End-to-end Learning for Autonomous Driving

by

Jiakai Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
May, 2019

Professor Kyunghyun Cho

© Jiakai Zhang

ALL RIGHTS RESERVED, 2019

Dedication

To my family and my daughter Claire Zhang.

Acknowledgments

Ph.D. program is a long journey. Throughout my years of study and through the process of researching and writing this thesis, I'm really grateful to have a lot of people who have helped me and encouraged me. Without their support, this work would not have been possible.

First and foremost, I owe a huge debt of gratitude to my advisor Prof. Kyunghyun Cho. I am grateful that Prof. Cho has given me freedom and support to work on what I am really excited at autonomous driving. Prof. Cho has immense knowledge of deep learning. Whenever I encountered obstacles in the research, he always provided me with key insights and invaluable advice. I would never forget the time we sat side by side to debug the code for emulation of a joystick controller in Linux and the great happiness in making it work. I sincerely appreciate Prof. Cho's continuous support and patience throughout my Ph.D. program.

I would also like to thank the rest of my thesis committee: Prof. Richard Bonneau, Prof. Davi Geiger, Dr. Larry Jackel, and Prof. Arjun Jain for their insightful comments as well as the challenging questions which have inspired me to widen my research from different perspectives.

My special thanks go to NVIDIA. I am honored to be one of the recipients of the NVIDIA

funding to support my research at NYU. I took three summer internships at NVIDIA, where I was fortunate to work with subject matter experts in autonomous driving - Urs Muller, Larry Jackel, Beat Flepp, Ben Firner, Mariusz Bojarski, and Chenyi Chen, etc. They taught me the value of truly creative thinking and showed me the power of imagination which can drive research successfully. I sincerely appreciate their trust in me and the resources provided to me while I was interning at NVIDIA, which enabled me to develop a convolutional neural network that can be deployed on a real car and to test it in the real environment. This thesis, especially Chapter Two, would not come into shape without NVIDIA's generous support as well as the insightful suggestions from my colleagues at NVIDIA.

I also learned a lot from my previous advisor Chris Bregler and my fellow labmates Chaoqiang Deng, Lisheng Fu, Xiang Zhang, Junbo Zhao, Elman Mansimov, Alfredo Canziani. Their knowledge and passion in computer science and deep learning have motivated me to dig deeper into this field. I will not forget those stimulating discussions we had, the collaborative projects we did, and more importantly, our shared love of computer science.

Last but not least, I would like to express my huge gratitude to my family for their unfailing support and unconditional love throughout my Ph.D. program and my life in general. Some people say that writing a thesis is similar to raising a child. I have the privilege to have experienced these two things at the same time, which is indeed an unforgettable memory. Being able to go to this step today is inseparable from the support of all my family and friends around. Thank you!

Abstract

The end-to-end learning approach for autonomous driving has sparked great interest in both academic and industry in recent years. The approach can be defined as learning a model that maps from sensory input, such as image frames from a camera, to driving actions for controlling the autonomous vehicle such as steering. Compared to the traditional autonomous driving system, which often includes perception, localization, mapping, and path planning, the end-to-end learning approach offers a more efficient method of utilizing large amounts of expert driver demonstrations to achieve fully autonomous driving without acquiring expensive labeling such as bounding box for objects.

The end-to-end learning for autonomous driving can be done by supervised learning, where a model is tuned to minimize the differences between predicted actions and ground-truth actions. The training data is usually obtained from expert demonstrations. A model trained in this way, however, suffers from bad performance due to the mismatch between the samples visited by a learned model and those collected by an expert driver. In the first part of the thesis, we introduce the end-to-end approaches via both supervised learning and imitation learning with different data

augmentation methods to address the data mismatch issue. The data augmentation in supervised learning approach is done by synthetically generating new samples to simulate driving situations that are different from the ones in expert demonstrations. We show that using such automatically-augmented data, a trained model using supervised learning can drive a car to follow a lane in various conditions on highways and local roads.

Instead of generating new samples, another way to do data augmentation is via collecting new samples during testing and querying an expert to label the new samples. Since it is costly to query an expert, we introduce an imitation learning algorithm called SafeDAgger that can significantly reduce the number of query times and train a driving model more efficiently. The experiments show that a trained model can successfully drive a car in a simulator to do lane following and overtaking.

The expert demonstrations provided by humans, however, often show significant variability due to latent factors such as different driving preferences of different human drivers. A model trained using such human demonstrations to minimize the differences between the expert and the predicted actions may drive a vehicle into a dangerous situation such as crashing into the vehicle ahead. In the second part of the thesis, therefore, we introduce a variational mixture density network with a discrete latent variable to address this issue. The experimental results show that the trained model can not only mimic expert behaviors but also learn the variability of driving actions from expert demonstrations.

In the last part of the thesis, we will introduce a simulator to support the training and evaluation of the end-to-end autonomous driving models in video games. Leveraging this simulator, we demonstrate that the trained model can drive a truck following a navigation map in a video game.

In summary, this thesis introduces the end-to-end learning approaches for autonomous driving

to address the data mismatch issue and to learn the variability of expert driving actions. Our results show that the trained models can drive vehicles to accomplish driving tasks such as lane following, overtaking, and making turns in the simulated driving environments.

Contents

DE	DICATI	ION	iii
Ac	KNOW	LEDGMENTS	iv
Ав	STRAC	Γ	vi
Lis	ST OF F	PIGURES	αii
Lis	ST OF T	CABLES	/ ii
1	Intro	ODUCTION	1
	1.1	Autonomous driving	1
	1.2	Thesis structure	7
2	End-	TO-END AUTONOMOUS DRIVING VIA SUPERVISED LEARNING	8
	2.1	End-to-end learning for autonomous driving	10
	2.2	Supervised learning	10

	2.3	Data collection	11
	2.4	Data augmentation	13
	2.5	Neural network structure and training	18
	2.6	Video simulator	21
	2.7	Evaluation	23
	2.8	Summary	24
3	B End	-to-end Autonomous Driving via Imitation Learning	26
	3.1	Imitation learning for autonomous driving	27
	3.2	DAgger: Beyond supervised learning	31
	3.3	SafeDAgger	33
	3.4	Experiment	39
	3.5	Results and analysis	49
	3.6	Summary	50
2	4 Vari	AATIONAL MIXTURE DENSITY NETWORKS	51
	4.1	Mixture density networks	53
	4.2	Variational mixture density networks	58
	4.3	Experiments	62
	4.4	Summary	66
Į.	5 Dri	ving Simulation for End-to-end Autonomous Driving	67
	5.1	Simulator for autonomous driving research	69
	5.2	Framework of the simulator	71
	5.3	End-to-end driving using navigation map	74

	5.4	Summary	 	 	 	 	82
6	Conc	CLUSION					83
	6.1	Future work	 	 	 	 	85
Він	BLIOGR	АРНҮ					91

List of Figures

1.1	Overview of a traditional autonomous driving system	4
1.2	Overview of an end-to-end autonomous driving system	6
2.1	Camera setup for data collection. The cameras are mounted using suction cups	
	and secured with ropes to avoid accidental loss. One camera is mounted on the	
	center of the hood, while the other two are mounted on the left and right sides	
	of the car. All cameras face straight ahead. We use a remote controller to control	
	all three cameras at the same time.	12
2.2	Representative training data samples showing various road textures and lane marker	
	conditions. Potential issues for autonomous lane following include missing lane	
	markers, bad road painting, and reflected light.	13
2.3	Sample image frames captured by the left camera (left), center camera (middle),	
	and right camera (right).	13

2.4	The mismatch between the samples visited by a learned model and those collected	
	by an expert driver. The rectangle represents a car driven by the trained model,	
	the red dashed line shows the driving trajectory of the trained model, and the blue	
	dashed line shows the trajectory of a human driver. The trained model makes a	
	small mistake and the car deviates from the center of the lane. Since the new	
	situation were never seen before, the trained model may output driving actions	
	that cause the car crash	14
2.5	Synthetic actions not represented by demonstration data: shift from the center	
	of the lane (left) and rotation from the direction of the lane (right). The blue	
	dashed line indicates the center of the lane.	15
2.6	Data augmentation: augmented data is generated via image transformation to	
	simulate shift and rotate situations. The corresponding steering command is ad-	
	justed to one that would return the vehicle to following the center of the lane	16
2.7	Sample images generated by data augmentation	17
2.8	Histograms of steering angles before and after data augmentation. In the raw data	
	set, most steering angles fall within the tiny range of [-0.05, 0.05], due to vehicles	
	driven by humans only requiring small adjustments to follow a lane	17
2.9	Structure of the convolutional neural network used for lane following	20
2.10	Training procedure for the CNN used for lane following.	21
2.11	Video simulator system	22
2.12	Screenshot of the simulator. The area on the left is filled black since it is unknown	
	due to image transformation. The highlighted wide rectangle below the horizon	
	is the image area that is sent to the CNN	24

3.1	Representative road textures and lane markers used with TORCs	4(
3.2	Training and test tracks with sample frames	41			
3.3	The histogram of the log square errors of the steering angle after supervised learn-				
	ing only. The dashed line is located at $\tau=0.0025$. 77.70% of the training				
	examples are considered safe	46			
3.4	Sample image frames sorted according to a safety classifier trained on a primary				
	policy right after the supervised learning stage. The number in each frame is the				
	probability of the safety classifier returning 1	47			
3.5	(a) Average number of laps (\uparrow), (b) damage per lap (\downarrow), (c) the mean squared error				
	of steering angle for each configuration (training strategy-driving strategy) over				
	each iteration and (d) the portion of time driven by the reference policy during				
	testing. We use dashed and solid curves for the cases with and without traffic,				
	respectively	48			
4.1	The uncertainty of driving actions in an overtaking scenario: the car can pass the				
	vehicle ahead either on the left or the right. The average of these two actions,				
	which would be learned by a model $\pi_{\theta}(s)$ trained to minimize the square errors of				
	driving actions between expert demonstrations and model predictions, will cause				
	the car to crash into the vehicle ahead	52			
4.2	Gaussian mixture model with three components. The red curve shows the mixture				
	densities	54			
4.3	Model architecture for a mixture density network	56			

