exponent of 2, such as 64, 128, and 256). Assuming that the sample size of the minibatch is m , then every time we perform gradient descent, we only need to randomly select and utilize m samples to train the neural network. After forward propagation, the loss function is calculated and the gradient of m samples is calculated in reverse. The gradient of m samples can be regarded as the estimate of the gradient of the entire sample set, which can be expressed as follows:

$$g = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(x_i, y_i, \theta).$$

Then, the following gradient estimator is used to update the parameters:

$$\theta \leftarrow \theta - \varepsilon g,$$

where ε is the learning rate.

SGD is the main method for training large linear models on large-scale data. Before the advent of deep learning, the main method for learning nonlinear models was linear models combined with kernel techniques. When the sample size of the dataset is large, the amount of computation time of this kind of method is unacceptable. In academia, the reason deep learning began to gain attention in 2006 was that, on medium-sized datasets with tens of thousands of samples, deep learning was better at generalizing new samples than many popular algorithms at the time.

5.4 Using Keras to implement the neural network

We use the Keras library to implement a three-layer neural network model rapidly. We use cross-entropy as the loss function and SGD as the optimization algorithm to train the model. After training, we saved the model as a “model.json” file and used our handwriting to verify the accuracy of the model. We completed the training, adjustment, and verification of the model in the Jupyter Notebook.

5.4.1 Data preparation

First, the MNIST handwritten digit dataset needs to be downloaded locally. Then, the dataset is read into memory and split into the training and

test sets. At the same time, the required library is imported and the superparameter is defined. After reading the dataset, the size of the dataset is printed.

Code List 5.4.1 Modified National Institute of Standards and Technology dataset preparation

```
from __future__ import print_function

import keras

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.optimizers import SGD

from matplotlib import pyplot as plt

batch_size = 128

num_classes = 10

epochs = 20

(x_train, y_train), (x_test, y_test) = mnist.load_data()

print(x_train.shape, x_test.shape)

print(y_train.shape, y_test.shape)
```

The results are as follows:

```
(60000, 28, 28) (10000, 28, 28)

(60000,) (10000,)
```

The training set of this dataset has 60,000 samples, and the test set of this dataset has 10,000 samples. Each sample is a 28×28 image, and we use the `matplotlib.pyplot` package to display part of these images.

Code list 5.4.2 Display of the first 16 images in the training set

```
def show_samples(samples, labels):  
    """  
    display 16 samples and labels  
    """  
  
    plt.figure(figsize=(12, 12))  
  
    for i in range(len(samples)):  
  
        plt.subplot(4, 4, i+1)  
  
        plt.imshow(samples[i], cmap='gray')  
  
        plt.title(labels[i])  
  
    plt.show()  
  
show_samples(x_train[:16], y_train[:16])
```

The results are shown in Fig. 5.15, which displays the images and labels of 16 samples of the MNIST handwritten digit dataset, indicating that the data are read correctly.

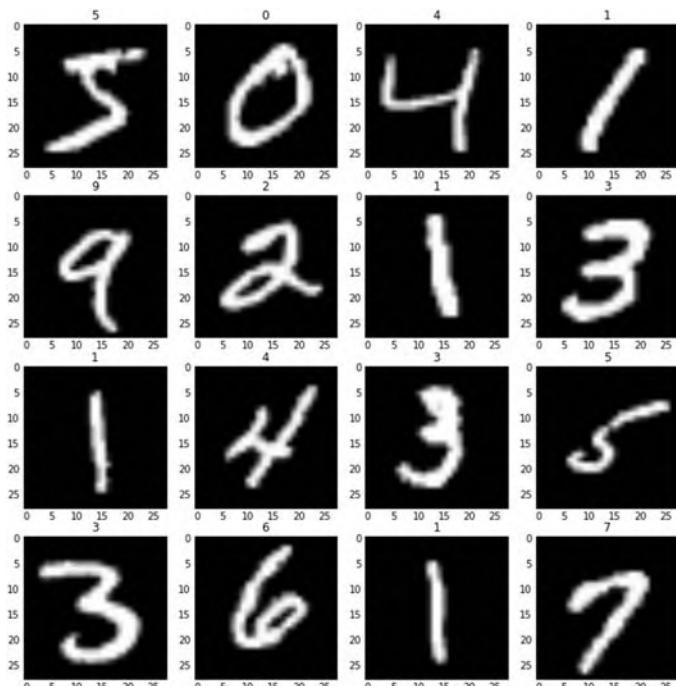


Figure 5.15 First 16 handwritings and their corresponding labels.

Because the input dimension received by the neural network is 784, the characteristics of the sample need to be adjusted, and the label should be one-hot encoded at the same time.

Code list 5.4.3 Structure adjustment, normalization, and one-hot encoding

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

print(x_train.shape, x_test.shape)
print(y_train.shape, y_test.shape)
```

After the transformation, the contents of the data and labels remain the same, but the structure (or format of the matrix) becomes the following:

```
(60000, 784) (10000, 784)
(60000, 10) (10000, 10)
```

We construct a three-layer neural network. First, we only use 15 hidden layer neurons to observe the training effect.

Code list 5.4.4 Constructing a neural network with only 15 hidden layer neurons

```
model = Sequential()

model.add(Dense(15, activation='relu', input_shape=(784,)))

model.add(Dense(num_classes, activation='softmax'))


model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.01),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                     batch_size=batch_size,
                     epochs=epochs,
                     verbose=1,
                     validation_data=(x_test, y_test))

print(history.history.keys())
```

Then, we draw the losses during the training process.

Code list 5.4.5 Drawing the losses during the training process

```
def plot_training(history):

    ### plot the training and validation loss for each epoch

    plt.plot(history.history['loss'])

    plt.plot(history.history['val_loss'])

    plt.title('model mean squared error loss')

    plt.ylabel('mean squared error loss')

    plt.xlabel('epoch')

    plt.legend(['training set', 'validation set'], loc='upper right')

    plt.show()


plot_training(history=history)
```

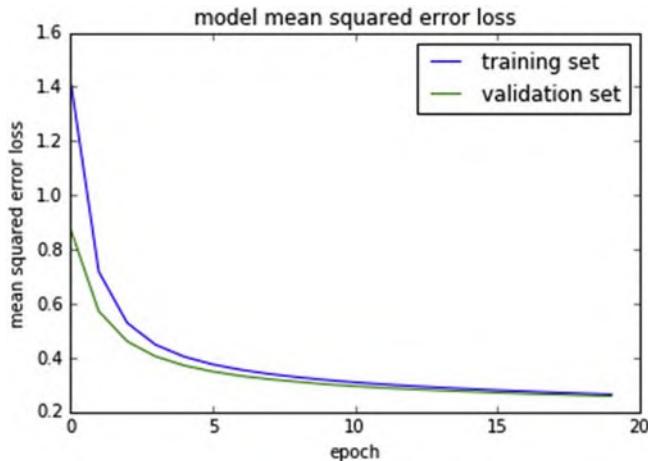


Figure 5.16 Loss changes during the training cycle.

We observed that the losses of both training and verification sets show a downward trend, and the trend became stable after approximately 20 epochs and approached 0.2, as shown in Fig. 5.16.

Then, we used the test set to verify the accuracy of model recognition.

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

The test results of this three-layer neural network on never seen handwritten digits (i.e., test set) are as follows:

```
Test loss: 0.2586659453
Test accuracy: 0.9249
```

We determine that this “simplest” neural network model has achieved 92% recognition accuracy in the test set, but it has not reached a practical level. To improve the recognition accuracy of our model, we change the code based on the deep neural network model. The reader may be confused with the code behind it; the relevant theoretical knowledge will be discussed in detail in the pertinent chapters of deep learning.

5.4.2 A small change in three-layer neural network—deep feedforward neural network

First, we deepen the original network layer into two hidden layers. At the same time, the number of neurons in each hidden layer is expanded to 512, and a regularization technique called Dropout⁸ is used behind each hidden layer. Finally, we use the RMSProp algorithm, a variant of SGD, as the learning algorithm for the model.

Code list 5.4.6 First deep neural network

```
from keras.layers import Dropout  
  
from keras.optimizers import RMSprop  
  
  
model = Sequential()  
  
model.add(Dense(512, activation='relu', input_shape=(784,)))  
  
model.add(Dropout(0.2))  
  
model.add(Dense(512, activation='relu'))  
  
model.add(Dropout(0.2))  
  
model.add(Dense(num_classes, activation='softmax'))  
  
  
model.summary()  
  
  
  
model.compile(loss='categorical_crossentropy',  
              optimizer=RMSprop(),  
              metrics=['accuracy'])  
  
  
history = model.fit(x_train, y_train,  
                     batch_size=batch_size,  
                     epochs=epochs,  
                     verbose=1,  
                     validation_data=(x_test, y_test))
```

```

    ### print the keys contained in the history object
    print(history.history.keys())

    plot_training(history=history)

    model.save('model.json')

    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test loss:', score[0])
    print('Test accuracy:', score[1])

```

Keras can output the network structure and the number of parameters during the compilation of the model, as follows:

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 512)	401920
dropout_1 (Dropout)	(None, 512)	0
dense_4 (Dense)	(None, 512)	262656
dropout_2 (Dropout)	(None, 512)	0
dense_5 (Dense)	(None, 10)	5130

Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0

Changes in the loss function during training are shown in Fig. 5.17. The test results of the test set are as follows:

```
Test loss: 0.117383948493
```

```
Test accuracy: 0.9824
```

The classification accuracy of the four-layer neural network is 98%, which is better than that of the previously discussed three-layer neural network. Then, we extract 16 images from the test set to observe the effect of model recognition.

```
import numpy as np

result = model.predict(x_test[:16])

result = np.argmax(result, 1)

print('predict: ', result)

true = np.argmax(y_test[:16], 1)

print('true: ', true)
```

The verification results are as follows:

```
predict: [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5]
```

```
true: [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5]
```

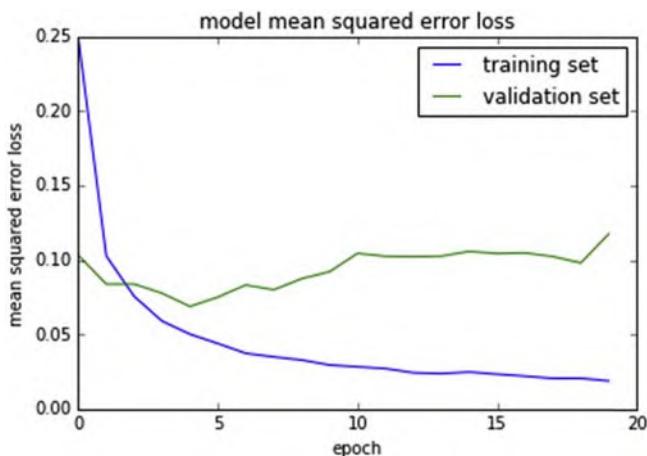


Figure 5.17 Curve of the loss function.

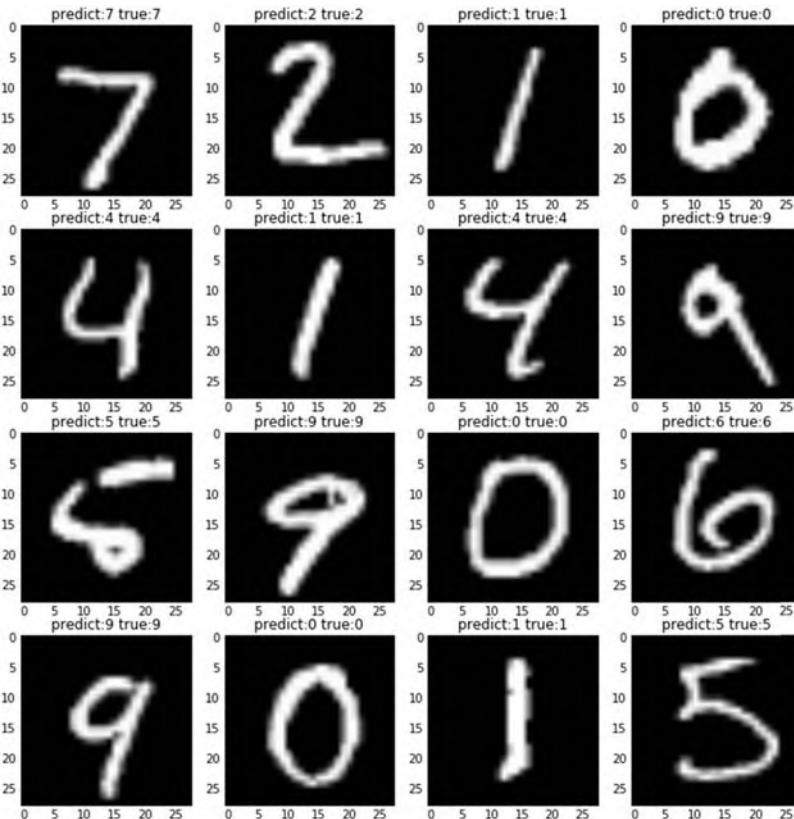


Figure 5.18 Classification results of the MNIST handwritten digit dataset.

The actual handwritten digit pictures are shown in Fig. 5.18 with the following code:

```
fig2 = plt.figure(figsize=(12, 12))

for i in range(16):

    plt.subplot(4, 4, i+1)

    plt.imshow(x_test[i].reshape((28, 28)), cmap='gray')

    plt.title('predict:' + str(result[i]) + ' true:' + str(true[i]))

plt.show()
```

5.4.3 Summary

This chapter introduces the basic principles of neural networks and implements the MNIST handwritten digit recognition neural network code based on the Keras deep learning application programming interface. Then, we will learn the application of deep learning in the unmanned sensing module and the end-to-end unmanned vehicle technology based on deep learning. We will further explore and apply the theory and technology of the combination of deep neural networks and reinforcement learning to unmanned vehicles.

References

1. Zhou Y, Tuzel O. *VoxelNet: End-To-End Learning for Point Cloud Based 3D Object detection[J]*. arXiv; 2017.
2. Lecun Y, Cortes C. *The Mnist Database of Handwritten Digits[Z]*. 2010.
3. Ian G, Bengio Y, Courville A. *Deep Learning[M]*. MIT Press; 2016.
4. MIT6. S094: Deep Learning for Self-Driving Cars[EB/OL]. <https://selfdrivingcars.mit.edu/>.
5. Li H. *Statistical Learning Methods[M]*. Beijing: Publishing House of Tsinghua University. 2012 (Ch).
6. Rosenblatt F. The perception: a probabilistic model for information storage and organization in the brain. *Psychol Rev*. 1958;65(6):368–408.
7. Deep Learning Tutorials[EB/OL]. <https://deeplearning.net/tutorial/>.
8. Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: a simple way to prevent neutral networks from overfitting. *J Mach Learn Res*. 2014;15(1):1929–1958.

This page intentionally left blank

CHAPTER 6

Deep learning and visual perception

Zebang Shen¹, Tingting Yu², Peng Zhi² and Rui Zhao²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

6.1 Deep feedforward neural networks—why is it necessary to be deep?	178
6.1.1 Efficiency of model training under big data	178
6.1.2 Representation learning	179
6.2 Regularization technology applied to deep neural networks	180
6.2.1 Data augmentation	180
6.2.2 Early stopping	183
6.2.3 Parameter normalization penalties	184
6.2.4 Dropout	185
6.3 Actual combat—traffic sign recognition	187
6.3.1 Belgium traffic sign dataset	187
6.3.2 Data preprocessing	193
6.3.3 Leverage Keras to construct and train a deep feedforward network	195
6.4 Introduction to convolutional neural networks	196
6.4.1 What is convolution and the motivation for convolution	197
6.4.2 Sparse interactions	199
6.4.3 Parameter sharing	202
6.4.4 Equivariant representations	202
6.4.5 Convolutional neural network	202
6.4.6 Some details of convolution	204
6.5 Vehicle detection based on YOLO2	206
6.5.1 Pretrained classification network	207
6.5.2 Train the detection network	208
6.5.3 Loss function of YOLO	209
6.5.4 Test	210
6.5.5 Vehicle and pedestrian detection based on YOLO	210
References	216

6.1 Deep feedforward neural networks—why is it necessary to be deep?

At the end of the last section of [Chapter 5](#), we have been introduced to deep feedforward neural networks, that is, a four-layer neural network is adopted to solve the Modified National Institute of Standards and Technology (MNIST) handwritten digit recognition problem¹, and it achieved a recognition rate of 98%. In simple terms, deep feedforward neural networks are based on the “deep” three-layer backpropagation neural network.

According to the previous discussion of neural networks, only three hidden layers of the neural network can fit any function. So, why do we deepen the number of layers in the network? This question should be viewed from two aspects: one is the efficiency of model training under big data, and the other is representation learning.

6.1.1 Efficiency of model training under big data

Breakthroughs in deep learning can be attributed to the following elements:

- Neural network theory (the long-established theoretical basis),
- Big data (through the development of the Internet),
- Stronger parallel computing capabilities (represented by the graphics processing unit).

Big data is an important factor in the improvement of the performance of deep neural networks. Traditional machine learning algorithms generally fall into a performance bottleneck after the amount of data increases to a certain order of magnitude, particularly for the support vector machine (SVM) that is based on structural risk minimization. The performance of the SVM will be saturated after the amount of data reaches a certain level. By contrast, the neural network is a machine learning algorithm that can be continuously expanded. When the amount of data increases, more powerful neural networks can be trained by increasing the number of neurons and hidden layers. Its change trend is shown in [Fig. 6.1](#).

As mentioned previously, by increasing the number of neurons in the hidden layer, a simple three-layer neural network can theoretically fit any function. So why don’t we directly use a simple three-layer neural network structure to fit complex problems?

Simply increasing the number of neurons in a single hidden layer can enhance the representation capability of the model. However, compared with increasing the number of layers and each layer using a relatively small number of neurons, the former is more difficult to train successfully in actual training and contains the overfitting problem that is difficult to control. To achieve the same performance, a deep-layer neural network often requires fewer neurons than a three-layer neural network.

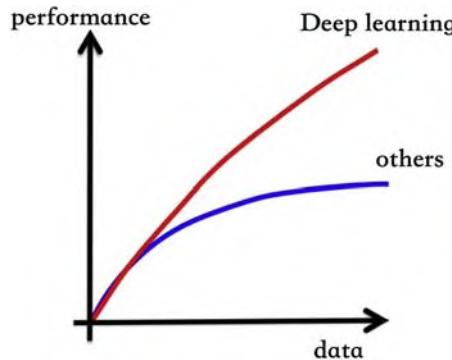


Figure 6.1 Impact of data volume on model performance.

6.1.2 Representation learning

Another explanation for deep learning is deep representation learning (also known as feature learning). [Fig. 6.2](#) is a visualization of the activation of hidden layers and neurons in a multilayer convolutional neural network (CNN) after inputting an image.

As shown in [Fig. 6.2](#), the first several layers of the neural network play a role in feature extraction and representation establishment. This is different

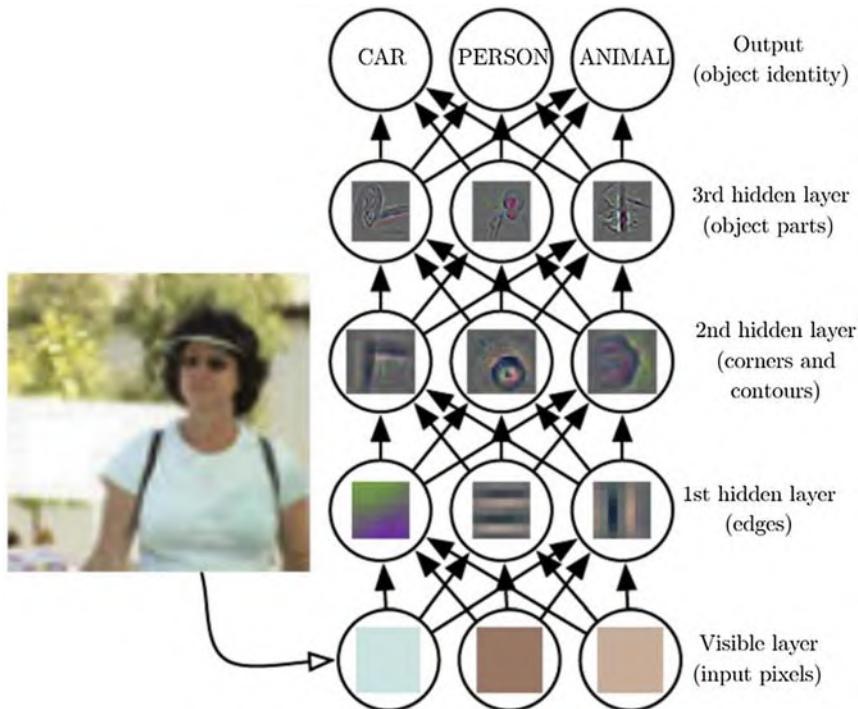


Figure 6.2 Deep learning is a process of spontaneously establishing representations.

from the artificial design features of traditional machine learning methods. The feature design of the neural network is executed with the training of the neural network, which is a process of automatic representation establishment. Fig. 6.2 also shows that the closer the layers are to the input, the simpler the features the layers will extract. By contrast, the further the layers are from the input, the more complex the features the layers will extract. For example, the first layer extracts “edge” features, the second layer extracts contour features, and the third layer combines simple underlying features to synthesize more advanced representations as local features of the recognition object. By abstracting the features layer by layer, the more the layers of the neural network are, the richer the feature representations it can acquire.

6.2 Regularization technology applied to deep neural networks

When the number of hidden layers and neurons in the neural network increases, the number of parameters increases substantially, which leads to an excessively large model capacity of the neural network. The model capacity of neural networks, particularly deep neural networks, is nearly always too large. An excessively large model capacity has advantages and disadvantages. On the one hand, an excessively large model capacity means a strong representation capability, that is, the network can learn more complex mapping relationships. On the other hand, an excessively large model capacity will make the training of the model “uncontrollable,” that is, the network may be more prone to overfitting during training; thus, the model performs well on the training set, but its generalization capability is poor. In machine learning, some strategies are proposed to reduce the gap between test and training errors, that is, improving the generalization capability of the model and enhancing the robustness of the model. These strategies are collectively referred to as “regularization.” The four common regularization techniques will be introduced in the following subsections of this chapter:

- Data augmentation,
- Early stopping,
- Parameter normalization penalties,
- Dropout.

6.2.1 Data augmentation

One of the most intuitive strategies for enhancing the robustness of machine learning (or improving the generalization capability of the model) is

to leverage more data to train the model (also known as data augmentation). However, the collected data are limited in reality; thus, new data are generated by transforming the raw data. For some machine learning tasks, such as image classification, the raw data can be easily modified. We take the MNIST handwritten digit dataset as an example to illustrate it.

Fig. 6.3 shows three characters in the training set of the MNIST handwritten digit dataset. For the raw image data, simple translation, such as rotation, can be utilized to generate new data.

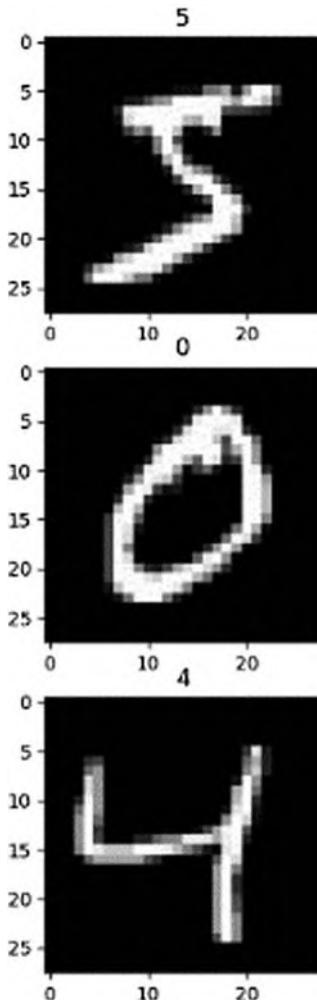


Figure 6.3 Three characters in the MNIST handwritten digit dataset.

Code list 6.1 Data expansion code

```

def expend_training_data(train_x, train_y):

    expanded_images = np.zeros([train_x.shape[0] * 5, train_x.shape[1],
train_x.shape[2]])

    expanded_labels = np.zeros([train_x.shape[0] * 5])

    counter = 0

    for x, y in zip(train_x, train_y):

        expanded_images[counter, :, :] = x

        expanded_labels[counter] = y

        counter = counter + 1

    bg_value = np.median(x) # this is regarded as background's value

    for i in range(4):

        # rotate the image with random degree

        angle = np.random.randint(-15, 15, 1)

        new_img = ndimage.rotate(x, angle, reshape=False,
cval=bg_value)

        # shift the image with random distance

        shift = np.random.randint(-2, 2, 2)

        new_img_ = ndimage.shift(new_img, shift, cval=bg_value)

        # register new training data

        expanded_images[counter, :, :] = new_img_

        expanded_labels[counter] = y

        counter = counter + 1

    return expanded_images, expanded_labels

agument_x, agument_y = expend_training_data(x_train[:3], y_train[:3])

```

Through the aforementioned operations, a new dataset is five times that the original dataset, which part of the data is shown as Fig. 6.4.

The four columns to the right of the image are obtained through transformation instead of adding the new data collected. By transforming

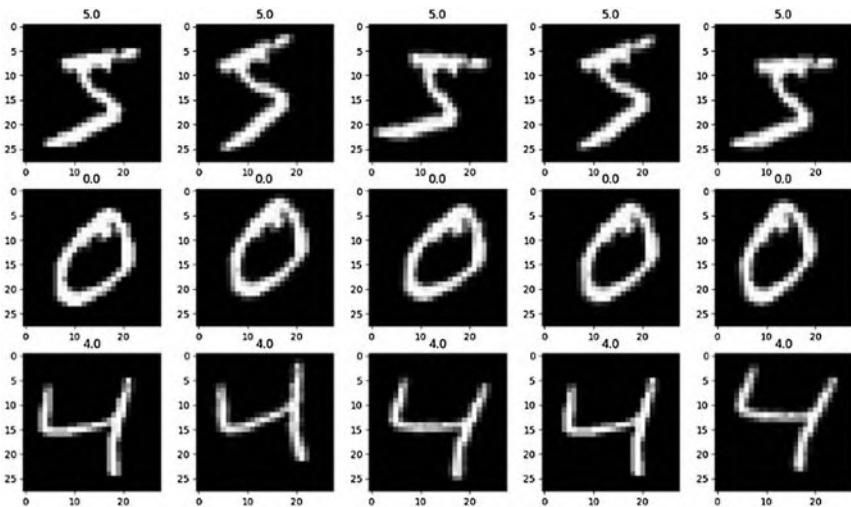


Figure 6.4 Data enlarged five times.

the data, our dataset has become several times larger than the original dataset. This processing method can significantly improve the generalization capability of the neural network. Even for a CNN with translation invariance, which will be discussed in detail subsequently, the new data obtained using this simple processing method can considerably improve the generalization capability of the model.

6.2.2 Early stopping

When the trained neural network is too complex, it always tends to overfit. In this case, the training error will reduce gradually over the training time, but the validation error will initially decrease and subsequently increase, as shown in Fig. 6.5.²

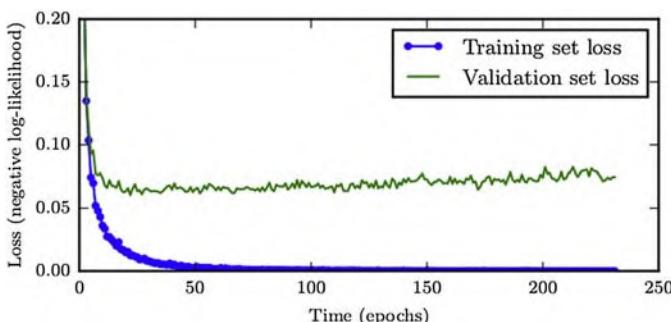


Figure 6.5 Changes in training and verification losses with training iterations.

Based on this phenomenon, we can save a copy of the model once the validation error has been corrected. If the error worsens, then the patience value is increased by 1. The training is terminated when the patience value reaches a predetermined threshold, and the last saved copy is retrieved. In this manner, we obtain the model with the lowest error.

6.2.3 Parameter normalization penalties

Many regularization methods will add a penalty term $\Omega(w)$ to the loss function $L(\theta)$ of the neural network to constrain the learning capability of the model, as follows:

$$L'(\theta) = L(\theta) + \alpha\Omega(w),$$

where θ is the parameter of the neural network that includes weight and bias (w, b) . Why is $\Omega(w)$ called the “penalty term”? The standard of neural network learning is to minimize the loss function. Therefore, the addition of a new item to the loss function means to punish or minimize that item. The formula shows that the penalty term $\Omega(w)$ is a function of the weight w . In other words, we want to optimize the loss function $L'(\theta)$, which is essentially to minimize the original loss function $L(\theta)$ and the function $\Omega(w)$ with respect to the weight w by adjusting (using the stochastic gradient descent algorithm) the parameters of the neural network. Notably, the penalty term usually only penalizes the weight (i.e., w) in the affine transformation, whereas the bias unit b will not be regularized. The reason is that each weight indicates how the two variables interact with each other. To fit the weights effectively, we need to observe these variables under various conditions. Each bias only controls a single variable, which means that we do not need to import too much variance when retaining the bias that is unregularized. Similarly, regularizing the bias parameters will introduce a considerable degree of underfitting possibility; thus, only the weights are penalized.

α is a hyperparameter that needs to be manually set called a penalty coefficient. When the value of α is 0, no parameter normalization penalty is needed. The larger the value of α is, the larger the penalty of the corresponding parameter. Parameter normalization penalty is divided into L1 and L2 norm penalties. The L2 norm penalty (also known as weight decay or L2 regularization³) will be illustrated as an example. The regularization term added after the loss function can be expressed as follows:

$$\alpha\Omega(w) = \frac{\alpha}{2}w^Tw.$$

The sum of the squares of all weights is added. So what will be the result of minimizing the square of the weights?

- The neural network will tend to make all weights small unless the error derivative is too large.
- Minimizing the square of the weights prevents incorrectly fitting samples.
- Minimizing the square of the weights makes the model “smoother,” that is, the input and output sensitivities are lower and small changes in the input will not be reflected on the output.
- If two same inputs are entered, then network weight allocation will tend to divide the weights based on the average instead of assigning all of the weights to one connection.

On the one hand, the L2 norm penalty reduces the degree of freedom of weight learning and weakens the learning capability of the network. On the other hand, relatively uniform weights can smoothen the model and make the model insensitive to subtle changes in the input, thereby enhancing the robustness of the model.

6.2.4 Dropout

The parameter normalization penalty is regularized by changing the loss function of the neural network, whereas dropout enhances the generalization capability of the network by changing the structure of the neural network during training. Fig. 6.6 shows the structure of a neural network during training.

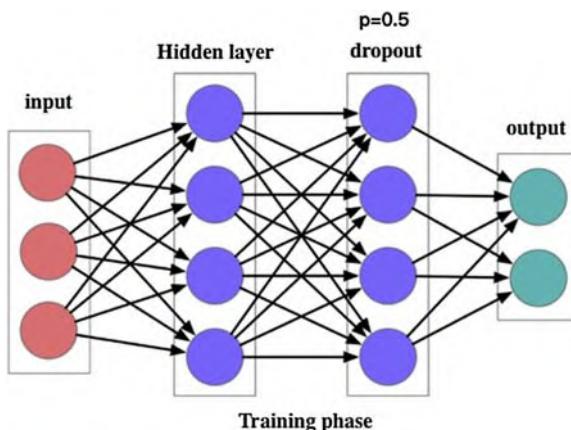


Figure 6.6 General neural network structure.

A dropout layer is attached after the first hidden layer. Dropout refers to randomly deleting nodes in a certain layer of the network, including the input and output edges of the node, as shown in Fig. 6.7.

Dropout is also equivalent to retaining nodes with a certain probability. In this example, p , which is the probability of retaining the node, is 50%. In practice, the retention probability is usually set at $[0.5, 1]$. So why does dropout help prevent overfitting? A simple explanation is that the use of the dropout training process is equivalent to training a large number of neural networks with only half of the hidden layer units (hereinafter referred to as

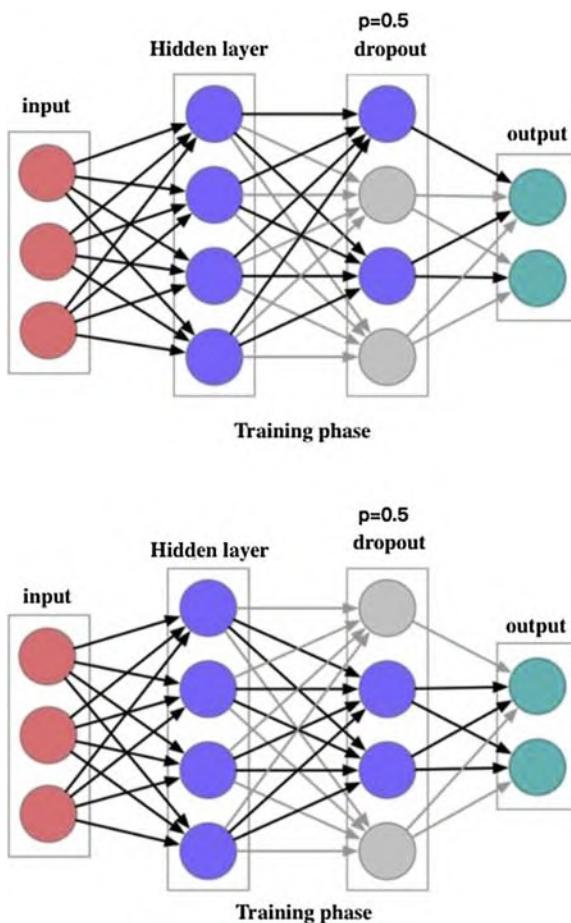


Figure 6.7 Dropout method randomly deletes connections in the neural network.

“half-networks”) where each of these half-networks can be given a classification result. Some of these results will be correct and some will be wrong. In terms of the training progress, most half-networks can provide correct classification results, whereas a small number of incorrect classification results will not considerably impact the final result. Thus, when the training process is over, we can consider this neural network an integrated model of many well-trained half-networks. During the application stage of the network, dropout will no longer be used. The final output of the network is the integration of all half-networks. As a result, the generalization capability of the network will be good.

6.3 Actual combat—traffic sign recognition

In this section, the traffic sign recognition tasks will be accomplished based on deep learning. Traffic signs, also called road signs and road traffic signs, are road facilities that use words or symbols to convey guidance, restrictions, warnings, and instructions. Providing unmanned vehicles with the capability to recognize traffic signs is an important prerequisite for ensuring that unmanned vehicles drive in accordance with actual traffic rules. At present, in the actual research and development of unmanned driving systems, the solution ideas for understanding traffic signs are divided into the following types:

- Directly write traffic sign information into a high-precision map: This solution writes the traffic sign information of a road section into a semantic map when the high-precision map is drawn so that the vehicle only needs to have accurate positioning to determine the traffic rules of the road section (i.e., speed limit and warning signs).
- Recognize the traffic signs in the images captured by the camera: This solution uses computer vision technology (including traditional vision and deep learning methods) to understand whether the unmanned vehicle camera sensor image contains traffic signs and what traffic signs are included.

6.3.1 Belgium traffic sign dataset

We use the Belgium Traffic Sign Dataset⁴ (BelgiumTS) for neural network exercises. BelgiumTS is a traffic signal dataset that includes 62 traffic signals, the data of which are shown in Fig. 6.8.



Figure 6.8 Belgium traffic sign dataset.

Download link for the training set:

http://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Training.zip

Download link for the test set:

http://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Testing.zip

After downloading and unzipping the data, we use the following directory structure to store the data:

```
data/Training/
```

```
  data/Testing/
```

Each of the training and test sets of this dataset contains 62 directories representing 62 types of traffic signals. Python is used to write the functions to read data, as follows:

Code list 6.2 Data reading function