4.4	Model architecture for the variational mixture density network. The sensor input	
	encoder is shared across all three parts. We use one hot encoding for the latent	
	variable z as the input of the policy network $\pi(s,z)$	62
4.5	Driving trajectory: For each time step, we assume the moving distance in the	
	y direction is a constant d_y . ϕ_t denotes the angle between the lane direction	
	and heading of the vehicle at time step t . The position of the vehicle coordinate	
	(x_t,y_t) is calculated by $y_t=y_{t-1}+d_y$ and $x_t=x_{t-1}+d_y an\phi_{t-1}$	65
4.6	Visualization of various driving trajectories based on the different latent variables	
	z_k . The top row shows the image frames of the sensor input. The second row	
	shows the corresponding human driving trajectories. The third row shows the	
	various driving trajectories based on different latent codes. The last row shows	
	the learned prior distribution $p(z \vert s)$. This figure empirically illustrates that the	
	three dimensional latent variable $z_{1:3}$ can be viewed as a set of high-level driving	
	commands, which are lane following, left lane changing, right lane changing	65
4.7	Screenshot of sensor inputs for VMDN evaluation in TORCS, illustrating that	
	the driving behavior of the car driven by the trained model can be controlled by	
	the latent variable	66
5.1	Schematic of the simulator used in autonomous driving research	69
5.2	Screenshots of representative racing simulators: Super Tux Kar (top-left), TORCS	
	(top-right), WRC6 (bottom-left), and Udacity's Self-Driving Car Simulator (bottom-	
	right)	70

5.3	Block diagram of the simulator framework. The simulator state, including sensor	
	inputs and control signals, is shared across all modules and updated at each time	
	step	72
5.4	Simulator interface	73
5.5	Samples of complex intersections based on actual intersections(of City Trans-	
	portation Officials, 2019). These irregular intersections, which result from suc-	
	cessive urban developments, make for difficult-to-describe navigation commands.	75
5.6	Model architecture for end-to-end learning using a navigation map	76
5.7	Setup for data collection in American Truck Simulator	78
5.8	Representative data collected in ATS	79
5.9	Representative sensor inputs and navigation map from our dataset	80
5.10	Our proposed model for autonomous driving using a navigation map	81

List of Tables

3.1	Configuration of a primary policy network. Each convolutional layer is denoted by	
	"Conv - # channels \times height \times width". Max pooling without overlap follows each	
	convolutional layer. We use rectified linear units for point-wise nonlinearities.	
	Only the shaded part of the full network is used during tests	45
4.1	Mean square error of steering determined through different inference strategies	64

1

Introduction

1.1 Autonomous driving

The rapid development of the technologies in computer vision and machine learning has enabled researchers and industry leaders to make significant progress in achieving autonomous driving. Today, autonomous driving technology is making a prominent appearance in our society. Vehi-

cles equipped with advanced driver assistance systems (ADAS) can accomplish autonomous driving in several scenarios such as highway driving. These technologies aim to reduce the number and severity of road accidents. Every year, approximately 1.35 million people lose their lives in automobile accidents, and up to 50 million people suffer accident-related injuries (Organization, 2019). Ninety-four percent of serious crashes are due to risky driving and to errors people make while behind the wheel (highway traffic safety administration, 2019). A self-driving revolution that reduces traffic accidents has the potential to positively impact the lives of millions of people. Furthermore, once fully autonomous driving is achieved, we no longer have to keep our hands on the wheel or our eyes on the road, which would free us to read or get work done while traveling. The average American spends nearly 300 hours in their car each year (Association, 2019), which is over seven full work weeks. That is a ton of potential productivity lost. Autonomous vehicles can eliminate or at least mitigate these issues.

1.1.1 History and background information

Since the 1980s, industrial leaders and researchers have been working on developing a fully autonomous vehicle that is comfortable, reliable, and safe for high-speed driving in the real world. Road tests and self-driving competitions held around the world identify shortcomings and difficulties in both software and hardware and allow autonomous driving technologies to be rapidly improved. Although some known problems remain unresolved, more and more vehicles with a certain level of autonomous driving ability are running on the road.

In 1995, "No Hands Across America" was introduced as one of the first long-distance road tests for autonomous driving (Jochem and Pomerleau, 1995). A trained neural network was used to steer a vehicle driving across the United States while human drivers controlled its acceleration and

braking. This was also one of the earliest research works to demonstrate end-to-end learning for autonomous lane keeping.

The next major competition called DARPA Grand Challenges was initiated by the Defense Advanced Research Projects Agency (DARPA) in 2003 (Rouff and Hinchey, 2011). It required autonomous vehicles to drive in off-road environments without the aid of road markings. Around the same time, the DAVE project was introduced to demonstrate an end-to-end learning system for off-road vehicle control using only visual input (Muller et al., 2006). In this project, a convolutional neural network was trained to predict steering angles based on images from left and right cameras.

After the DARPA Grand Challenges competition, researchers started to tackle the challenges of urban driving in complex environments with dense traffic. Since road tests for autonomous urban driving were not permitted at that time, they faced considerable difficulty in addressing the challenges and making evaluations in real-world environments.

In 2007, the DARPA Urban Challenge was held to provide a real urban driving environment, which included an intersection and simulated highway on-ramp (Rouff and Hinchey, 2011). This competition provided opportunities for researchers to assess the capabilities and limits of autonomous driving in complex urban environments. The autonomous vehicles in the challenge were developed with the capability of localizing to the environment; they detected and tracked objects using various sensing, perception, and localization technologies (Leonard et al., 2008). Four vehicles completed the challenge, including teams from Stanford University, Carnegie Mellon University, Massachusetts Institute of Technology and Virginia Polytechnic Institute and State University. Although the road-test scenarios used in the DARPA Urban Challenge were much more similar to daily driving compared to those in previous competitions, they were still constrained to low-speed driving and simple situations that lack some common roadway obstacles,

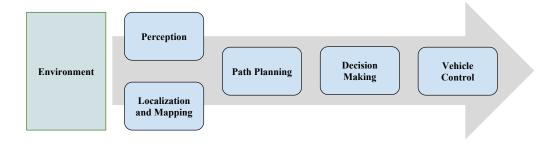


Figure 1.1: Overview of a traditional autonomous driving system.

including pedestrians and cyclists.

1.1.2 Overview of autonomous driving systems

To date, most of these autonomous driving systems can be categorized into two main classes: traditional autonomous driving systems and end-to-end autonomous driving systems.

Traditional autonomous driving system

As a brief overview, a traditional autonomous driving system can be divided into five main components, shown in Fig. 1.1: perception, localization and mapping, path planning, decision making, and vehicle control (Cheng, 2011).

Similar to human vision, the perception component uses sensors to continuously scan and analyze its surrounding environment. This component usually consists of functions for obstacle detection and tracking, traffic sign recognition, and lane marker detection based on various sensor inputs. The localization and mapping component calculates the global and local locations of the vehicle and also maps the environment based on sensor data. Path planning uses perception and localization information to calculate possible safe and ideal routes for the vehicle to drive. The

decision-making component is designed to generate the optimal path based on the available routes, the vehicle's state, and environmental information. Finally, the vehicle control module determines the driving commands, such as steering angle, acceleration, etc., that drives the vehicle along the determined optimal route.

Although promising progress has been made in developing the traditional autonomous driving system, there are still many challenges towards building a fully autonomous driving vehicle. The perception component in the traditional autonomous driving systems needs to recognize specific human-designed features such as traffic signs, traffic lights, lane markers and bounding boxes for driving-relevant objects, which requires a large amount of expensive manually-labeled data. Some of the tasks such as semantic segmentation are still considered as open research questions in computer vision. Besides, the traditional autonomous driving systems today widely utilize pre-built high definition (HD) maps. These maps are detailed, static, and highly accurate records of the surrounding environment. Typically, the autonomous vehicles rely heavily on this prior information for accurate localization and for detecting and recognizing obstacles. However, there are several obvious limitations to the use of HD maps. First, the static maps are quite expensive to build and to keep current with dynamic environments that change over time. Second, the reliance of autonomous vehicles on pre-built maps limits their capability to react and adapt to new situations such as construction zones. These drawbacks have inspired research into the end-to-end learning approach for autonomous driving that does not require manual decomposition of the autonomous driving system and detailed maps (Bojarski et al., 2016; Zhang and Cho, 2017).

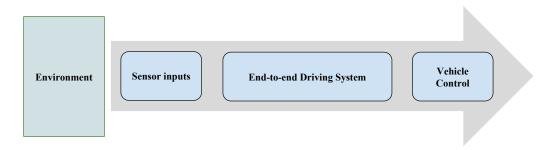


Figure 1.2: Overview of an end-to-end autonomous driving system.

End-to-end autonomous driving system

We define end-to-end autonomous driving as a single, self-contained driving system that carries out all processes automatically, from mapping based on sensory input, such as a front-facing camera, to the actions necessary for driving, such as steering, braking and acceleration. An end-to-end autonomous driving system is often designed to learn from expert demonstrations rather than depend upon manually-designed tasks and modules.

Although the end-to-end learning approach simplifies self-driving systems, it is challenging to train a model that encompasses everything from mapping very high-dimension, pixel-level sensor inputs to controlling low-dimension continuous control signals. In this thesis, we will introduce several methods to address the following major challenges of using the end-to-end learning approach for autonomous driving:

- Mismatch in data distribution at test time between human drivers and the learned driving model.
- Uncertainty in the human driving actions used for training.
- Difficulty of testing and evaluating the driving model.

1.2 Thesis structure

The organization of this dissertation is as follows: In the first two chapters, we aim to address the mismatched data distribution issue using data augmentation. Chapter 2 introduces an end-to-end learning approach for the lane following task based on supervised learning with data augmentation. Chapter 3 presents an end-to-end learning approach for both following and changing lane tasks based on query-efficient imitation learning. Chapter 4 introduces a variational mixture density network to address the uncertainty in the human driving actions. Chapter 5 presents a general driving simulator for training and testing models in video games that use end-to-end learning approaches. Leveraging this simulator, we demonstrate that the trained model can drive a truck following a navigation map in a video game.

2

End-to-end Autonomous Driving via Supervised Learning

In 1989, Autonomous Land Vehicle In a Neural Network (ALVINN), a three-layer neural network trained for the task of lane following, drove a retrofitted Army ambulance around Carnegie Mellon University under controlled field conditions without any human intervention (Pomerleau, 1989).

ALVINN is one of the first examples of an autonomous vehicle using the end-to-end learning approach.

The ALVINN net featured two kinds of sensory inputs: 1) a 30x32 image streamed from a camera mounted on the top of the vehicle, and 2) 8x32 image encoding range information captured by a laser range finder.

The output layer of the ALVINN network could be divided into two groups of units. The first group, consisting of one unit, indicated whether the texture of the road in the current image was lighter or darker than the non-road. During testing, this unit was also recursively sent to the network's input layer. The second group, consisting of 45 units, represented the turning curvature along the direction that the vehicle should travel in order to remain in the center of the road. Activation of the middle part of the units represented straight driving, while activation of the left or right units represented left or right turns. In order to convert the network's output active levels into a steering direction, a Gaussian curve with a fixed width was used to fit the output units. The peak of the best fit Gaussian determined the vehicle's steering.

The ALVINN net inspired our work in many ways. It demonstrated that an end-to-end trained neural network could successfully drive a vehicle along a road. It also showed the importance of having sufficient variability in the training data set to cover different driving conditions. In this chapter, we will introduce an end-to-end learning method that trains a neural network to steer a vehicle for lane following.

2.1 End-to-end learning for autonomous driving

We define the end-to-end learning for autonomous driving as learning a function to map sensor inputs, such as the images from a front-facing camera, to driving commands, such as steering angle. The mapping function can be of any appropriate form; here we use a convolutional neural network (CNN), as a CNN takes advantage of local spatial coherence in the sensor inputs, which are often represented as RGB images.

2.2 Supervised learning

One means for learning such a mapping function is the reinforcement learning method. In reinforcement learning, the learning models are guided by sparse rewards such as travel time, vehicle condition, and the like. Although reinforcement learning has been applied to many promising and successful projects, such as playing computer games, it is challenging to utilize directly for developing an autonomous vehicle. There are three major issues in training an autonomous driving system via reinforcement learning. First, damage to the environment or other agents is a severe safety concern for autonomous driving. In a simulation, we can afford many failures without worrying about safety issues. However, in a real environment, car crashes and accidents are not acceptable. Second, many failures while learning in a real environment are also not affordable. For example, we cannot just let the car hit objects many times in order to learn how to avoid obstacles. Third, it is costly to run experiments in real environments, as well as time-consuming; reality cannot be sped up like a simulation.

Another means for learning the mapping function is the supervised learning method, in which

a set of demonstrations are provided that consist of pairs of sensor inputs and driving commands. More specifically, both sensor inputs and driving signals are recorded while a human drives the car, and these are used to train the mapping function. However, solely using captured data to train such a mapping function is not effective because the demonstrations only provide correct actions; the model does not learn how to recover from mistakes. In this chapter, we introduce an approach to address this issue via data augmentation.

2.3 Data collection

To train the mapping function, we need to capture both sensor inputs and the corresponding driving commands. For sensor inputs, we decided to use images captured from the three front-facing cameras mounted on the car hood. Much like the viewpoint of a human driver, these images should contain enough visual information for performing the lane following task. The specific position of the cameras is not critical; in fact, a position behind the windshield might be a better choice for camera mounting, as its higher placement would provide a better view and is the interior would be more stable with regard to wind while driving. We used three GoPro cameras rather than just one in order to capture sequences of images simultaneously; the rationale behind this setup is related to data augmentation, and will be explained later in section 2.4. The final camera positions are shown in Fig. 2.1.

During demonstrations, the steering angle is captured via the vehicle's Controller Area Network (CAN) Bus. The CAN Bus can be viewed as a simple network that permits any system in the car to listen to and send commands. In order to make the system independent of car geometry, we represent the steering command as 1/r, where r is the turning radius in meters. We use 1/r



Figure 2.1: Camera setup for data collection. The cameras are mounted using suction cups and secured with ropes to avoid accidental loss. One camera is mounted on the center of the hood, while the other two are mounted on the left and right sides of the car. All cameras face straight ahead. We use a remote controller to control all three cameras at the same time.

instead of r to prevent a singularity when driving straight (the turning radius for driving straight would be infinity). 1/r smoothly transitions through zero from left turns (negative values) to right turns (positive values).

Training data was collected through driving on a wide variety of roads and in a diverse set of lighting and weather conditions. Road types represented include highways, two-lane roads (with and without lane markings), residential roads with parked cars, tunnels, and unpaved roads. Various road textures were captured in the dataset, including different road paint colors and different lane marker conditions (Bojarski et al., 2016). Representative training samples in Fig. 2.2 illustrate the various road textures and lane marker conditions in the dataset.



Figure 2.2: Representative training data samples showing various road textures and lane marker conditions. Potential issues for autonomous lane following include missing lane markers, bad road painting, and reflected light.

2.4 Data augmentation

The dataset contains single images s sampled from video, paired with the corresponding steering command a. A sample image is shown in Fig. 2.3.



Figure 2.3: Sample image frames captured by the left camera (left), center camera (middle), and right camera (right).

Training only with data from human demonstrations is not sufficient. A model trained using

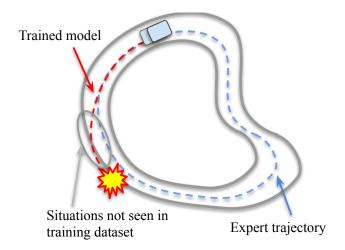


Figure 2.4: The mismatch between the samples visited by a learned model and those collected by an expert driver. The rectangle represents a car driven by the trained model, the red dashed line shows the driving trajectory of the trained model, and the blue dashed line shows the trajectory of a human driver. The trained model makes a small mistake and the car deviates from the center of the lane. Since the new situation were never seen before, the trained model may output driving actions that cause the car crash.

only human demonstrations can suffer from bad performance due to the mismatch between the samples visited by a learned model and those collected by an expert driver. In our case, most of the training samples were collected while driving in the center of the lane, which is not guaranteed to be the case during testing; the trained model makes small mistakes during testing and will drift from the center of the lane. Therefore, a vehicle driven by the trained model will ultimately crash because it does not know how to recover from the new, non-center state that was not represented in the training set as shown in Fig. 2.4.

The data mismatch issue can be addressed by adding new samples to cover situations the car will encounter when trying out the trained model. In order to generate the actions for these new samples, we could ask the human expert to label all samples generated by the trained model and then retrain the model. However, doing such labeling work for autonomous driving is time-



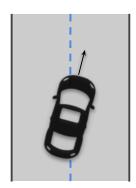


Figure 2.5: Synthetic actions not represented by demonstration data: shift from the center of the lane (left) and rotation from the direction of the lane (right). The blue dashed line indicates the center of the lane.

consuming and challenging. For example, it is difficult to identify the ideal degree of the steering wheel from an image. Instead of manually labeling, we introduce a data augmentation method that synthetically generates samples. Each generated sample represents the car in one of the various shifts from the center of the lane and rotations from the direction of the road. Fig. 2.5 shows the two kinds of scenarios generated by data augmentation in which a vehicle does not correctly follow the lane.

We obtained the images for two specific off-center shifts from the left and right cameras, and simulated additional shifts between the cameras and all rotations by camera viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires depth information for each pixel in an image, which we did not have. We therefore approximated the transformation by assuming all points below the horizon to be on flat ground and all points above the horizon to be infinitely far away. We manually chose the position of the horizon and use it for all images in the training set.

The shift operation to transfer the raw pixel position (u, v) below the horizon to the new pixel

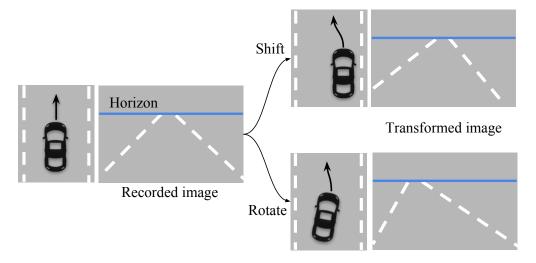


Figure 2.6: Data augmentation: augmented data is generated via image transformation to simulate *shift* and *rotate* situations. The corresponding steering command is adjusted to one that would return the vehicle to following the center of the lane.

position (u', v') is defined as:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} u \\ v * (1 + s * (u - u_h)) \end{bmatrix}$$
 (2.1)

 u_h is the vertical coordinate of the horizon. s is the shift parameter, which represents how much we want to shift.