```

def load_data(data_dir):

    directories = [d for d in os.listdir(data_dir)
                  if os.path.isdir(os.path.join(data_dir, d))]

    labels = []
    images = []

    for d in directories:

        label_dir = os.path.join(data_dir, d)
        file_names = [os.path.join(label_dir, f)
                      for f in os.listdir(label_dir) if f.endswith(".ppm")]

        for f in file_names:

            images.append(skimage.data.imread(f))
            labels.append(int(d))

    return images, labels

# Load training and testing datasets.

ROOT_PATH = "data"

train_data_dir = os.path.join(ROOT_PATH, "Training")
test_data_dir = os.path.join(ROOT_PATH, "Testing")

images, labels = load_data(train_data_dir)

```

The number of categories and the total number of images in the training set are obtained as follows:

```

print("Unique Labels: {0}\nTotal Images:
{1}".format(len(set(labels)), len(images)))

```

Then, the total number of categories and the total number of sample images are obtained as follows:

```

Unique Labels: 62
Total Images: 4575

```

The first picture of each category is as follows:

Code list 6.3 First picture of each category

```
def display_images_and_labels(images, labels):

    unique_labels = set(labels)

    plt.figure(figsize=(15, 15))

    i = 1

    for label in unique_labels:

        # Pick the first image for each label.

        image = images[labels.index(label)]

        plt.subplot(8, 8, i) # A grid of 8 rows x 8 columns

        plt.axis('off')

        plt.title("Label {0} ({1})".format(label, labels.count(label)))

        i += 1

        _ = plt.imshow(image)

    plt.show()

display_images_and_labels(images, labels)
```

The results are shown in Fig. 6.9.

Notably, the pictures in the dataset do not have a uniform size. To train the neural network, we need to resize all pictures into the same size. In this chapter, the resizing format is 32×32 .

Code list 6.4 Using skimage to unify the image size to 32×32

```
images32 = [skimage.transform.resize(image, (32, 32))

            for image in images]

display_images_and_labels(images32, labels)
```



Figure 6.9 62 Types of traffic sign samples.

The sample obtained after resizing is shown in Fig. 6.10.

Code list 6.5 Output picture information after resizing

```
for image in images32[:5]:  
    print("shape: {0}, min: {1}, max: {2}".format(image.shape,  
image.min(), image.max()))
```



Figure 6.10 Image samples after resizing.

Output information:

```
shape: (32, 32, 3), min: 0.0, max: 1.0
shape: (32, 32, 3), min: 0.13088235294117614, max: 1.0
shape: (32, 32, 3), min: 0.057059972426470276, max: 0.9011967677696078
shape: (32, 32, 3), min: 0.023820465686273988, max: 1.0
shape: (32, 32, 3), min: 0.023690257352941196, max: 1.0
```

The value range of the image is normalized. Then, Keras is adopted to construct a neural network to train a deep feedforward network to recognize traffic signals.

6.3.2 Data preprocessing

During data preprocessing, we convert the three-channel RGB image into a grayscale image.

Code list 6.6 Three-channel RGB image into a grayscale image

```
images_a = color.rgb2gray(images_a)

display_images_and_labels(images_a, labels)
```

We use *matplotlib* to visualize the results of our conversion, as shown in Fig. 6.11.

Notably, what is shown in Fig. 6.11 is not a grayscale image. The reason is that we used the previous *display_images_and_labels* function where only the parameter *cmap* is set to gray to display the grayscale image.

The previously presented method is adopted to expand the data up to five times than the original dataset. Fig. 6.12 visualizes the three pictures adopted the above method.



Figure 6.11 Three-channel RGB image converted into a grayscale image.

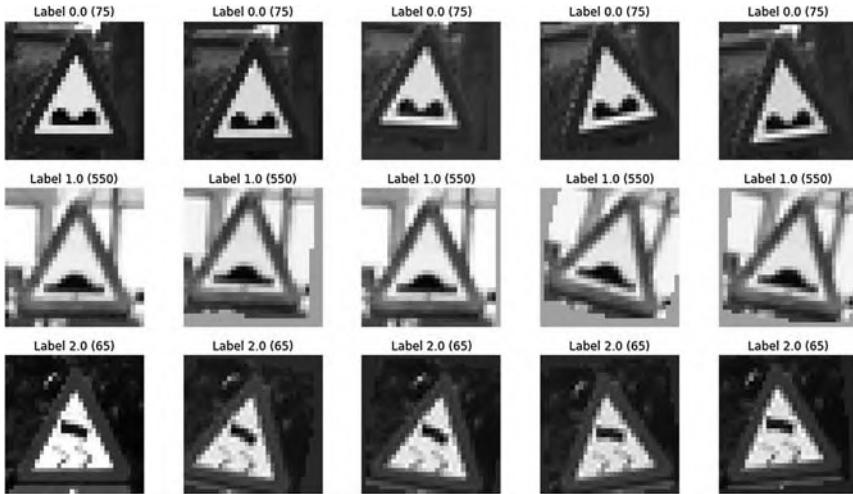


Figure 6.12 Dataset after data enhancement.

Then, a shuffle operation is performed on the pictures to disrupt the order of the samples in our dataset, that is, rearrange the indices of the samples. The shuffle operation is a necessary step in the stochastic gradient descent algorithm. The dataset is divided into the training and validation sets, the first 20,000 samples are the training set, and the remaining samples are the test set. Finally, the one-hot encoding operation is executed on the labels.

Code list 6.7 Data preparation

```

from sklearn.utils import shuffle

idx = np.arange(0, len(labels_a))

idx = shuffle(idx)

images_a = images_a[idx]

labels_a = labels_a[idx]

print(images_a.shape, labels_a.shape)

train_x, val_x = images_a[:20000], images_a[20000:]

train_y, val_y = labels_a[:20000], labels_a[20000:]

train_y = keras.utils.to_categorical(train_y, 62)

val_y = keras.utils.to_categorical(val_y, 62)

print(train_x.shape, train_y.shape)

```

6.3.3 Leverage Keras to construct and train a deep feedforward network

The deep feedforward network mentioned previously can, on the one hand, verify the performance in more complex situations and illustrate the “universality” of neural networks for pattern recognition problems, and on the other hand, help us understand the deep neural network code presented previously.

Code list 6.8 Constructing a multilayer feedforward neural network

```
model = Sequential()

model.add(Flatten(input_shape=(32, 32)))

model.add(Dense(512, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(512, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(62, activation='softmax'))


model.summary()


model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])


history = model.fit(train_x, train_y,
                     batch_size=128,
                     epochs=20,
                     verbose=1,
                     validation_data=(val_x, val_y))

### print the keys contained in the history object
print(history.history.keys())

model.save('model.json')
```

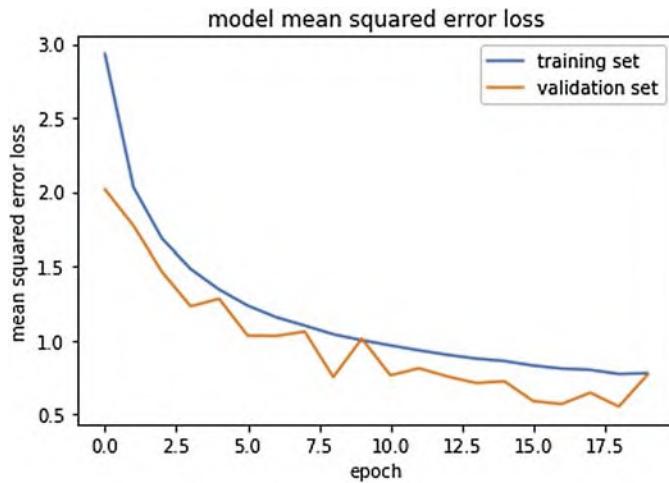


Figure 6.13 Training and validation error curves.

The changes in the training and verification losses of the network are shown in Fig. 6.13.

The test set is loaded, and its accuracy is checked.

```
('Test loss:', 0.8060373229994661)
('Test accuracy:', 0.7932539684431893)
```

This simple neural network achieved 79% accuracy on the test set. The prediction results of several test samples are shown in Fig. 6.14.

The simple feedforward neural network applied previously to the MNIST handwritten digit recognition problem only achieved 79% recognition accuracy in the traffic sign recognition tasks. Of course, this example is just an introductory network. First, the three-channel RGB image is discarded; thus, loss of information will occur to a certain extent. Second, a fully connected neural network initially vectorizes the image. A deeper network that is more in line with the two-dimensional characteristics of the picture will continuously be explored in the subsequent section.

6.4 Introduction to convolutional neural networks

In this section, a type of neural network, that is, CNN, which is specifically used to process data with a grid structure, will be illustrated.



Figure 6.14 Recognition effect on the test set.

6.4.1 What is convolution and the motivation for convolution

Convolution is a special linear operation, that is, a mathematical operation on two real-valued functions. Convolution operations are usually represented by the asterisk symbol (*). For ease of understanding, a one-dimensional convolution with a discrete form will be discussed.

Suppose a recoverable spacecraft is landing, and its sensors continuously measure its altitude information. $h(i)$ denotes the altitude measurement at the moment i . This measurement occurs at a certain frequency (i.e., it is measured every other time interval; thus, the measurement $h(i)$ is discrete). Limited by the sensor, the measurement is inaccurate; thus, a weighted average method is adopted. Specifically, the closer to the measurement of time i , the more in line with the true height of time i , i.e., the weight given to the measurement $s(i) = w_i h(i) + w_{i-1} h(i-1) + w_{i-2} h(i-2) \dots$ satisfies

$w_i > w_{i-1} > w_{i-2} \dots$. This is a one-dimensional convolution of the discrete form. Because the “future measurement” in this example cannot be measured, only half of the one-dimensional discrete convolution is included. The complete formula for one-dimensional discrete convolution can be expressed as follows:

$$s(i) = (h * w)(i) = \sum_{j=-\infty}^{\infty} h(j)w(i-j),$$

where i denotes the state calculated (e.g., time and position), j denotes the distance to state i (e.g., time difference and spatial distance), and h and w denote two real-valued functions. In the application of CNNs, the first function h is called the input, the second function w is called the kernel function, and the output s is called the feature map. Notably, in practice, j (i.e., the interval we consider) is generally not within the range of negative infinity to positive infinity but in a small one. In deep learning applications, the input is usually a high-dimensional array (such as an image), and the kernel function is a high-dimensional parameter array generated by an algorithm (such as stochastic gradient descent). If a two-dimensional image I is inputted, then it needs to use a two-dimensional kernel K . Thus, two-dimensional convolution can be written as follows:

$$S(m, n) = (I * K)(m, n) = \sum_i \sum_j I(i, j)K(m - i, n - j),$$

where (m, n) is the calculated pixel position and (i, j) is the considered range. The two-dimensional convolution operation can be expressed in a more intuitive form, as shown in Fig. 6.15.

After defining convolution, we explain why the linear operation of convolution is adopted. First, the CNN refers to a neural network that uses at least one layer of convolution operation in the network to replace the general matrix multiplication operation. In the previous feedforward neural network example, the fully connected layer is used as the structure of the network. The input of the fully connected layer is the accumulation of the upper layer input multiplied by the weight, which is essentially a matrix multiplication. The convolutional layer uses the convolution operation to replace the matrix multiplication in the fully connected layer. The purpose of convolution is to improve the machine learning system through the following techniques:

- Sparse interactions,
- Parameter sharing,
- Equivariant representations.

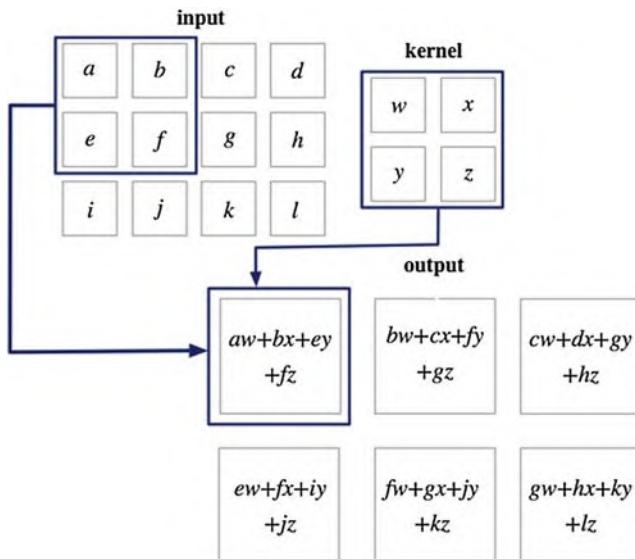


Figure 6.15 Two-dimensional convolution operation.

6.4.2 Sparse interactions

For an ordinary fully connected neural network, the nodes between layers are fully connected, as shown in Fig. 6.16.

By contrast, for a CNN, the nodes of the next layer are only related to the nodes that the convolution kernel acts on, as shown in Fig. 6.17.

An intuitive benefit of using sparse connections is that the network has fewer parameters. We take a grayscale image with a width and height of 200 pixels as an example, and its input in the fully connected neural network is shown in Fig. 6.18.

Assuming that the first hidden layer of this network has 40,000 neurons, the network has two billion parameters close to this layer. For the case where the input sample is 40,000 dimensions, 40,000 hidden layer nodes

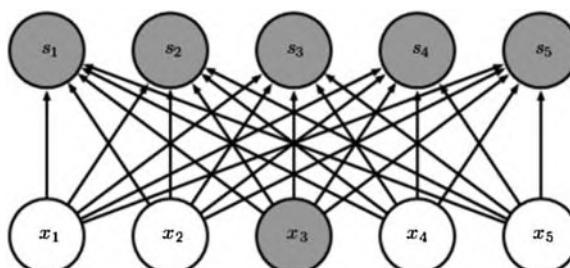


Figure 6.16 Fully connected layer.

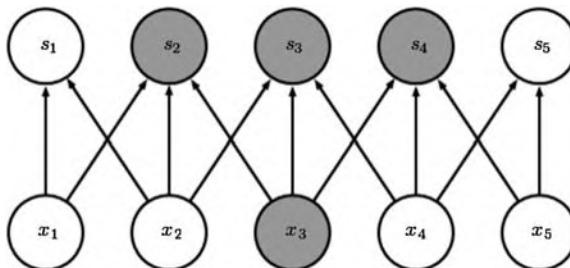


Figure 6.17 Convolutional layer.

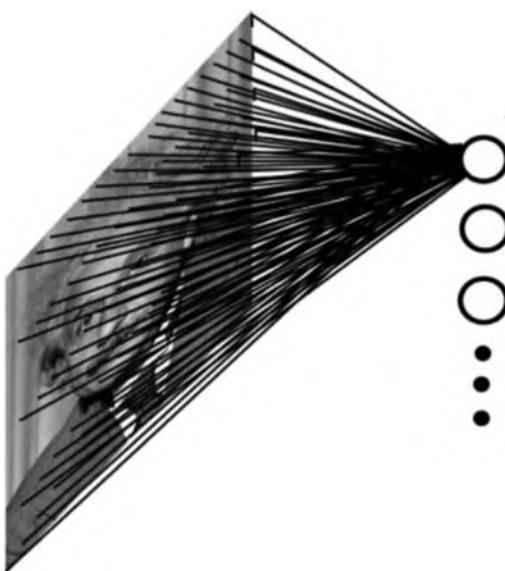


Figure 6.18 Input of a single neuron in the fully connected layer.

are appropriate. The computational load of training such a model is large and requires a large storage space. For CNN, the situation is different as shown in Fig. 6.19.

Here, we still use 40,000 hidden layer neurons. The size of our convolution kernel (also known as a filter) is 10×10 . The number of parameters for such a layer of convolution is only four million approximately, which is less than that of the fully connected neural network.

We are uncertain whether the output of a convolution is only related to a part of the input. If a certain character is not based on local features but is related to the entire input, then is the representation established by convolution incomplete?

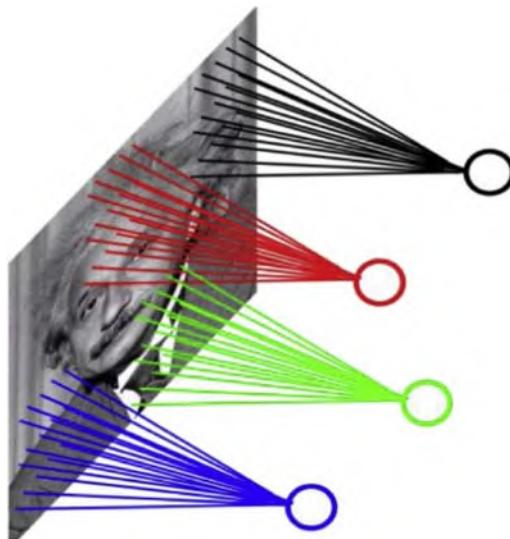


Figure 6.19 Input of the convolutional layer neuron.

This is not true. Modern CNNs often need to overlay multiple convolutional layers. Although CNNs are sparse in direct connection, units in deeper layers can be indirectly connected to all or most of the input images, as shown in Fig. 6.20.

Notably, in the related literature on CNNs, the terms “neuron,” “kernel,” and “filter” refer to the same thing, that is, the kernel function. In this book, we collectively call it the convolution kernel.

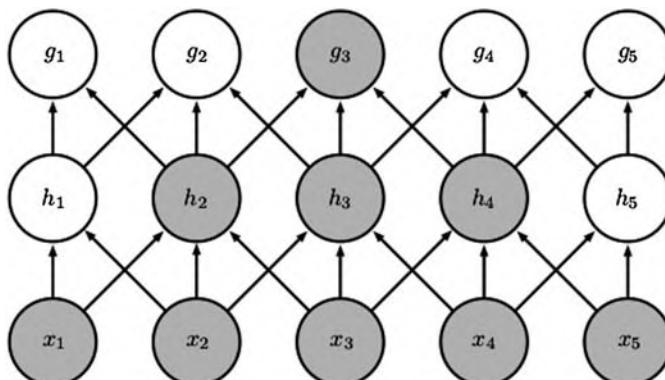


Figure 6.20 Although the convolutional neural network is sparsely connected, it can establish a global connection by increasing the number of layers.

6.4.3 Parameter sharing

The convolution kernel is the main parameter of the CNN. The convolution kernel slides the window on the input image, which means that the pixels of the input image share this set of parameters, as shown in Fig. 6.21.

Parameter sharing in the CNN enables the algorithm to learn only one parameter set instead of learning a separate parameter set for each pixel, which considerably reduces the storage space required by the model.

6.4.4 Equivariant representations

Because the entire input picture shares a parameter set, the model has equivariance to the translation of some features in the image. This property is useful in detecting certain shared structures in the input (such as edges), particularly in the first few layers of CNNs (i.e., layers close to the input).

6.4.5 Convolutional neural network

Through the previously discussed techniques, the convolution operation improves the pure matrix operation in the fully connected neural network. Similar to the fully connected neural network layer, the convolutional layer usually includes activation functions and develops a new transfer structure called pooling. Fig. 6.22 shows a typical convolutional layer. The traditional convolutional layer contains the following structures:

- Convolution operation,
- Activation function (nonlinear transformation),
- Pooling.

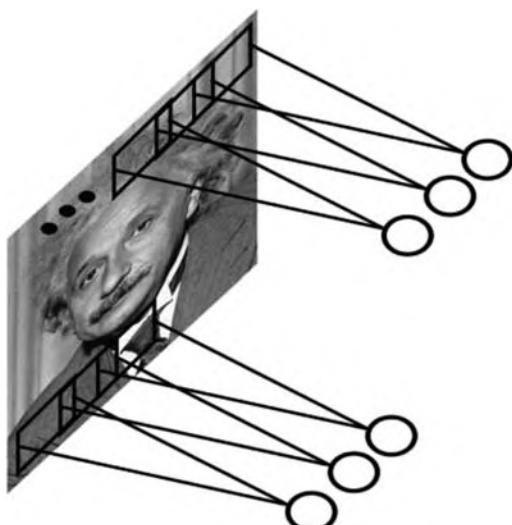


Figure 6.21 Shared parameters.

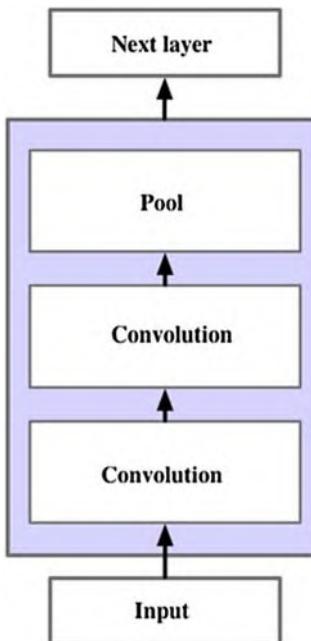


Figure 6.22 General form of the convolutional layer.

The convolution operation has already been discussed in the previous section. A two-dimensional convolution operation is adopted to process the image. The activation function in the convolutional layer plays the role of a nonlinear network like a fully connected neural network. The rectified linear unit is the most commonly used activation function. Let us discuss pooling in detail.

Pooling, also called the pooling function, is a method that uses the overall statistical characteristics of adjacent positions to replace the value of that position. The idea of pooling is a bit similar to sliding window averaging in time series problems. Fig. 6.23 shows a pooling method, that is, maximum pooling.⁵

Fig. 6.23 shows a 2×2 maximum pooling with a stride of 2. This pooling method involves sliding a 2×2 window on the input image with a step of two and calculating and outputting the maximum value of the input elements within the window. Notably, the pooling layer does not contain new parameters (weights). Thus, the pooling layer is only a transformation layer and does not participate in learning. After such a pooling function, the

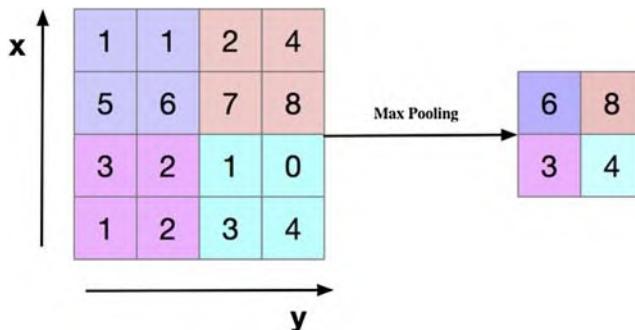


Figure 6.23 Maximum pooling.

input size is “compressed,” which means fewer parameters in the subsequent convolutional layer are needed. Therefore, after adopting pooling, the number of parameters of the entire neural network can be further reduced. Then, we derive the calculation formula for the size of the pooled input and output.

Assuming that the input size is $w \times h \times d$, the stride is s , and the window size is $f \times f$, the output width, height, and depth can be derived as follows:

$$w_{\text{out}} = \frac{(w - f)}{s} + 1,$$

$$h_{\text{out}} = \frac{(h - f)}{s} + 1,$$

$$d_{\text{out}} = d.$$

The commonly used pooling functions mainly include maximum pooling and average pooling that output the maximum and average values of adjacent matrix regions, respectively. Pooling, regardless of its kind, is invariant to a small amount of translation of the object in the input image because the network can further learn which transformations should be invariant.

6.4.6 Some details of convolution

In CNNs, convolution calculations still need to consider some details, including:

- Padding,
- Input and output sizes,
- Depth of the convolution kernel.

We need to analyze the convolution size conversion from input to output. For boundary problems (e.g., whether the convolution kernel crosses the boundary when sliding and how to deal with the boundary), the processing method called padding is executed in the CNN. To prevent the convolution kernel from sliding across the image boundary, a method called valid padding can be used. If p is the value of the pixel to be padded, then p equals 0 when an effective padding method is used to process the boundary. However, in the first few layers of the CNN, as much original input information as possible needs to be retained so that the low-level features can be extracted. The same convolutional layer is applied to ensure that the output has the same width and height as the input. To achieve the target that the output and input of convolution have the same width and height, a certain number of zero padding around the boundary with the same padding function is adopted. Suppose the size of the input is $w \times h \times d$, the stride of convolution is s , and the size of the convolution kernel is $f \times f$, the output width, height, and depth can be calculated as follows:

$$\begin{aligned} w_{\text{out}} &= \frac{(w - f + 2p)}{s} + 1, \\ h_{\text{out}} &= \frac{(h - f + 2p)}{s} + 1, \\ d_{\text{out}} &= k, \end{aligned}$$

where k denotes the depth of the convolution kernel.

We also need to consider the depth of the convolution kernel. Generally, a convolutional layer contains multiple convolution kernels, as shown in Fig. 6.24.

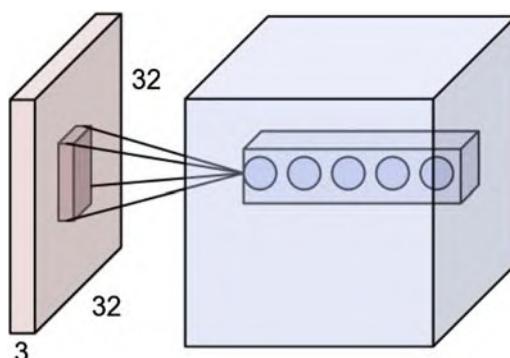


Figure 6.24 Using multiple convolution kernels in a convolutional layer.

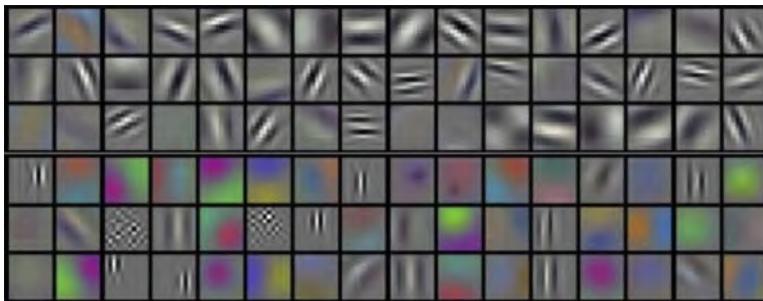


Figure 6.25 Visualization effect of each convolution kernel in the same convolutional layer.

Different convolution kernels will learn different features, that is, some will learn the color features, and some will learn the edge and shape features. Fig. 6.25 shows the visualization effect of the convolution kernel of the trained CNN in the same layer.⁶

The number of convolution kernels is called the depth of the convolution kernel. Fig. 6.25 shows that some convolution kernels learn shape (edge) features during the training process, whereas some convolution kernels learn color features. These features of different characters belong to the representation of low-level features learned by the neural network.

6.5 Vehicle detection based on YOLO2

You Only Look Once (YOLO)⁷ is an object detection model. Before the advent of deep learning, the main steps of traditional object detection methods were as follows:

- Extract the features of the object (i.e., HIST, HOG, and SIFT);
- Train the corresponding classifier (i.e., train a classifier that can judge whether an image is the object; as it is a two-class classification task, the SVM method is usually adopted);
- Sliding window search;
- Duplicate and false positive filtering.

Traditional object detection has two main defects. On the one hand, the sliding window selection strategy has no pertinence, high time complexity, and window redundancy. On the other hand, the hand-designed features are not robust, and the classifier is unreliable.

Since the advent of deep learning, object detection has made significant breakthroughs in the challenges of ImageNet.⁸ Two significant methods

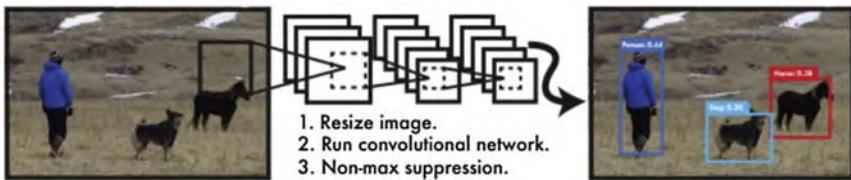


Figure 6.26 Workflow of YOLO.

are used for object detection. The first is a deep learning object detection algorithm based on region proposal represented by RCNN (i.e., RCNN, SPP-NET, Fast-RCNN, and Faster-RCNN). The second is a deep learning object detection algorithm based on regression methods represented by YOLO (i.e., YOLO and SSD). In this section, a deep learning object detection method based on regression methods represented by YOLO will be illustrated. Then, a simple YOLO model will be used for real-time vehicle and pedestrian detection.

YOLO regards object detection as a regression problem, the workflow of which is simple, as shown in Fig. 6.26.

As shown in Fig. 6.26, as an end-to-end method, the input is the original image and the output is the location of the object, its label, and the corresponding confidence probability. Different from the traditional sliding window detection algorithm, YOLO receives the entire picture as input in the training and application phases. The algorithm of YOLO can be divided into the following steps:

- Resize the image to 448×448 as the input of the neural network;
- Run the neural network to determine the bounding box coordinates, the confidence probabilities of the objects contained in the bounding box, and the class probabilities;
- Perform nonmaximum suppression and filter the bounding boxes.

The YOLO network architecture is modified from GoogleNet. The network has a total of 24 layers, of which the first 22 layers are used to extract features, and the remaining two fully connected layers are used to predict the probabilities and coordinates. The structure of the network is shown in Fig. 6.27.

The feature extraction and training processes of YOLO are discussed in detail in the following subsections.

6.5.1 Pretrained classification network

As shown in Fig. 6.27, the first step is to leverage the first 20 convolutional layers, an average pooling layer, and a fully connected layer to train a

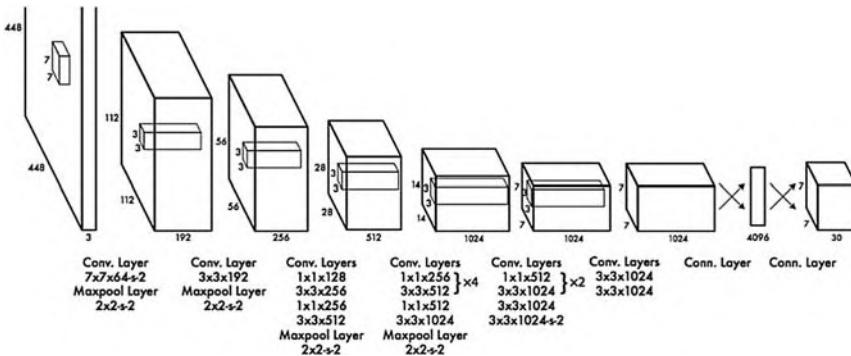


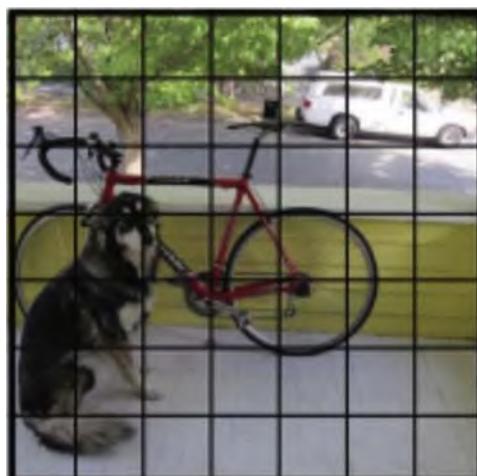
Figure 6.27 YOLO network structure.

classification network on the ImageNet2012 dataset. The classification network achieves an accuracy of 88%, that is, at the Top five on the ImageNet2012 dataset.

6.5.2 Train the detection network

The next step is to detect using the trained classification network and add four convolutional layers and two fully connected layers, that is, the last four convolutional layers and the last two fully connected layers in the structure diagram, after the 20 pretrained convolutional layers. By doing this, the input of the network becomes 448×448 , and the output is a tensor of $7 \times 7 \times 32$.

The image inputted into the detection network will first be resized into 448×448 and then be divided into 7×7 grids as shown in Fig. 6.28.

Figure 6.28 Divide the input image into a 7×7 grid.

The output of the network is $7 \times 7 \times 30$, which is responsible for the regression prediction of this 7×7 grid. Let us consider the 30 output components of each grid. Each grid has to predict two bounding boxes, which are the rectangular boxes used to enclose the object (i.e., the rectangular area where the object is located). Each bounding box contains the following information:

- Center coordinates (x, y) , that is, the coordinates of the center of the rectangular area where the object is predicted to be located;
- Width and height of the bounding box (w, h) ;
- Confidence probability of the predicted object and the accuracy of the box.

Because the anchor size in YOLO is five and each grid has to predict two bounding boxes, it has 10 outputs. Moreover, because YOLO detects a total of 20 types of objects during training, there are 20 categories of outputs. If the output of each grid is marked as C , then the value of C is 30.

6.5.3 Loss function of YOLO

To accurately predict the 30 output components of each grid, the design of the loss function must achieve a balance between bounding box coordinates, width, and height of the bounding box, confidence probability, and categories. YOLO uses the following function as the loss function of the detection network:

$$\begin{aligned}
& \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{\text{obj}} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \\
& + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{\text{obj}} (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\
& + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 \\
& + \sum_{i=0}^{S^2} 1_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

6.5.4 Test

In the test phase, the class-specific confidence score is obtained by multiplying the predicted category information and confidence probability of the bounding box. If this operation is performed on each grid of the entire image, then $7 \times 7 \times 2 = 98$ bounding boxes can be obtained. These bounding boxes contain not only information such as bounding box coordinates but also category information.

After obtaining the class-specific confidence score of each box, we set the threshold, filter out the boxes with low scores, and perform NMS processing on the reserved boxes to obtain the final detection result.

NMS is known as nonmaximum suppression. First, a detection bounding box is generated based on the object detection score. Then, the detection bounding box M with the highest score is selected, and other detection bounding boxes that have a significant overlap with the detection bounding box M are ignored. In this example, the YOLO network is used to predict a series of preselected boxes with scores. When the detection bounding box M with the highest score is selected, it is removed from Set B and placed in the final detection result Set D. At the same time, the detection bounding boxes in Set B will also be removed when their overlap with the detection bounding box M is greater than the overlap threshold N_t .

6.5.5 Vehicle and pedestrian detection based on YOLO

Because of the high real-time requirements for vehicle and pedestrian detection, a simplified version of YOLO, that is, Fast YOLO, is implemented. This model leverages a simple nine-layer convolution instead of the original 24-layer convolution, which sacrifices a certain degree of accuracy and has a faster processing speed to meet the needs of real-time object detection.

We use Keras to implement the Fast YOLO network structure.

Code list 6.9 Use Keras to implement fast YOLO

```
model = Sequential()

model.add(Convolution2D(16, 3,
3,input_shape=(3,448,448),border_mode='same',subsample=(1,1)))

model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(32,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))

model.add(Convolution2D(64,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))

model.add(Convolution2D(128,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))

model.add(Convolution2D(256,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))

model.add(Convolution2D(512,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))

model.add(Convolution2D(1024,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(Convolution2D(1024,3,3 ,border_mode='same'))

model.add(LeakyReLU(alpha=0.1))

model.add(Flatten())

model.add(Dense(256))

model.add(Dense(4096))

model.add(LeakyReLU(alpha=0.1))

model.add(Dense(1470))
```

Because the YOLO network needs a long training process, the trained model can be directly adopted to load the model parameters into the Keras model. The parameter download address is <https://pan.baidu.com/s/1o9twnPo>.

We load the parameter file into our network.

Code list 6.10 Load pretrained weights

```
def load_weights(model, yolo_weight_file):

    tiny_data = np.fromfile(yolo_weight_file, np.float32)[4:]

    index = 0

    for layer in model.layers:

        weights = layer.get_weights()

        if len(weights) > 0:

            filter_shape, bias_shape = [w.shape for w in weights]

            if len(filter_shape) > 2: # For convolutional layers

                filter_shape_i = filter_shape[::-1]

                bias_weight = tiny_data[index:index +
                    np.prod(bias_shape)].reshape(bias_shape)

                index += np.prod(bias_shape)

                filter_weight = tiny_data[index:index +
                    np.prod(filter_shape_i)].reshape(filter_shape_i)

                filter_weight = np.transpose(filter_weight, (2, 3, 1, 0))

                index += np.prod(filter_shape)

                layer.set_weights([filter_weight, bias_weight])

            else: # For regular hidden layers

                bias_weight = tiny_data[index:index +
                    np.prod(bias_shape)].reshape(bias_shape)

                index += np.prod(bias_shape)

                filter_weight = tiny_data[index:index +
                    np.prod(filter_shape)].reshape(filter_shape)

                index += np.prod(filter_shape)

                layer.set_weights([filter_weight, bias_weight])
```

Extract the detection result of the vehicle from the output of the YOLO network.

```
def yolo_net_out_to_car_boxes(net_out, threshold=0.2, sqrt=1.8, C=20,
B=2, S=7):

    class_num = 6

    boxes = []

    SS = S * S # number of grid cells

    prob_size = SS * C # class probabilities

    conf_size = SS * B # confidences for each grid cell

    probs = net_out[0: prob_size]

    confs = net_out[prob_size: (prob_size + conf_size)]

    cords = net_out[(prob_size + conf_size):]

    probs = probs.reshape([SS, C])

    confs = confs.reshape([SS, B])

    cords = cords.reshape([SS, B, 4])

    for grid in range(SS):

        for b in range(B):

            bx = Box()

            bx.c = confs[grid, b]

            bx.x = (cords[grid, b, 0] + grid % S) / S

            bx.y = (cords[grid, b, 1] + grid // S) / S

            bx.w = cords[grid, b, 2] ** sqrt

            bx.h = cords[grid, b, 3] ** sqrt

            p = probs[grid, :] * bx.c
```

```
if p[class_num] >= threshold:

    bx.prob = p[class_num]

    boxes.append(bx)

# combine boxes that are overlap

boxes.sort(key=lambda b: b.prob, reverse=True)

for i in range(len(boxes)):

    boxi = boxes[i]

    if boxi.prob == 0: continue

    for j in range(i + 1, len(boxes)):

        boxj = boxes[j]

        if box_iou(boxi, boxj) >= .4:

            boxes[j].prob = 0.

boxes = [b for b in boxes if b.prob > 0.]

return boxes
```

Then, we extract the detection result of the vehicle from the output of the YOLO network.

The results of the test images are shown in Fig. 6.29.

The effect on the test video is shown in Fig. 6.30.

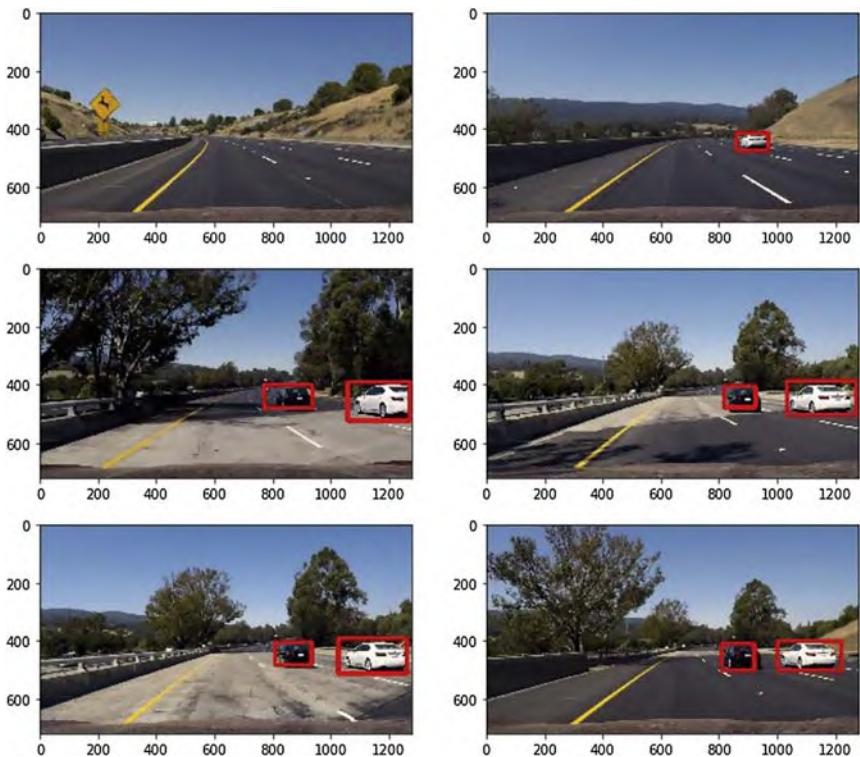


Figure 6.29 Detection effect of fast YOLO on the test picture.



Figure 6.30 Detection effect of fast YOLO on the test video.

References

1. Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE*. Nov 1998;86(11):2278–2324. <https://doi.org/10.1109/5.726791>.
2. Goodfellow I, Bengio Y, Courville A. *Deep Learning*. MIT Press; 2016.
3. Krogh A, Hertz JA. A simple weight decay can improve generalization. In: *International Conference on Neural Information Processing Systems*. Morgan Kaufmann Publishers Inc.; 1991:950–957.
4. Radu T, Mathias M, Benenson R, Van Gool L. Traffic Sign Recognitiondhow far are we from the solution? In: *International Joint Conference on Neural Networks*. Dallas, USA: IJCNN; August 2013.
5. Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition. <http://cs231n.github.io/convolutional-networks/>.
6. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Commun ACM*. May. 2017;60:84–90.
7. Redmon J, Divvala S, Girshick R, et al. *You Only Look once: Unified, Real-Time Object Detection*. 2015:779–788.
8. Russakovsky O, Deng J, Su H, et al. ImageNet large scale visual recognition challenge. *Int J Comput Vis*. 2015;115(3):211–252.

CHAPTER 7

Transfer learning and end-to-end self-driving

Zebang Shen¹, Yunfei Che² and Rui Zhao³

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²Huawei Technologies Co., Ltd., Xi'an, China; ³School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

7.1 Transfer learning	218
7.2 End-to-end selfdriving	220
7.3 End-to-end selfdriving simulation	221
7.3.1 Selection of the simulator	221
7.3.2 Data collection and processing	222
7.3.3 Construction of the deep neural network model	223
7.3.3.1 LeNet deep selfdriving model	224
7.3.3.2 NVIDIA deep selfdriving model	226
7.4 Summary of this chapter	228
References	229

In the first two chapters, we introduced the basic knowledge of neural networks and deep learning. In this chapter, we will introduce the concept of transfer learning and apply it to the end-to-end selfdriving model. Generally, the training of large-scale deep neural networks requires a large amount of computational resources. However, many ordinary developers only possess a crude deep learning computing environment, which has become one of the bottlenecks hindering the widespread application of deep learning. For example, to train a 50-layer residual neural network, it takes approximately 14 days to complete the model training using an NVIDIA M40 graphics processing unit (GPU). If we replace it with an ordinary personal computer (PC), then it may take decades to complete the model training. For such problems, transfer learning is a good strategy.

End-to-end selfdriving is a basic technology for selfdriving cars. In the implementation process, only the camera loaded on the driverless vehicle is used to obtain the image data of road conditions to train the deep neural network model. Then, the real-time image data collected by the camera are inputted into the trained deep neural network model, and the control

parameters are outputted to determine the driving strategy of the driverless vehicle. In essence, end-to-end selfdriving is a simplified autonomous vehicle model, and its actual road condition processing operations may be complex but can intuitively help us understand the application of deep learning to selfdriving cars.

7.1 Transfer learning

Deep learning is widely used in the industry. However, the process of building and training the deep learning model from the beginning is time-consuming and laborious. Engineers need to redesign the network architecture and conduct numerous training and testing experiments to obtain the appropriate model. A good strategy is to adopt and fine-tune an existing deep learning model based on the original model to adapt to the new application scenario, which is called transfer learning.¹

Therefore, training the model from the beginning when developing deep learning applications is unnecessary. At present, in many common application scenarios, relevant research teams have trained high-precision deep learning models. ImageNet² is a large-scale dataset with 16 million images, which has been marked by a large number of volunteers on the Internet. The ImageNet dataset covers many important image application scenarios; thus, it is a valuable public dataset. To solve the problems of recognition and classification on the ImageNet dataset, the most well-known work is the development of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Although the ILSVRC ended in 2017, many excellent recognition models for ImageNet, such as AlexNet, VGGNet, Google Inception Net, and ResNet, have emerged. Transfer learning can easily transplant these classical models to new application scenarios, which mainly depends on two factors, namely, the amount of data in the new application and the similarity between the new application and the original model. Generally, transfer learning is mainly applicable to four application scenarios, as shown in Table 7.1.

Table 7.1 Four application scenarios of transfer learning.

Dataset size	Application similarity	Model training method
More	Higher	Fine-tuning
More	Lower	Fine-tuning or retraining
Less	Higher	Modifying and training the fully connected classification layer
Less	Lower	Redesigning and training a new model

In the first scenario, if we want to develop a common object recognition application with a large amount of data, then we can directly learn from excellent mature models. In practice, for example, we only need to download the weights of one model, load them into the model, and input the new dataset into a network corresponding to the model for fine-tuning to obtain the ideal model.

In the second scenario, the dataset of a new application is relatively large, but no suitable pretrained model can be used. In this case, datasets are mostly collected in special application scenarios, such as medical image recognition. To deal with this situation, we first select a deep learning model and then fine-tune it with a new dataset. If the effect is not ideal, then we need to retrain the model with a new dataset.

In the third scenario, the dataset of a new application is relatively small, but similar recognition models are already existing. For example, to develop a face recognition application for dozens of employees in a department, we can only download the trained face recognition classification model based on a convolutional neural network (CNN) from the Internet and input the new data into the model for forward propagation. Afterward, the output is obtained before it reaches the fully connected layer, the forward propagation is stopped, and the output is regarded as the input of the fully connected layer; meanwhile, the original output is trained as the output of the fully connected layer. In other words, during the entire training process, we keep the parameters of the convolutional and pooling layers unchanged to maintain the feature extraction capability of the original network. The fully connected layer classifies the face features extracted from the convolutional and pooling layers. Therefore, the retraining of the fully connected layer ensures that the new model can adapt to the classification of new data. Deep CNN for image recognition may be complex and need a long training time. However, in this case, the actual calculation of the entire training process includes only one forward propagation process of sample data and the reverse propagation training of the fully connected layers (usually no more than two layers). Therefore, the training time of the model can be considerably reduced, and the development efficiency can be improved.

The worst case is that only a few new training datasets exist, and no similar models are available. In this case, a new network model needs to be redesigned and trained to obtain the best prediction effect. Because no successful model can be used as a reference, both network design and model training will take a long time but will ensure that the redesigned training model has high academic and engineering value in this field.

7.2 End-to-end selfdriving

Deep learning plays an important role in the field of unmanned driving, in which the end-to-end selfdriving technology based on deep neural networks is an important component. In practice, a simulator is usually used to imitate the effect of end-to-end selfdriving. The principles of end-to-end selfdriving are simple: First, the vehicle or simulator is manually operated to collect the control data. In this process, we need to collect the real-time road scene forward-looking images recorded by the cameras in the vehicle. In practice, to improve the generalization capability of the model, multiple cameras (e.g., three cameras) can be used to collect road images from different perspectives at the same time. Then, the control parameters generated by a human to drive the vehicle correctly, such as the steering wheel angle, brake, and throttle, need to be recorded. When training the deep neural network model, the collected road images are used as the input parameters of the model, whereas the vehicle control parameters are used as the output data of the model. After training the neural network model based on these road images and control parameters, it can predict the real-time road scene collected by the cameras and can output the control parameters of the vehicle. These parameters are inputted into the vehicle by a control unit (or as the input parameters of the simulator) to achieve the purpose of controlling the automatic driving of the vehicle.

As shown in Fig. 7.1, the core of end-to-end selfdriving is the deep learning model. During the driving process, the real-time image data

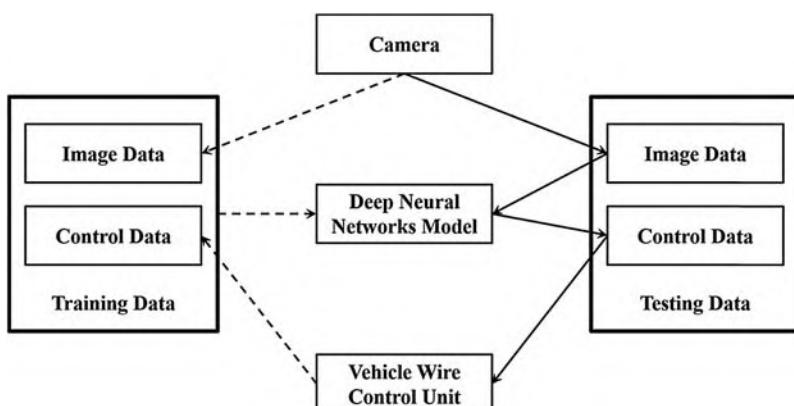


Figure 7.1 Schematic diagram of end-to-end selfdriving: the dotted line denotes the training process, and the solid line denotes the test process.

under different road conditions are collected, and the human control parameters of the car under different road conditions are recorded. These data are set as the training data of the deep learning model. When the deep learning model is used to control the selfdriving car, the real-time road images are collected by the camera and inputted into the deep learning model to obtain the vehicle wire control parameters of the autonomous driving control unit. Therefore, end-to-end selfdriving is a simple unmanned driving model, which not only uses the data collected by the camera and ignores other sensors but also makes operational decisions based only on the camera data. This scheme is certainly inappropriate for actual unmanned driving. However, for research purposes, we can test the capability of the deep learning model to control unmanned vehicles.

7.3 End-to-end selfdriving simulation

When simulating end-to-end selfdriving, several components, including simulators, image processing, and deep learning frameworks, are required.

7.3.1 Selection of the simulator

To simulate the control of unmanned vehicles by the deep learning model, an unmanned vehicle simulator installed on a PC can be used. Udacity³ is an online education provider that offers online courses; it also has an unmanned vehicle simulator. The details of the simulator can be found in GitHub.⁴ Many other open-source unmanned vehicle simulators can also be found in GitHub that can be used for the research. We can flexibly choose these simulators when conducting our experiments, but we need to use a simulator that can collect road condition image data and control data in real time. Before training the autonomous driving model, the training data need to be collected, that is, the image data are inputted into the model, and the control data are the output of the model. To simplify the problem, the operating speed of the simulator can be set to a fixed value. In this case, we only need the steering wheel to control the car simulator for unmanned driving. Overall, the most common method used to build this kind of deep learning model is the CNN model. The input is the image data, and the output is a single control parameter.

7.3.2 Data collection and processing

(1) For the data collection operations, the following basic principles need to be followed:

- We need to continuously control the car to drive at the center of the road. End-to-end selfdriving lets the model learn how to achieve this target. Therefore, when collecting data, we need to control the vehicle stably at the center of the road and try to keep it running smoothly, which is consistent with the actual control of vehicles by human beings.
- In addition to a stable environment, the car should be driven on curved roads as much as possible so that the model can make correct control operations for the road conditions in which the car needs to turn.
- We should simulate the driving of cars under different road conditions as much as possible. We need to collect more driving data to generalize the operational capability of the model so that the model can make correct control predictions for unknown conditions. We also need to collect the control data of the car under reversing conditions to ensure that the model will react correctly in extreme situations. In the case of using simulators, at least 10 laps of operation data under the correct driving conditions of unmanned vehicles need to be collected to train the end-to-end model to improve its generalization capability.
- Multiple cameras can be used to collect road condition information in different directions, such as collecting road image data from the left, center, and right of the car at the same time. On the one hand, this method increases the amount of data and improves the generalization capability of the model. On the other hand, the road conditions in different directions can help the model make more appropriate decisions.
- The frame rate of image acquisition should not be high. If the image acquisition frame rate is high, then repeated or similar road condition data will be collected. If some samples in the test set have already appeared in the training set, then the model's test accuracy will be falsely high. Moreover, computational resources will be wasted, and the trained model will frequently send similar control signals to the simulator. Under normal circumstances, the frame rate of road image acquisition should be controlled at approximately 10 frames/s.

(2) For the collected data, the following preprocessing operations should be performed:

- All images should be cropped to a suitable size. For example, the standard image required by the NVIDIA driverless model is 66 pixels high and 200 pixels wide. Some areas of the image that do not contribute to model decision-making, such as the car chassis at the bottom of the image and the sky and clouds at the top, should also be cropped. Some deep learning libraries (such as Keras) have built-in image cropping functions, and the GPU will also make these operations more efficient.
- The normalization operation should be applied to the image pixel value. The pixel value ranges from 0 to 255, which can be transformed into the range $[-1,1]$ according to the following formula. In this range, the activation function of the neural network can work better. The Python programming language can be used to define the operations employed to normalize the image pixel values.

7.3.3 Construction of the deep neural network model

When building an end-to-end neural network control model, a variety of neural network structures can be utilized. The most common neural network structure is a fully connected neural network. We can start by trying a four-layer fully connected neural network model (including two hidden layers). After training the model with the collected data, we can use the trained model to output the control parameters of the car simulator. Some open-source vehicle simulators have a driverless mode. In this case, we only need to turn on the driverless mode of the simulator and use the trained deep learning model to control the car to perform autonomous driving operations.

In actual operation, the performance of the four-layer fully connected neural network is poor. Thus, we need to design a suitable deep neural network model. According to the previously discussed transfer learning theory, we can transfer some classic image recognition models to this task, which has a large dataset but low application similarity. Therefore, we can learn from the structure of classic models and use the newly collected data to retrain or fine-tune these models.

In the previous chapters, we have introduced some content related to CNNs. CNNs are particularly suitable for handling two-dimensional image problems. The input of the end-to-end selfdriving model is road condition

images. Thus, we can try to use CNNs to build the model. We can learn from the design experience of many classic deep neural network models and migrate them to the construction of end-to-end driverless deep neural network models.

7.3.3.1 LeNet deep selfdriving model

Many classic visual neural network models are based on CNN models. LeNet⁵ is the earliest CNN model used for commercial handwritten digit recognition. Although it still retains the structure of two convolutional layers and two pooling layers, the reconstructed LeNet has some differences from the original structure. Notably, the current LeNet generally uses the rectified linear unit (ReLU) series of activation functions to replace the original sigmoid function. The ReLU activation function has a better performance on image feature extraction than the original sigmoid function. At the same time, the ReLU activation function can considerably reduce the amount of calculation for training and activating the network. Moreover, the input received by the original LeNet is a 32×32 grayscale (single channel) image, whereas that received by our model is a 66×200 (66 pixels in height and 200 pixels in width) color image (three channels). Therefore, the input of our model is different from that of the original LeNet. For convenience, when constructing the model, the convolution operation uses the same padding for the padding of the input feature map and the stride set to a value of 1. In this case, the size of the feature map obtained by convolution is the same as the size of the input feature map. The pooling layer uses a common pooling method with its parameters (i.e., width, height, and step size) assigned a value of 2. Thus, the pooling layer plays a significant role in dimensionality reduction (the output feature map is a quarter of the size of the input feature map). After two convolution layers and two pooling layers, the model connects two fully connected layers with 1024 nodes, followed by the output node of the model. The actual output of the model is no longer the original LeNet classification function but the control parameter. When simulating end-to-end unmanned driving, we only output a parameter that controls the car's steering (for convenience, we fix the speed of the car). Thus, the entire model has only one output node. The reconstructed LeNet selfdriving model is shown in Fig. 7.2.

Many deep learning frameworks, such as TensorFlow, TFLearn, Theano, Caffe, PyTorch, MXNet, and Keras, can be used to build deep learning models. Among them, Keras is a concise framework that uses TensorFlow or Theano as its backend. Currently, Keras has been officially supported by Google. This chapter uses the Keras library to build an end-to-end selfdriving model. By adopting the Keras deep learning framework,

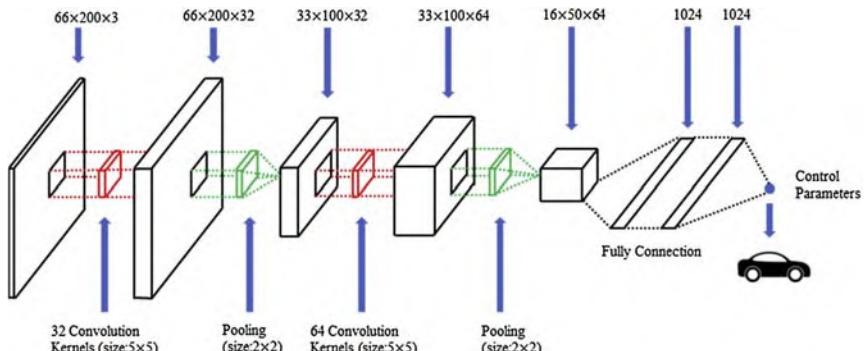


Figure 7.2 LeNet end-to-end self-driving model.

we can build and train the LeNet model with only a dozen lines of code. The code is as follows:

```
model = Sequential()

model.add(Conv2D(32, (5,5), padding='same', activation='relu',
input_shape=(32,32,3)))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (5,5), padding='same', activation='relu',
input_shape=input_shape))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(1024, activation='relu'))

model.add(Dense(1024, activation='relu'))

model.add(Dense(1))

model.compile(loss='mse', optimizer='adam')

model.fit(x_train, y_train, batch_size = 128, epochs = 10,
verbose=1, validation_data=(x_val, y_val))

model.save('selfdriver.h5')
```

The loss function of the model is set as the mean squared error (MSE), which means the difference between the control parameters given by the model to the currently inputted road condition image and the control

parameters when humans (correctly) operate the car. The optimization algorithm uses the adaptive moment estimation (Adam)⁶ optimization algorithm, which is the optimized version of stochastic gradient descent. Because the Adam optimization algorithm is also archived in the Keras and TensorFlow package, we do not need to illustrate the details here. Specifically, the model only needs to call the “fit()” function and pass in the training data. Then, the function will execute the Adam optimization algorithm to complete the optimization operations, such as automatic differentiation. After the model is trained, it is saved to a file named “selfdriver.h5,” which will be called when the simulator executes selfdriving.

When using the aforementioned trained LeNet model to perform selfdriving simulation, the end-to-end self-driving performance of the LeNet model is better than that of the four-layer fully connected neural network, and the entire driving process is stable. However, in the entire driving process, the collected training data show that the car is not always driving in the optimal position, that is, at the center of the road. Thus, the model needs to be improved.

7.3.3.2 NVIDIA deep selfdriving model

NVIDIA has released an end-to-end selfdriving model,⁷ as shown in Fig. 7.3.

The model still uses a 66×200 (66 pixels in height and 200 pixels in width) three-channel color picture as the input image. The difference between the NVIDIA and LeNet deep selfdriving models is that the pooling operation no longer appears in the entire model. First, the input image is subjected to three convolution operations, with the size of the convolution kernel being 55 and the number of convolution kernels being

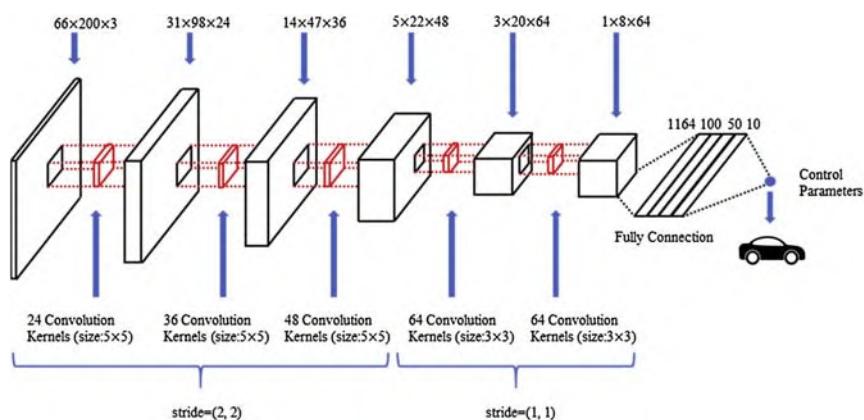


Figure 7.3 NVIDIA end-to-end self-driving model.

24, 36, and 48, which increase in sequence. Because the three convolution operations all use a step size of 22, which is equivalent to performing a pooling operation at the same time, the dimension of the feature map is reduced from 66×200 to 5×22 . Then, the output is subjected to two convolution operations, with the size of the convolution kernel reduced to 33 and the number of convolution kernels set to 64. Thus, the dimension of the feature map will be reduced to 18 until the feature extraction operation of the input image is completed. To output appropriate control parameters, the last 64 feature maps, with the size of 1×8 , are connected to four fully connected layers after they are flattened. The number of hidden layer nodes in each layer is 1,164, 100, 50, and 10. Then, the output control node is obtained. In practice, to improve the generalization capability of the model, we use the dropout technology in each fully connected layer, with the dropout rate set to 0.5. The optimization operation of the model still adopts the MSE and Adam optimization algorithm. Using the convolution application programming interface provided by Keras, we can still implement the model concisely. The code is as follows:

```
model = Sequential()

model.add(Conv2D(24, (5, 5), strides=(2, 2), activation='relu',
                input_shape=(66,200,3)))

model.add(Conv2D(36, (5, 5), strides=(2, 2),activation='relu'))

model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='relu'))

model.add(Conv2D(64, (3, 3), strides=(1, 1), activation='relu'))

model.add(Flatten())

model.add(Dense(1164, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(50, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(10, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(1))

model.compile(loss='mse', optimizer='adam')

model.fit(x_train, y_train, batch_size = 128, epochs = 10,
           verbose=1, validation_data=(x_val, y_val))