We define the rotate operation of the image transformation as:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} u \\ v * (1 + \theta * (u_b - u)) \end{bmatrix}$$
 (2.2)

 u_b is the vertical coordinate of the bottom of the image.

The assumption works fine for flat terrain, but introduces distortions for objects that stick above



Figure 2.7: Sample images generated by data augmentation.

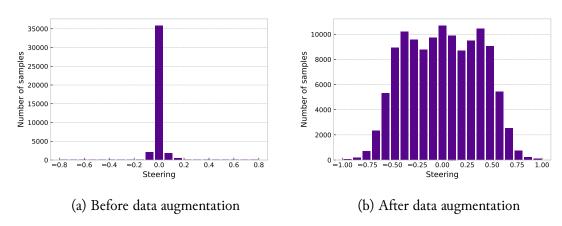


Figure 2.8: Histograms of steering angles before and after data augmentation. In the raw data set, most steering angles fall within the tiny range of [-0.05, 0.05], due to vehicles driven by humans only requiring small adjustments to follow a lane.

the ground, such as cars, poles, trees, and buildings. Fortunately, these distortions do not greatly affect network training; the three-camera setting helps to reduce artifacts introduced by image transformation. Additionally, we crop the raw image, only showing the part below the horizon and removing any black pixels introduced by the transformation. Fig. 2.2 shows a mini-batch of image samples generated using the data augmentation approach.

For transformed images, we adjusted each steering label to one that would steer the vehicle back to the desired location and orientation within a certain driving distance. Fig.2.8 shows the histograms of steering angles before and after data augmentation.

Finally, the augmented data contains both raw data and the images transformed with random shift s and rotation θ parameters and adjusted steering labels. The transformed images have dimensions of 66x200, which is an ideal input size for the convolutional neural network performing the lane following task. A high-resolution input image costs more in computing time without carrying additional useful information. Importantly, system latency is a critical concern for autonomous driving. In any single loop of a real-time autonomous driving system, we need to perform sensor input capture, image transformation, model prediction, and feeding of the control signal to the vehicle. This enforces strict constraints on the run-time of the model's predictions.

2.4.1 Data Selection

In order to train a neural network, we need to select training frames from the raw recorded video data. We labeled our collected data with road type, weather condition, and driver activity (staying in a lane, switching lanes, turning, and so forth) (Bojarski et al., 2016). To train a CNN for lane following, we only selected data where the driver was staying in a lane; other activities were discarded. We then sample that video at ten frames per second (FPS). A higher sampling rate would include images that are highly similar and thus would not provide much useful information.

2.5 Neural network structure and training

To run the system in real time, we started with a simple feed-forward CNN include a normalization layer, five convolutional layers, and three fully-connected layers, shown in Fig. 2.9. This convolutional neural network works surprisingly well for the lane following task. The input image is split into YUV planes and passed to the network. This convolutional neural network structure

works surprisingly well for the lane following task.

The first layer of the network performs image normalization. The normalizer is hard-coded and will remain fixed throughout the learning process. Performing normalization in the network allows the normalization scheme to be altered with the network architecture and to be accelerated via GPU processing.

The convolutional layers were chosen empirically through a series of experiments that varied layer configurations. We use strided convolutions in the first three convolutional layers, with a 2×2 stride and a 5×5 kernel, and non-strided convolutions with a 3×3 kernel in the last two convolutional layers. The five convolutional layers are followed by three fully-connected layers that lead to an output vehicle control value, the inverse turning radius.

The procedure for training the neural network is shown in Fig. 2.10. We trained the neural network on the augmented data set with randomly transformed images. For the loss function, we used the Least Square (L2) error between the steering label and model prediction.

$$Loss(\pi_{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \|\pi_{\theta}(s_n) - a_n\|^2$$
 (2.3)

Neural network weights were updated at each batch step using the stochastic gradient descent (SGD) optimization method.

After training, the neural network can generate steering commands using only images from the center camera. Each generated steering command is fed to the controller module, which drives the vehicle.

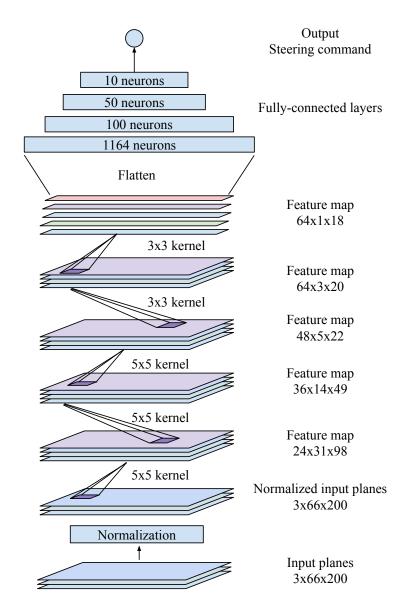


Figure 2.9: Structure of the convolutional neural network used for lane following.

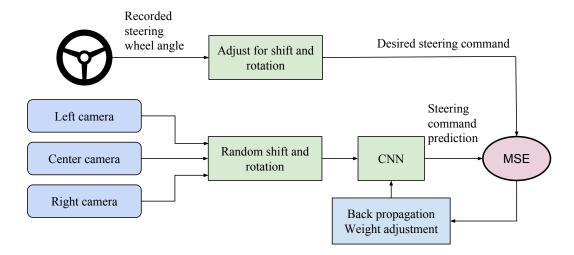


Figure 2.10: Training procedure for the CNN used for lane following.

2.6 Video simulator

We can evaluate actual lane following performance of the model by letting it drive in a real environment; however, it is time-consuming and not safe to perform a road test for every evaluation. Therefore, we need a simulation method by which to evaluate autonomous driving performance in order to generate results quickly without worrying about real-world failures.

Instead of building a sophisticated simulation platform, we proposed to develop a video simulator that simulates trained models performing the lane following task using recorded video. Specifically, an image frame from a recorded video is fed to our trained model to generate a steering command. Then we can apply image transformation on the next frame of that recorded video to simulate driving based on the predicted steering command. Running this process in a loop allows us to measure driving performance in the recorded video. A simplified block diagram of the simulation system is shown in Fig. 2.11.

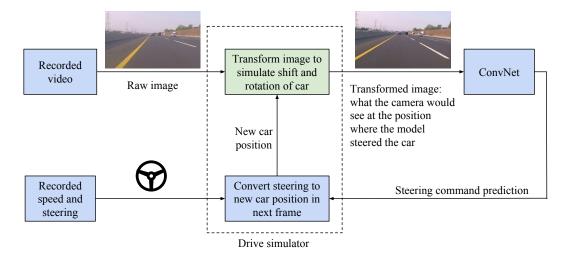


Figure 2.11: Video simulator system.

The simulator takes pre-recorded videos from a forward-facing camera on a human-driven vehicle and generates images that approximate what would appear if the CNN were instead steering the vehicle. These test videos are time-synchronized with recorded steering commands generated by the human driver.

Since human drivers might not drive in the center of the lane at all times, we manually calibrated the lane center associated with each frame used by the simulator. We used this position as the ground truth to measure lane following performance.

The simulator transforms the original images to account for departures from ground truth. Note that this transformation also includes any discrepancy between the human-driven path and the ground truth, and is accomplished by the same methods described in Section 2. The simulator accesses the recorded test video along with the synchronized steering commands that occurred when the video was captured. It sends the first frame of the video, adjusted for any departures from ground truth, to the input of the trained CNN. The CNN then returns a steering command for that frame. The CNN steering commands, as well as the recorded human-driver commands,

are fed into the dynamic model of the vehicle to update its simulated position and orientation.

The simulator then modifies the next frame in the test video so that the image appears as if the vehicle were at the position that resulted from following the CNN-generated steering command. We then feed this new image to the CNN and repeat the process.

The simulator records the off-center distance (distance from the car to the lane center), the yaw, and the distance traveled by the virtual car. When the off-center distance exceeds one meter, a virtual human intervention is triggered, and the virtual vehicle position and orientation are reset to match the ground truth of the corresponding frame of the original test video.

2.7 Evaluation

The model's driving performance is measured using both the video simulator and on-road tests. Video simulator evaluation is usually done first in order to develop a better understanding of the model's performance before carrying out the on-road test. In simulation tests, we have the network provide steering commands for an ensemble of prerecorded test routes that correspond to about a total of three hours and 100 miles of driving in Monmouth County, NJ. The test data was taken in various lighting and weather conditions and included highways, local roads, and residential streets.

2.7.1 Simulation tests

We estimate what percentage of time the network is capable of appropriately driving the car (autonomy), determined by counting simulated human interventions (see Section 2.6). Such interventions occur when the simulated vehicle departs from the center of the lane by more than one meter. We assume that in real life, an actual intervention would require a total of six seconds: this



Figure 2.12: Screenshot of the simulator. The area on the left is filled black since it is unknown due to image transformation. The highlighted wide rectangle below the horizon is the image area that is sent to the CNN.

is the time required for a human to retake control of the vehicle, re-center it, and then restart the self-steering mode. The percentage of autonomy r is defined as follows:

$$r = 1 - \frac{n * T}{t} \tag{2.4}$$

n is the number of human interventions. T is the time for human intervention in seconds, 6 in our case. t is the elapsed time of the simulated test.

2.8 Summary

In this chapter, we present an end-to-end learning approach via supervised learning to train a convolutional neural network. To address the data mismatch issue, we introduce a data augmentation method by generating augmented sensor inputs and steering automatically. For fast evaluation, we developed a video simulator which allows measuring the lane following performance before road tests. The results show that a small amount of training data from less than a hundred hours of driving was sufficient to train a vehicle to drive in different conditions, on highways, local roads, and residential roads in sunny, cloudy, and rainy conditions.