model.save('selfdriver.h5')
```

In practice, we determined that the selfdriving model provided by NVIDIA has good predictive performance: After sufficient iterative training, the car can drive relatively smoothly at the center of the road in the simulator. However, on some rugged winding mountain roads, long-term unmanned driving simulations may cause operational errors. At this time, on the one hand, we need to collect more selfdriving road scene data to increase the number of model training samples. On the other hand, we can control the driving speed to maintain stability. We can manually set the driving speed in the simulator to a low value or use the driving speed as a control parameter to learn the laws of the deep learning model. In this case, the output of the deep learning model has two nodes, namely, steering control and speed control.

We can also migrate many other classic deep learning models to the construction of end-to-end selfdriving models. Long short-term memory (LSTM)⁸ has been investigated by numerous scholars because it can memorize historical data. At present, applying LSTM to end-to-end unmanned driving is a well-known research method. A distinguishing feature of LSTM is making decisions on subsequent scenes based on previous scenes. In theory, LSTM is also a deep learning model that is suitable for end-to-end model construction. A good strategy is to combine the image feature extraction function of the CNN model and the time series memory function of the LSTM model to form a hybrid deep learning model that can be applied to end-to-end unmanned driving. Using Keras, a concise deep learning library (whose details will not be described in this chapter), readers can easily build their own end-to-end unmanned driving model.

7.4 Summary of this chapter

This chapter mainly introduces the concepts of transfer learning and end-to-end self-driving. To develop new deep learning applications efficiently, transfer learning strategies are often applied to engineering practices. Although many deep learning application scenarios have considerable differences, in most cases, we can just fine-tune or retrain the model to meet the application requirements of new scenarios. In the worst case, we can still refer to some classic deep learning models to design our deep learning application models.

End-to-end self-driving is an intuitive model of unmanned driving, which ignores the various sensor data of unmanned vehicles and simulates unmanned driving only through the road condition data collected by the

camera. End-to-end selfdriving also simplifies the entire unmanned driving process, which can be used as the basis for unmanned driving research. When discussing the end-to-end selfdriving model, we mainly considered three deep learning models, namely, shallow neural network, LeNet-like CNN, and NVIDIA end-to-end selfdriving models. However, in a real unmanned driving environment, relying solely on the road condition images collected by the camera to make driving decisions is unsafe. Therefore, we also need to comprehensively consider the data collected by multiple sensors to make joint decisions.

References

1. Pan SJ, Yang Q. A survey on transfer learning. *IEEE Trans Knowl Data Eng*. 2010;22(10):1345–1359.
2. Deng J, Dong W, Socher R, et al. ImageNet: a large-scale hierarchical image database. In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE; 2009:248–255.
3. <https://cn.udacity.com/>.
4. <https://github.com/udacity/self-driving-car-sim>.
5. Lecun Y, Bengio Y. Convolutional networks for images, speech, and time series. In: *The Handbook of Brain Theory and Neural Networks*. MIT Press; 1998.
6. Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.
7. Bojarski M, Del Testa D, Dworakowski D, et al. *End to End Learning for Self-driving Cars*. 2016.
8. Graves A. *Long Short-Term Memory. Supervised Sequence Labelling With Recurrent Neural Networks*. Springer Berlin Heidelberg; 2012:1735–1780.

This page intentionally left blank

CHAPTER 8

Getting started with self-driving planning

Zebang Shen¹, Wei Wang² and Rui Zhao²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

8.1 A* algorithm	232
8.1.1 Directed graph	232
8.1.2 Breadth-first search (BFS) algorithm	233
8.1.3 Data structure	235
8.1.4 How to generate a route	236
8.1.5 Directional search (heuristic)	238
8.1.6 Dijkstra algorithm	239
8.1.7 A* algorithm	240
8.2 Hierarchical finite state machine (HFSM) and autonomous vehicle behavior planning	241
8.2.1 Design criteria for decision-making plan system of autonomous vehicles	242
8.2.2 FSM	242
8.2.3 Hierarchical FSM	244
8.2.4 Use of state machines in behavior planning	245
8.3 Autonomous vehicle route generation based on free boundary cubic spline interpolation	247
8.3.1 Cubic spline interpolation	247
8.3.2 Cubic spline interpolation algorithm	251
8.3.3 Using Python to implement cubic spline interpolation for path generation	253
8.4 Motion planning method of the autonomous vehicle based on Frenet optimization trajectory	255
8.4.1 Why use the Frenet coordinate system	257
8.4.2 Jerk minimization and polynomial solution of fifth degree trajectory	259
8.4.3 Collision avoidance	264
8.4.4 Example of motion planning for autonomous vehicles based on Frenet optimization trajectory	265
References	272

This chapter focuses on the planning layer of autonomous vehicle systems. As described in [Chapter 1](#), the plan module consists of three layers: task planning, behavior planning, and action planning.

Before introducing the planning module, the A* algorithm is mainly introduced in this chapter. In task planning, the A* algorithm, as a discrete space search algorithm, is primarily used to solve the problem of optimal path search in discrete space.

Behavior planning is a core part of the decision-making process for autonomous vehicle systems. The finite state machine (FSM) is often used to design a behavioral decision model. This chapter also covers the hierarchical FSM (HFSM) that has been successfully applied to DARPA's autonomous vehicle challenge.

In addition introducing the motion planning method, this chapter also explains and uses spline interpolation to implement a simple path generation. On the basis of the results, Moritz Werling's trajectory optimization motion plan method based on the Frenet coordinate system is illustrated. A simple motion planner is also implemented using Python.

8.1 A* algorithm

The A* algorithm has been applied to many aspects of the field of autonomous driving algorithms. It is a classic and efficient discrete space path search algorithm. In task planning, the A* algorithm can be used to find the shortest route in an urban road network. In action planning, the custom A* can also be used to search for the local optimal path.

In the A* algorithm, the key is to find the shortest path while avoiding the obstacle from “start” to “end” on the basis of a graph. This section gradually reveals the core idea of the A* algorithm through deductive iteration. We need a certain data structure to represent the “graph.” In task planning for autonomous driving, a route network definition file is usually used for path planning, and the data of the road network is a directed graph.

8.1.1 Directed graph

In general, the input of the A* algorithm is a directed graph. A simple directed graph D consists of a vertex set $V(D)$ and an edge set $E(D)$, as shown in [Fig. 8.1](#), where the circles with numbers represent the vertices and the arrows between the circles represent edges attached by different weights, which illustrate the distance or length of the edges.

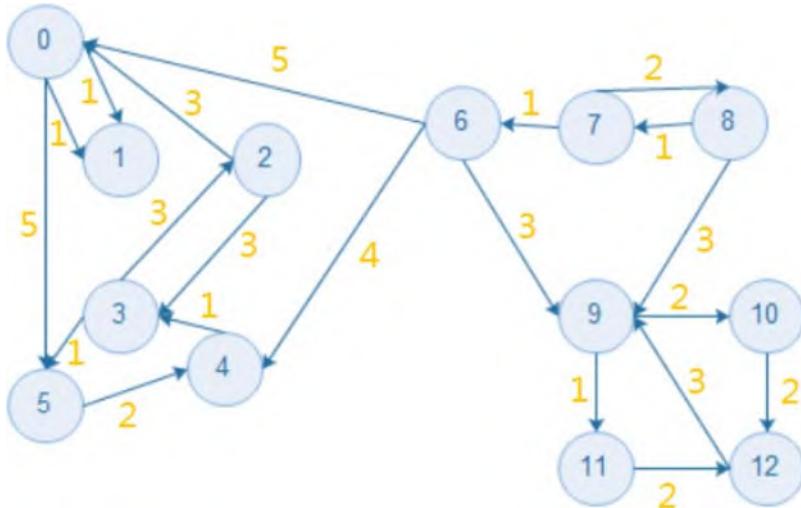


Figure 8.1 Representation of a directed graph.

In this directed graph, the direction of node 12 points to node 9, while node 9 cannot directly point to node 12. The important idea is to find a set named Neighbors in which one node can directly reach all nodes. For each node in the directed graph, a list of Neighbors can be found. For example:

$$\text{Neighbors}[3] = \{5,2\}$$

$$\text{Neighbors}[4] = \{3\}$$

$$\text{Neighbors}[6] = \{0,4,9\}$$

Start with Neighbors for a particular node saved in a list, and then find the other Neighbors for each node in the list on the basis of those Neighbors. The basic idea of path search is to iterate the process repeatedly and finally traverse the whole graph (as long as the graph is connected) to find all feasible paths.

To simplify the process of deriving the algorithm, the grid representing the search space with a default weight of one for all edges is adopted. The grid, as shown on the left of Fig. 8.2, can essentially be represented by a directed graph on the right.

8.1.2 Breadth-first search (BFS) algorithm

To understand the A* and hybrid A* algorithms, we introduce some background knowledge about search algorithms, which we then apply. Typically, search algorithms are mainly divided into depth-first search and BFS algorithms.

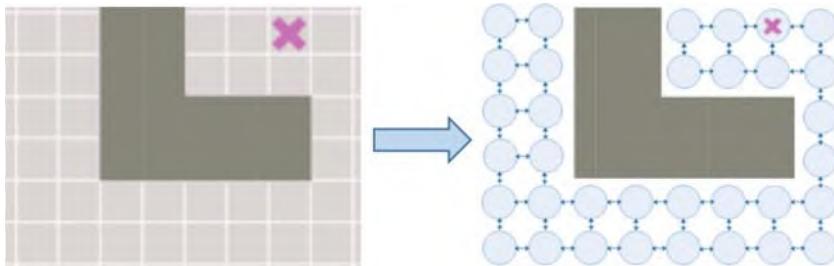


Figure 8.2 Grid as a directed graph.

The BFS algorithm starts with a start node and adds all of its neighbors to the frontier to be searched next. The frontier is represented by the FIFO (first in—first out) queue in this algorithm. The core idea of BFS algorithms is to have the start node traverse the graph and spread out to find the shortest path. As long a connection exists between the start node and the end node, there is at least one path between them. As the BFS algorithm diffuses radially from the start node to search, the path it finds is a local shortest path. The diffusion process of the BFS algorithm is illustrated in Fig. 8.3. The blue square represents the current diffusion boundary, which can be approximated (as the graph is discrete) as a circle around the start point.

However, because the BFS algorithm needs to traverse the entire discrete space when searching the shortest path, its search efficiency is relatively low and inefficient. Therefore, to improve the search efficiency of the BFS algorithm, the A* algorithm is proposed. The pseudocode of A* is given. Then, we discuss how to improve the BFS algorithm to the A* algorithm gradually.

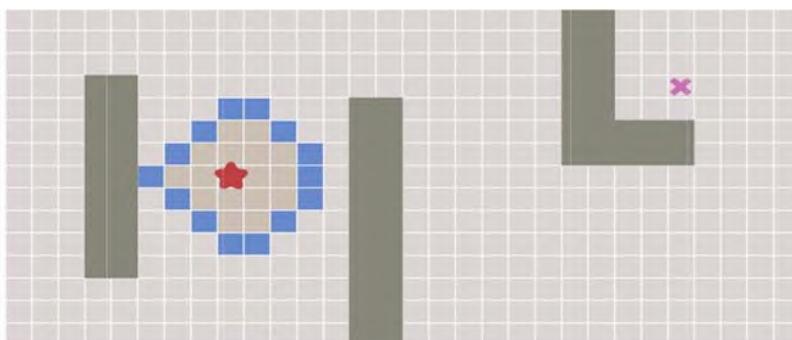


Figure 8.3 Diffusion process of BFS.

Code list 8.1.1 A* pseudocode

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0
while not frontier.empty():
    current = frontier.get()
    if current == goal:
        break
    for next in graph.neighbors(current):
        new_cost = cost_so_far[current] + 1
        if next not in cost_so_far or new_cost < cost_so_far[next]:
            cost_so_far[next] = new_cost
            priority = new_cost + heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

8.1.3 Data structure

Graph: As the directed graph, it is mainly aimed at finding the neighboring nodes of a certain node. In the code, the *Graph.Neighbors* function can be called to return a list of all neighboring nodes around the current node that will be traversed iteratively by the loop.

Queue: The adoption of a queue is based on a characteristic of the FIFO principle. As shown in Fig. 8.4, if the queue of the frontier (open list) is empty and the current node is A, then its Neighbors will return four nodes B, C, D, and E. After these four nodes are added to the frontier, in the next round of the while loop, *frontier.get()* returns node B (according to the FIFO principle, point B, as the first to join the queue, should be the first to leave the queue). The *Neighbors* function is called, and G, F, A, and H are returned. Expect for node A, the other three nodes are added to the next frontier (open list) to be traversed iteratively by the next round of the loop. At this moment, the frontier has C, D, F, G, and H. According to the FIFO principle of the queue, *frontier.get()* returns the C node. In this way, the whole process of diffusion becomes a phenomenon from near to far and

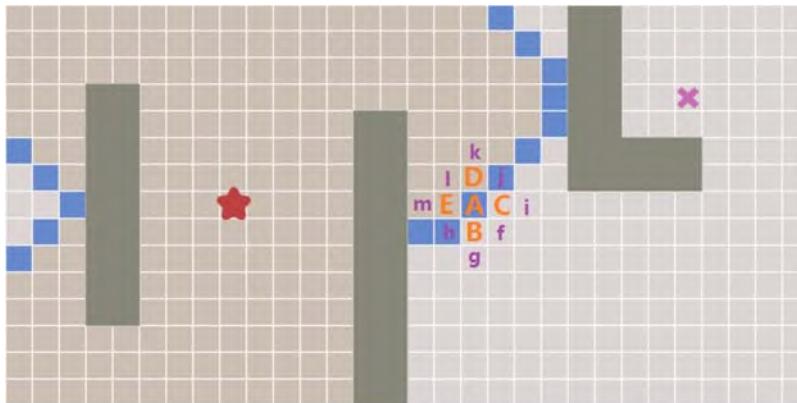


Figure 8.4 Neighbors at a certain point.

from inside to outside, which is also the principle of BFS. In the code, an element from the queue is pulled by `frontier.get()` (which is removed from the queue), and the current neighbors are added to it by `frontier.put()`. This process is repeated until all nodes in the graph have been traversed.

Visited list: Graph.Neighbors (A) returns four nodes B, C, D, and E. These four nodes are added to the frontier. The next round of Graph.-Neighbors (B) returns four nodes A, H, F, and G. If node A is added to the frontier list at this time, the iteration falls into an infinite loop. To avoid this situation, we need to add the traversed node to the visited list for backup. In this way, before putting the node into the frontier, we can first judge whether this node already exists in the visited list. The visited list is equivalent to the close list described in other materials. A node is added to the close list to indicate that the point has been traversed.

The steps of the BFS algorithm are illustrated in Fig. 8.5, in which the green box represents the neighbors of the current node returned by `Neighbors`, and the blue box represents the nodes in the current frontier queue. Given the FIFO principle of the queue, the node in front of the queue (the number of the block in Fig. 8.5) is traversed earlier by `current = Frontier.Get()` in the while loop. Dark brown squares represent nodes that have been added to the visited queue.

8.1.4 How to generate a route

As the above algorithm can traverse all nodes in the graph as long as the graph is connected, it must be able to find the target node. Therefore, the route from the start node to the end node must exist. To generate this

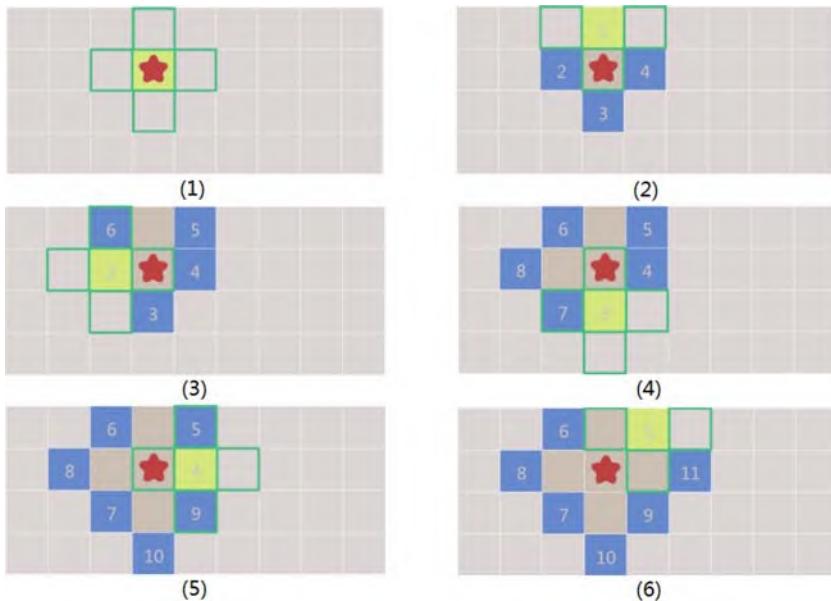


Figure 8.5 Running steps for BFS.

route, the process of diffusion needs to be recorded, the source of each node (which points to the current node) should be saved, and the full path is then obtained through these records. In Fig. 8.6, the arrows on each square point to the source node. Through the arrow, the start node of any square can be found, and then a complete path is formed.

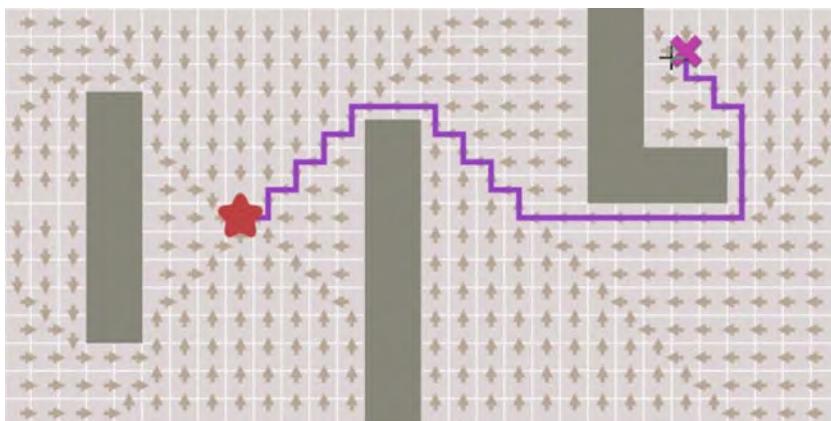


Figure 8.6 Use of arrows to construct the route.

8.1.5 Directional search (heuristic)

Thus far, we have presented the idea of the BFS algorithm for path search. Each iteration of the algorithm moves in all directions, diffuses outward from the start node in concentric circles, traverses every square around the start node, and forms the entire graph until the end node appears.

During the implementation of the algorithm above, several nodes are always kept in the frontier (*frontier.get()* function pulls a node while *frontier.Put()* pushes a node to the queue in each iteration). The *frontier.Get()* function returns the nodes that determine the diffusion direction of BFS in the next iteration. As the frontier is a FIFO queue, the earlier the nodes join the frontier, the sooner they are taken out. Thus, a radial expansion from the start node with concentric circles is performed.

How can the diffusion process be in a certain order? We should note that the coordinates of the start node and end node are always known but are not maximized. If the iteration can always be diffused toward the end node, can better results be achieved?

As *frontier.get()* returns one node from the frontier, the entire graph will be traversed by whichever node is pulled first from the frontier. To achieve heuristic expansion in the graph, the node closest to the end node should be returned by *Frontier.Get()*. As the grid graph is used, each node has (x, y) coordinates, and thus, the distance between two nodes can be calculated by (x, y) (owing to the existence of obstacles, the calculated distance may be different from the real value, but these two values are close so that the true distance can be estimated through this method).

Next, the FIFO mode of the original queue is changed. The priority is added to form a PriorityQueue. The smaller the second parameter of *frontier.put(next, priority)* is, the higher the priority of the current node will be. Obviously, the shorter the Manhattan distance from the end node is, the earlier this node will return from *frontier.get()*.

[Fig. 8.7](#) shows the results of the comparison of the numbers of nodes in the graph traversed by BFS and heuristic search algorithms to arrive at the end node. In total, the BFS algorithm traverses 222 nodes while the heuristic search traverses only 25 nodes to find the end node. The efficiency is increased by nearly 10 times.

However, the heuristic search algorithm also creates new problems, as shown in [Fig. 8.8](#).

In this case, although the heuristic search algorithm traverses fewer nodes than the BFS algorithm, it does not generate the globally shortest route. To solve this problem, the Dijkstra algorithm is introduced.

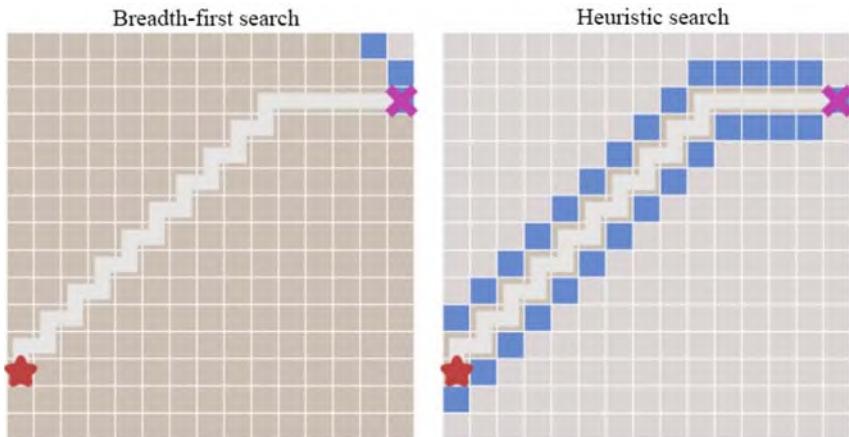


Figure 8.7 Heuristic search that converges quickly.

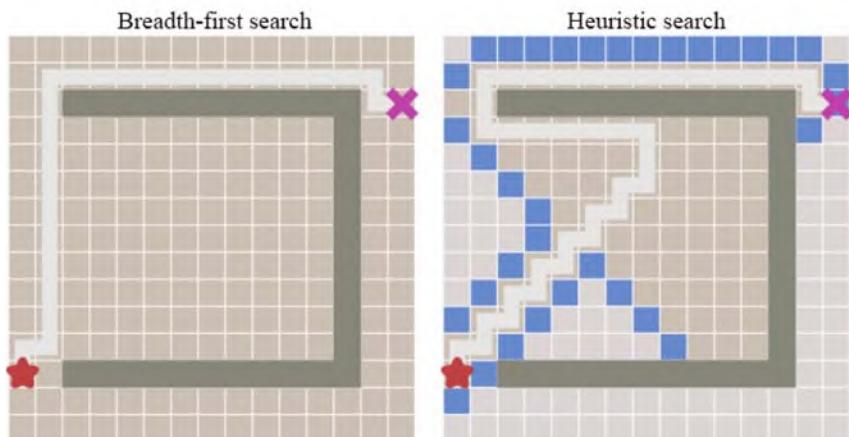


Figure 8.8 Problems with heuristic search.

8.1.6 Dijkstra algorithm

There are always multiple routes from the start node to the end node. The previous methods leverage the visited array to avoid traversing the same node repeatedly. However, this approach leads to a problem in which the earliest traversed path is considered to be the shortest path. To consider the requirements of the shortest route and the efficiency of the algorithm, we should introduce and refer to the idea of Dijkstra's algorithm.

The process of Dijkstra's algorithm is to choose the shortest path among multiple paths and record the lowest cost of every node through which the

route traverses from the start node to the end node. Once a traversed node is traversed again through another route, the algorithm determines whether the current route is less expensive than the route initially traversed by this node. If the cost is less, then the node is included in the new route. The current shortest path cost (length) is recorded from every node to the start node, and the cost is treated as the priority of the node in the PriorityQueue.

8.1.7 A* algorithm

The search algorithm is expected to be not only heuristic but also capable of obtaining the shortest path possible. Combining these two considerations results in the A* algorithm, which integrates the advantages of Dijkstra's algorithm and the heuristic search algorithm. The priority of the node in the PriorityQueue is defined as the sum of the distance and the estimated distance from the start node to this node. The effect of the A* algorithm is shown in Fig. 8.9. The A* algorithm successfully overcomes the problems encountered by heuristic search.

The constraints of the A* algorithm are expressed as

$$f(n) = g(n) + h(n).$$

Corresponding to the code,

$$\text{priority} = \text{new_cost} + \text{heuristic(goal, next)}.$$

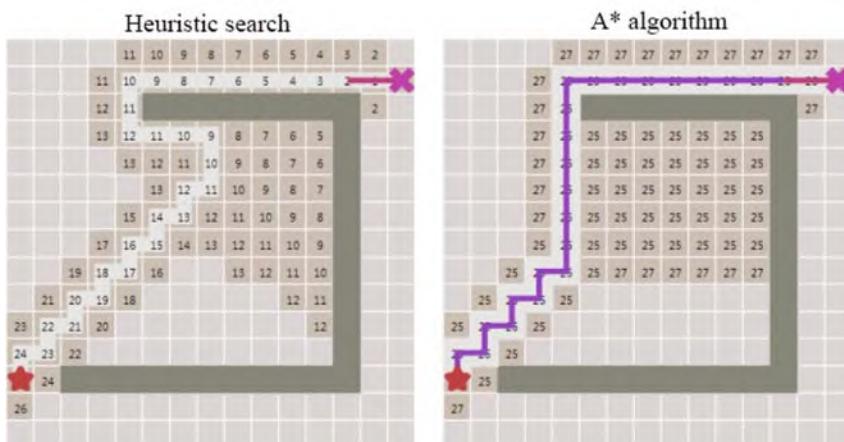


Figure 8.9 Heuristic search and A* algorithm.

In the formula, $f(n)$ is the cost value from the start node to the current node and is also called priority. The lower the total cost is, the smaller the priority will be. The higher the priority is, the earlier the node will be traversed by `frontier.get()`. $g(n)$ is the new_cost as the known cost value from the start node to the current node, and $h(n)$ is the final estimated cost from the current node to the end node.

The A* algorithm mainly maintains two lists: open list and close list. The open list refers to the frontier and indicates the series of nodes that may be traversed next. The close list (implied by visited, `cost_so_far` in the code) contains all the nodes that have been traversed.

8.2 Hierarchical finite state machine (HFSM) and autonomous vehicle behavior planning

Behavioral planning is also called behavioral decision-making. It is the middle layer of the three-layer system (tasks, behaviors, actions) of autonomous vehicle planning modules. This section mainly introduces the basic concepts of behavior planning. At the same time, the HFSM, which is the widely used method for the behavior planning of autonomous vehicles, is introduced.

Driving behavior planning is also called driving behavior decision making. The role of this layer is mainly to generate the driving state and behavior on the basis of the global optimal driving route trajectory from the upper layer (task planning layer), the current traffic and environmental perception information, the constraints of traffic rules, and the guidance of driving experience. The information flow of the decision-making layer of autonomous vehicles¹ is shown in Fig. 8.10.

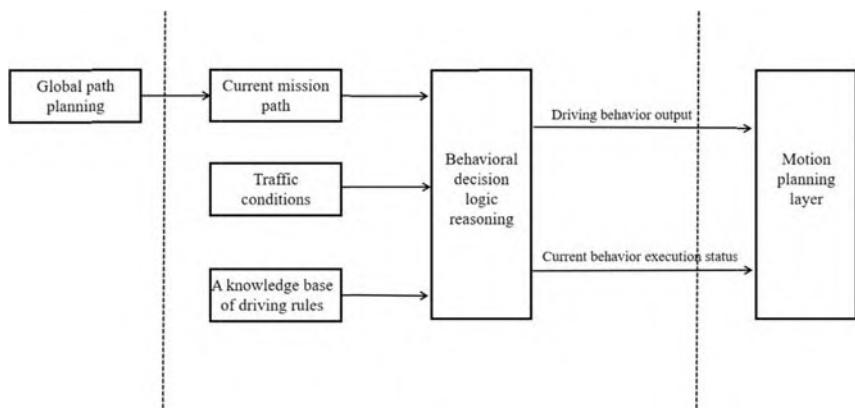


Figure 8.10 Flow of behavior planning in the autonomous vehicle plan system.

8.2.1 Design criteria for decision-making plan system of autonomous vehicles

The content of behavior planning is directly related to the reliability and safety of autonomous vehicles, hence designing a behavior plan system that fully conforms to driving experience and traffic rules remains challenging. However, there are many methods to achieve behavior planning, and the design concept can be summarized as two points:

- Reasonableness: The reasonableness of autonomous vehicles is based on two points: traffic laws and driving experience. The priority of traffic laws is higher than that of driving experiences. The traffic laws and regulations should be considered, and they include driving on the correct lane, not speeding, using the turn signal when changing lanes and overtaking, driving the vehicle in accordance with traffic instructions, braking immediately in any dangerous situation. The following contents of driving experience need to be considered: try to stay in the original lane, change lane at will, do not accelerate at will when driving in urban roads to ensure driving comfort, overtake decisively if the front vehicle is slow and conditions are permitted, and so on. Therefore, these two factors must be considered when designing the behavior planning system.
- Real time: The behavior planning in any autonomous vehicle system should be real time, which means that the behavior plan should be able to deal with complex dynamic traffic scenarios and be able to quickly adjust driving behavior to avoid danger according to changes in the environment.

8.2.2 FSM

At present, no “best solution” is able to achieve the behavior planning layer of autonomous vehicles. Currently, the HFSM is widely accepted and adopted. The HFSM is still adopted by many teams in the early DARPA challenge competition, and it is based on the FSM.

The FSM is a straightforward abstraction reaction system. It is simple because it produces limited responses through specific external inputs. In an FSM, a limited number of states can be constructed. External inputs can only make the state machine transition from one state to another in this limited set of states. A simple FSM² is shown in Fig. 8.11.

An FSM usually consists of the following parts:

- (a) Set of inputs: Usually, it is called a set of stimuli. It contains all the inputs that the state machine might receive. The symbol Σ is used to represent this set. As a simple example, given an autonomous vehicle with two buttons, namely, start and stop (a and b, respectively, which cannot be pressed at the same time), $\Sigma = \{a, b\}$ is used to represent the input set of the FSM with these two buttons.

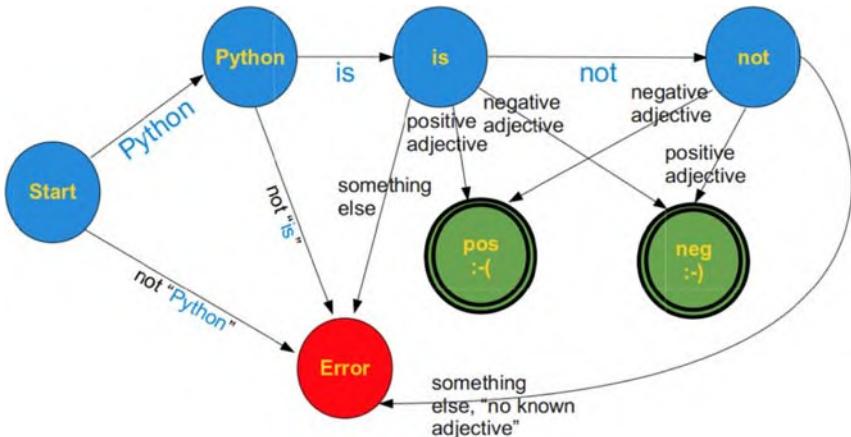


Figure 8.11 Simple finite state machine.

- (b) Output set: The set of responses that the FSM can make. This set is also finite, and the symbol Λ is used to represent this set. In some cases, the FSM output is empty.
- (c) Usually, a directed graph is used to describe the state and transition logic within the FSM. The symbol S is used to represent the set of states in the directed graph.
- (d) The FSM usually has a fixed initial state (the default state of the state machine when it does not have any inputs). The symbol s_0 is used to represent this state.
- (e) The set of end states, which is a subset of the state machine S , may be an empty set (the entire state machine has no end state) and is usually represented by the symbol F .
- (f) Transition logic: The conditions of a state machine transfer from one state to another (the current state and the input need to be combined). For example, in Fig. 8.11, the conditions from the Python state to the error state are as follows: (1) the state machine is in the Python state; (2) the input is not “is.” A state transition function is usually used to describe the transition logic: $\delta : S \times \Sigma \rightarrow S$.

Note: Acceptor and transducer are two categories according to whether they have an output in the state machines. The acceptor has no output but an end state, while the transducer has a set of outputs.

The FSM can be further divided into deterministic automata and nondeterministic automata. In the deterministic automata, each state has exactly one transition for each possible input. In the nondeterministic automata, a state may have no transition or more than one transition for a given possible input.

8.2.3 Hierarchical FSM

Given a large number of states, the FSM system is likely to become very large. If the FSM has N states, then there may be $N \times N$ state transitions. When the number of N is large, the structure of the state machine becomes more complex. In addition, FSMs have the following problems:

- Poor maintainability: When a state is added or removed, all associated states need to be changed. Thus, large changes in the state machine can cause problems.
- Poor scalability: When the FSM contains a large number of states, the readability and scalability of the directed graph are poor.
- Poor reusability: Using the same FSM in different projects is almost impossible.

Therefore, HFSMs need to be introduced. Take the state machines of the same type as one state machine and then create a larger state machine to maintain the substate machines ([Fig. 8.12](#)).

Unlike that in the FSM, a super state is added in the HFSM. Essentially, a set of states of the same type are combined into a set (the box in [Fig. 8.12](#)), and a transition logic exists between the super states. Hence, the HFSM

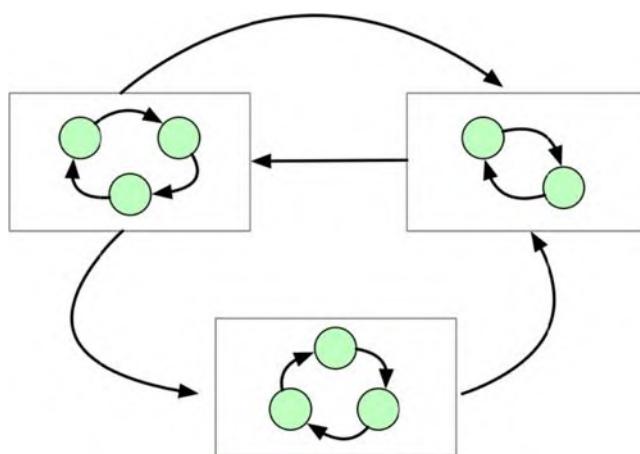


Figure 8.12 Layered finite state machine.

does not need to build a transition logic for each state and other states because the state is classified into the set. Moreover, transfer logic exists between sets, and the state transition between them can be achieved by inheriting the transition logic. The transition of inheritance in the HFSM is the same as object-oriented programming.

8.2.4 Use of state machines in behavior planning

Why should a state machine be adopted in the behavior planning layer of autonomous vehicles? As the decision of an autonomous vehicle is based on the current state of the autonomous vehicle and the input information from the perception module, the behavior planning module is also a reaction system.

We illustrate the HFSM with an example: “Junior”³ was the autonomous vehicle of Stanford University, and it participated in the DARPA Urban Challenge in 2007 and won second place in the autonomous vehicle competition. Junior’s behavior planning system is implemented through the HFSM. The top-level driving behavior is divided into 13 super states (the big box above) by Junior’s team. Each driving behavior super state corresponds to a set of subbehavior states to complete the behavior. The top-level behavior is managed by an FSM, as shown in Fig. 8.13.

- LOCATE_VEHICLE: This is Junior’s initial positioning state, which determines the position of the autonomous vehicle on the map before it leaves.

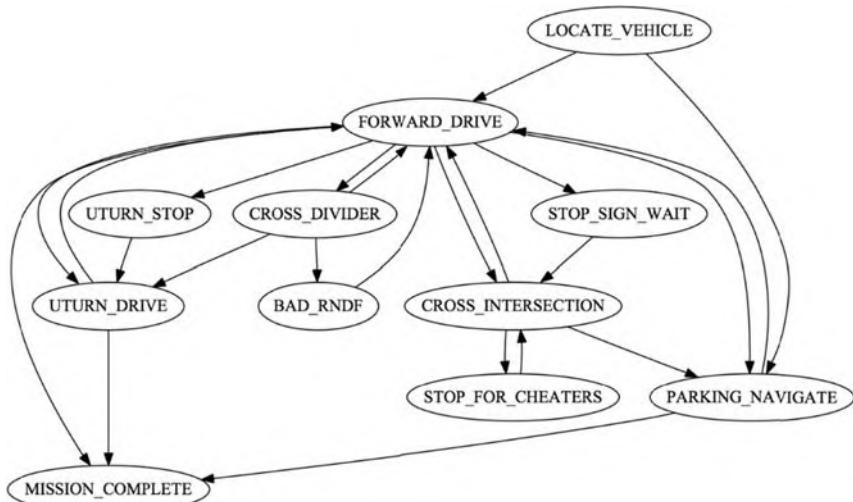


Figure 8.13 Stanford Junior using a top-level finite state machine.

- FORWARD_DRIVE: This super state consists of the following behaviors: going straight ahead, lane holding, and avoiding obstacles. It is the first state for the state machine when the vehicle is not parked.
- STOP_SIGN_WAIT: The vehicle enters this state when an autonomous vehicle waits at a stop sign (a stop sign is a common traffic signal at an intersection in the United States).
- CROSS_INTERSECTION: In this state, the autonomous vehicle processes the crossing scene. The autonomous vehicle waits until it confirms a safe environment.
- UTURN_DRIVE: The vehicle turns around in this state.
- UTURN_STOP: The vehicle performs a short stop before turning around.
- CROSS_DIVIDER: To avoid local congestion, after the opposite car passes, the autonomous vehicle crosses the yellow line and drives.
- PARKING_NAVIGATE: This behavior is the standard driving mode while parking.
- BAD_RNDF: If the current road is different from the network diagram of the system, the vehicle enters this state. In this state, the hybrid A* algorithm is adopted to complete the path plan of the vehicle.
- MISSION_COMPLETE: When the mission is complete, the autonomous vehicle enters this state, which is the end of the state machine.

The state machine is in its standard driving mode (FORWARD_DRIVE and PARKING_NAVIGATE) when the vehicle is normally driving. Through stuck detectors, the system judges whether the states change from standard driving states to other states. If the other states are completed, then the behavior planning module returns to the standard driving mode.

This design would allow autonomous vehicles to handle complex situations, such as the following:

- If the current lane is blocked, vehicles will consider driving into the opposite lane. If the opposite lane is also blocked, the vehicles enter the UTURN_STOP or UTURN_DRIVE state.
- For the traffic jam problem at the intersection, if the waiting time is over, the hybrid A* algorithm will be called to find the nearest exit to make the vehicle leave the blocked area.
- If the one-way road is blocked and the path planning fails, the hybrid A* algorithm is also called to plan the next waypoint.

- If some waypoints are not reachable after multiple loops, these waypoints will be skipped to avoid vehicles entering an infinite loop to reach an infeasible waypoint.
- If the autonomous vehicle does not make any progress for a long time, the vehicle will call hybrid A* to plan a route to reach the nearby GPS waypoint. This plan ignores traffic rules.

Although Junior's strategy was designed for the DRAPA Challenge with the aim of winning the competition as much as possible, its design concept still offers great reference value today. In the actual autonomous driving application of other scenarios, the implementation of the state machine will be more complicated. The HFSM is more modular than the basic FSM. However, it still has many disadvantages, such as limited reusability, large architecture, and behavioral decision-making problem in complex scenarios. These issues remain an important topic in current autonomous research.

8.3 Autonomous vehicle route generation based on free boundary cubic spline interpolation

The purpose of route generation is to generate a viable and efficient path from a start node to an end node. The algorithm based on spline interpolation is different from previous path search algorithms, such as A* and Hybrid A*. In the path generation problem, the start node and end node are known. A basic demand of path generation is that the path can be executed through the control of the vehicle. A simplified method based on the cubic spline function for generating a path that is continuous and smooth is illustrated in this section. A simple program is also implemented in Python.

8.3.1 Cubic spline interpolation

Before discussing the path generation based on spline interpolation, we need to understand what spline is. Spline interpolation was originally used for function fitting. What is function fitting? Some nodes are shown in Fig. 8.14.

What kind of function is necessary to fit these isolated nodes to form a connecting line segment? The easiest way is to connect discrete nodes through straight lines, as shown in Fig. 8.15.

Obviously, the fitting line segment is not “smooth” enough. To make the line smoother, a conic for connecting each node is recommended.

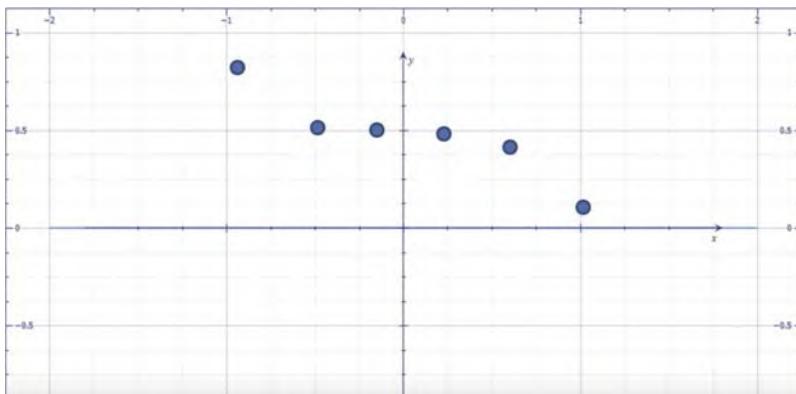


Figure 8.14 Points to be fitted.

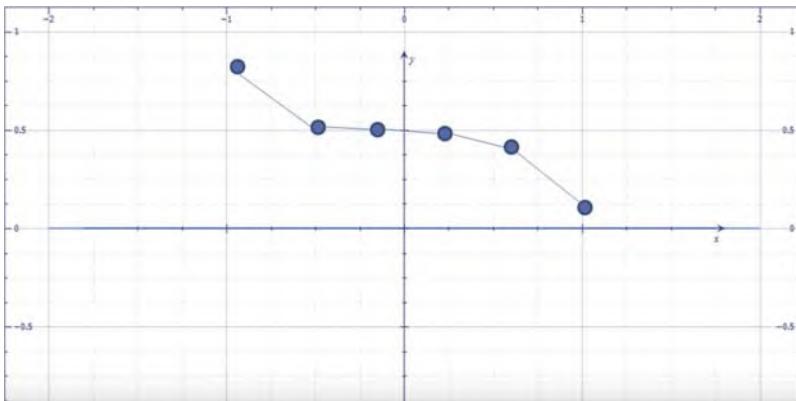


Figure 8.15 Using line to fit.

As shown in Fig. 8.16, the fitting effect is significantly improved relative to the first curve. As the quadratic curve is adopted, we advance further to leverage the cubic curve for fitting.

As shown in Fig. 8.17, the curve fitting effect of a cubic polynomial seems to be better, but all fitting methods are based on N-order lines to easily connect all points. Next, the cubic spline interpolation algorithm is adopted to fit these points, as shown in Fig. 8.18.

Compared with that by simple cubic polynomial connection methods, the curve fitted by cubic spline interpolation is more consistent with vehicle motion form and is smoother. As cubic polynomial connection methods produce a curve radian at some straight lines connecting line segments, the vehicle motion characteristics combined with the kinematics model of the

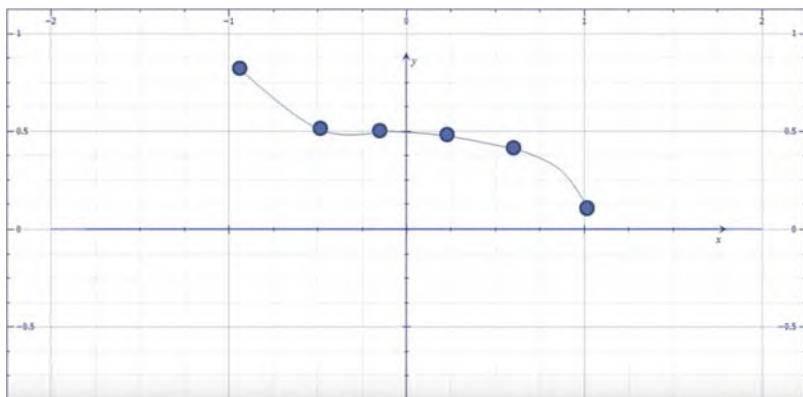


Figure 8.16 Connections using a conic.

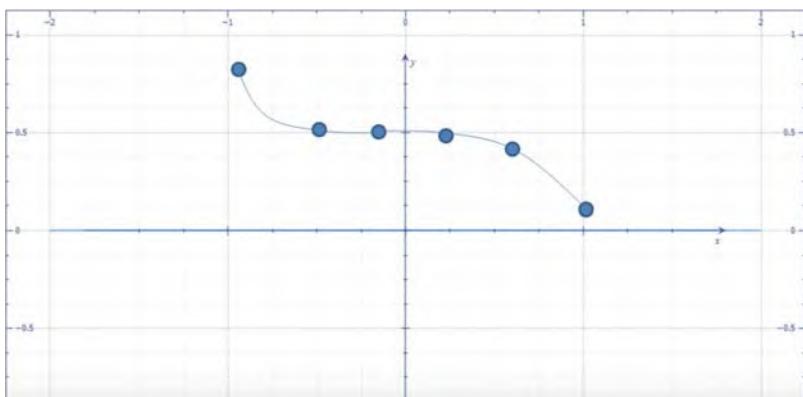


Figure 8.17 Cubic curve connection.

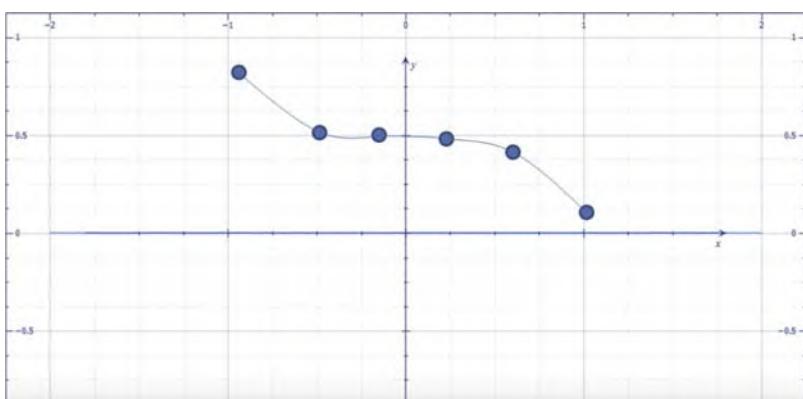


Figure 8.18 Cubic spline fitting.

vehicle make the vehicle move in a straight line as far as possible and only perform a turn at the area where the direction changes. Therefore, the cubic polynomial connection method is not suitable. Some properties of the cubic spline interpolation algorithm are introduced.

- The cubic spline curve is continuous and smooth at the junction node.
- The first and second derivatives of cubic splines are continuously differentiable.
- The boundary second derivative of the nature cubic spline is also continuous.
- A single point does not affect the entire interpolation curve.

The continuity difference at the junction node is shown in Fig. 8.19.

The first in Fig. 8.19 is discontinuous at the junction, the second is continuous at the junction but not smooth (the first derivative is discontinuous at the junction), and the third is smooth while the first derivative is continuous at the junction. Therefore, how do we compute a spline? If there are three nodes to be fitted, the set of three nodes can be expressed as $S_1 = (x_1, y_1)$, $S_2 = (x_2, y_2)$, $S_3 = (x_3, y_3)$. Then, a cubic function is used to fit two points (S_1, S_2) , and another cubic function is used to fit two points (S_2, S_3) . The two cubic functions are, respectively, denoted as

$$\begin{aligned}y &= ax^3 + bx^2 + cx + d \\y &= ex^3 + fx^2 + gx + h.\end{aligned}$$

Through these two functions, the following two equations can be obtained:

$$\begin{aligned}y_1 &= ax_1^3 + bx_1^2 + cx_1 + d, \\y_3 &= ex_3^3 + fx_3^2 + gx_3 + h.\end{aligned}$$

As both curves pass through S_2 , the formula is as follows:

$$ax_2^3 + bx_2^2 + cx_2 + d = ex_2^3 + fx_2^2 + gx_2 + h = y_2.$$

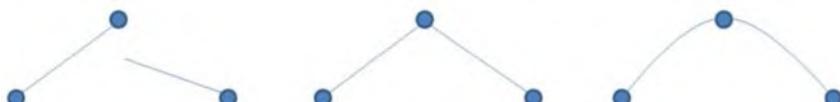


Figure 8.19 Continuity comparison.

The derivative of the spline curve at the junction is also continuous; that is, the first derivatives of the two cubic functions at the junction are also equal. The first derivative of both sides can be obtained as follows:

$$6ax_2 + 2b = 6ex_2 + 2f.$$

For a natural cubic spline, the second derivatives at the start and end nodes are also continuous.

$$6ax_1 + 2b = 0$$

$$6ex_3 + 2f = 0$$

Combined with the above six equations and the given set of nodes, the polynomial coefficients of two cubic splines (a, b, c, d, e, f, g, h) are calculated by algebraic equations. In the actual spline parameter solution, if the polynomial coefficients are determined (the parameters a, b, c, d , and so on are determined), then the subsequent spline solution can be easily solved. The programming is described as follows. The Python program is introduced later. Given a set of points ($S_1, S_2, S_3, \dots, S_i$), the cubic spline is used to fit the curve. The curve fitting process is a simple path generation.

8.3.2 Cubic spline interpolation algorithm

When calculating the coefficient of a cubic spline, the algebraic equation solution method is adopted. But the algebraic equation is usually not to solve the coefficient of cubic spline in the actual algorithm implementation. A computer algorithm for cubic spline interpolation is shown below. Considering that the derivation process of this algorithm involves numerical analysis and is far from the topic of autonomous driving, we do not discuss the derivation of this algorithm.

If there are currently $n + 1$ waypoints, they are $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.

Methods are available to solve the coefficients of each spline curve (a_i, b_i, c_i, d_i) .

- Calculate the step size between points.

$$h_i = x_{i+1} - x_i, (i = 0, 1, \dots, n + 1)$$

- Substitute the waypoint and end node conditions (if the endpoint conditions in the natural cubic spline are satisfied, $S'' = 0$) into the following matrix equation:

$$\begin{bmatrix}
 1 & 0 & 0 & \cdots & 0 \\
 h_0 & 2(h_0 + h_1) & h_1 & 0 & \\
 0 & h_1 & 2(h_1 + h_2) & h_2 & 0 \\
 \vdots & 0 & h_2 & \ddots & \vdots \\
 0 & \cdots & 0 & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\
 0 & \cdots & & & 0 & 1
 \end{bmatrix}$$

$$\times \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_n \end{bmatrix} = 6 \begin{bmatrix} 0 \\ \frac{y_2 - y_1}{h_1} - \frac{y_1 - y_0}{h_0} \\ \frac{y_3 - y_2}{h_2} - \frac{y_2 - y_1}{h_1} \\ \frac{y_4 - y_3}{h_3} - \frac{y_3 - y_2}{h_2} \\ \vdots \\ \frac{y_n - y_{n-1}}{h_{n-1}} - \frac{y_{n-1} - y_{n-2}}{h_{n-2}} \\ 0 \end{bmatrix}$$

- Solve the matrix equation and obtain the second derivative m_i .
- Calculate the cubic spline curve coefficient of each section.

$$\begin{aligned}
 a_i &= y_i \\
 b_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{2}m_i - \frac{h_i}{6}(m_{i+1} - m_i) \\
 c_i &= \frac{m_i}{2} \\
 d_i &= \frac{m_{i+1} - m_i}{6h_i}
 \end{aligned}$$

- In each subinterval $x_i \leq x \leq x_{i+1}$, the corresponding spline function is expressed as

$$f_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

Let us implement the algorithm in Python.

8.3.3 Using Python to implement cubic spline interpolation for path generation

A new Python file cubic_spline.py is created, and the class of the natural cubic spline is defined.

Code list 8.3.1 Class of cubic spline interpolation

```
# coding=utf-8

import numpy as np
import bisect

class Spline:

    def __init__(self, x, y):
        self.a, self.b, self.c, self.d = [], [], [], []
        self.x = x
        self.y = y
        self.nx = len(x) # dimension of x
        h = np.diff(x)
        # calc coefficient c
        self.a = [iy for iy in y]
        # calc coefficient c
        A = self.__calc_A(h)
        B = self.__calc_B(h)
        self.m = np.linalg.solve(A, B)
        self.c = self.m / 2.0
        # calc spline coefficient b and d
        for i in range(self.nx - 1):
            self.d.append(((self.c[i + 1] - self.c[i]) / (3.0 * h[i])))
            tb = (self.a[i + 1] - self.a[i]) / h[i] - h[i] * (self.c[i + 1] + 2.0
            * self.c[i]) / 3.0
            self.b.append(tb)

    def calc(self, t):
        if t < self.x[0]:
            return None
        elif t > self.x[-1]:
            return None
        else:
            index = bisect.bisect(self.x, t) - 1
            tx = self.x[index]
            ty = self.y[index]
            dx = self.x[index + 1] - tx
            dy = self.y[index + 1] - ty
            dt = t - tx
            dtx = self.d[index]
            dty = self.d[index + 1]
            a = (dty - dtx) / (3.0 * dx)
            b = (4.0 * dy - dtx * (2.0 * dx) - dty * (dx)) / (dx * dx)
            c = (3.0 * dy - dtx * (dx) - dty * (2.0 * dx)) / (dx * dx)
            d = ty - a * tx - b * tx * tx - c * tx * tx * tx
            return [a, b, c, d]
```

```

        return None
    i = self.__search_index(t)
    dx = t - self.x[i]
    result = self.a[i] + self.b[i] * dx + \
             self.c[i] * dx ** 2.0 + self.d[i] * dx ** 3.0
    return result

def __search_index(self, x):
    return bisect.bisect(self.x, x) - 1

def __calc_A(self, h):
    A = np.zeros((self.nx, self.nx))
    A[0, 0] = 1.0
    for i in range(self.nx - 1):
        if i != (self.nx - 2):
            A[i + 1, i + 1] = 2.0 * (h[i] + h[i + 1])
            A[i + 1, i] = h[i]
            A[i, i + 1] = h[i]
        A[0, 1] = 0.0
    A[self.nx - 1, self.nx - 2] = 0.0
    A[self.nx - 1, self.nx - 1] = 1.0
    return A

def __calc_B(self, h):
    B = np.zeros(self.nx)
    for i in range(self.nx - 2):
        B[i + 1] = 6.0 * (self.a[i + 2] - self.a[i + 1]) / h[i + 1] - 6.0 * \
                   (self.a[i + 1] - self.a[i]) / h[i]
    return B

```

Functions `__calc_A` and `__calc_B` are, respectively, used to construct the left and right matrices in the second step of the aforementioned equation. As m is a diagonal matrix, `linalg.solve` from the Numpy library is used to calculate m . Next, a new test.py file is created to execute the test code.

Code list 8.3.2 Cubic spline interpolation used to generate paths

```

import cubic_spline
import numpy as np
import matplotlib.pyplot as plt

def main():
    x = [-4., -2, 0.0, 2, 4, 6, 10]
    y = [1.2, 0.6, 0.0, 1.5, 3.8, 5.0, 3.0]
    spline = cubic_spline.Spline(x, y)
    rx = np.arange(-4.0, 10, 0.01)
    ry = [spline.calc(i) for i in rx]
    plt.plot(x, y, "og")
    plt.plot(rx, ry, "-r")
    plt.grid(True)
    plt.axis("equal")
    plt.show()

if __name__ == '__main__':
    main()

```

The result of the generated path is shown in Fig. 8.20.

The dots are the node sets to be fitted, and the red curve is the cubic spline curve obtained by fitting.

Note: In practice, spline interpolation need not be implemented because there are many open-source codes, such as the code in <http://kluge.in-chemnitz.de/opensource/spliner>, to implement the cubic spline curve algorithm in C++.

8.4 Motion planning method of the autonomous vehicle based on Frenet optimization trajectory

The method of generating a simple path based on a cubic spline interpolation algorithm was discussed in the previous section. However, the path

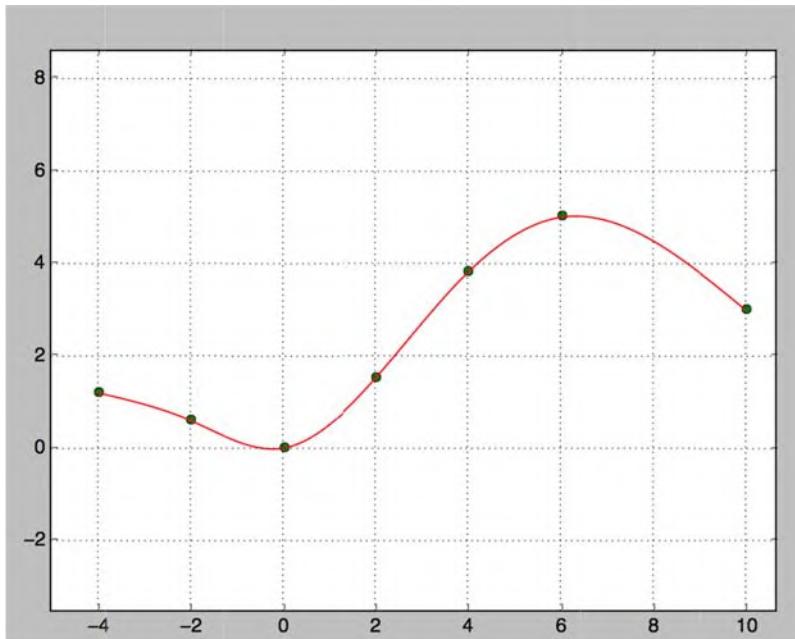


Figure 8.20 Path generated by cubic spline interpolation.

only contains position information and does not contain control signals that are transmitted to the control layer for execution. In this section, we explain the concept of a motion plan and how to transmit paths as control signals to the control layer for execution.

Motion planning is located at the bottom of the autonomous vehicle planning module. Its main task is to generate a series of actions according to the current configuration and the target configuration. These action sequences are the motion trajectory that the control layer can execute. This section focuses on one of the motion plan algorithms, that is, the optimized trajectory method based on the Frenet coordinate system. This method has strong practicability in assisted driving and autonomous driving at high speeds. It is one of the motion plan algorithms that are widely used at present.

The method of motion planning based on the Frenet coordinate system⁴ was proposed by Moritz Werling. Before discussing the method of motion planning based on the Frenet coordinate system, we need to know which is the optimal motion sequence. For lateral control, if the vehicle deviates from the desired lane to avoid obstacles, change lanes, and so on,

then the optimal motion sequence is a trajectory sequence to achieve relatively safe, comfortable, simple, and efficient driving under the constraints of vehicle dynamics.

Similarly, the optimal longitudinal trajectory can also be defined as follows: If the vehicle is too fast at this time or too close to the vehicle in front, then it must decelerate. Deceleration also has different degrees, such as a sharp brake or comfortable deceleration, which may cause different sensations. What is “comfortable” deceleration? We can define it using the concept of a jerk, which is the rate of change of acceleration. Why do we want to study this physical quantity? Generally, for passengers, when the acceleration of a vehicle causes discomfort, it does not result in the size of the acceleration value. Imaging that while in an airplane, the acceleration may be high, but passengers do not feel any discomfort. Physicists use the rate of change of acceleration, which is the derivative of acceleration, to measure the comfort level of deceleration and acceleration. In general, excessively high acceleration causes discomfort for passengers. Thus, from the perspective of passenger comfort, the jerk should be optimized. At the same time, the control cycle T of trajectory, that is, the operation time of a control action, should be introduced.

$$T = t_{\text{end}} - t_{\text{start}}$$

8.4.1 Why use the Frenet coordinate system

In the Frenet coordinate system, the center line of the road is generally taken as the road reference line. The tangent vector t and normal vector n of the reference line are used to establish a coordinate system, that is, the Frenet coordinate system, as shown in Fig. 8.21. It takes the center line of the start road as the original point. The coordinate axes are perpendicular and divided into the s -direction (the direction along the road reference line as longitudinal) and d -direction (the current normal vector of the reference line; it is the distance away from the road center line as lateral). Unlike the Cartesian coordinate system, the Frenet coordinate system is able to simplify the road curve fitting problem. In the highway environment, we can always find the road reference line, which is the road center line. On the basis of the reference line, the representation of the vehicle location can simply use the longitudinal distance (the distance along the road direction) and horizontal distance (the distance deviation from the reference line). In the same way, the two directions (s and \dot{d}) of the speed calculation are relatively

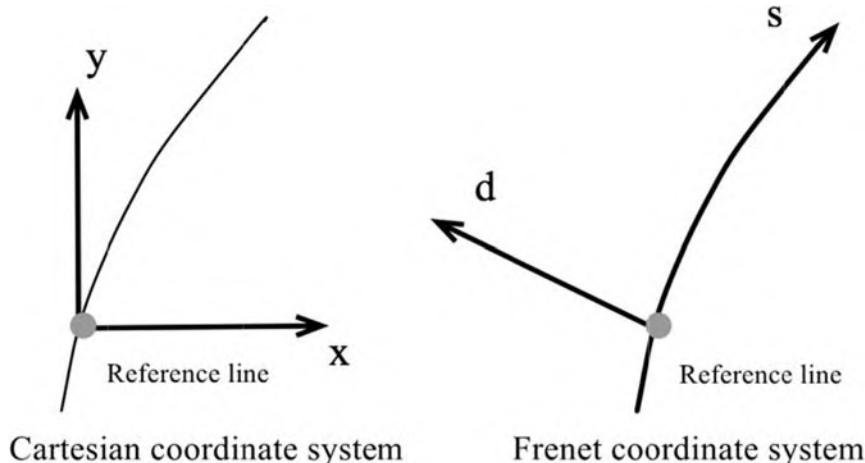


Figure 8.21 Comparison between Cartesian and Frenet coordinate systems.

simple. When the Cartesian coordinate system is used, if the road environment is complex, then a complicated curve will be fitted. Hence, it is very unrealistic.

Therefore, the configuration space in the current action plan problem can be divided into three dimensions: (s, d, t) , where t is the time of each action in path planning. The essential difference between the trajectory and the path is that the dimension of time is considered in the trajectory.

The key concept of the motion planning method based on the Frenet coordinate system is to decompose the high-dimensional optimization problem of motion planning into two independent optimization problems in the lateral and longitudinal directions (Fig. 8.22).

If the current vehicle is required to cross the dotted line to complete a lane change behavior at the moment t_8 in the behavior planning layer, then the vehicle is required to complete Δd action in the lateral direction and Δs action in the longitudinal direction.

The functions of s and d are, respectively, expressed by t : $s(t)$ and $d(t)$, as shown on the right of Fig. 8.22. Which path should be chosen for the optimal path of d, s ? Through this transformation, the original motion plan problem is decomposed into two independent optimization problems, namely, the lateral and longitudinal trajectory optimization problems. The loss function C is selected, and the trajectory with the smallest C is taken as the final planned action sequence. The definition of the loss function based on the Frenet coordinate system method is also associated with the jerk described above.

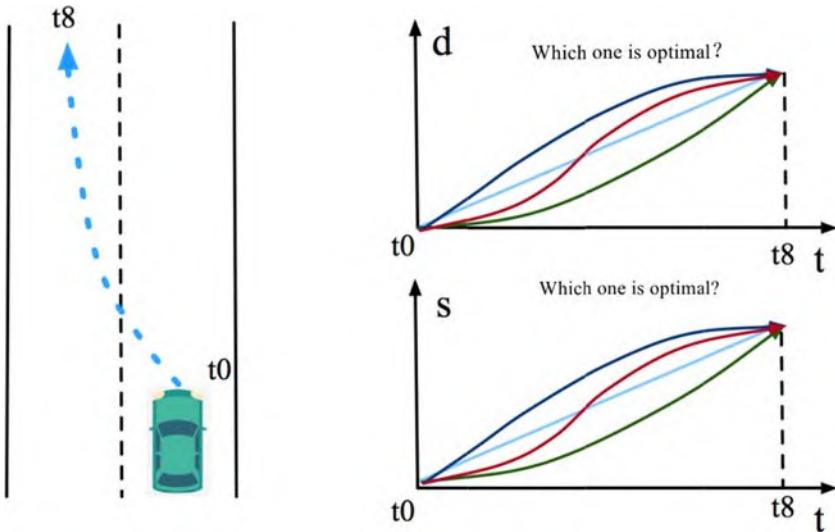


Figure 8.22 Dividing the trajectory optimization problem into longitudinal and lateral ones.

8.4.2 Jerk minimization and polynomial solution of fifth degree trajectory

As a result of the introduction of the Frenet coordinate system, the trajectory optimization problem is decomposed into the independent optimization problems of the s and d directions. Thus, the minimization of jerk can be performed from the lateral and longitudinal directions. Let p be the configuration we want to consider ($p = (s, d)$). Then, the expression of J_t in time segment $t_1 - t_0$ is

$$J_t(p(t)) = \int_{t_0}^{t_1} p(\tau)^2 d\tau.$$

Then, the problem is simplified to find $p(t)$ that can minimize $J_t(p(t))$. Takahashi et al.⁵ proved that the solution of any jerk optimization problem can be expressed by a fifth polynomial:

$$p(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5.$$

To solve this equation, some initial configuration and target configuration are required. The lateral path planning is taken as an example. The

initial configuration is $D_0 = [d_0, \dot{d}_0, \ddot{d}_0]$, where the initial lateral offset of the vehicle, the initial lateral velocity, and the initial lateral acceleration at the moment t_0 are denoted by $d_0, \dot{d}_0, \ddot{d}_0$, respectively. The following equations can be obtained:

$$\begin{aligned} d(t_0) &= \alpha_{d0} + \alpha_{d1}t_0 + \alpha_{d2}t_0^2 + \alpha_{d3}t_0^3 + \alpha_{d4}t_0^4 + \alpha_{d5}t_0^5, \\ \dot{d}(t_0) &= \alpha_{d1} + 2\alpha_{d2}t_0 + 3\alpha_{d3}t_0^2 + 4\alpha_{d4}t_0^3 + 5\alpha_{d5}t_0^4, \\ \ddot{d}(t_0) &= 2\alpha_{d2} + 6\alpha_{d3}t_0 + 12\alpha_{d4}t_0^2 + 20\alpha_{d5}t_0^3. \end{aligned}$$

To distinguish the lateral and longitudinal directions, α_{di} and α_{si} are used in representing polynomial coefficients in directions d and s , respectively. Similarly, the equations can be obtained according to the lateral objective config. $D_1 = [d_1, \dot{d}_1, \ddot{d}_1]$.

$$\begin{aligned} d(t_1) &= \alpha_{d0} + \alpha_{d1}t_1 + \alpha_{d2}t_1^2 + \alpha_{d3}t_1^3 + \alpha_{d4}t_1^4 + \alpha_{d5}t_1^5 \\ \dot{d}(t_1) &= \alpha_{d1} + 2\alpha_{d2}t_1 + 3\alpha_{d3}t_1^2 + 4\alpha_{d4}t_1^3 + 5\alpha_{d5}t_1^4 \\ \ddot{d}(t_1) &= 2\alpha_{d2} + 6\alpha_{d3}t_1 + 12\alpha_{d4}t_1^2 + 20\alpha_{d5}t_1^3 \end{aligned}$$

If $t_0 = 0$, then the solution of the six-element equation can be simplified, and the initial conditions α_{d0} , α_{d1} , and α_{d2} can be directly obtained.

$$\begin{aligned} d(t_1) &= \alpha_{d0} + \alpha_{d1}t_1 + \alpha_{d2}t_1^2 + \alpha_{d3}t_1^3 + \alpha_{d4}t_1^4 + \alpha_{d5}t_1^5 \\ \dot{d}(t_1) &= \alpha_{d1} + 2\alpha_{d2}t_1 + 3\alpha_{d3}t_1^2 + 4\alpha_{d4}t_1^3 + 5\alpha_{d5}t_1^4 \\ \ddot{d}(t_1) &= 2\alpha_{d2} + 6\alpha_{d3}t_1 + 12\alpha_{d4}t_1^2 + 20\alpha_{d5}t_1^3 \end{aligned}$$

If $t_0 = 0$, then the solution of the six-element equation can be simplified. α_{d0} , α_{d1} , and α_{d2} can be found directly.

$$\begin{aligned} \alpha_{d0} &= d(t_0) \\ \alpha_{d1} &= \dot{d}(t_0) \\ \alpha_{d2} &= \frac{\ddot{d}(t_0)}{2} \end{aligned}$$

If $T = t_1 - t_0$, then the remaining three coefficients $\alpha_{d3}, \alpha_{d4}, \alpha_{d5}$ can be obtained by solving the following matrix equation:

$$\begin{bmatrix} T^3 & T^4 & T^5 \\ 3T^2 & 4T^3 & 5T^4 \\ 6T & 12T^2 & 20T^3 \end{bmatrix} \times \begin{bmatrix} \alpha_{d3} \\ \alpha_{d4} \\ \alpha_{d5} \end{bmatrix} = \begin{bmatrix} d(t_1) - \left(d(t_0) + \dot{d}(t_0)T + \frac{1}{2}\ddot{d}(t_0)T^2 \right) \\ \dot{d}(t_1) - \left(\dot{d}(t_0) + \ddot{d}(t_0)T \right) \\ \ddot{d}(t_1) - \ddot{d}(t_0) \end{bmatrix}$$

The solution of this equation can be found simply by using `np.linalg.solve` in Python's Numpy library.

Thus far, any initial configuration D_0 , target configuration D_1 , and control period time T are given. The coefficient of the quintic polynomial in the corresponding dd -direction with regard to time t can be solved. Similarly, the same method is used to solve the coefficient of the quintic polynomial in the longitudinal direction (the ss -direction). Therefore, how do we determine the optimal trajectory? The idea of the method based on the Frenet coordinate system is to obtain the alternative set of trajectories through a set of target configurations. Then, the optimal trajectory in the alternative set based on the minimum principle of jerk is selected. This chapter still takes the optimized trajectory in the direction of d as an example to explain.

First, the following target configuration set can be selected to calculate a set of alternative polynomials:

$$\begin{bmatrix} d_1, \dot{d}_1, \ddot{d}_1, T \end{bmatrix}_{ij} = [d_i, 0, 0, T_j].$$

For the optimization problem, we actually expect that the vehicle can always run parallel along the reference line (road center line). d_1 is a constant. Therefore, $\dot{d}_i = \ddot{d}_i = 0$. The target configuration only involves the combination of d_i and T_j . The two variables in autonomous application scenarios are actually limited. The target configuration value range can be constrained by defining (d_{\min}, d_{\max}) and (T_{\min}, T_{\max}) . The sampling density can be limited by Δd and ΔT . Thus, a finite set of alternative trajectories can be obtained in each control braking cycle, as shown in Fig. 8.23.

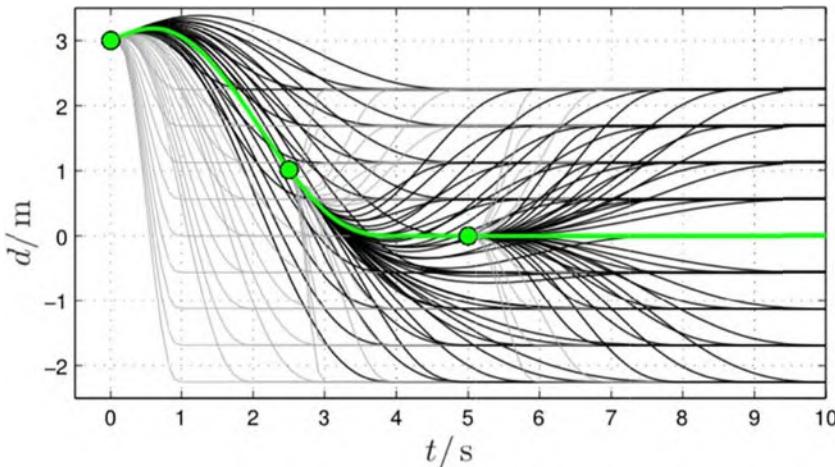


Figure 8.23 Horizontal optimization trajectory set.

To select the optimal trajectory (the green trajectory in Fig. 8.23) from the alternative set, a better loss function needs to be designed. The loss function is also different if the scenarios are different. Taking the lateral trajectory as an example, the loss function can be designed as

$$C_d = k_J J_t(d(t)) + k_t T + k_d d_1^2.$$

The loss function contains three penalty terms:

- $k_J J_t(d(t))$: Alternative trajectory with the larger value of Jerk
- $k_t T$: Braking should be rapid and short
- $k_d d_1^2$: The target state should not deviate too far from the road center line

k_J , k_t , and k_d are the proportionality coefficients of the three penalty terms and are also called weight values. Their values determine the preferred aspect of optimization of the loss function. Thus, the loss of all alternative trajectories can be calculated, and the alternative trajectory with the lowest loss can be selected as the final lateral trajectory.

The above loss function is only applicable to relatively high-speed scenarios. In the case of extremely low speed, the braking capacity of the vehicle is incomplete. At this time, d can no longer be expressed as a quintic polynomial about time t , and the loss function will also be different. However, this idea based on finite sampled trajectories and the basic principle of searching optimal trajectories by optimizing loss functions are still similar and will not be described here.

After discussing the horizontal trajectory optimization problem, we consider longitudinal trajectory optimization. The loss function of the longitudinal trajectory optimization is different if the scenarios are different. The optimization scenarios of the longitudinal trajectory based on the Frenet coordinate system are divided into the following three categories:

- Follow the car;
- Junction and parking;
- Maintain speed.

In this section, longitudinal trajectory optimization in the scenario of maintaining speed is described in detail. This method is widely used in the application scenarios of assisted driving on expressways. In this scenario, the target position (s_1) does not need to be considered in the target configuration. Therefore, in this scenario, the initial target configuration is still $s_0, \dot{s}_0, \ddot{s}_0$; the target configuration becomes \dot{s}_1, \ddot{s}_1 ; and the loss function is

$$C_s = k_d J_t(s(t)) + k_t T + k_s (\dot{s}_1 - \dot{s}_c)^2.$$

\dot{s}_c is the longitudinal speed to be maintained, and the third penalty term is introduced to make the longitudinal speed in the target configuration as close as possible to the set speed. In this scenario, the target configuration set is

$$\left[\dot{s}_1, \ddot{s}_1, T \right]_{ij} = [[\dot{s}_c + \Delta \dot{s}_i], 0, T_j].$$

In other words, the variable parameters in the optimization process are $\Delta \dot{s}_i$ and T_j . Similarly, the density of trajectory sampling can be set by setting ΔT and $\Delta \dot{s}_i$ so as to obtain a finite longitudinal trajectory set.

In Fig. 8.24, the green line is the longitudinal optimal trajectory.

We have discussed the lateral and longitudinal optimal trajectory search methods. In applications, the loss functions of two directions can also be combined into one:

$$C_{\text{total}} = k_{\text{lat}} C_d + k_{\text{lon}} C_s.$$

In this way, the optimized trajectory set can be obtained by minimizing C_{total} . Apart from the “optimal” trajectory polynomial parameters, “sub-optimal” trajectories can be obtained to meet various requirements in real scenes.

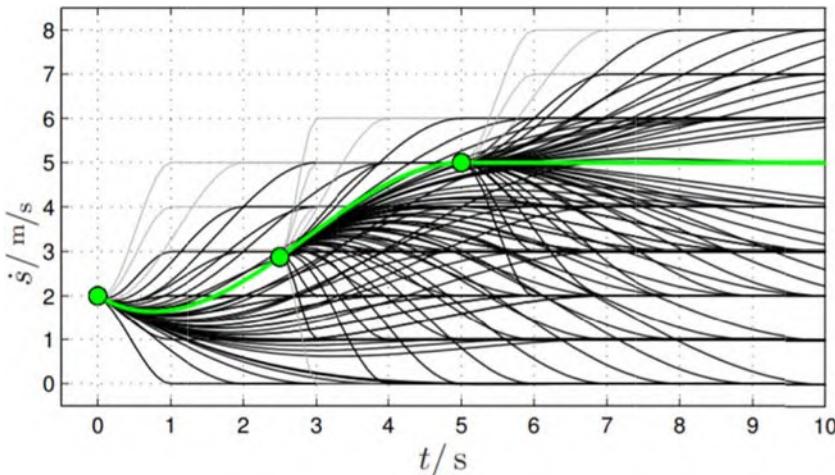


Figure 8.24 Set of longitudinal optimization trajectories.

8.4.3 Collision avoidance

Obviously, the trajectory optimization loss function mentioned above does not contain relevant penalties for obstacle avoidance, and the loss function does not contain control constraints, such as maximum speed, maximum acceleration, and maximum curvature. That is to say, the current optimization trajectory set does not consider obstacle avoidance and control constraints. An important reason for not adding obstacle avoidance into the loss function is that if the collision penalty term is added, then a large number of parameters would need a manual adjustment (weight coefficient). Moreover, the loss function of design would become complex. The consideration of these factors will be independent of the method based on the Frenet coordinate system. After completing the optimal trajectory, that is, after the loss calculation of all alternative tracks is completed, a track check should be performed. Through this track check, the tracks that do not conform to the control constraints and may collide with obstacles will be filtered. The check contents include the following:

- Check whether the speed in direction S exceeds the limited maximum speed;
- Check whether the acceleration in the S direction exceeds the maximum acceleration;
- Check whether the curvature of the track exceeds the maximum curvature;
- Check whether the track can cause a collision (accident).

Generally, obstacle avoidance is related to target behavior prediction, and both of them are complex subjects. Advanced autonomous driving systems usually have the ability to predict target behavior so as to determine whether the generated trajectory will have a collision accident. This section focuses on the motion plan of autonomous vehicles. Thus, the examples in the following only consider static obstacle avoidance and motion planning.

8.4.4 Example of motion planning for autonomous vehicles based on Frenet optimization trajectory

As a result of the length of the code in the Planner, the complete code of this example can be found in the present code routine provided with this book, and only the core content of the action plan algorithm is explained here. Python is also used to achieve the action plan algorithm in this section.

We should generate the center reference line of the road to be tracked and the static obstacle model. The cubic spline interpolation algorithm mentioned in the previous section is used to generate the reference line, and the code is as follows:

Code list 8.4.1 Generation of the reference line

```

wx = [0.0, 10.0, 20.5, 30.0, 40.5, 50.0, 60.0]
wy = [0.0, -4.0, 1.0, 6.5, 8.0, 10.0, 6.0]
ob = np.array([[20.0, 10.0],
               [30.0, 6.0],
               [30.0, 5.0],
               [35.0, 7.0],
               [50.0, 12.0]
              ])
tx, ty, tyaw, tc, csp = generate_target_course(wx, wy)

```

The reference line and static obstacles are generated, as shown in Fig. 8.25.

The red line is the global path fitted, and the blue node is the static obstacle model. Then, we define some parameters, such as road and vehicle control attributes.

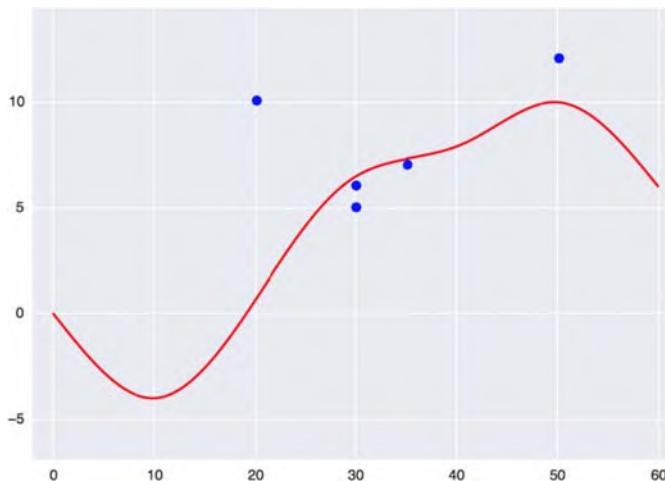


Figure 8.25 Reference paths and obstacle positions.

Code list 8.4.2 Definition of some parameters

```
MAX_SPEED = 50.0 / 3.6
MAX_ACCEL = 2.0
MAX_CURVATURE = 1.0
MAX_ROAD_WIDTH = 7.0
D_ROAD_W = 1.0
DT = 0.2 # Delta T [s]
MAXT = 5.0
MINT = 4.0
TARGET_SPEED = 30.0 / 3.6
D_T_S = 5.0 / 3.6
N_S_SAMPLE = 1
ROBOT_RADIUS = 2.0 # robot radius [m]
KJ = 0.1
KT = 0.1
KD = 1.0
KLAT = 1.0
KLON = 1.0
```

A series of horizontal and vertical trajectories are generated using the optimization trajectory method based on the Frenet coordinate system, and the corresponding loss value of each trajectory is calculated as.

Code list 8.4.3 Generation of the reference line

```
def calc_frenet_paths(c_speed, c_d, c_d_d, c_d_dd, s0):
    frenet_paths = []

    for di in np.arange(-MAX_ROAD_WIDTH, MAX_ROAD_WIDTH,
D_ROAD_W):
        for Ti in np.arange(MINT, MAXT, DT):
            fp = Frenet_path()
            lat_qp = quintic_polynomial(c_d, c_d_d, c_d_dd, di, 0.0,
0.0, Ti)

            fp.t = [t for t in np.arange(0.0, Ti, DT)]
            fp.d = [lat_qp.calc_point(t) for t in fp.t]
            fp.d_d = [lat_qp.calc_first_derivative(t) for t in fp.t]
            fp.d_dd = [lat_qp.calc_second_derivative(t) for t in fp.t]
            fp.d_ddd = [lat_qp.calc_third_derivative(t) for t in fp.t]

            for tv in np.arange(TARGET_SPEED - D_T_S * N_S_SAMPLE,
TARGET_SPEED + D_T_S * N_S_SAMPLE, D_T_S):
                tfp = copy.deepcopy(fp)
                lon_qp = quartic_polynomial(s0, c_speed, 0.0, tv, 0.0,
Ti)

                tfp.s = [lon_qp.calc_point(t) for t in fp.t]
                tfp.s_d = [lon_qp.calc_first_derivative(t) for t in fp.t]
                tfp.s_dd = [lon_qp.calc_second_derivative(t) for t in fp.t]
```

```
tfp.s_ddd = [lon_qp.calc_third_derivative(t) for t in
fp.t]

Jp = sum(np.power(tfp.d_ddd, 2)) # square of jerk
Js = sum(np.power(tfp.s_ddd, 2)) # square of jerk