3

End-to-end Autonomous Driving via Imitation Learning

In the previous chapter, we have introduced an end-to-end learning approach via supervised learning for the lane-following task. We address the data distribution mismatch issue by using data augmentation. The data augmentation is done by synthetically generating new samples to cover

the situations where the vehicle does not drive in the center of a lane. The data augmentation works well for lane following as off-center samples can be generated using image transformation and the corresponding steering label can be estimated automatically. However, what if we want to go beyond lane following to do more complex driving tasks such as changing lanes? This requires an alternative approach for data augmentation. In this chapter, we therefore introduce a data aggregation approach to imitation learning to address the data mismatch issue of end-to-end learning for autonomous driving.

Imitation plays a crucial role in human learning. None of us learns in a vacuum; rather, the way we learn is very often a direct result of our observation of those around us. We learn driving skills by imitating what an expert does; we learn language from hearing our parents converse; my two-year-old daughter Claire learns to use a spoon from the way I use mine. This is true regardless of whether we directly imitate or use an idea in a different way, and we do this from a very young age (Williamson et al., 2010). Imitation learning algorithms have been successfully applied to many applications, including playing computer games such as car racing (Ross et al., 2010), building controllers (Coates et al., 2008), and sports analytics (Le et al., 2017). In this chapter, we introduce how to carry out end-to-end learning for autonomous driving via imitation learning in an efficient and safe manner.

3.1 Imitation learning for autonomous driving

In this section, we describe imitation learning in the context of learning a policy for driving a car.

3.1.1 State transition and reward

When an agent, a car in our case, travels around the world, the surrounding environment (the world) is defined as a set of states S. Each state is accompanied by a set of possible actions A(S). According to the state transition function $\delta: S \times A(S) \to S$, any given state $s \in S$ transitions to another state $s' \in S$ when an action $a \in A(S)$ is performed. The transition function may be either deterministic or stochastic.

For each sequence of state-action pairs, there is an associated (accumulated) reward r:

$$r(\Omega = ((s_0, a_0), (s_1, a_1), (s_2, a_2), \ldots)),$$

where $s_t = \delta(s_{t-1}, a_{t-1})$.

A reward may be implicit in the sense that it is observed only when there is a failure. Such a reward comes in the form of a binary value, with 0 corresponding to any unsuccessful run (e.g., crashing into another car so that the car breaks down), while any successful run (e.g., driving indefinitely without crashing) receives a reward. This is the case in which we are interested in. In learning to drive, the reward is simply defined as follows:

$$r(\Omega) = \begin{cases} 1, & \text{if there was no crash,} \\ 0, & \text{otherwise} \end{cases}$$

3.1.2 Policies

A policy is a function that maps a state observation $\phi(s)$ to one a of the actions available A(s) at the state s. An underlying state s describes the surrounding environment perfectly, while a policy

often has only limited access to the state via its observation $\phi(s)$. In the context of end-to-end autonomous driving, s summarizes all necessary information about the road (e.g., number of lanes, existence of other cars or pedestrians), while $\phi(s)$ is, for instance, an image frame captured by a front-facing camera.

We have two separate policies. First, the *primary policy* π is a policy that learns to drive a car, which is also called the *learned policy*. Usually, this policy does not observe the full, underlying state s but only has access to the state observation $\phi(s)$, which is a pixel-level RGB image frame from a front-facing camera. This policy is implemented as a function parameterized by a set of parameters θ .

The other policy is the *reference policy* π^* . This policy may or may not be optimal, but is assumed to be a good policy that we want the primary policy to *imitate*. The reference policy often refers to the policy executed by an expert, thus is also called the *expert policy*. In the context of autonomous driving, a reference policy can be a human driver.

Policy cost

Unlike previous works on imitation learning (e.g. (Daumé Iii et al., 2009; Ross et al., 2010; Chang et al., 2015)), we introduce a concept of cost to a policy. The cost of querying a policy for an appropriate action varies significantly based on how the policy is implemented. For instance, it is expensive to query a reference policy if it is a human driver. On the other hand, it is much cheaper to query a primary policy, which is often implemented as a classifier. Therefore, we analyze an imitation learning algorithm in terms of *how many queries it makes to a reference policy*.

3.1.3 Driving

A car is driven by querying a policy for an action based on a state observation $\phi(s)$ at each time step. The policy observes an image frame s from a front-facing camera and returns both the angle of the steering wheel $(u \in [-1, 1])$ and a binary indicator for braking $(b \in \{0, 1\})$. We call this strategy of relying on a single fixed policy a *naive strategy*.

Reachable states

With a set of initial state $S_0^\pi \subset S$, each policy π defines a subset of the reachable states S^π . That is, $S^\pi = \bigcup_{t=1}^\infty S_t^\pi$, where $S_t^\pi = \left\{ s \middle| s = \delta(s', \pi(\phi(s'))) \middle| \forall s' \in S_{t-1}^\pi \right\}$. In other words, a car driven by a policy π will only visit the states in S^π .

We use S^* to refer to a set reachable by the reference policy. When learning to drive, this reference set is intuitively smaller than that of any other reasonable, non-reference policy. This happens because the reference policy avoids any state that is likely to lead to a low reward, such as crashing into other cars and roadblocks or driving off the road.

3.1.4 Supervised learning

Imitation learning aims to find a primary policy π_{θ} that imitates a reference policy π^* . As we discussed in Chapter 2, we can carry out imitation learning as supervised learning, which is called behavior cloning. In supervised learning, a car is first driven by a reference policy while collecting the state observations $\phi(s)$ of the visited states, resulting in $D = \{\phi(s)_1, \phi(s)_2, \dots, \phi(s)_N\}$.

Based on this dataset, we define a loss function as

$$l_{\text{supervised}}(\pi, \pi^*, D) = \frac{1}{N} \sum_{n=1}^{N} \|\pi(\phi(s)_n) - \pi^*(\phi(s)_n)\|^2.$$
 (3.1)

Then, a desired primary policy can be written as $\hat{\pi} = \arg\min_{\pi} l_{\text{supervised}}(\pi, \pi^*, D)$.

A major issue with this supervised learning approach stems from the imperfection of the primary policy $\hat{\pi}$ even after training. This imperfection likely leads the primary policy to a state s that is not included in the reachable set S^* of the reference policy, i.e., $s \notin S^*$. As this state cannot have been included in the training set $D \subseteq S^*$, the behavior of the primary policy becomes unpredictable. Such imperfection arises from many possible factors, including sub-optimal loss minimization, bias in the primary policy, stochastic state transition, and partial observability.

3.2 DAgger: Beyond supervised learning

A major characteristic of the supervised learning approach described above is that only the reference policy π^* generates training examples. A direct consequence of this is that the training set is almost a subset of the reference reachable set S^* . This issue can be addressed by imitation learning or learning-to-search (Daumé Iii et al., 2009; Ross et al., 2010).

In an imitation learning framework, the primary policy, which is currently being estimated, is used in addition to the reference policy when generating training examples. The overall training set used to tune the primary policy then consists of both the states reachable by the reference policy and those reachable by the intermediate primary policies. This makes it possible for the primary policy to correct its path toward a good state when it visits a state unreachable by the reference policy, i.e., $s \in S^{\pi} \backslash S^*$.

DAgger is one such imitation learning algorithm, proposed in (Ross et al., 2010). This algorithm fine-tunes a primary policy initially trained with the supervised learning approach described earlier. Let D_0 and π_0 be the supervised training set (generated by a reference policy) and the initial primary policy trained in a supervised manner. Then, DAgger iteratively performs the following steps: At each iteration i, first, additional training examples are generated from a mixture of the reference π^* and primary π_{i-1} policies (i.e.

$$\beta_i \pi^* + (1 - \beta_i) \pi_{i-1} \tag{3.2}$$

) and combined with all the previous training sets: $D_i = D_{i-1} \cup \{\phi(s)_1^i, \dots, \phi(s)_N^i\}$. The primary policy is then fine-tuned, or trained from scratch, by minimizing $l_{\text{supervised}}(\theta, D_i)$ (see Eq. (3.1).) This iteration continues until the supervised cost on a validation set stops improving.

DAgger does not rely on the availability of explicit reward; this makes it suitable for building an end-to-end autonomous driving model that drives on the road indefinitely. However, it is certainly possible to incorporate an explicit reward with other imitation learning algorithms, such as SEARN (Daumé Iii et al., 2009), AggreVaTe (Ross and Bagnell, 2014) and LOLS (Chang et al., 2015). Although we focus on DAgger in this section, our proposal also applies generally to any learning-to-search type of imitation learning algorithm.

Limitations of DAgger

At each iteration, DAgger queries the reference policy for each and every collected state. In other words, the cost of DAgger C_i^{DAgger} at the i-th iteration is equivalent to the number of training examples collected, i.e. $C_i^{\mathrm{DAgger}} = |D_i|$. In total, the cost of DAgger for learning a primary policy is $C^{\mathrm{DAgger}} = \sum_{i=1}^M |D_i|$, excluding the initial supervised learning stage.

This high cost of DAgger comes with a more practical issue when the reference policy is a human operator, or in our case a human driver. First, as noted in (Ross et al., 2013), a human operator cannot drive well without actual feedback, which is the case when using DAgger as the primary policy drives most of the time. This leads to suboptimal labelling of the collected training examples. Furthermore, this constant operation easily exhausts a human operator, making it difficult to scale the algorithm toward more iterations.