# square of diff from target speed
ds = (TARGET_SPEED - tfp.s_d[-1]) ** 2
tfp.cd = KJ * Jp + KT * Ti + KD * tfp.d[-1] ** 2
tfp.cv = KJ * Js + KT * Ti + KD * ds
tfp.cf = KLAT * tfp.cd + KLON * tfp.cv
frenet_paths.append(tfp)

return frenet_paths
```

An important class is the class of quintic polynomial fitted, which is used to generate quintic polynomial coefficients by fitting; its definition is as follows:

Code list 8.4.4 Generation of longitudinal and lateral trajectories

```

class quintic_polynomial:

    def __init__(self, xs, vxs, axs, xe, vxe, axe, T):
        self.xs = xs
        self.vxs = vxs
        self.axs = axs
        self.xe = xe
        self.vxe = vxe
        self.axe = axe
        self.a0 = xs
        self.a1 = vxs
        self.a2 = axs / 2.0
        A = np.array([[T ** 3, T ** 4, T ** 5],
                      [3 * T ** 2, 4 * T ** 3, 5 * T ** 4],
                      [6 * T, 12 * T ** 2, 20 * T ** 3]])
        b = np.array([xe - self.a0 - self.a1 * T - self.a2 * T ** 2,
                      vxe - self.a1 - 2 * self.a2 * T,
                      axe - 2 * self.a2])
        x = np.linalg.solve(A, b)
        self.a3 = x[0]
        self.a4 = x[1]
        self.a5 = x[2]

    def calc_point(self, t):
        xt = self.a0 + self.a1 * t + self.a2 * t ** 2 + \
            self.a3 * t ** 3 + self.a4 * t ** 4 + self.a5 * t ** 5
        return xt

    def calc_first_derivative(self, t):

```

```

        xt = self.a1 + 2 * self.a2 * t + \
            3 * self.a3 * t ** 2 + 4 * self.a4 * t ** 3 + 5 * self.a5 * t ** 4
    return xt

def calc_second_derivative(self, t):
    xt = 2 * self.a2 + 6 * self.a3 * t + 12 * self.a4 * t ** 2 + 20 * self.a5 * t ** 3
    return xt

def calc_third_derivative(self, t):
    xt = 6 * self.a3 + 24 * self.a4 * t + 60 * self.a5 * t ** 2
    return xt

```

The solution process of quintic polynomial coefficients is the same as that described previously. One can easily solve the matrix by using the np.linalg.solve (A, b) method in the Numpy library. Finally, a simple obstacle avoidance algorithm is implemented.

Code list 8.4.5 Class of obstacle avoidance

```

def check_collision(fp, ob):
    for i in range(len(ob[:, 0])):
        d = [(ix - ob[i, 0]) ** 2 + (iy - ob[i, 1]) ** 2]
        for (ix, iy) in zip(fp.x, fp.y)]
        collision = any([di <= ROBOT_RADIUS ** 2 for di in d])
        if collision:
            return False
    return True

```

As obstacle avoidance has been simplified to static obstacle avoidance, we simply calculate the distance between all path plan nodes and obstacles

and predict whether collisions will occur according to the distance. The following is the complete optimized trajectory checking function:

Code list 8.4.6 Function of trajectory checking

```
def check_paths(fplist, ob):
    okind = []
    for i in range(len(fplist)):
        if any([v > MAX_SPEED for v in fplist[i].s_d]):
            continue
        elif any([abs(a) > MAX_ACCEL for a in fplist[i].s_dd]):
            continue
        elif any([abs(c) > MAX_CURVATURE for c in fplist[i].c]):
            continue
        elif not check_collision(fplist[i], ob):
            continue
        okind.append(i)
    return [fplist[i] for i in okind]
```

Therefore, the selection of the final optimization trajectory is not only based on the minimum loss function. Some trajectories that do not conform to the constraints will be filtered by trajectory checking. Therefore, the Frenet-based optimization trajectory is used to generate the action plan of autonomous vehicles, and it can usually find the optimal solution of finite sets. When the optimal solution fails to pass the check, it is automatically replaced with a “suboptimal solution.”

The complete action plan effect is shown in Fig. 8.26. The set of green nodes represents the optimized trajectory of the final solution. The model obviously achieves obstacle avoidance and control trajectory output.

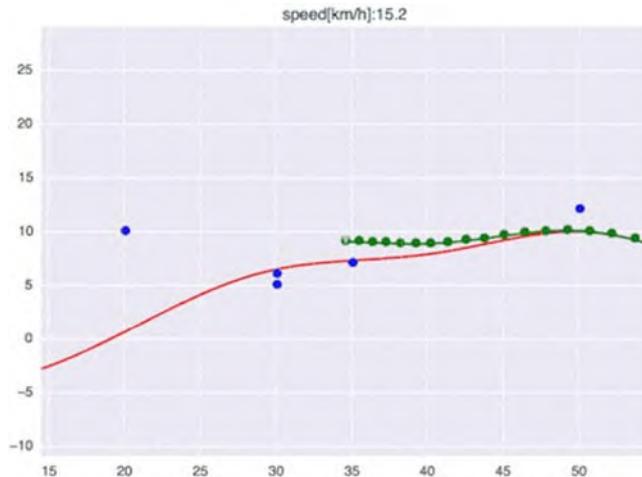


Figure 8.26 Effect of action planning.

References

1. Wikipedia contributors. Finite-state machine. In: *Wikipedia, The Free Encyclopedia*; 2022, February 8. Retrieved 01:10, March 23, 2022, from https://en.wikipedia.org/w/index.php?title=Finite-state_machine&oldid=1070640030.
2. Montemerlo M, Becker J, Bhat S, et al. Junior: the Stanford Entry in the urban challenge. *J Field Robot.* 2009;25(9):569–597.
3. Werling M, Ziegler J, Kammel S, et al. Optimal trajectory generation for dynamic street scenarios in a Frenét frame. In: *IEEE International Conference on Robotics and Automation*. IEEE; 2010:987–993.
4. Takahashi A, Hongo T, Ninomiya Y, et al. Local path planning and motion control for Agy in positioning. In: *IEEE/Rsj International Workshop on Intelligent Robots and Systems '89*. The Autonomous Mobile Robots and ITS Applications. IROS '89. Proceedings of IEEE; 2002:392–397.
5. Wikipedia contributors. Directed graph. In: *Wikipedia, The Free Encyclopedia*; 2022, March 11. Retrieved 14:50, March 24, 2022, from https://en.wikipedia.org/w/index.php?title=Directed_graph&oldid=1076467468.
6. Wikipedia contributors. A* search algorithm. In: *Wikipedia, The Free Encyclopedia*. 2022, February 23. Retrieved 14:51, March 24, 2022, from https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1073628314.
7. Karl K. Path planning in unstructured environments. In: *Degree Project in Vehicle Engineering*. 2016. Second cycle, 30 Credits. Stockholm, Sweden.
8. Dolgov D, Thrun S. *Autonomous Driving in Semistructured Environments: Mapping and Planning*. 2009.
9. Dolgov D, Thrun S, Montemerlo M, James D. Path planning for autonomous vehicles in unknown semi-structured environments. *Int J Robot Res.* 2010.
10. Function approximation with Cubic Splines. <https://www.youtube.com/watch?v=f4iNbNRKZKU>.

CHAPTER 9

Vehicle model and advanced control

Zebang Shen¹ and Rui Zhao²

¹R&D Center, Mercedes-Benz Group AG, Beijing, China; ²School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

9.1 Kinematic bicycle model and dynamic bicycle model	273
9.1.1 Bicycle model	274
9.1.2 Kinematic bicycle model	276
9.1.3 Dynamic bicycle model	277
9.2 Rudiments of autonomous vehicle control	278
9.2.1 Need for control theory	278
9.2.2 PID control algorithm	279
9.2.2.1 <i>Proportional control</i>	280
9.2.2.2 <i>Proportional and derivative control</i>	282
9.3 MPC based on kinematic model	291
9.3.1 Applying PID controller forwards to steering control	291
9.3.2 Predictive model	291
9.3.3 Online optimal loop based on time series	293
9.3.4 Feedback correction	294
9.4 Trajectory tracking	295
References	305

The control module, as an actual processing module in an unmanned vehicle, aims to extend vehicle movement as far as possible according to the trajectory computed by the planning module. This chapter introduces the control theory and technology required in the field of autonomous driving, including some simple vehicle models, PID control, and predictive control model.

9.1 Kinematic bicycle model and dynamic bicycle model

Before understanding advanced control algorithms for vehicles, we should understand the vehicle model, which is a type of model that can describe

vehicle motion. Obviously, the more complex the model is, the closer it is to the actual vehicle motion in reality. This section introduces two widely used vehicle models¹: the kinematic bicycle model and the dynamic bicycle model.

Given a top-to-bottom approach, the autonomous driving system is usually divided into three layers: perception, planning, and control. The planning layer specifies the driving path on the basis of higher-level layer information and real-time feedback information from the lower layer. Then, the output is the reference action sequence of the vehicle. Meanwhile, the control system needs to drive the vehicle strictly in accordance with this reference action sequence. Generally, it is described in the form of a polynomial (e.g., the third-degree polynomial shown below can describe the greatest majority path).

$$\gamma = ax^3 + bx^2 + cx + d$$

The control of unmanned vehicles relies on model predictive control (MPC), which generates a series of feasible control states that the vehicle can actually do and optimize on the basis of some algorithms (usually a constrained nonlinear optimization algorithm) by minimizing a certain kind of cost function. The optimization result of the cost function depends on the state computed by the kinematic or dynamic model of the vehicle and the reference path. These models are illustrated in this section.

9.1.1 Bicycle model

A model is usually built to simplify a complex problem. For example, the bicycle model is a simple and effective way to simplify the motion of a car. Some assumptions based on the bicycle model are described herein.

- Ignore the movement of the vehicle at the z-axis direction, that is, assuming that a vehicle is a moving object on a two-dimensional plane (it can be equivalent to the overhead view in the sky).
- The structure of the vehicle is like a bicycle so that the front two wheels of the vehicle have the same angle and speed, and the rear two wheels are also the same. Then, the front and rear tires can be described by one tire, like that in a bicycle.
- The movement manner of the vehicle is also the same as that of a bicycle. Hence, the front tires control the turning angle of the vehicle.

We describe a vehicle on a two-dimensional plane in Fig. 9.1.

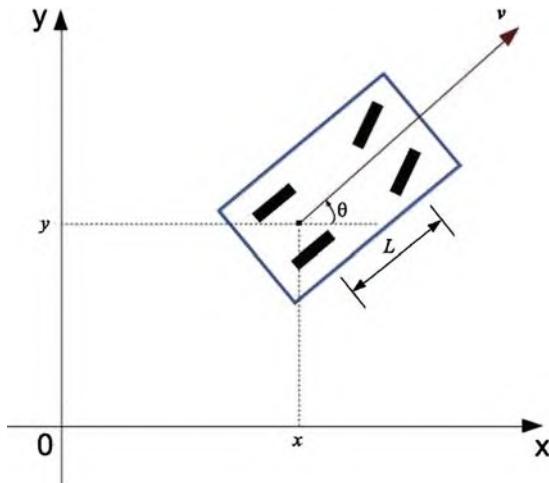


Figure 9.1 Vehicle description on a two-dimensional plane.

θ is the yaw angle, which is the counterclockwise angle related to the x -axis.

v is the velocity in the θ direction (the speed of the center of the rear wheel of the vehicle).

L is the wheelbase of the vehicle (the distance between the front and rear wheels).

(x, y) is the vehicle coordinates (the coordinates of the center of the rear wheel).

Fig. 9.2 shows the corresponding bicycle model.

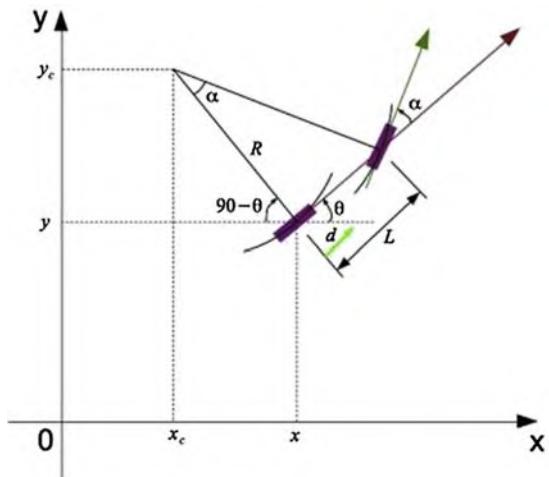


Figure 9.2 Bicycle model of vehicle.

9.1.2 Kinematic bicycle model

The variables of the kinematic bicycle model can be simplified as (a, δ_f) , where a is the acceleration of the vehicle, in which stepping on the accelerator means positive acceleration and stepping on the brake pedal means negative acceleration. δ_f is the turning angle of the steering wheel that is assumed as the current turning angle of the front wheel. In this way, two variables are used to describe the control input of the vehicle. Then, the four state variables for describing the current state of the vehicle are defined in the model.

- x : The coordinate in the x-axis at the current time of the vehicle
- y : The coordinate in the y-axis at the current time of the vehicle
- ψ : The yaw angle at the current time of the vehicle described in radians and with a positive counterclockwise direction
- v : Instantaneous velocity of the vehicle

A simple kinematic bicycle model is shown in Fig. 9.3.

l_f and l_r are the distances from the front and rear wheels to the center of gravity of the vehicle, respectively. According to the theorem, the update formulas of the state parameters in the kinematic bicycle model are as follows:

$$\begin{aligned}x_{t+1} &= x_t + v_t \cos(\psi_t + \beta) \times dt \\y_{t+1} &= y_t + v_t \sin(\psi_t + \beta) \times dt \\\psi_{t+1} &= \psi_t + \frac{v_t}{l_r} \sin(\beta) \times dt \\v_{t+1} &= v_t + a \times dt\end{aligned}$$

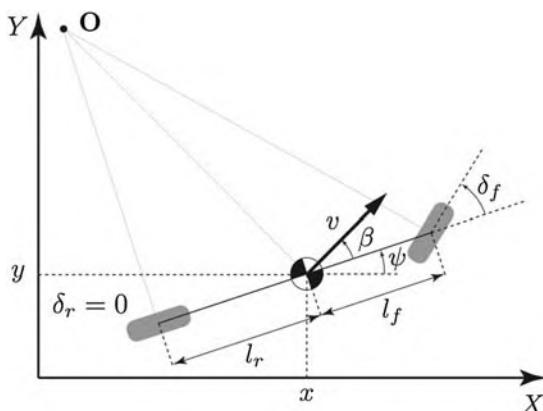


Figure 9.3 Simple kinematic bicycle model.

β can be calculated by the following formula:

$$\beta = \tan^{-1} \left(\frac{l_r}{l_f + l_r} \tan(\delta_f) \right)$$

As most of the vehicle's rear wheels cannot be deflected, the bicycle model assumes that the angle control input of the rear wheel δ_r equals 0. Thus, the control input on the steering wheel is reflected on the angle of the front wheel.

Given a certain time of control input, by calculating the state information (coordinates, yaw angle, and velocity) of the vehicle after dt time, this simple kinematic bicycle model can be adopted as a vehicle model in MPC.

9.1.3 Dynamic bicycle model

The kinematic bicycle model actually implies an important assumption: the direction of the front wheel of the vehicle is the direction of the current velocity. However, in a high-speed vehicle movement process, the direction of the wheels may not necessarily be the current speed direction of the vehicle. Given this deficiency of the kinematic model, the dynamic bicycle model of the vehicle needs to be introduced.

The vehicle dynamic model uses the complex interaction between the wheel tires and the road surface to describe the movement of the vehicle. In this model, the key is to consider the various forces roughly divided into two categories: longitudinal force and lateral force. Longitudinal force makes the vehicle move forward and backward, while lateral force promotes the vehicle to move laterally. In the process of force interaction, wheel tires play a decisive role (according to the laws of physics, tires are an important source of force for vehicle movement).

As shown in Fig. 9.3, a simple dynamic vehicle model is described by the state quantities $(\dot{x}, \dot{y}, \dot{\psi}, X, Y)$, where \dot{x} and \dot{y} , respectively, denote the longitudinal and lateral velocity of the vehicle body. Then, $\dot{\psi}$ is the yaw angular velocity, and (X, Y) is the current coordinate of the vehicle.

The differential equations of the state quantities are as follows:

$$\begin{aligned}\ddot{x} &= \dot{\psi} \dot{y} + a_x \\ \ddot{y} &= -\dot{\psi} \dot{x} + \frac{2}{m} (F_{c,f} \cos \delta_f + f_{c,r})\end{aligned}$$

$$\ddot{\psi} = \frac{2}{I_z} (l_f F_{c,f} - l_r F_{c,r})$$

$$\dot{X} = \dot{x} \cos \psi - \dot{y} \sin \psi$$

$$\dot{Y} = \dot{x} \sin \psi + \dot{y} \cos \psi$$

m and I_z , respectively, describe the quality and yaw inertia of the vehicle. $F_{c,f}$ and $F_{c,r}$, respectively, describe the lateral force received by the front and rear wheel tires computed by a certain tire model. In a simple linear tire model, $F_{c,i}$, ($i=f$ or r) is expressed as

$$F_{c,i} = -C_{\alpha_i} \alpha_i$$

In the above formula, α_i represents the deflection angle of the wheel tire and is referred to as the angle between the current direction of the wheel tire and the current instantaneous velocity. C_{α_i} is usually called the tire cornering stiffness.

9.2 Rudiments of autonomous vehicle control

9.2.1 Need for control theory

Imagine the scene described in Fig. 9.4. When you drive a car through this curve, how would you operate the vehicle?

Unless the driver is a professional racer, one cannot achieve one-step control to drive smoothly through this curve. The deviation of the vehicle trajectory relative to the lane should be carefully observed while adjusting the angle of the steering wheel and the strength of the accelerator pedal. This method based on environmental interaction is called feedback



Figure 9.4 Real scene of a road curve.

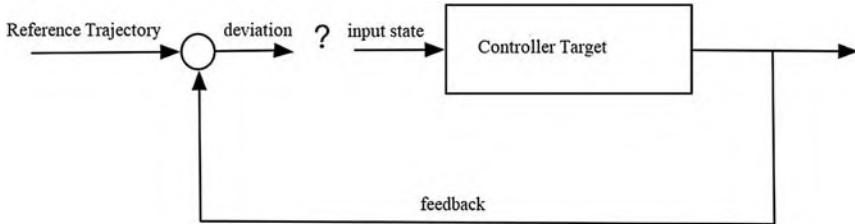


Figure 9.5 Schematic of the feedback control algorithm.

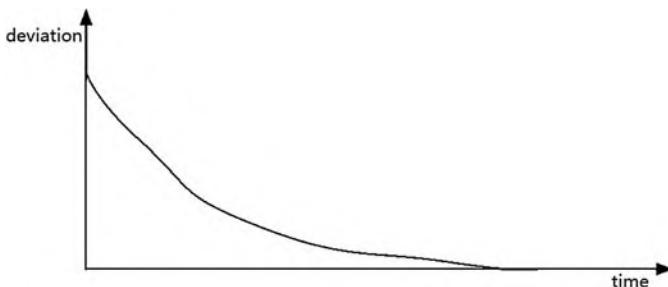


Figure 9.6 Line chart to reflect the relationship between deviation and time.

control, which is the basis of modern control theory. The concept of feedback control is shown in Fig. 9.5.

The purpose of vehicle control is to enable an unmanned vehicle to follow the planned trajectory. The control process first obtains the deviation relative to the current step's reference state with the current feedback of the environment. Then, a certain algorithm that can continuously generate the next step state value is designed to minimize the deviation. How can control states be generated on the basis of this deviation? The most direct way is to make the deviation gradually become relatively small until it is equal to zero, as shown in Fig. 9.6.

Zero deviation means that the vehicle would always drive on the planned trajectory. The approach to reducing the deviation is introduced. To understand feedback control, we need to fully comprehend the most widely used control theory, that is, PID control, and then leverage it to discuss control theory.

9.2.2 PID control algorithm

The name of the PID control algorithm refers to proportion, integral, and derivative. The three terms indicate how to use deviation to generate a control state. The whole process is shown in Fig. 9.7.

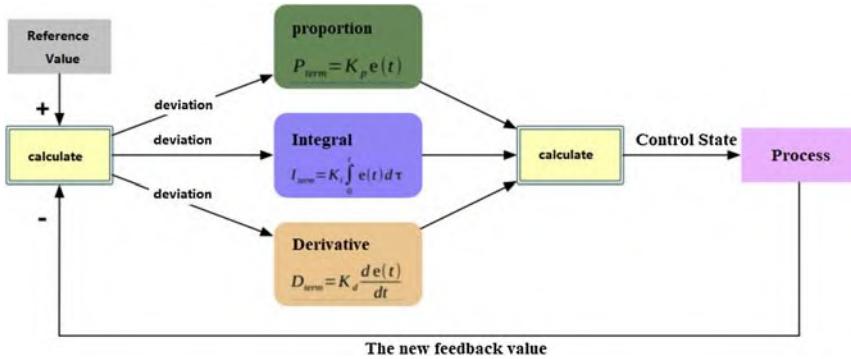


Figure 9.7 Flow chart of PID algorithm.

Based on the feedback and reference value, the first step is to compute the deviation using various metrics according to specific situations. If the PID algorithm used to control the vehicle to follow a certain trajectory, then the metric will be the distance between the current position of the vehicle and the reference path. If it is used to control the speed of the vehicle, this metric will be the difference between the current speed and the set speed. In the second step, these three terms (proportional, integral, and derivative) are calculated according to the deviation to obtain their coefficients, respectively named as K_p , K_i , and K_d , which impact the specific gravity of the final output. The final step for the PID algorithm is to take the sum of P , I , D as the final output state. The following sections discuss the meaning of these three items separately.

9.2.2.1 Proportional control

Given a simple situation, assume that an unmanned vehicle needs to follow the green line. However, the actual vehicle position is in the position shown in Fig. 9.8.

If the control state is the turning angle of the vehicle and the turning value is a fixed angle, then the trajectory of the vehicle will be that shown in Fig. 9.9.



Figure 9.8 Relative position of the vehicle and the *green line*.

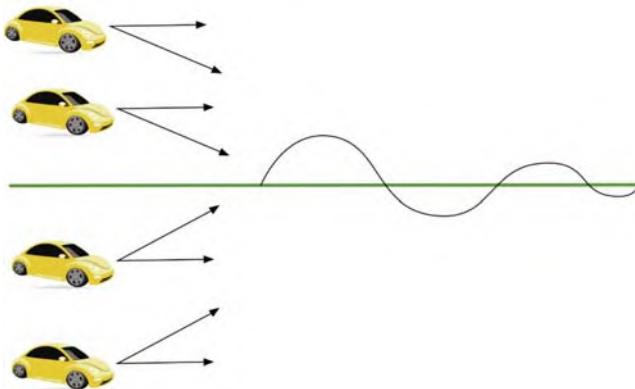


Figure 9.9 Vehicle adopting a fixed angle to turn.

Obviously, fixed and equivalent control values cause nonconvergence oscillations. An intuitive solution is to use proportional control. As shown in Fig. 9.10, when the deviation is large, the deflection is larger; when the deviation is small, the deflection is smaller.

This is a proportional controller, usually called a P controller. Here the cross track error (CTE) is signed as the deviation metric, which is the distance from the vehicle to the reference line. Thus, the corner becomes

$$\text{steeringangle} = K_p \cdot e(t)$$

$e(t)$ is the metric CTE of the vehicle at time t . In the P controller, the coefficient K_p directly affects the final control effect. Within a reasonable value range, the larger K_p is, the better the control effect will be, and the faster it will return to the reference line. However, when the position of the vehicle is far away from the reference line and the K_p coefficient is large, then the vehicle will lose control, as shown in Fig. 9.11.

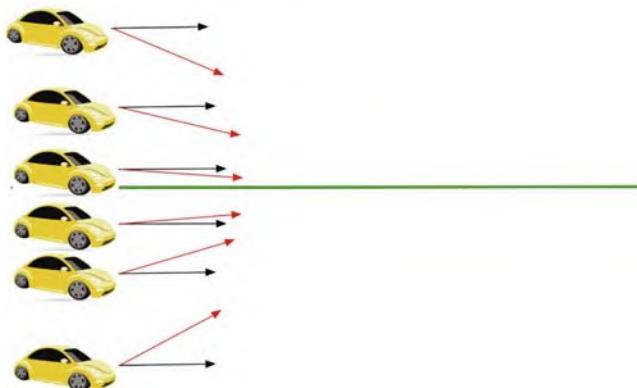


Figure 9.10 Vehicle adopting P controller.

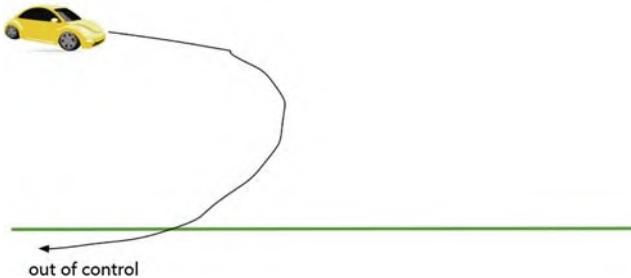


Figure 9.11 Vehicle loses control due to oversized K_p



Figure 9.12 Scenario in which the vehicle is on the reference line.

Therefore, if the parameter K_p is reasonable, then the P controller is better than the fixed equivalent control. However, as the P controller has its drawbacks that are affected by zero value, the simple P controller cannot meet complicated control requirements, as shown in Fig. 9.12.

Fig. 9.12 shows that although at the current moment, the vehicle is on the reference line, it is not the expected state because it will deviate in the next moment. However, for the P controller, this state is the ideal one in which the control angle is zero at this moment. Therefore, the vehicle will overshoot the line repeatedly. To correct this drawback, an additional error term will be introduced.

9.2.2.2 Proportional and derivative control

The change rate of the CTE actually characterizes how fast the unmanned vehicle moves toward the reference line. Suppose that the unmanned vehicle always maintains perfect movement on the reference line. Thus, the change rate of the CTE will be zero. Therefore, the term (description of the change rate of deviation) can be expressed as a derivative residual, and the control formula is the sum of the proportional term and the derivative term.

$$\text{steering angle} = K_p \cdot e(t) + K_d \frac{de(t)}{dt}$$

K_p is the coefficient of the derivative term, which determines the influence of the change rate of the CTE on the feedback control. This formula is named

the PD controller, which has two coefficients K_p and K_d that can be adjusted manually. Enlarging K_p will increase the tendency of the unmanned vehicle to move toward the reference line while enlarging K_d will increase the “resistance” of the unmanned vehicle’s rapidly changing corners, thereby making the steering increasingly gentle. A PD controller vehicle system that keeps the oversized K_p and undersized K_d coefficients is underdamped because the vehicle’s behavior will oscillate along the reference line. Meanwhile, the system that keeps the undersized K_p and oversized K_d is overdamped, and thus, correcting its deviation will take a long time.

Although the PD controller can ensure normal control requirements, disturbances in the real environment cause the P term in the controller to move toward the reference line. Meanwhile, these disturbances may be offset by the D term because of the slight disturbances of the force on the vehicle. This tendency that makes the unmanned vehicle always unable to move along the reference line is called steady-state error, as shown in Fig. 9.13.

To solve this deflection, one more term, that is, the integral term, is introduced.

Proportional, Integral, and Derivative Control

Adding the integral term to the PD controller, the corner output of the unmanned vehicle can be expressed as

$$\text{steering angle} = K_p \cdot e(t) + K_d \frac{de(t)}{dt} + K_i \int_0^t e(t) dt$$

K_i is the coefficient of the integral term that essentially represents the area of the graph from the actual route of the vehicle to the reference line. After adding the integral term, the controller makes the integral of the vehicle route as small as possible (making the area between the vehicle route and the actual motion reference line as small as possible) so as to avoid the steady-state error.

Similarly, the size of K_i will also affect the stability of the entire control system. Oversized K_i will cause the system to “oscillate,” while undersized K_i will cause the controlled vehicle to encounter disturbances (in steady-state error) and spend extra time returning to the reference line; in some cases, the vehicle may find itself in a dangerous situation.



Figure 9.13 Scenario of environment disturbance.

The PID controller is determined by the above three terms. Thus, the realization of the PID controller is not complicated, but the parameter choices of K_p , K_d , and K_i are complex and are usually determined by the actual performance of the control system in practice. The basic PID is implemented in C++.

Code list 9.2.1 Basic PID algorithm implemented by C++

```
#c++

#include <limits>

#include <iostream>

#include "PID.h"

//using namespace std;

PID::PID() {}

PID::~PID() {}


void PID::Init(double Kp, double Ki, double Kd) {
    parameter.push_back(Kp);
    parameter.push_back(Ki);
    parameter.push_back(Kd);

    this->p_error = 99999999.;

    this->d_error = 0.0;
    this->i_error = 0.0;

    //twiddle parameters
    need_twiddle = false;

    step = 1;

    // let the car run at first 100 steps, then in the next 3000 steps
    add the cte^2 to the total_error

    val_step = 100;
    test_step = 2000;
```

```
for (int i = 0; i < 3; ++i) {  
  
    // init the change rate with the value of 0.1*parameter  
  
    changes.push_back(0.1 * parameter[i]);  
  
}  
  
index_param = 0;  
  
  
best_error = std::numeric_limits<double>::max();  
  
total_error = 0;  
  
// fail to make the total_error better times  
  
fail_counter = 0;  
  
}  
  
  
void PID::UpdateError(double cte) {  
  
    if(step == 1){  
  
        p_error = cte;  
  
    }  
  
    d_error = cte - p_error;  
  
    p_error = cte;  
  
    i_error += cte;  
  
  
    if(need_twiddle){  
  
        if(step % (val_step + test_step) > val_step){  
  
            total_error += (cte * cte);  
  
        }  
    }  
}
```

```
if(step % (val_step + test_step) == 0){

    std::cout<<"======"><step<<""
    ====="><std::endl;

    std::cout<< "P: "<< parameter[0]<<" I: "<<parameter[1]<<" D:
"<<parameter[2]<<std::endl;

    if (step == (val_step + test_step)) {

        if(total_error<best_error){

            best_error = total_error;

        }

        parameter[index_param] += changes[index_param];

    } else{

        if(total_error<best_error){

            best_error = total_error;

            changes[index_param] *= 1.1;

        IndexMove();

        parameter[index_param] += changes[index_param];

        fail_counter = 0;

    } else if(fail_counter == 0){

        parameter[index_param] -= (2*changes[index_param]);

        fail_counter++;

    } else{

        parameter[index_param] += changes[index_param];

        changes[index_param] *= 0.9;

    IndexMove();

    parameter[index_param] += changes[index_param];
```

```
fail_counter = 0;

    }

}

std::cout<< "best_error: "<<best_error<<" total_error:
"<<total_error<<std::endl;

std::cout<< "change_index: "<<index_param<<" new_parameter:
"<<parameter[index_param]<<std::endl;

std::cout<<std::endl;

total_error = 0;

    }

}

step++;

}

double PID::TotalError() {

    return -parameter[0] * p_error - parameter[1] * i_error - parameter[2]
* d_error;

}

void PID::IndexMove() {

    index_param++;

    if(index_param>=3){

        index_param = 0;

    }

}
```

Code list 9.2.2 pid.h

```
#c++

#ifndef PID_H
#define PID_H


#include <cmath>
#include <vector>

class PID {
private:
    int step;
    std::vector<double> changes;
    double best_error;
    double total_error;
    int index_param;
    int val_step;
    int test_step;
    int fail_counter;
    void IndexMove();
    bool need_twiddle;
}
```

```
public:

    /*
     * Errors
     */
    double p_error;
    double i_error;
    double d_error;

    /*
     * Coefficients, the order is P, I, D
     */
    std::vector<double> parameter;

    /*
     * Constructor
     */
    PID();

    /*
     * Destructor.
     */
    virtual ~PID();

    /*
     * Initialize PID.
     */
    void Init(double Kp, double Ki, double Kd);
```

```
/*
 * Update the PID error variables given cross track error.
 */

void UpdateError(double cte);

/* 
 * Calculate the total PID error.
 */

double TotalError();

};

#endif /* PID_H */
```

Usage guide.

In the control loop code, call the PID package.

Code list 9.2.3 Call method

```
#c++

PID pid;

pid.Init(0.3345, 0.0011011, 2.662); //your init parameters


for (in your control loop) {

    pid.UpdateError(cte);

    steer_value = pid.TotalError();