Another issue is that it is not safe to run DAgger in real-world road tests since it may seriously damage the environment, especially in the beginning when the reference policy π_{θ} is not good.

3.3 SafeDAgger

We propose an extension of DAgger that minimizes the number of queries made to the reference policy both during training and testing. In this section, we describe this extension, called *SafeDAgger*, in detail.

3.3.1 Safety classifier

Distinct from previous approaches to imitation learning, often as learning-to-search (Daumé Iii et al., 2009; Ross et al., 2010; Chang et al., 2015), we introduce a classifier c_{safe} , to which we refer as a *safety classifier*. This classifier takes as input both the partial observation of a state $\phi(s)$ and a primary policy π and returns a binary label indicating whether the primary policy π is likely to deviate from a reference policy π^* without querying it.

We define the deviation of a primary policy π from a reference policy π^* as

$$\epsilon(\pi, \pi^*, \phi(s)) = \|\pi(\phi(s)) - \pi^*(\phi(s))\|^2.$$

Note that the error metric can be flexibly chosen depending on the target task. For autonomous driving, the metric can be any information that is helpful in determining whether a given state is safe or not. For instance, we can use the distances between the autonomous vehicle and the surrounding objects. We simply use the L_2 distance between the reference steering angle and the predicted steering angle, ignoring the brake indicator.

Then, with this defined deviation, the optimal safety classifier c_{safe}^* is defined as

$$c_{\text{safe}}^*(\pi, \phi(s)) = \begin{cases} 0, & \text{if } \epsilon(\pi, \pi^*, \phi(s)) > \tau \\ 1, & \text{otherwise} \end{cases}$$
(3.3)

where τ is a predefined threshold. The safety classifier decides whether the choice made by the policy π at the current state can be trusted with respect to the reference policy. We emphasize again that this determination is done without querying the reference policy.

Learning

A safety classifier is *not* given, meaning that it needs to be estimated during learning. A safety classifier c_{safe} can be learned by collecting another set of training examples:*

$$D' = \{\phi(s)'_1, \phi(s)'_2, \dots, \phi(s)'_N\}$$

^{*}It is possible to simply set aside a subset of the original training set for this purpose.

The safety classifier is trained to minimize a binary cross-entropy loss defined as follows:

$$l_{\text{safe}}(c_{\text{safe}}, \pi, \pi^*, D') = -\frac{1}{N} \sum_{n=1}^{N} c_{\text{safe}}^*(\phi(s)'_n) \log c_{\text{safe}}(\phi(s)'_n, \pi) + (1 - c_{\text{safe}}^*(\phi(s)'_n)) \log(1 - c_{\text{safe}}(\phi(s)'_n, \pi)),$$
(3.4)

where we model the safety classifier as returning a Bernoulli distribution over $\{0, 1\}$.

Driving: Safe strategy

The naive strategy is a default go-to strategy in most applications of reinforcement learning or imitation learning. We can improve on the naive strategy by designing a safe strategy that utilizes the proposed safety classifier c_{safe} . In this strategy, the safety classifier determines at each point in time whether it is safe to let the primary policy drive. If so (i.e. $c_{\text{safe}}(\pi, \phi(s)) = 1$,) we use the action returned by the primary policy (i.e. $\pi(\phi(s))$.) If not (i.e. $c_{\text{safe}}(\pi, \phi(s)) = 0$,) we let the reference policy drive instead (i.e. $\pi^*(\phi(s))$.)

Assuming a good safety classifier is available, this strategy avoids any dangerous situation arising from an imperfect primary policy that may lead to a low reward (e.g., a crash.) In the context of learning to drive, this safe strategy can be thought of as letting a human driver take control based on an automated decision.[†] Note that this driving strategy is applicable regardless of the learning algorithm used to train the primary policy.

[†]Such intervention has been carried out manually by a human driver (Pomerleau, 1992).

Discussion

The proposed safety classifier has the potential to address the unsafety issue up to a certain point. First, since a separate training set is used to train the safety classifier, it is more robust to unseen states than the primary policy. Second and more importantly, the safety classifier finds and exploits a simpler decision boundary between safe and unsafe states instead of trying to learn a complex mapping from state observation to control variable. For instance, in learning to drive, the safety classifier may simply learn to distinguish between a crowded road and an empty road and determine that it is safer to let the primary policy drive on an empty road.

However, the safety classifier is not without problem. A major potential issue with the safe strategy of driving is that the safety classifier may pass control to the reference policy, or a human driver, most of the time, defeating the whole purpose of the model learning to drive. This tendency can be mitigated by setting the threshold τ to a lower value, but doing so inevitably results in the unsafe driving experience. Later on, however, we show that SafeDAgger automatically decreases the number of queries to the reference policy as learning progresses (with a fixed τ).

Relationship to a value function

A value function $V^{\pi}(s)$ in reinforcement learning computes the reward a given policy π can achieve in the future when starting from a given state s (Sutton and Barto, 1998). This description reveals a clear connection between the safety classifier and the value function. The safety classifier $c_{\text{safe}}(\pi,s)$ determines whether a given policy π is likely to fail if it operates at a given state s, in terms of deviation from the reference policy. By assuming that a reward is only given at the very end of a policy run, and that the reward is 1 if the current policy acts exactly like the reference policy and otherwise 0, the safety classifier precisely returns the value of the current state.

A natural question that follows is whether the safety classifier can drive a car on its own or not. This perspective on the safety classifier as a value function suggests a way to directly use the safety classifier to drive a vehicle. At a given state s, the best action \hat{a} can be selected to be $\arg\max_{a\in A(s)}c_{\text{safe}}(\pi,\delta(s,a))$. This is however not possible under the current formulation, as the transition function δ is unknown. We may extend the definition of the proposed safety classifier so that it considers a state-action pair (s,a) instead of a state alone and predicts the safety in the next time step, which makes it closer to a Q function. This still encounters an issue when the action is continuous, such as the steering wheel angle, as it requires us to run a potentially expensive optimization routine.

Algorithm 1 SafeDAgger Blue fonts highlight differences from vanilla DAgger.

```
1: Collect D_0 using a reference policy \pi^*
 2: Collect D_{\text{safe}} using a reference policy \pi^*
 3: \pi_0 = \arg\min_{\pi} l_{\text{supervised}}(\pi, \pi^*, D_0)
 4: c_{\text{safe},0} = \arg\min_{c_{\text{safe}}} l_{\text{safe}}(c_{\text{safe}}, \pi_0, \pi^*, D_{\text{safe}} \cup D_0)
 5: for i = 1 \text{ do} M
            Collect D' using the safe strategy \pi_{i-1} and c_{\text{safe},i-1}
            Subset Selection:
 7:
                 D' \leftarrow \{ \phi(s) \in D' | c_{\text{safe},i-1}(\pi_{i-1},\phi(s)) = 0 \}
            D_i = D_{i-1} \cup D'
 8:
            \pi_i = \arg\min_{\pi} l_{\text{supervised}}(\pi, \pi^*, D_i)
            c_{\text{safe},i} = \arg\min_{c_{\text{safe}}} l_{\text{safe}}(c_{\text{safe}}, \pi_i, \pi^*, D_{\text{safe}} \cup D_i)
10:
11: end for
12: return \pi_M and c_{\text{safe},M}
```

3.3.2 SafeDAgger: Safety classifier in the loop

We describe here the proposed SafeDAgger, presented in Alg. 1, which aims to reduce the number of queries to a reference policy during iterations. At the core of SafeDAgger lies the safety classifier

introduced earlier in this section. This implementation incorporates two major modifications from the original DAgger.

First, we use the safe strategy, instead of the naive strategy, to collect training examples (line 6 in Alg. 1). This allows an agent to simply give up and hand over the control to the reference policy when it is not safe to drive itself, thereby collecting training examples with a much further horizon without crashing. This would have been impossible with the original DAgger unless a manually-forced take-over measure was implemented (Ross et al., 2013).

Furthermore, unlike the original DAgger's random switch in Eq. (3.2), the proposed approach often gives a longer-term window of control to the reference policy, thereby making it more suitable for a human driver.

Second, the subset selection (line 7 in Alg. 1) drastically reduces the number of queries made of the reference policy. Only the small subset of states where the safety classifier returns 0 need to be labelled with reference actions. This is in contrast to the original DAgger, where all collected states had to be queried against the reference policy. The cost of the SafeDAgger at each iteration, in the context of reference queries, is $C_i^{\text{SafeDAgger}} = O(\tilde{\tau})C_i^{\text{DAgger}}$, where $\tilde{\tau} = \frac{\tau}{\max_{s \in S} \epsilon(\pi, \pi^*, \phi(s))} \leq 1$.

Furthermore, this subset selection allows subsequent supervised learning to focus more on difficult cases, which almost always correspond to the states that are problematic (i.e., $S \setminus S^*$.) This reduces the total number of training examples without losing important ones, thereby making this algorithm data-efficient.

Once the primary policy is updated with D_i , which is the union of the initial training set D_0 and all difficult examples collected so far, we update the safety classifier. This step ensures that the safety classifier correctly identifies which states are difficult/dangerous for the latest primary policy. This has an effect of automated curriculum learning (Bengio et al., 2009) with a mixed

strategy (Zaremba and Sutskever, 2014), where the safety classifier selects training examples of appropriate difficulty at each iteration.

Despite these differences, the proposed SafeDAgger inherits many of the theoretical guarantees from DAgger. This is achieved by gradually increasing the threshold τ of the safety classifier (Eq. (3.3)). If $\tau > \epsilon(\pi, \phi(s))$ for all $s \in S$, SafeDAgger reduces to the original DAgger with β_i (from Eq. (3.2)) set to 0. However, we later observe empirically that this is not necessary, and that training with the proposed SafeDAgger with a fixed τ automatically and gradually reduces the portion of the reference policy during data collection over iterations.