})
```

9.3 MPC based on kinematic model

In the previous section, two common vehicle models, namely, the kinematic bicycle model and the dynamic bicycle model, were introduced. As the traditional PID controller, which is widely used because of its simple and easy implementation but is not suitable in lateral vehicle control (steering control) due to real-time delay, this section introduces a new control theory called MPC based on the kinematic vehicle model.² In lateral control, the optimal control method is MPC, a theory that pursues optimal control in a short time interval by taking the control delay in the vehicle model to avoid the delay drawback of the PID controller.

9.3.1 Applying PID controller forwards to steering control

As a kind of feedback control method, PID is widely popular because of its simplicity and ease of implementation. In the scenario of actual vehicle control, the PID controller computes the control state determined by the current state of the vehicle. However, the vehicle often cannot execute the control command immediately. Hence, there is a certain delay, and the control state will be executed in the “next vehicle state.” This condition may cause the vehicle in the “future state” to face danger in some fields, for example, generating the brake order state. This is the biggest dilemma of using the PID algorithm in autonomous steering control.

MPC is a theory that pursues optimal solutions by decomposing and transforming an optimal control problem with a longer time span or even infinite time into a series of optimal control problems with a shorter time span or the largest time span; it is composed of the following three elements²:

- Predictive model: predicts the change of the system state in a short period
- Online optimal loop based on time series: minimizes the output of the predictive model and reference value
- Feedback correction: predicts and optimizes according to the new state at the next moment.

The MPC theory will be discussed from these three aspects.

9.3.2 Predictive model

The configuration of the kinematic bicycle model as a predictive model is shown in Fig. 9.14.

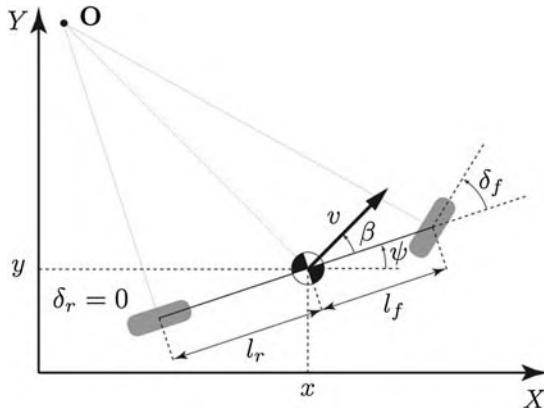


Figure 9.14 Kinematic bicycle model.

The update formulas of the state variables are as follows:

$$\begin{aligned}x_{t+1} &= x_t + v_t \cos(\psi_t + \beta) \times dt \\y_{t+1} &= y_t + v_t \sin(\psi_t + \beta) \times dt \\\psi_{t+1} &= \psi_t + \frac{v_t}{l_r} \sin(\beta) \times dt \\v_{t+1} &= v_t + a \times dt\end{aligned}$$

β can be calculated by the following formula:

$$\beta = \tan^{-1} \left(\frac{l_r}{l_f + l_r} \tan(\delta_f) \right)$$

On the basis of the above formula, given a control state, the predictive model can calculate the state of the vehicle (x, y, ψ, v) after dt time according to the kinematic bicycle model. This state is also the theoretically possible state of the vehicle as the kinematic bicycle model itself is built on a series of assumptions.

As shown in Fig. 9.15, an S-shaped curve road includes a dotted line serving as the reference line of control. The goal of the controller is to make the vehicle drive along the reference line as much as possible.

Assuming $dt = 0.05s$ and calculating the state of the vehicle in the next 0.5 s, according to the prediction model, based on the premise of a set of control inputs, as shown in the red dot in Fig. 9.16.

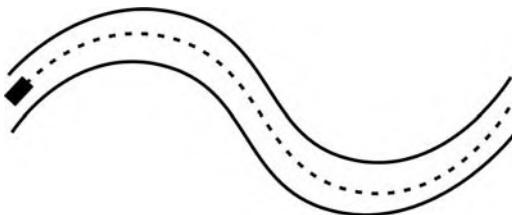


Figure 9.15 15S-shaped curve road including the dotted line.

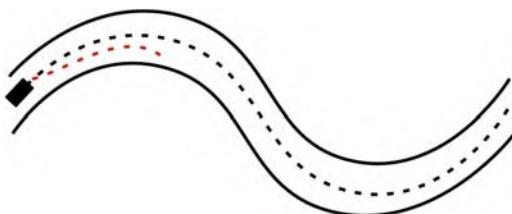


Figure 9.16 State of the vehicle in the next 0.5 s.

9.3.3 Online optimal loop based on time series

At this point, the predict problem becomes an optimization problem. With reference to the experience in neural network optimization, the important step is to define the loss function. In this example, the loss function is the CTE between the trajectory predicted by the model and the reference line. The goal of optimization is to minimize a set of control variables, namely, (a, δ_f) , which represent the accelerator brake coefficient and steering wheel angle, respectively.

$$\text{Loss} = \text{CTE} = \sum_{i=1}^{10} (z_i - z_{\text{ref},i})^2$$

The prediction model in the next 10 time intervals is considered so that the value range of $i \in (1, 10)$ and $z_i - z_{\text{ref},i}$ represent the distance from the predicted point to the actual reference line. Adding more items to the loss function can improve the predictive control, for example, if the goal is not only to optimize the vehicle to follow the reference line but also to optimize the velocity of the vehicle at each point in a certain short time range, then the loss function adds a term of the squared difference with velocity.

$$\text{Loss} = \sum_{i=1}^{10} [(z_i - z_{\text{ref},i})^2 + (v_i - v_{\text{ref},i})^2]$$

Furthermore, some items can be added to the loss function to make the control smooth. If the goal is for the throttle coefficient not to change abruptly (slowly stepping on the accelerator and brake), the square of the difference between the front and rear throttle coefficients may be added as an item in the loss function.

$$\text{Loss} = \sum_{i=1}^{10} [(z_i - z_{\text{ref},i})^2 + (\nu_i - \nu_{\text{ref},i})^2 + (a_{i+1} - a_i)^2]$$

Similarly, the more perfect the loss function is, and the more reasonable the loss function is designed, the better the output of the MPC will be able to meet the requirements of riding comfort. In addition to the loss function, the variables in the optimization problem, such as the value range of the front vehicle wheel angle δ_f and the value range of the vehicle's accelerator a ($a \in (-1, 1)$, -1 means full brake while one means full throttle), are still constrained. The braking delay problem encountered in PID control can also be solved by adding constraints. In this section, the predictive model uses 10 time intervals (each time interval is 50 ms) to predict the motion and assume that the braking delay of the vehicle is 100 ms. Hence, the first two steps are actually within the braking delay time. In fact, the vehicle is still executing the braking command in the previous state in these two steps. To make the model closer to reality, the braking command of these two steps is constrained, that is, (a, δ_f) , as the instruction of the previous state so that the MPC takes the braking delay into consideration.

9.3.4 Feedback correction

Fig. 9.17 shows a simplified schematic of MPC, essentially a feedback control, which can obtain a series of control states through optimization methods (in this case, it is the control output of the next 10 steps). The vehicle executes the control states and feeds back the vehicle state z_t of this current moment, which can also be published to the planning module and

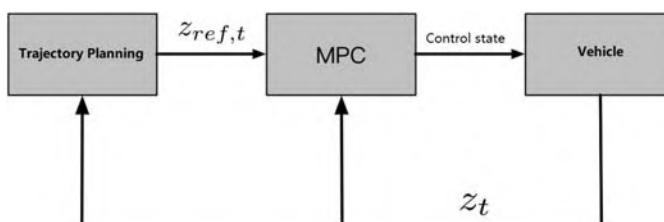


Figure 9.17 Simplified diagram of MPC.

MPC controller module. The path planning module makes a new plan on the basis of the z_t state combined with the information of the perception module and the map information. MPC then performs a new round of predictive control on the basis of the new reference trajectory and the z_t state of the vehicle. We should note that the feedback of the real state of the vehicle is not fed back after the predicted time period is executed. The feedback time interval is often less than a predicted time period (in this case, the predicted time period is $0.05 \times 10 = 0.5$ s).

In summary, by combining different predictive models with loss functions, various model predictive controllers can be constructed. The structure of the model predictive controller, in general, is fixed and can be decomposed into the following steps:

- (a) Predict the next states of t steps, starting from t moment.
- (b) Construct a loss function on the basis of the controller target of the model and optimize the loss function by adjusting the control parameters.
- (c) Input the control state to the system.
- (d) Repeat step 3 in the next period.

MPC has its natural advantages in multimodel constraint processing and can be combined well with sensor data preprocessing algorithms for perception and planning processes. It is also an ideal method to reflect vehicle kinematic and dynamic constraints in the process of autonomous driving control. However, in the actual development process, the complexity of a vehicle model is affected by the degree of freedom of the vehicle. When a large number of degrees of freedom are involved, the vehicle model becomes very complicated. In recent years, methods based on neural networks and deep learning have been applied to the field of control. They offer certain significance in processing high-level degrees of freedom because their ability to automatically learn state features makes them useful in autonomous driving systems.

9.4 Trajectory tracking

For autonomous driving systems, the trajectory computed by the planning module usually consists of a series of waypoints called global waypoints, which contain spatial position information, attitude information, velocity, and acceleration, etc. The difference between global waypoints and trajectory is that the trajectory is also a kind of path point containing time information through the addition of time constraints to the path point.

These trajectory points containing time information are usually called local waypoints. Fig. 9.18 shows the sequence of global path points in the map.

The current popular methods to track the trajectory for an unmanned vehicle are divided into two categories: methods based on geometric tracking and methods based on model prediction. In this section, a simple and widely used method based on geometric tracking called Pure Pursuit is introduced.

Before discussing the Pure Pursuit tracking algorithm, we review the geometric bicycle model in Fig. 9.19.

The geometric bicycle model as a simplification of Ackerman's steering geometry simplifies a four-wheeled vehicle into a two-wheeled model and

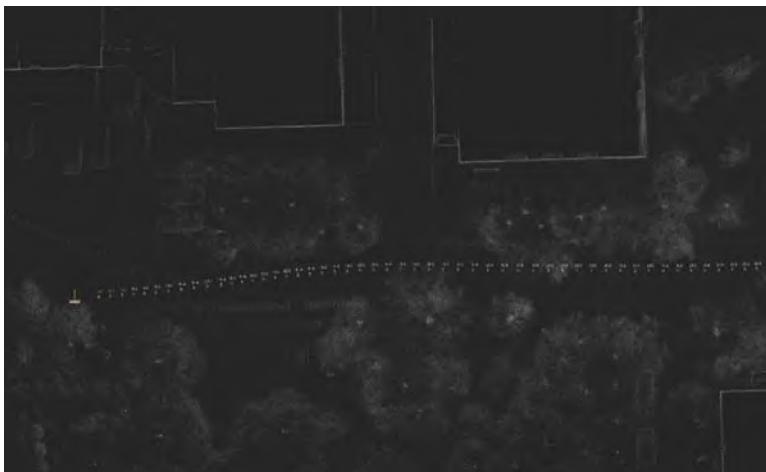


Figure 9.18 Global waypoints in the point cloud.

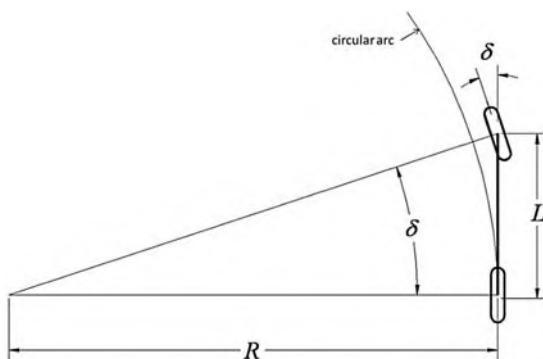


Figure 9.19 Geometric bicycle model.

assumes that the vehicle only travels on a flat surface. The advantage of using the bicycle model is that it simplifies the geometric relationship between the steering angle of the front wheel and the rear wheelbase.

$$\tan(\delta) = \frac{L}{R}$$

δ represents the turning angle of the front wheel, L is the rear wheelbase, and R is the radius of the circle followed by the rear wheelbase under a given turning angle. This formula can estimate vehicle motion in lower speed scenarios.

The Pure Pursuit tracking algorithm takes the rear wheelbase as the tangent point and the longitudinal body of the vehicle as the tangent based on the geometric bicycle model. By controlling the front wheel angle δ , the vehicle can drive along a circular arc passing the goal point, as shown in Fig. 9.20.

(g_x, g_y) is the next waypoint to be tracked in the above figure, and it is located in the planned global waypoints. The target is to control the vehicle, especially the rear wheelbase, to pass this waypoint. l_d represents the distance between the current position of the vehicle (the position at the center of the rear wheelbase) and the target waypoint (g_x, g_y) . α represents the angle between the current vehicle pose and the target waypoint. Thus,

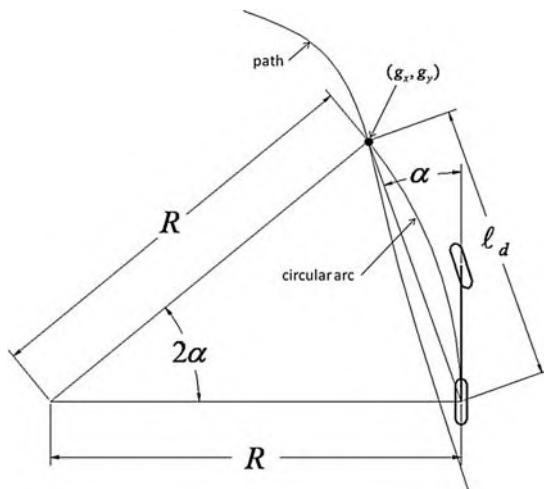


Figure 9.20 Pure Pursuit tracking based on the geometric bicycle model.

the following conversion formula can be derived according to the law of sine:

$$\begin{aligned}\frac{l_d}{\sin(2\alpha)} &= \frac{R}{\sin\left(\frac{\pi}{2} - \alpha\right)} \\ \frac{l_d}{2 \sin \alpha \cos \alpha} &= \frac{R}{\cos \alpha} \\ \frac{l_d}{\sin \alpha} &= 2R\end{aligned}$$

The formula can also be represented as

$$\kappa = \frac{2 \sin \alpha}{l_d}$$

where κ is the calculated curvature of the arc so that the front wheel's turning angle δ expresses

$$\delta = \tan^{-1}(\kappa L)$$

Combining the above two formulas, we can obtain the final expression of δ for the Pure Pursuit tracking algorithm.

$$\delta(t) = \tan^{-1}\left(\frac{2L \sin(\alpha(t))}{l_d}\right)$$

If the time term is considered, under the premise of knowing the angle $\alpha(t)$ between the vehicle and the target waypoint and the forward-looking distance l_d from the target waypoint at time t , the above formula can be used to estimate the front wheel angle $\delta(t)$ as the wheelbase of the vehicle is fixed. To better understand the Pure Pursuit tracking algorithm, the variable e_l , defined as the error between the current attitude of the vehicle and the target waypoint in the lateral direction, can be obtained; the angle α sine is

$$\sin(\alpha) = \frac{e_l}{l_d}$$

Thus, arc κ can be rewritten as

$$\kappa = \frac{2}{l_d^2} e_l$$

As e_l is essentially a horizontal CTE, the Pure Pursuit tracking algorithm is actually a P controller with a horizontal rotation angle, and its $K_p = \frac{2}{l_d^2}$.

As this P controller is greatly affected by the parameter l_d (the forward-looking distance), the key is to adjust the parameter l_d in the Pure Pursuit tracking algorithm. Generally, l_d , considered as a function of vehicle velocity, can have different values on the basis of different vehicle velocities.

One of the most common ways to adjust the parameter l_d is to express the forward-looking distance as a linear function of the vehicle's longitudinal velocity, that is, $l_d = kv_x$. Then, the front wheel angle formula becomes

$$\delta(t) = \tan^{-1} \left(\frac{2L \sin(\alpha(t))}{kv_x(t)} \right)$$

Then, the adjustment of the pure tracking algorithm becomes the adjustment coefficient k . The maximum and minimum forward-looking distances are typically used to constrain the forward-looking distance. The larger the forward-looking distance is, the smoother the tracking of the trajectory will be. The smaller the forward-looking distance is, the more accurate the tracking will be (of course, it will also bring about oscillation). The implementation of a simple Pure Pursuit tracking algorithm with Python is described.

This implementation includes pure tracking to control the steering angle and a simple P controller to control the velocity.

Define the parameter values as follows:

Code list 9.3.1 Definition

```
import numpy as np
import math
import matplotlib.pyplot as plt

k = 0.1 # Forward looking distance coefficient
Lfc = 2.0 # Forward looking distance
Kp = 1.0 # velocity coefficient in P controller
dt = 0.1 # time interval
L = 2.9 # wheelbase
```

The minimum forward-looking distance is set to 2, the coefficient k of the forward-looking distance on the vehicle velocity is set to 0.1, the proportional coefficient K_p of the velocity in the P controller is set to 1.0, the time interval is 0.1 s, and the wheelbase of the vehicle is set to 2.9 m.

The following defines the vehicle state class. In a simple bicycle model, only the current position (x, y) of the vehicle, the yaw angle of the vehicle, and the speed v of the vehicle are considered. To simulate the model on the software program, the state update function of the vehicle is defined to simulate the status update.

Code list 9.3.2 Vehicle state and update function

```
class VehicleState:

    def __init__(self, x=0.0, y=0.0, yaw=0.0, v=0.0):
        self.x = x
        self.y = y
        self.yaw = yaw
        self.v = v

    def update(state, a, delta):
        state.x = state.x + state.v * math.cos(state.yaw) * dt
        state.y = state.y + state.v * math.sin(state.yaw) * dt
        state.yaw = state.yaw + state.v / L * math.tan(delta) * dt
        state.v = state.v + a * dt
        return state
```

The P controller is used for longitudinal control, while the Pure Pursuit tracking controller is used for lateral control (corner control). The two controllers are defined as follows.

```
defPControl(target, current):  
  
    a = Kp * (target - current)  
  
    return a  
  
  
defpure_pursuit_control(state, cx, cy, pind):  
  
    ind = calc_target_index(state, cx, cy)  
  
    if pind>= ind:  
  
        ind = pind  
  
    if ind<len(cx):  
  
        tx = cx[ind]  
  
        ty = cy[ind]  
  
    else:  
  
        tx = cx[-1]  
  
        ty = cy[-1]  
  
    ind = len(cx) - 1  
  
    alpha = math.atan2(ty - state.y, tx - state.x) - state.yaw  
  
    if state.v< 0: # back  
  
        alpha = math.pi - alpha  
  
    Lf = k * state.v + Lfc  
  
    delta = math.atan2(2.0 * L * math.sin(alpha) / Lf, 1.0)  
  
    return delta, ind
```

The function defined to search for the nearest waypoint:

```
defcalc_target_index(state, cx, cy):
    # search for the nearest waypoint
    dx = [state.x - icx for icx in cx]
    dy = [state.y - icy for icy in cy]
    d = [abs(math.sqrt(idx ** 2 + idy ** 2)) for (idx, idy) in zip(dx, dy)]
    ind = d.index(min(d))

    L = 0.0
    Lf = k * state.v + Lfc
    while Lf > L and (ind + 1) < len(cx):
        dx = cx[ind + 1] - cx[ind]
        dy = cx[ind + 1] - cx[ind]
        L += math.sqrt(dx ** 2 + dy ** 2)
        ind += 1
    return ind
```

Main function:

```
defmain():
    # target position
    cx = np.arange(0, 50, 1)
    cy = [math.sin(ix / 5.0) * ix / 2.0 for ix in cx]
    target_speed = 10.0 / 3.6 # [m/s]
    T = 100.0 # time peroid
    # vehicle state
    state = VehicleState(x=-0.0, y=-3.0, yaw=0.0, v=0.0)
    lastIndex = len(cx) - 1
```

```
time = 0.0

x = [state.x]

y = [state.y]

yaw = [state.yaw]

v = [state.v]

t = [0.0]

target_ind = calc_target_index(state, cx, cy)

while T >= time and lastIndex>target_ind:

    ai = PControl(target_speed, state.v)

    di, target_ind = pure_pursuit_control(state, cx, cy, target_ind)

    state = update(state, ai, di)

    time = time + dt

x.append(state.x)

y.append(state.y)

yaw.append(state.yaw)

v.append(state.v)

t.append(time)

plt.cla()

plt.plot(cx, cy, ".r", label="course")

plt.plot(x, y, "-b", label="trajectory")

plt.plot(cx[target_ind], cy[target_ind], "go", label="target")

plt.axis("equal")

plt.grid(True)

plt.title("Speed[km/h]:" + str(state.v * 3.6)[:4])

plt.pause(0.001)

if __name__ == '__main__':

    main()
```

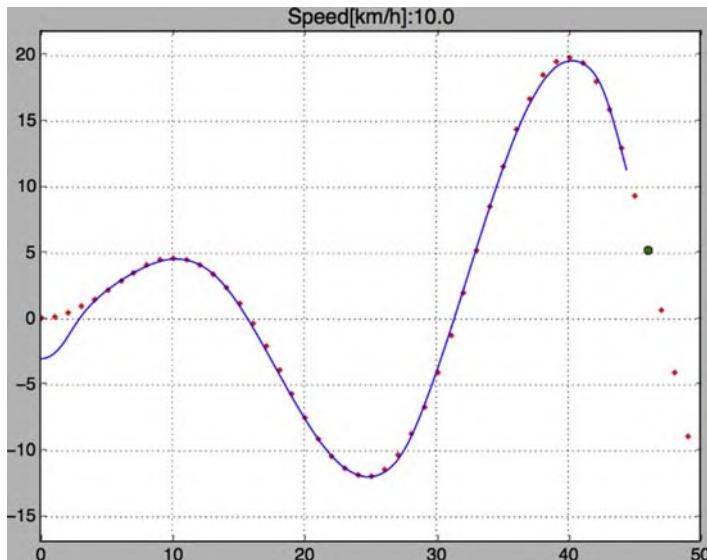


Figure 9.21 Tracking effect of Pure Pursuit controller.

The result is shown in Fig. 9.21.

In the figure, the red dot represents the planned waypoint, the blue line represents the actual trajectory of the vehicle, and the green dot in front represents the current forward-looking distance. As the minimum forward-looking distance is set to 2 m, further optimization and comparison experiments may be performed. To set a larger forward-looking distance, for example, the pure tracking controller will behave more “smoothly.” This condition results in understeering at some sharp corners.

The trajectory tracking algorithm based on geometric relationships, represented by the Pure Pursuit tracking controller, is easy to understand and implement. The geometric method introduced in this section implements basic path tracking, but it will encounter bottlenecks when faced with a significant speed change. Although the Pure Pursuit tracking method uses the forward-looking distance to consider the path information, it is almost independent of the trajectory shape under scenarios with low velocity. This method is commonly used to express the forward-looking distance as a function of velocity because the method of selecting the best forward-looking distance is not clear. However, extra attention should be paid to the adjustment of the forward-looking distance of the Pure Pursuit tracking controller as it may also be a function of path curvature and

may even be related to CTEs other than longitudinal velocity. A short forward-looking distance will cause the vehicle control to be unstable or even shocked. In ensuring the stability of the vehicle, setting a long forward-looking distance will cause the vehicle to understeer as it turns at a large corner.

References

1. Kong J, Pfeiffer M, Schildbach G, et al. Kinematic and dynamic vehicle models for autonomous driving control design. In: *Intelligent Vehicles Symposium*. IEEE; 2015:1094–1099.
2. Xi W, Baras JS. MPC based motion control of car-like vehicle swarms. In: *Control and Automation, 2007. MED '07. Mediterranean Conference on*. IEEE; 2002:1–6.

This page intentionally left blank

CHAPTER 10

Deep reinforcement learning and application in self-driving

Jinqiang Wang and Rui Zhao

School of Information Science & Engineering, Lanzhou University, Lanzhou, Gansu, China

Contents

10.1 Overview of reinforcement learning	308
10.2 Reinforcement learning	309
10.2.1 Markov decision process	309
10.2.2 Constituent elements	310
10.2.2.1 <i>Policy</i>	310
10.2.2.2 <i>Reward</i>	311
10.2.3 Value function	311
10.3 Approximate value function	314
10.4 Deep Q network algorithm	315
10.4.1 <i>Q</i> learning algorithm	315
10.4.2 DQN algorithm	315
10.4.2.1 <i>Reward function</i>	316
10.4.2.2 <i>Objective function</i>	317
10.5 Policy gradient	319
10.6 Deep deterministic policy gradient and TORCS game control	319
10.6.1 About TORCS game	319
10.6.2 TORCS game environment installation	320
10.6.3 Deep deterministic strategy gradient algorithm	322
10.6.3.1 <i>Theory</i>	322
10.6.3.2 <i>Reward function setting</i>	323
10.6.3.3 <i>Running program</i>	324
10.7 Summary	325
References	325

As an important branch of machine learning, reinforcement learning (RL) is a method of gaining experience and learning a complete set of strategies through continuous interaction with the environment.¹ Since the launch of Google DeepMind, which uses neural networks to simulate RL

strategies and surpasses the level of human players in the Atari 2600 game,^{2,3} deep RL has developed rapidly and achieved a series of breakthroughs in the field of control. The autonomous driving system is a complex system integrating multiple interdisciplinary technologies. In recent years, the application of RL to autonomous driving has attracted considerable attention. By introducing the basic principles of RL and explaining the leveraging of neural networks based on the function approximation theory in this chapter, an unmanned vehicle control method, adopted in The Open Racing Car Simulator (TORCS) game that integrates a strategy gradient based on RL and a deep deterministic strategy gradient algorithm will be discussed.

10.1 Overview of reinforcement learning

Machine learning is usually divided into three categories, namely, supervised learning, unsupervised learning, and RL. Inspired by behavioral theory in psychology, RL is a subfield of machine learning used to determine how the agent gradually forms a habitual behavior that maximizes returns under the stimulation of the reward or punishment given by the environment. By learning a set of strategies according to the environment to maximize the expected reward, RL, which is characterized by universality, has been adopted in many fields, such as autonomous driving, game theory, cybernetics, simulation optimization, multiagent system learning, and genetic algorithms. The main differences between RL and other machine learning methods, such as supervised learning and unsupervised learning, are as follows:

- Compared with supervised learning, RL does not require correct input/output pairs or precise correction of suboptimal behavior. RL usually needs to balance exploration (unknown domain) and utilization (existing knowledge). The learning process of RL is a repeated practice of continuous interaction between the agent and the environment and continuous trial and error.
- RL is different from other machine learning methods in that there is no supervisor, only a delayed reward signal, and feedback is not generated immediately. Hence, timing is a crucial concept in RL.
- In general, RL does not require labeled data as a training environment and is a process that uses interactive training.

10.2 Reinforcement learning

Generally, RL consists of three parts, namely, environment, agent, and rewards.⁴ The composition and process of RL are shown in Fig. 10.1.

By obtaining state O_t from the environment at the beginning, the agent will exhibit behavior A_t according to its policy and feed it back to the environment. At the same time, the environment will give the agent the reward R_t based on the agent's feedback. Through continuous interactive learning between the agent and the environment, an interactive historical sequence consisting of three parts, namely, observation, behavior, and reward, that is, $\{O_t, A_t, R_t\}$, is obtained, which can be expressed as follows:

$$H_t = O_1, A_1, R_1, \dots, O_t, A_t, R_t.$$

Because the state is defined as the mapping of the $f(\cdot)$ function, that is, $S_t = f(H_t)$, the interaction sequence can be described using the Markov decision process (MDP).

10.2.1 Markov decision process

The essence of RL is a sequential decision-making process, which is similar to the MDP. Before explaining the MDP, two terms need to be defined, as follows:

Markov property: The next state S_{t+1} of the system is only related to the current state S_t , which can be expressed as follows:

$$P[S_{t+1}|S_t] = P[S_{t+1}|s_1, s_2, \dots, s_t].$$

Markov process: This process consists of a two-tuple (S, P) , where S is a finite state machine, and P is a state transition probability matrix, which can be expressed as follows:

$$P = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix}.$$

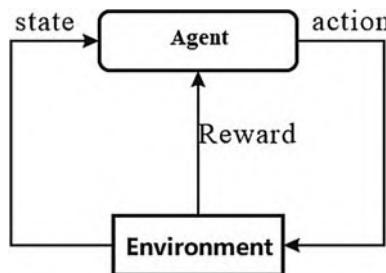


Figure 10.1 Schematic diagram of reinforcement learning.

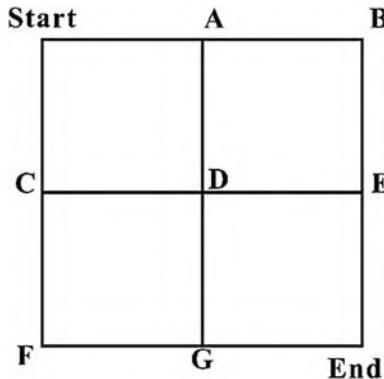


Figure 10.2 Markov sequence decision.

As shown in Fig. 10.2, the state transition probability matrix P is taken as an example. However, the actions and rewards in the Markov process are not defined. Multiple Markov chains (paths) from “Start” to “End” are considered the description of the Markov process. Therefore, as a sequential decision-making problem, the selection of a path from “Start” to “End” requires a constant to obtain maximum benefits.

Usually, the MDP is described by tuples $\langle S, A, P, R, \gamma \rangle$, where S is the finite state set, A is the limited action set, P is the probability of state transition, R is the payoff function, and γ is the discount factor used to calculate the cumulative return.

For the grid path selection problem, the key problem is how to choose the optimal route according to different state transition probabilities and rewards from the “Start” point to the “End” point, which will form different sequences, for example,

$$\{(Start, East, P, R1), (A, East, P, R2), (B, South, P, R3), \\ (E, South, P, R4), (End, South, P, R5)\}$$

Which path will bring the most benefit?

The answer is choosing a policy.

10.2.2 Constituent elements

10.2.2.1 Policy

A policy is usually defined as the mapping from state to action, that is, an action probability is specified in each state, and can be divided into stochastic and deterministic.

Stochastic policy: For the same state, the output state is not unique but satisfies a certain probability distribution, which can be expressed as follows:

$$\pi(a|s) = P[A_t = a | S_t = s].$$

Deterministic policy: Under the same state, the output action is deterministic, which can be expressed as follows:

$$a = \mu(s).$$

10.2.2.2 Reward

- Timely reward: This reward is given to the agent in real time. For example, when a toy helicopter makes a flight control gesture based on the current state, it will be immediately rewarded if it is good or bad.
- Cumulative expected reward: This is the expected total reward of a process, such as the entire process of a helicopter from flying to landing. In general, the cumulative expected reward can be expressed as follows:

$$G_t = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^{k-1} R_k = \sum_{k=0} \gamma^k R_{t+k+1}.$$

Under the policy π , starting from the previous “Start” state shown in Fig. 10.2, different paths lead to the same “End” state (illustrated as follows) but have a different cumulative return G_t .

$$Start \rightarrow A \rightarrow B \rightarrow E \rightarrow End$$

$$Start \rightarrow C \rightarrow D \rightarrow G \rightarrow End$$

Under the stochastic policy, although G_t is a random variable under different paths, it can be estimated using a value function to approximate the expectation at state S when different paths have a certain value.

10.2.3 Value function

When the agent (i.e., the interactive subject in RL) uses a certain policy π , the cumulative return follows the distribution where the expectation at state S is usually defined as a state-value function, which can be mathematically expressed as follows:

$$V_\pi(s) = E_\pi [R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^{k-1} R_k | S_t = s].$$

$\gamma \in [0, 1]$ is the discount factor, which is used to estimate the future impact on the current moment. $\gamma = 0$ is the short-sighted state that only considers

the current moment without paying attention to long-term returns. Generally, calculating the expected reward using the state—value function is insufficient. The expected reward brought about by taking action in a certain state, which is called the “state—behavior value function” used to determine whether the current behavior is good or bad, should also be considered and can be mathematically expressed as follows:

$$q_{\pi}(s, a) = E_{\pi} \left[\sum_{t=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right].$$

In general, the relationship between “state—value function” and “state—action value function” can be expressed as follows:

$$V_{\pi}(s) = \sum_{a \in A} \pi(a|s) q_{\pi}(s, a).$$

The specific relationship diagram is shown in Fig. 10.3.

To calculate the value function in a certain state S or the state—action value function of the action taken in the state S to evaluate the cumulative expected reward, the relationship between v_{π} and q_{π} can be derived from the graph, as follows:

$$q_{\pi}(s, a) = R_s^a + \gamma \sum_{a' \in A} P_{s'}^{a'} v_{\pi}(s'),$$

$$v_{\pi}(s') = \sum_{a' \in A} \pi(a'|s') q_{\pi}(s', a').$$

By substituting the previously presented formulas into each other, the following expressions can be obtained:

$$v_{\pi}(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi}(s') \right),$$

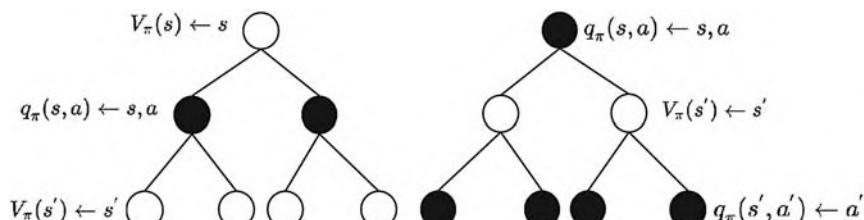


Figure 10.3 Relationship diagram.

$$q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_{\pi}(s', a').$$

By summarizing the previously presented formulas, the Bellman equations of the “state–value function” and “state–action value function” can be obtained, as follows:

$$\begin{aligned}\nu(s_t) &= E_{\pi}[R_{t+1} + \gamma \nu(s_{t+1})], \\ q(s_t, a_t) &= E_{\pi}[R_{t+1} + \gamma q(s_{t+1}, a_{t+1})].\end{aligned}$$

The purpose of calculating the value function is to construct learning to obtain the optimal policy from the data. Each policy corresponds to a value function, and the optimal policy corresponds to the optimal value function. If and only if $\pi \geq \pi^*$ and $V_{\pi}(s) \geq V^*(s)$, the optimal state–value function $V^*(s)$ is the largest value function among all policies and can be expressed as follows:

$$V^*(s) = \max_{\pi} \nu_{\pi}(s).$$

The optimal “state–action value function” is the largest state–action value among all policies and can be expressed as follows:

$$q^*(s, a) = \max_{\pi} q_{\pi}(s, a).$$

In the same manner, the optimal “state–value function” and “state–action value function” can be expressed as follows:

$$\begin{aligned}V_{\pi}(s) &= \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s'), \\ q_{\pi}(s, a) &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q^*(s', a').\end{aligned}$$

Finally, if the optimal “state–action value function” is known, then the optimal policy can be obtained directly through maximization, as follows:

$$\pi^*(a|s) = \begin{cases} 1, & a = \operatorname{argmax}_{\pi} q^*(s, a) \\ 0, & \text{otherwise} \end{cases}.$$

10.3 Approximate value function

At the beginning of the 20th century, a significant breakthrough in the theory of function approximation was made. Subsequently, many numerical methods of function approximation were proposed. “Approximation” literally means infinite approximation to the true value. Similarly, the approximation function is an infinite approximation to the true value function, that is, state–value function (aa) and state–behavior value function (bb), and can be mathematically expressed as follows:

$$\begin{aligned}\hat{V}(s, w) &\approx V_\pi(s), \\ \hat{Q}(s, a, w) &\approx Q_\pi(s, a).\end{aligned}$$

The purpose of function approximation is to determine whether w is a suitable parameter to approximate the value function. w can be a parametric or nonparametric approximation as a kernel function. According to the input of RL, the output is usually expressed as two approximation methods, as shown in Fig. 10.4.

In Fig. 10.4, the left panel shows the approximate value of the state according to the state itself, and the right panel shows the approximate value of the state–behavior according to the state–behavior itself. The two rectangular boxes can be regarded as black boxes, which can be considered a function fitter or can be imagined as anything, e.g., “neural network,” “decision tree,” “Taylor polynomial,” “linear function,” or everything that can or cannot be described. Because this chapter mainly discusses deep RL, the depth here mainly leverages deep neural networks as a function fitter to approximate the state.

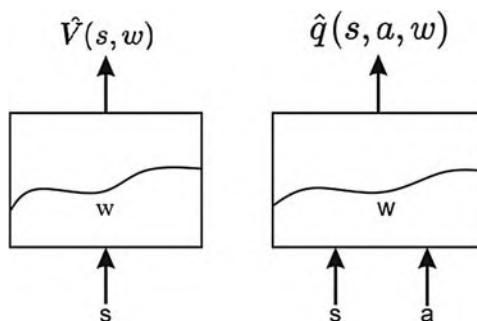


Figure 10.4 Approximate value function.

10.4 Deep Q network algorithm

Deep Q network (DQN) is a deep RL method established by Google DeepMind in 2013. DQN uses neural network approximation in RL to achieve results that surpass the level of human players in the Atari 2600 game.

10.4.1 Q learning algorithm

Q learning is a model-free RL technology proposed by Watkins in 1989. Q learning can handle random transitions and reward issues without adjustments. For any limited MDP, Q learning has been proven to eventually obtain the optimal policy. Starting from the current state, the expected value of the total return of all consecutive steps is the maximum that can be achieved. In the learning process, at each time t , the agent chooses an action a , gets a reward r , enters a new state S_{t+1} , and updates the Q value. The value function is iterated according to the following formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \left[r_t + \gamma \max_{\pi} Q(s_{t+1}, a_t) - Q(s_t, a_t) \right],$$

where α is the learning rate and γ is the discount factor. The algorithm executes the pseudocode as follows: first, select the behavior according to the policy (using the greedy policy); then, get a reward based on the interaction between the behavior and the environment.