3.4 Experiment

3.4.1 Simulation environment

We use TORCS (tor, accessed May 12, 2016), a racing car simulator, for empirical evaluation. We chose TORCS for the following reasons: First, it has been used widely and successfully as a platform for research on autonomous racing (Loiacono et al., 2008), although most of the previous work, except for (Koutník et al., 2013; Chen et al., 2015), is not comparable as radars instead of cameras were used for observing states. Second, TORCS is a light-weight simulator that can be run on an off-the-shelf workstation. Third, TORCS is open-source software, making it easy to interface with other software, in our case the program Torch.

The original TORCS was designed as a car racing game, where racing cars can drive freely within the track and even hit each other for overtaking. In order to incorporate normal driving environments such as highway driving, we needed to modify the TORCS by creating the lane markets and programming the car agent to follow pre-defined traffic rules.



Figure 3.1: Representative road textures and lane markers used with TORCs.

Tracks

To simulate highway driving with multiple lanes, we modified the original TORCS road surface textures by adding various lane configurations that incorporated different numbers and types of lanes. The maximum number of lanes for a track is three, which is enough to demonstrate changing lanes. Fig. 3.1 shows representative lane configurations.

We used ten different tracks in total for our experiments, splitting them into two disjointed sets: seven training tracks and three test tracks. All training examples as well as validation examples were collected from the training tracks only, and the trained primary policy was tested on the test tracks.

The ten tracks shown in Fig. 3.2 contain different road textures, different lane configurations, various weather conditions, and different surrounding environments, providing a rich simulation environment for autonomous driving.

Reference policy π^*

We implement our own reference policy, which has access to an underlying state configuration. The state includes car position, car heading direction, car speed, and distances to other cars. The reference policy follows a simple traffic rule. Cars driven by the reference policy either follow the current lane (accelerating up to the speed limit), change lanes if there is a slower car in front and

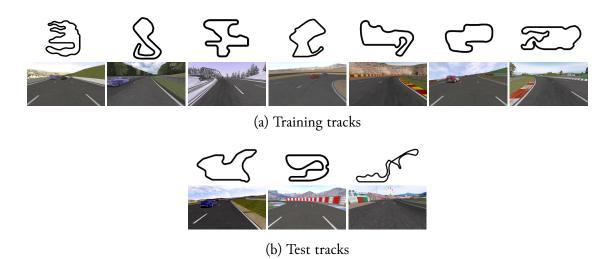


Figure 3.2: Training and test tracks with sample frames.

an available lane to the left or right, or brake to avoid occlusion. The reference policy is defined in Alg. 2.

We follow the TORCS robot car tutorial (Wymann, 2006) to program the car to follow the reference policy via computing the desired steering, acceleration, and brake commands. Once a target lane is set, a desired steering command s at each time step is calculated with Eq. (3.5) so that the car follows the target lane.

$$s = \left(\theta - \frac{|L_c - L_t|}{w_{\text{road}}}\right) * C \tag{3.5}$$

 θ is the angle between the lane direction and car direction. L_c represents the coordinate of the lane we are driving on. L_t presents the coordinate of the target lane we want to change to. $|L_c - L_t|$ is the distance between the current lane and the target lane. w_{road} the width of the road. C is a pre-defined parameter to control how quickly we want to make the change.

Algorithm 2 Reference Policy

```
1: initialize the target lane, car speed, and car position randomly within the track
2: while simulator is running do
       compute the distance to the preceding cars in each lane
3:
4:
       if distance to the preceding car in the same lane within certain threshold then
5:
           if left lane exists and left lane change available then
6:
               set target lane to be left lane
7:
           else if right lane exists and right lane change available then
               set target lane to be right lane
8:
           else
9:
               Slow down
10:
           end if
11:
       end if
12:
13:
       compute steering command
       compute acceleration/brake command
14:
15: end while
```

The desired speed of the vehicle is determined by road curvature and the distance to other cars. The car speed controller controls the brake and acceleration paddle so that the car adheres to the desired speed. To achieve smooth car-following behavior, we use the simplified velocity car-following model (Newell, 1961) to compute the desired speed as shown in Eg. (3.6)

$$v_d = v_{\text{max}} (1 - \exp\frac{-c * d}{v_{\text{max}}})$$
 (3.6)

d is the distance to the car in front, and c is a pre-defined parameter to control how quickly the car changes speed. $v_{\rm max}$ is the maximum car speed allowed on the track. We randomly initialize a maximum speed $v_{\rm max}$ for each car, therefore a great deal of overtaking occurs during the simulation, which produces many interesting traffic situations.

3.4.2 Data collection

To collect data, we use a car in TORCS driven by a pre-defined reference policy. To simulate traffic on each training track, we add 40 cars driven by the same reference policy. In addition to the initial supervised learning stage, we run up to three iterations. In the case of SafeDAgger, we collect 30k, 30k, and 10k training examples (after the subset selection in line 6 of Alg. 1.) In the case of the original DAgger, we collect up to 390k samples each iteration and uniform-randomly select 30k, 30k, and 10k training examples.

Instead of using only the initial training set D_{safe} to train the safety classifier network, we also use the training data D_i .

This data collection strategy was designed to keep consistent the amount of data used for training a primary policy network. Note that the number of queries to the reference policy is much higher (up to 39-fold) with the original DAgger, as it queries both primary and reference policy simultaneously for every single image frame.

Data representation

The collected data contains videos recorded by a front-facing camera in the game, and labels corresponding to each video frame. When used as input for the neural network, the video frames are scaled and cropped to 160×72 pixel images with three color channels (red, green and blue). Sample images from each track are shown in Fig. 3.2. The label contains twelve variables, given in the following list:

- 1. $S_c \in [-1,1]$: angle of the steering wheel
- 2. $I_b \in \{0,1\}$: if the car is braking

- 3. $I_{ll} \in \{0,1\}$: if there is a lane to the left
- 4. $I_{lr} \in \{0,1\}$: if there is a lane to the right
- 5. $I_{cl} \in \{0,1\}$: if there is a car ahead in the left lane
- 6. $I_{cm} \in \{0,1\}$: if there is a car ahead in the same lane
- 7. $I_{cr} \in \{0,1\}$: if there is a car ahead in the right lane
- 8. $D_{cl} \in \mathbb{R}$: distance to the car ahead in the left lane
- 9. $D_{cm} \in \mathbb{R}$: distance to the car ahead in the same lane
- 10. $D_{cr} \in \mathbb{R}$: distance to the car ahead in the right lane
- 11. $P_c \in [-1, 1]$: position of the car within the lane
- 12. $A_c \in [-1, 1]$: angle between the direction of the car and the direction of the lane

The first two variables in the list, both underlined, are driving command variables. The other ten variables are state configurations that are observed only by the reference policy; they are hidden to both the primary policy and safety classifier. All variables are used as target labels during training, but only the driving commands (steering S_c and braking I_b) are used to drive a car during testing.

3.4.3 Policy networks

Primary policy π_{θ}

We use a deep convolutional neural network that has five convolutional layers followed by a set of fully-connected layers. This convolutional network takes as input a pixel-level image from a front-facing camera. It predicts the angle of the steering wheel ([-1,1]) and whether to brake

Input - 3×160×72											
Conv1 - 64×3×3											
Max Pooling - 2×2											
Conv2 - 64×3×3											
Max Pooling - 2×2											
Conv3 - 64×3×3											
Max Pooling - 2×2											
Conv4 - 64×3×3											
Max Pooling - 2×2											
<u>Conv5</u> - 128×5×5											
FC-2	FC-2	FC-2	FC-2	FC-2	FC-2	FC-64 FC-1	FC-1	FC-1	FC-1	FC-1	FC-1
I_{ll}	I_{lr}	I_{cl}	I_{cm}	I_{cr}	$\underline{I_b}$	$\underline{S_c}$	D_{cl}	D_{cm}	D_{cr}	P_c	A_c

Table 3.1: Configuration of a primary policy network. Each convolutional layer is denoted by "Conv - # channels \times height \times width". Max pooling without overlap follows each convolutional layer. We use rectified linear units for point-wise nonlinearities. Only the shaded part of the full network is used during tests.

 $(\{0,1\})$. Furthermore, the network predicts as an auxiliary task the car's affordances, including the existence of a lane to the left or right and the existence of another car to the left, right, or in front of the car. We have found this multi-task approach to easily outperform a single-task network, confirming the promise of multi-task learning from (Caruana, 1997). The details of the convolutional neural network are shown in Table 3.1.

Safety classifier c_{safe}

We use a feed-forward network to implement a safety classifier. The activation of the primary policy network's last hidden convolutional layer is fed through two fully-connected layers followed by a softmax layer with two categories corresponding to 0 and 1. We choose $\tau=0.0025$ as our safety classifier threshold so that approximately 20% of initial training examples are considered

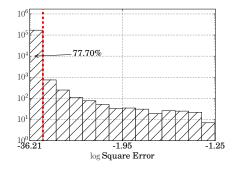


Figure 3.3: The histogram of the log square errors of the steering angle after *supervised* learning only. The dashed line is located at $\tau = 0.0025$. 77.70% of the training examples are considered safe.

unsafe, as shown in Fig. 3.3.

In Fig. 3.4, we present twenty sample frames. The top ten frames were considered *safe* (0) by a trained safety classifier, while the bottom ones were considered *unsafe* (1). At this point, it seems that the safety classifier determines the safety of a current state observation based on two criteria: (1) the existence of other cars, and (2) whether entering a sharp curve.