```

Initialize Q(s, a), ∀s ∈ S, a ∈ A(s), arbitrarily, and Q(terminal-state)=0

Repeat(for each episode) :

    Initialize S

    Repeat(for each step of the episode) :

        Choose A and S using policy derived from Q(e.g.e-greedy)

        Take action A, observe R, A'

        Q(S,A) ← Q(S,A) + α[R + γ max_a(Q(S',A) − Q(S,A))]

        S ← S'

    Until S is terminal

```

10.4.2 DQN algorithm

The Atari 2600 game is a classic control game. For example, Atari brick-breaking is a high-dimensional state input (original image pixel input) and low-dimensional action output (discrete actions, e.g., up, down, left,

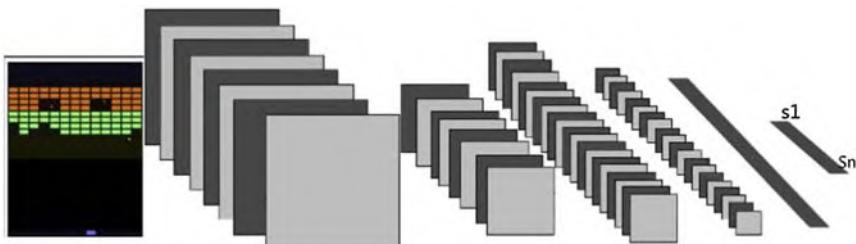


Figure 10.5 Convolutional neural network reads the Atari game screen structure diagram.

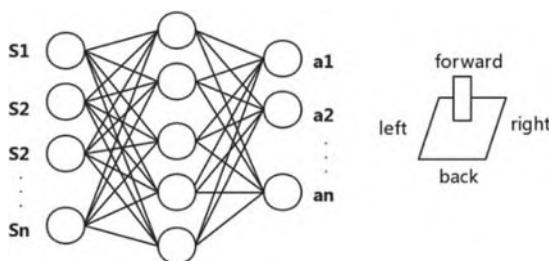


Figure 10.6 DQN algorithm neural network approximation.

and right, or firing cannonballs) game. Before the computer can process the image, the image needs to be read into the computer. Usually, the convolutional neural network (CNN) is used to read the image. The structure of the CNN is shown in Fig. 10.5.

First, the program will extract the Y channel representing the luminance from a frame of RGB image of the Atari game, resize it to 84×84 , take the continuous m frames of this image as input, and obtain n states after convolution pooling. Finally, K discrete actions will be the output after neural network approximation, as shown in Fig. 10.6.

10.4.2.1 Reward function

The reward is a crucial part of RL. Reasonable settings have a significant impact on the learning process and convergence speed. In the Atari game, at the current time t and state s , the score value change of the game after action a is taken as r_t , expressed as follows:

$$r_t = \begin{cases} 1, & \text{increase} \\ 0, & \text{no exchange} \\ -1 & \text{decrease} \end{cases}$$

The long-term cumulative discount reward can be expressed as follows:

$$R_t = \sum_{k=0}^T \gamma^k r_{t+k+1}.$$

10.4.2.2 Objective function

The core of DQN² is to identify the neural network parameters that can be utilized to approximate the value function. Currently, the gradient descent method is used to minimize the objective function to update the network weights continuously. The objective function of the DQN algorithm can be expressed as follows:

$$L_i(\theta_i) = E_{(s,a,r,s')} \sim U(D) \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right],$$

where θ_i^- is the target network parameter of the n -th iteration and θ_i is the Q network parameter. The gradient of the loss function is obtained using the gradient descent method, as follows:

$$\frac{\partial L_i(\theta_i)}{\partial \theta_i} = E_{(s,a,r,s')} \sim U(D) \left[\left(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right].$$

Moreover, the training data (quadruple) are stored in the replay memory during the learning process. As shown in Fig. 10.7, the replay memory stores and reads each learning process sequence in minimum batch size during the learning process, which can disrupt the correlation between the

$\langle s_1, a_1, r_2, s_2 \rangle$
$\langle s_2, a_2, r_3, s_3 \rangle$
$\langle s_3, a_3, r_4, s_4 \rangle$
$\langle s_4, a_4, r_5, s_5 \rangle$
$\langle s_5, a_5, r_6, s_6 \rangle$
\vdots

Figure 10.7 Experience replay buffer.

data. Thus, the data follow an independent and identical distribution when using the gradient descent method.

Note: Two important ideas, namely, experience replay and target network, are used in the DQN algorithm.

(a) Experience replay: The data obtained from the system's exploration environment are stored and randomly sampled to update the parameters of the deep neural network. As a supervised learning model, deep neural networks usually require data to be independently and identically distributed. However, the samples obtained by the learning sequence are related before and after. The experience replay buffer breaks the correlation between the data through the storage sampling method.

(b) Target network: During the learning process, the target Q value remains unchanged for a certain period, reducing the correlation between the current Q value and the target Q value to a certain extent to improve the stability of the algorithm. Another copy of the target network is used to generate the target Q value. Specifically, $Q(s, a; \theta_i)$ is the output of the current network MainNet, which is used to evaluate the value function of the current state-action pair, and $Q(s, a; \theta_i^-)$ is the output of the target network.

After satisfying the independent and identical distribution, the agent can learn from the replay memory using gradient descent and continuously update the network weights by approximating the optimal value function and policy. The pseudocode executed by the specific DQN algorithm is as follows:

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$

For episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1, T$ **do**

 With probability ε select a random action a_t

 otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

 Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

 Every C steps reset $\hat{Q} = Q$

10.5 Policy gradient

In the DQN algorithm, a neural network to approximate the features obtained after convolution is adopted. Then, the optimal policy is set to calculate the value function. For some discrete action Atari games, this can be a perfect solution. However, the DQN algorithm will learn slowly for some situations, such as continuously controlled robots or a large action space. Hence, scientists propose the use of a separate neural network to approximate the policy gradient. Usually, the objective function $J(\theta)$ is defined according to the expectations of the trajectory, as follows:

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)] = \int_{\tau \sim \pi_\theta(\tau)} \pi_\theta(\tau) r(\tau) d\tau.$$

By deriving the objective function, the following expression can be obtained:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_{\tau \sim \pi_\theta(\tau)} \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \\ &= E_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]. \end{aligned}$$

However, the objective function is difficult to compute in the process of calculating the gradient. The final gradient formula is obtained using the maximum likelihood and Monte Carlo methods, as follows:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^N \left[\sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_i, t | s_i, t) \left(\sum_{t=0}^T r(s_i, t, a_i, t) \right) \right].$$

At this point, the policy gradient can be calculated and updated, as follows:

$$\theta = \theta + \alpha \nabla_\theta J(\theta).$$

10.6 Deep deterministic policy gradient and TORCS game control

10.6.1 About TORCS game

TORCS is a popular open-source 3D racing simulation game on the Linux operating system written in C and C++ with 50 vehicles, 20 tracks, and simple visual effects. TORCS is released under the GPL agreement. Some interface controls and instructions are as follows:

No.	Name	Scope	Description
1	ob.angle	$(-\pi, +\pi)$	The angle between the direction of the car and the direction of the track axis
2	ob.track	(0.200) (m)	19 ranging sensors, each sensor returns the distance between the edge of the runway and the car within a range of 200 m
3	ob.trackPos	$(-\infty, +\infty)$	The distance between the car and the track axis, where greater than 1 or -1 means that the car is outside the track
4	ob.speedX	$(-\infty, +\infty)$ (km/h)	The speed of the car along the longitudinal axis of the car
5	ob.speedY	$(-\infty, +\infty)$ (km/h)	The speed of the car along the transverse axis
6	ob.speedZ	$(-\infty, +\infty)$ (km/h)	The speed of the car along the Z-axis
7	ob.wheelSpinVel	$(0, +\infty)$ (km/h)	The vector representing the four rotating sensors
8	ob.rpm	$(0, +\infty)$ (km/h)	Car engine revolutions per minute

10.6.2 TORCS game environment installation

This section of the experiment depends on the environment, that is, Ubuntu 16.04, Tensorflow 1.3, Python 2.7, and Keras 1.1. TORCS can be installed directly by executing the code block in the corresponding computer environment. The installation process is shown in Fig. 10.8.



Figure 10.8 Experience replay buffer.

```
sudo apt-get install xautomation  
  
// installing package numpy  
  
sudo pip install numpy  
  
// installing gym  
  
sudo pip install gym  
  
git clone https://github.com/ugo-nama-kun/gym\_torcs.git  
  
cd gym_torcs/vtorcs-RL-color/src/modules/simu/simuv2/  
  
sudo vim simu.cpp  
  
// annotation lines-64 and add the following codes  
  
//if(isnan((float)(car->ctrl->gear))  
  
||isinf((float)(car->ctrl->gear))) car->ctrl->gear = 0;  
  
cd gym_torcs/vtorcs-RL-color  
  
// install depending package  
  
sudo apt-get install libglib2.0-dev libgl1-mesa-dev libglu1-mesa-dev  
  
freeglut3-dev libplib-dev libopenal-dev libalut-dev libxi-dev  
  
libxmu-dev libxrender-dev libxrandr-dev libpng12-dev  
  
../conFig.  
  
// complie games  
  
make  
  
sudo make install  
  
sudo make datainstall  
  
// starting  
  
torcs
```

In this chapter, the TORCS game is adopted for the simulation experiment, and the deep deterministic policy gradient (DDPG) algorithm is used to simulate the racing car running along the track. Because the parameters in the game, such as throttle or rake, are all continuous variables, algorithms, such as DQN, are not particularly compatible in terms of the continuous policy. After David Silver proved the existence of deterministic policy gradients in 2015, he used deep neural networks to implement the

DDPG algorithm,⁵ which exhibited excellent performance in dealing with continuous and extremely large state spaces. The DDPG algorithm in this section leverages the actor–critic (AC) framework for learning and adopts the off-policy method, in which the actor uses a random policy to explore the environment, whereas the critic network uses a deterministic policy for function approximation.

10.6.3 Deep deterministic strategy gradient algorithm

10.6.3.1 Theory

By adopting two important thoughts, namely, the experience replay of the DQN algorithm and the target network, the DDPG algorithm successfully eliminates the relevance of RL arising from the data collection process. The AC framework is equivalent to a continuous interactive process, in which the actor dances on the stage and the critics in the audience comment on his/her performance, motivating the actor to constantly improve his/her performance. The off-policy method is adopted in the DDPG algorithm, in which the actor uses a random strategy to explore, whereas the critic uses a deterministic strategy. The overall structure is shown in Fig. 10.9.

The gradient of the actor’s random strategy can be expressed as follows:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a | \theta^Q) \Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s | \theta^{\mu}) \Big|_{s_i}$$

The critic’s update method can be expressed as follows:

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \theta^Q))^2.$$

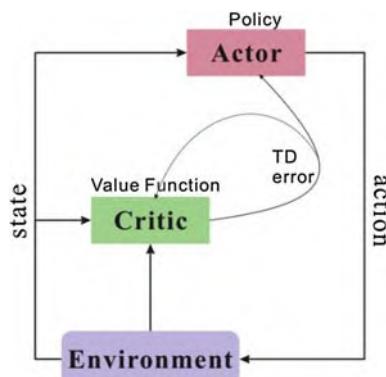


Figure 10.9 Structure diagram of the actor–critic framework.

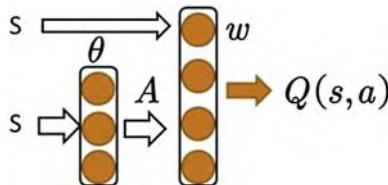


Figure 10.10 Structure diagram of the DDPG algorithm.

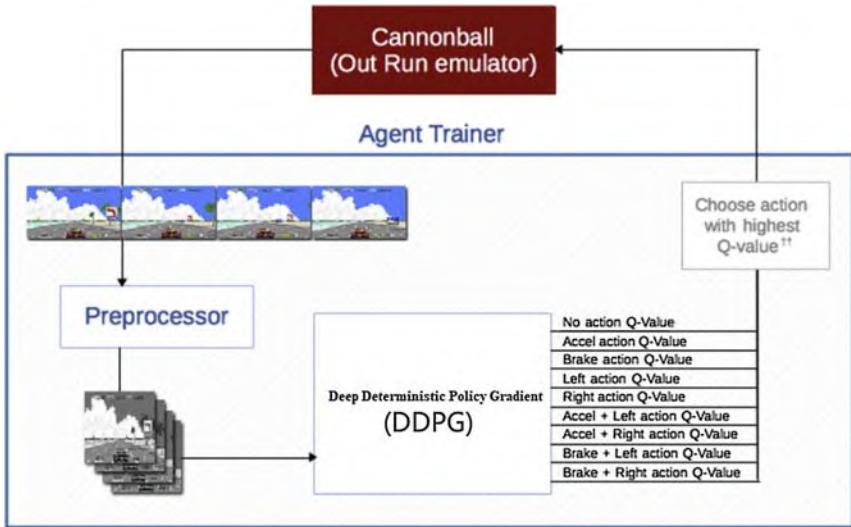


Figure 10.11 DDPG algorithm and TORCS game control structure diagram.⁶

The algorithm execution structure is shown in Fig. 10.10.

The algorithm execution structure is integrated with the game screen. The structure is shown in Fig. 10.11.

10.6.3.2 Reward function setting

Compared with the previous Atari game simulation environment, the reward function is used to determine the highest score of the game. The delayed reward signal is one of the most difficult problems in RL because it not only needs a large amount of exploratory work but also encounters a difficult reliability assignment problem. That is, for RL, rewards can be easily obtained in games with good settings because of a clear reward function. However, rewards in the real environment are often not achieved, regardless of the amount of data.

As shown in Fig. 10.12, the rewards in this section are set based on the speed and deflection angle of the car. The reward function can be expressed as follows:

$$R_t = V_x \cos(\theta) - V_x \sin(\theta) - V_x |\text{TrackPos}|.$$

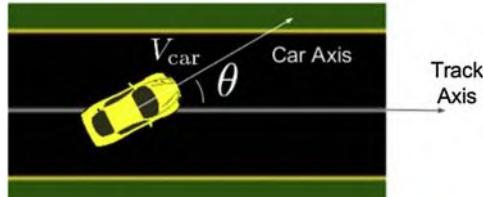


Figure 10.12 Vehicle turning angle.

Based on the environmental reward function and algorithm principle, the pseudocode for the execution of the DDPG algorithm is as follows:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for episode = 1, M **do**

 Initialize a random process \mathcal{N} for action exploration

 Receive initial observation state s_1

for t = 1, T **do**

 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

 Execute action a_t and observe reward r_t and observe new state s_{t+1}

 Store transition (s_t, a_t, r_t, s_{t+1}) in R

 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

 Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

end for

end for

10.6.3.3 Running program

Based on the explanation of the algorithm and pseudocode, the execution code is as follows:

```
pip install keras
pip install tensorflow

git clone https://github.com/yanpanlau/DDPG-Keras-Torcs.git
cd DDPG-Keras-Torcs
cp *.* ..../gym_torcs
cd ../../gym_torcs
python ddpg.py
```



Figure 10.13 TORCS game.

The running effect is shown in Fig. 10.13.

10.7 Summary

The autopilot system is a complex system, which mainly includes three parts, namely, perception, planning, and control. Perception is usually obtained by sensors (i.e., camera, LiDAR, GPS/IMU, radar, and sonar) to obtain information about the current environment. Decision-making through the planning module predicts the next behavior (including routing planning and behavior planning). The control module involves a series of control actions on the vehicle, such as how much throttle, how hard to brake, or how to turn the direction of the car, all of which are related to safe driving on the road. This chapter first introduces the basic concepts and principles of RL and then introduces the TORCS game and simulation experiments. Because of the complexity of the autonomous driving system and the high level of safety requirements, its control process is subject to multilevel safety protection. This chapter only discusses the steering angle and throttle control based on the simulation environment and uses RL to model and simulate the process. Because the autonomous driving system involves many safety considerations and ethical constraints, the application of autonomous driving by controlling the TORCS game through RL is adopted to demonstrate the application value of RL in unmanned driving.

References

1. Li Y. Deep Reinforcement Learning: An Overview[EB/OL]. <https://arxiv.org/pdf/1701.07274.pdf>.
2. Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning. *Nature* 518, 529–533 (2015). <https://doi.org/10.1038/nature14236>.

3. Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M, Playing atari with deep reinforcement learning. 2013. arXiv:1312.5602. <http://arxiv.org/abs/1312.5602>.
4. Arul Kumaran K, Deisenroth M.P, Brundage M. A Brief Survey of Deep Reinforcement Learning[EB/OL]. <https://arxiv.org/abs/1708.05886>.
5. Continuous Control With Deep Reinforcement Learning[EB/OL]. <https://arxiv.org/pdf/1509.02971.pdf>.
6. Using Keras and Deep Deterministic Policy Gradient to play TORCS[EB/OL]. <https://github.com/yanpanlau/DDPG-Keras-Torcs>.

Index

‘Note: Page numbers followed by “f” indicate figures, “t” indicate tables and “b” indicate boxes.’

A

- A* algorithm, 230–239
 - Breadth-first search (BFS) algorithm, 231–233, 232f
 - data structure, 233–234, 234f
 - definition, 238–239
 - Dijkstra algorithm, 237–238
 - directed graph, 230–231, 231f
 - directional search (heuristic), 236, 237f
 - heuristic search, 238f
 - route generate, 234–235, 235f
- Actual combat–traffic sign recognition
 - Belgium traffic sign recognition, 185–190, 186f
 - construct and train, leverage Keras to, 193–194, 194f
 - data preprocessing, 191–192, 191f–192f
- Actual combat–traffic sign recognition, 185–194
- Adaptive cruise control (ACC), 3
- Adaptive moment estimation (Adam), 223–224
- Advanced driver assistance system (ADAS), 3
- Advanced motion modeling, 113–136
- Approximate value function, 312, 312f
- Artificial intelligence (AI), 155
- Artificial neurons, 156f
- Autonomous vehicle based, motion planning method of, 253–269
- Autonomous vehicle control, rudiments of
 - control theory, 276–277, 276f–277f
 - PID control algorithm, 277–288, 278f
 - proportional and derivative control, 280–288, 281f
 - proportional control, 278–280, 279f–280f

- Autonomous vehicle perception module
 - sensors, 100
- Autonomous vehicle route generation, 245–253
- Autonomous vehicle sensing module, 100–113
 - autonomous vehicle perception module sensors, 100
- Kalman filter-based pedestrian localization estimation, 100–103
- Kalman filter pedestrian state estimation, Python, 103–113

B

- Bayesian formula, 94
- Behavior planning, 16–17, 16f
 - state machines in, 243–245, 243f
- Belgium traffic sign recognition, 185–190, 186f
- Bicycle model, 272–273, 291f
- Biological neurons, 156f
- Black box, 152f
- Breadth-first search (BFS) algorithm, 231–233, 232f

C

- Cartesian and Frenet coordinate systems, 256f
- Catkin create system, 32–33
- Classification standards, 2–5, 3t
- CMakelists.txt file, 34–35
- Collision avoidance, 262–263
- Communication platform, 28
- COMPONENTS, 47
- Constant curvature and acceleration (CCA) model, 114
- Constant turn rate and velocity (CTRV) model, 113
- Constant velocity model, 101–102

- Constituent elements, 308–309
 policy, 308–309
 reward, 309
 value function, 309–311
- Control system, 18–21, 20f
- Control theory, 276–277, 276f–277f
- Convolutional neural networks (CNN),
 12, 194–204, 195f, 201f, 217,
 313–314
 definition, 200–202
 equivariant representations, 200
 motivation, 195–196
 parameter sharing, 200
 sparse interactions, 197–199,
 197f–199f
- Cubic spline interpolation algorithm,
 245–250, 246f–248f
- Curvature and acceleration (CCA)
 model, 114
- D**
- Data augmentation, 178–181, 179f,
 181f
- Data collection, 220–221
- Data preprocessing, 191–192,
 191f–192f
- Decision-making plan system, design criteria for, 240
- Deep deterministic policy gradient (DDPG) algorithm, 317–323
 reward function setting, 321–322, 322f
 running program, 322–323, 323f
 theory, 320–321
- Deep feedforward neural networks,
 176–178
 big data, model training under, 176
 representation learning, 177–178, 177f
 support vector machine (SVM), 176
- Deep learning
 actual combat–traffic sign recognition,
 185–194
 Belgium traffic sign recognition,
 185–190, 186f
 construct and train, leverage Keras to,
 193–194, 194f
 data preprocessing, 191–192,
 191f–192f
- convolutional neural networks (CNN),
 194–204, 195f, 201f
 definition, 200–202
 equivariant representations, 200
 motivation, 195–196
 parameter sharing, 200
 sparse interactions, 197–199,
 197f–199f
- deep feedforward neural networks,
 176–178
 big data, model training under, 176
 representation learning, 177–178,
 177f
- support vector machine (SVM), 176
- deep neural networks, regularization
 technology applied to, 178–185
 data augmentation, 178–181, 179f,
 181f
 dropout, 183–185, 183f–184f
 early stopping, 181–182, 181f
 parameter normalization penalties,
 182–183
- You Only Look Once (YOLO),
 204–212, 205f, 213f
 detection network, train, 206–207,
 206f
 loss function, 207
 pedestrian detection, 208–212
 pretrained classification network,
 205–206, 206f
 test, 208
 vehicle, 208–212
- Deep neural network model,
 221–226
- LeNet deep selfdriving model,
 222–224, 223f
- mean squared error (MSE), 223–224
- NVIDIA deep selfdriving model,
 224–226, 224f
- Deep neural networks, regularization
 technology applied to, 178–185
 data augmentation, 178–181, 179f,
 181f
 dropout, 183–185, 183f–184f
 early stopping, 181–182, 181f
 parameter normalization penalties,
 182–183

Deep Q network (DQN) algorithm, 313–316
 convolutional neural network (CNN), 313–314
 objective function, 315–316
 Q learning algorithm, 313
 reward function, 314–315
 Differential GPS (DGPS), 83
 Dijkstra algorithm, 237–238
 Dimension disaster, 17
 Directed graph, 230–231, 231f
 Directional search (heuristic), 236, 237f
 Distributed design, 29
 Dropout, 183–185, 183f–184f
 Dynamic bicycle model, 275–276

E
 Early stopping, 181–182, 181f
 End-to-end selfdriving
 convolutional neural network (CNN), 217
 data collection, 220–221
 deep neural network model, 221–226
 LeNet deep selfdriving model, 222–224, 223f
 mean squared error (MSE), 223–224
 NVIDIA deep selfdriving model, 224–226, 224f
 definition, 218–219
 processing, 220–221
 simulator selection, 219
 transfer learning, 216–217, 216t
 Environmental perception, 9–12, 11f
 Environment configuration, 22–25
 install OpenCV, 24–25
 robot operating system (ROS), 23–24
 simple environment installation, 22–23
 Exclusive OR (XOR) relationship, 157
 Experience replay, 316b
 Extended Kalman filter (EKF), 100, 113–136
 Jacobi matrix, 116–119
 measurement, 122–123
 process noise, 116–119
 Python implementation, 124–136

F
 Feedback control, 18–19
 Feedback correction, 19, 292–293, 292f
 Fifth degree trajectory, 257–261
 Filter algorithm, 99
 Finite state machine (FSM), 16, 230
 Fitting arbitrary functions, 158–159
 Forecast error calculation, 96
 Forward propagation, 162
 Forward transmission, 160–162, 160f
 Free boundary cubic spline interpolation
 autonomous vehicle route generation, 245–253
 cubic spline interpolation, 245–249, 246f–248f
 cubic spline interpolation algorithm, 249–250
 Python, 251–253
 Frenet optimization trajectory
 autonomous vehicle based, motion planning method of, 253–269
 Cartesian and Frenet coordinate systems, 256f
 collision avoidance, 262–263
 fifth degree trajectory, 257–261
 Jerk minimization, 257–261
 motion planning for autonomous vehicles, 263–269
 polynomial solution, 257–261
 FSM, 240–242

G
 Geometric path tracking, 20
 Geometric path tracking algorithms, 20–21
 GitHub, 219
 Global Navigation Satellite System (GNSS), 81–82
 Global positioning system (GPS), 81–86
 different sensors, localization fusion of, 84–86, 86f
 localization principle, 82–84, 83f
 Graphics processing unit (GPU), 215

H
 Handwritten digit recognition task, 147–148

- Hello World program, 39
 Heuristic search, 238f
H
 Hierarchical finite state machine (HFSM), 242–243, 242f
 autonomous vehicle behavior planning, 239–245, 239f
 behavior planning, state machines in, 243–245, 243f
 decision-making plan system, design criteria for, 240
 FSM, 240–242
 hierarchical FSM, 242–243, 242f
I
 High-definition maps (HD maps), 4–5
 Husky robot, small case based on, 45–52
 Husky simulator, practice based on, 35–39, 35f
J
 ICP algorithm, 65–71
 ImageNet, 216
 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), 216
 Inertial measurement unit (IMU), 85
 Inertial navigation system (INS), 13, 81–86
 Install OpenCV, 24–25
 Iterative closest point (ICP), 14, 65
K
 Jacobi matrix, 116–119
 Jerk minimization, 257–261
L
 Kalman algorithm, 85
 Kalman filter
 advanced motion modeling, 113–136
 EKF algorithm, 113–136
 unscented Kalman filter (UKF), 136–142
 vehicle tracking, 113–116
 autonomous vehicle sensing module, 100–113
 autonomous vehicle perception module sensors, 100
 Kalman filter-based pedestrian localization estimation, 100–103
 Kalman filter pedestrian state estimation, Python, 103–113
 Bayesian formula, 94
 definition, 94
 forecast error calculation, 96
 Kalman gain calculation, 97
 likelihood estimation $P(Z/X)$, 94
 measurement error, 96–97
 optimal estimate error, 97–99
 posterior probability $P(X/Z)$, 94
 priori probability $P(X)$, 94
 status forecast, 95–96
 Kalman gain calculation, 97
 Keras library, 163–173
 data preparation, 163–168
 deep feedforward neural network, 169–172, 171f–172f
 three-layer neural network, 169–172
 Kinematic bicycle model, 274–275, 274f
 Kinematic model, MPC based on, 289–293
 feedback correction, 292–293, 292f
 predictive model, 289–290, 290f–291f
 steering control, PID controller forwards to, 289
 time series, online optimal loop based on, 291–292
M
 Laser radar, 11
 Light Detection and Ranging (LiDAR) system, 63
 Light weight, 29
 Likelihood estimation $P(Z/X)$, 94
 Localization, 13–14
 Localization principle, 82–84, 83f
N
 Machine learning (ML)
 basic concepts of, 146–149
 handwritten digit recognition task, 147–148

- Modified National Institute of Standards and Technology (MNIST), 146, 147f
- optimization, 148
- performance criteria, 148
- reinforcement learning, 148. *See also* Reinforcement learning
- supervised learning. *See* Supervised learning
- unsupervised learning, Unsupervised learning
- Markov decision process, 307–308, 307f
- Master, 29–30
- Mean squared error (MSE), 65, 223–224
- Measurement error, 96–97
- Message, 32
- Mission planning, 14–15, 15f
- Model predictive control (MPC), 19
- Modified National Institute of Standards and Technology (MNIST), 146, 147f, 149, 156
- Motion planning, 17–18, 18f
autonomous vehicles, 263–269
- Multilanguage support, 29
- Multiple cameras, 220
- Multiple nodes, 30–31
- N**
- National Highway Traffic Safety Administration (NHTSA), 2–3
- Neural network
artificial neurons, 156f
basic structure of, 156–158
biological neurons, 156f
black box, 152f
conceptual relationships, 155, 155f
example of, 151f
exclusive OR (XOR) relationship, 157
fitting arbitrary functions, 158–159
forward transmission, 160–162, 160f
fundamentals of, 155–163
hierarchies, 155, 155f
Keras library, 163–173
data preparation, 163–168
deep feedforward neural network, 169–172, 171f–172f
- three-layer neural network, 169–172
- NOT-AND (NAND) gate, 158
- stochastic gradient descent (SGD), 162–163
- unlimited capacity, 158–159
- Newton's method optimization, 78
- Node, 30
handle, 41
- Normal distribution transform (NDT), 14, 72–81
algorithm, 72–73, 73f
algorithm example, 75–81
basic steps of, 73–74
- NOT-AND (NAND) gate, 158
- O**
- Object-oriented node coding, 44
- Optimal estimate error, 97–99
- Optimization, 148
parameters, 74
- P**
- Package.xml, 34
- Parameter normalization penalties, 182–183
- Parameter services, 44–45
- Path planning, 14–15
- Perception, 9
- Personal computer (PC), 215
- PID control algorithm, 277–288, 278f
- Platform support, 28
- Plugins, 59
- Point Cloud Library (PCL), 63
- Point-to-point design, 29
- Policy gradient, 317
- Polynomial solution, 257–261
- Posterior probability $P(X/Z)$, 94
- Prediction model, 19
- Predictive model, 289–290, 290f–291f
- Priori probability $P(X)$, 94
- Probability density function (PDF), 72
- Process noise, 116–119
- Project organization structure, 33–35
- Proportional and derivative control, 280–288, 281f
- Proportional control, 278–280, 279f–280f

- Proportional–integral–derivative (PID)
controller, 18–19
- Python, 251–253
implementation, 124–136
- ## R
- Real-time kinematic (RTK), 82
- Rectified linear unit (ReLU), 222
- Reference trajectory, 19
- Reinforcement learning (RL), 148
approximate value function, 312, 312f
constituent elements, 308–309
policy, 308–309
reward, 309
value function, 309–311
- deep deterministic policy gradient (DDPG) algorithm, 317–323
reward function setting, 321–322, 322f
running program, 322–323, 323f
theory, 320–321
- deep Q network algorithm, 313–316
convolutional neural network (CNN), 313–314
objective function, 315–316
Q learning algorithm, 313
reward function, 314–315
- definition, 307–311
experience replay, 316b
Markov decision process, 307–308, 307f
overview of, 306
policy gradient, 317
target network, 316b
TORCS game, 317, 323f
environment installation, 318–320, 318f
- Road traffic safety, improvement of, 7
- Robot operating system (ROS), 23–24
abilities, 28
action, 57, 57f
basic programming, 39–52
Catkin create system, 32–33
CMakelists.txt file, 34–35
common tools, 57–62
communication platform, 28
concept of, 29–32
- definition, 28
distributed design, 29
features of, 29
free and open source, 29
history of, 28–29
husky robot, small case based on, 45–52
husky simulator, practice based on, 35–39, 35f
light weight, 29
master, 29–30
message, 32
multilanguage support, 29
node, 30
object-oriented node coding, 44
parameter services, 44–45
platform support, 28
point-to-point design, 29
project organization structure, 33–35
- ROS C++ client library (roscpp), 39–42
node handle, 41
ROS logging method, 42
rqt, 58–59, 59f
Rviz, 58, 58f
SDF, 61–62
services, 53–56, 54f
TF coordinate conversion system, 59–61, 60f
tools, 28
topic, 30–31
URDF, 61–62
write simple publish and subscribe code, 42–44
- Rolling optimization, 19
- ROS C++ client library (roscpp), 39–42
node handle, 41
ROS logging method, 42
- ROS logging method, 42
- Route generate, 234–235, 235f
- Route planning, 14–15
- Rqt, 58–59, 59f
Rviz, 58, 58f

S

Sampling-based methods, 18
 SDF, 61–62
 Services, 53–56, 54f
 Simple environment installation, 22–23
 Simulation Description Format (SDF),
 57
 Simulator selection, 219
 Simultaneous localization and mapping
 (SLAM), 13–14
 Single-point activation encoding
 method, 161
 SLAM-based localization system, 87–92,
 87f
 applications, 90–92
 localization principle, 88–90, 89f
 Sobel operator, 12
 Society of Automotive Engineers (SAE),
 2–3
 Software module, 30
 Specific code implementation, 65
 Standard positioning service (SPS),
 81–82
 Status forecast, 95–96
 Steering control, PID controller forwards
 to, 289
 Stochastic gradient descent (SGD),
 162–163
 Supervised learning, 148
 artificial neural network (ANN), 151
 certain algorithm, 153–154
 empirical risk minimization (ERM)
 strategy, 149–150
 gradient descent algorithm, 153–154
 overfitting, 150–153
 underfitting, 150–153
 Support vector machine (SVM), 12

T

Target network, 316b
 Test error, 152
 TF coordinate conversion system,
 59–61, 60f
 The Open Racing Car Simulator
 (TORCS) game, 317
 environment installation, 318–320,
 318f

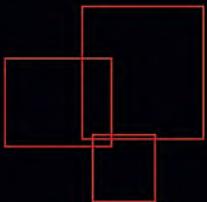
Time series, online optimal loop based
 on, 291–292
 Training error, 152
 Trajectory generation, 19–20
 Trajectory tracking, 20, 292–293,
 294f–295f, 302f
 Transfer learning, 216–217, 216t
 Travel efficiency, 8–9

U

Unified Robot Description Format
 (URDF), 57, 61–62
 Unlimited capacity, 158–159
 Unmanned systems, localization methods
 for
 achieving localization, principle of,
 63–65
 differential GPS (DGPS), 83
 disadvantages, 65
 global positioning system (GPS), 81–86
 different sensors, localization fusion
 of, 84–86, 86f
 localization principle, 82–84, 83f
 ICP algorithm, 65–71
 inertial navigation system (INS), 81–86
 iterative closest point (ICP), 65
 localization principle, 82–84, 83f
 mean squared error (MSE), 65
 normal distribution transform (NDT),
 72–81
 advantages of, 74–75
 algorithm, 72–73, 73f
 algorithm example, 75–81
 basic steps of, 73–74
 real-time kinematic (RTK), 82
 SLAM-based localization system, 87–92,
 87f
 applications, 90–92
 localization principle, 88–90, 89f
 specific code implementation, 65
 Unmanned vehicles
 adaptive cruise control (ACC), 3
 advanced driver assistance system
 (ADAS), 3
 basic framework of, 9–22, 10f
 behavior planning, 16–17, 16f
 control system, 18–21, 20f

- Unmanned vehicles (*Continued*)
 environmental perception, 10–12, 11f
 localization, 13–14
 mission planning, 14–15, 15f
 motion planning, 17–18, 18f
 classification standards, 2–5, 3t
 definition, 2–7
 environment configuration, 22–25
 install OpenCV, 24–25
 robot operating system (ROS), 23–24
 simple environment installation, 22–23
 high-definition maps (HD maps), 4–5
 implementation of, 5–7, 5f
 lowering the threshold for drivers, 9
 road traffic safety, improvement of, 7
 travel efficiency, 8–9
 urban traffic congestion alleviation, 8
- Unscented Kalman filter (UKF), 136–142
 lossless transformation, 138, 139f
 measurement updates, 141–142
 movement model, 137–138
 nonlinear processing/measurement models, 138
 predicted mean and variance, 140
 projections, 139–140
 sigma point prediction, 140
 update status, 142
- Unsupervised learning, 148
- Urban traffic congestion alleviation, 8
- V**
- Vehicle tracking, 113–116
 linear motion model, 113
 nonlinear motion model, 113
- W**
- Write simple publish and subscribe code, 42–44
- Y**
- You Only Look Once (YOLO), 204–212, 205f, 213f
 detection network, train, 206–207, 206f
 loss function, 207
 pedestrian detection, 208–212
 pretrained classification network, 205–206, 206f
 test, 208
 vehicle, 208–212

Theories and Practices of Self-Driving Vehicles



Qingguo Zhou, Zebang Shen, Binbin Yong, Rui Zhao, Peng Zhi

Systematically introduces the full stack of self-driving vehicle systems from principles to practice

Theories and Practice of Self-Driving Vehicles presents a comprehensive introduction to the technology of self-driving vehicles across the three domains of perception, planning, and control. This book systematically introduces vehicle systems from principles to practice, including basic knowledge of Robot Operating System (ROS) programming, machine and deep learning, as well as basic modules such as environmental perception and sensor fusion. The book introduces advanced control algorithms as well as important areas of new research offering engineers, technicians, and students an accessible handbook to the entire stack of technology in a self-driving vehicle in this rapidly growing area of research and expertise.

Key Features

- Provides a comprehensive introduction to the technology stack of self-driving vehicles
- Offers foundational theory and best practices
- Introduces advanced control algorithms and high-potential areas of new research

About the Authors

Qingguo Zhou is a Professor at Lanzhou University and Deputy Director of the Engineering Research Center for Open Source Software and Real-Time Systems, at the Ministry of Education, China. He is also the Director of the School of Computer Science and Engineering and the Embedded System Laboratory at Lanzhou University. His research focuses on intelligent driving, AI, embedded and real-time systems.

Zebang Shen is a Senior Autonomous Driving Engineer at Daimler AG and a Google Developer Expert (GDE) in machine learning. His research focuses on the development of L4 autonomous driving systems. In particular his interests include 3D perception, multisensor fusion, multisensor automatic calibration, computer vision, and 3D SLAM. He is an advocate for innovation, open source, and knowledge sharing.

Binbin Yong is an Associate Professor and Master's Supervisor in the School of Information Science and Engineering, Lanzhou University, China. He is mainly engaged in research on high-performance computing, neural networks, and deep learning.

Rui Zhao is a PhD candidate in computer science at Lanzhou University, China, who is devoted to the development of search recommendation models in a multinational company. He is currently focused on research on perception in the self-driving vehicle.

Peng Zhi is a PhD candidate in computer science at Lanzhou University, China. His research interests include computer vision, deep learning, and autonomous driving.



ELSEVIER

elsevier.com/books-and-journals

ISBN 978-0-323-99448-4



9 780323 994484