3.4.4 Evaluation

Training and driving strategies

We mainly compare three training strategies; (1)Supervised Learning, (2) DAgger (with $\beta_i = \mathbf{I}_{i=0}$) and (3) SafeDAgger. For each training strategy, we evaluate trained policies with each of two driving strategies: (1) naive and (2) safe.

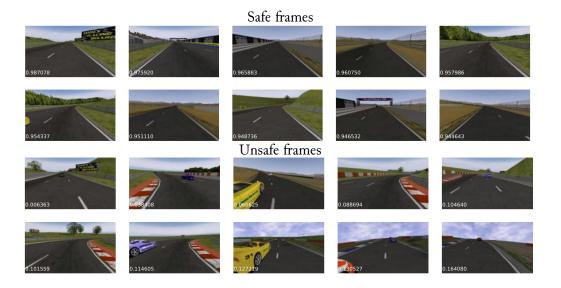


Figure 3.4: Sample image frames sorted according to a safety classifier trained on a primary policy right after the supervised learning stage. The number in each frame is the probability of the safety classifier returning 1.

Evaluation metrics

We evaluate each combination by letting it drive up to three laps on the three test tracks. All runs are repeated under two conditions, with and without traffic, while recording three metrics. The first metric is the number of completed laps in which the car did not go outside the track, averaged over all three tracks. When a car drives out of the track, we immediately halt. Second, we look at the damage accumulated while driving. Damage occurs each time the car bumps into another car. Instead of a raw, accumulated damage level, we report the damage per lap. Lastly, we report the mean squared error of steering angle, computed while the primary policy drives.

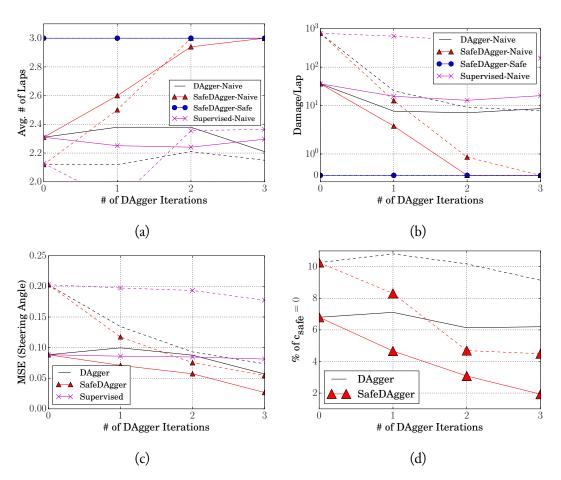


Figure 3.5: (a) Average number of laps (\uparrow), (b) damage per lap (\downarrow), (c) the mean squared error of steering angle for each configuration (training strategy–driving strategy) over each iteration and (d) the portion of time driven by the reference policy during testing. We use dashed and solid curves for the cases with and without traffic, respectively.

3.5 Results and analysis

In Fig. 3.5, we present the results in terms of both average laps and damage per lap. The first thing we notice is that a primary policy trained using supervised learning alone (the 0-th iteration) works perfectly when used in tandem with a safety classifier. The safety classifier switched to the reference policy for 7.11% and 10.81% of the time without and with traffic during the test.

Second, in terms of both metrics, the primary policy trained with the proposed SafeDAgger makes much faster progress than the original DAgger. After the third iteration, the primary policy trained with SafeDAgger is perfect. We conjecture that this is due to the automated curriculum learning effect. Furthermore, the examination of the mean squared difference between the primary policy and the reference policy reveals that SafeDAgger brings the primary policy closer to the reference policy more rapidly.

As a baseline, we show the performance of a primary policy trained using purely supervised learning in Fig. 3.5 (a)–(b). The performance clearly illustrates that supervised learning alone cannot train a primary policy well even when an increasing number of training examples are presented.

In Fig. 3.5 (d), we observe that the portion of time the safety classifier switches to the reference policy while driving decreases as the SafeDAgger iteration progresses. We conjecture that this happens because SafeDAgger encourages the primary policy's learning to focus on those cases deemed difficult by the safety classifier. When the primary policy was trained with the original DAgger (which does not take into account the difficulty of each collected state), the rate of decrease was much smaller. Essentially, using the safety classifier and SafeDAgger together results in a virtuous cycle of fewer and fewer queries to the reference policy during both training and testing.

predict the deviation of a primary policy from the reference policy one second in advance. We observe a similar trend, which makes the SafeDAgger a realistic algorithm for deployment in practice.

3.6 Summary

In this chapter, we have proposed an extension of DAgger, called SafeDAgger, which allows a primary policy to learn without catastrophic experience. We first introduced a safety classifier that prevents a primary policy from falling into a dangerous state by automatically switching between it and the reference policy *without* querying the reference policy. This safety classifier is used during data collection stages in the proposed SafeDAgger, which can collect a set of progressively difficult examples while minimizing the number of queries to the reference policy. Extensive experiments with simulated autonomous driving showed that SafeDAgger not only queries the reference policy less but also trains the primary policy more efficiently.

4

Variational Mixture Density Networks

As illustrated in the previous chapter, the goal of imitation learning is to train a primary policy to mimic expert demonstrations without access to an explicit reward signal. Expert demonstrations provided by humans, however, often show significant variability due to latent factors such as different driving preferences of different human drivers. It is also the case that the same person may make different choices when encountering similar situations. For example, in an overtaking

scenario, a human driver can choose to pass via either the left or the right lane. A model trained by minimizing the square errors between such expert actions and predicted actions may drive the car into a dangerous situation at test time. In the example of overtaking shown in Fig. 4.1, the model trained in such way outputs actions that will cause the car to crash into the vehicle ahead.

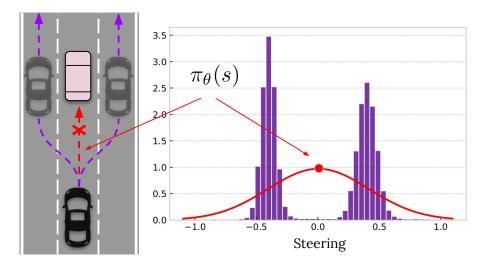


Figure 4.1: The uncertainty of driving actions in an overtaking scenario: the car can pass the vehicle ahead either on the left or the right. The average of these two actions, which would be learned by a model $\pi_{\theta}(s)$ trained to minimize the square errors of driving actions between expert demonstrations and model predictions, will cause the car to crash into the vehicle ahead.

In this chapter, we will introduce a proposed approach, variational mixture density networks (VMDN) with a categorical latent variable to address the expert demonstration uncertainty issue. Our experiences with this method show that it can not only mimic expert demonstrations but also capture their underlying latent structure.

4.1 Mixture density networks

Before introducing our proposed variational mixture density network, we will first discuss how to address this driving actions uncertainty issue by learning the conditional probability distribution of driving actions with a Gaussian Mixture Model (GMM) using Mixture Density Networks (MDN) (Bishop, 1994). The mixture density networks (Bishop, 1994) is defined as the weighted sum of a number of component densities. The component densities are chosen from a particular parametric class of densities, such as Gaussian, which is considered to model the data distribution in hand. A k components mixture density network is defined as:

$$p(y;\theta) = \sum_{k=1}^{K} \pi_k p(y;\theta_k), \tag{4.1}$$

where $p(y;\theta_k)$ is denoted as the k-th component density and θ_k represents the density parameters. We use π_k to denote the weight for the k-th component in the mixture densities. The weights π_k can be viewed as the probability p(k) that a data sample will be drawn from the component k, which must satisfy the following properties: (i) $\pi_k \in [0,1]$, (ii) $\sum_{k=1}^K \pi_k = 1$. θ contains all parameters $\{\theta_1, ..., \theta_k, \pi_1, ..., \pi_K\}$.

Supposing we use Gaussian components, the mixture density network for approximating the conditional distribution p(a|s) is defined as:

$$p(a|s) = \sum_{k=1}^{K} \pi_k(s) \mathcal{N}(a|\mu_k(s), \operatorname{diag}(\sigma_k^2(s))), \tag{4.2}$$

s is the state, which is usually represented as sensor inputs, and a is the driving action.

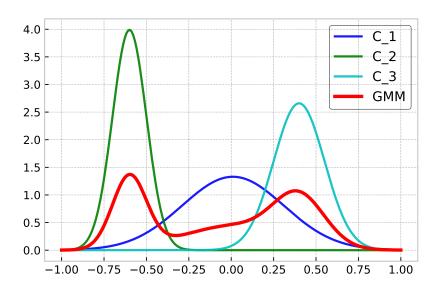


Figure 4.2: Gaussian mixture model with three components. The red curve shows the mixture densities.

4.1.1 Model approach

We can use the outputs of a conventional neural network to govern the parameters of the Gaussian mixture model. The output unit activations of the network are denoted as $\mathbf{a} = \{\mathbf{a}^{\pi}, \mathbf{a}^{\mu}, \mathbf{a}^{\sigma}\}$. If there are K components, the output unit activations can be viewed as a group of: a_k^{π} that determines the coefficients $\pi_k(s)$, a_k^{σ} that determines the variances $\sigma_k(s)$, and a_k^{μ} that determines the means $\mu_k(s)$.

The constraints for the mixing coefficients can be achieved using softmax outputs:

$$\pi_k(s) = \frac{exp(a_k^{\pi})}{\sum_{l=1}^{K} exp(a_l^{\pi})}$$
(4.3)

The variances must satisfy $\sigma_k \geq 0$ and can be represented using the exponentials of the activations a_k^{σ}

$$\sigma_k(s) = \exp(\sigma_k^{\sigma}) \tag{4.4}$$

Finally, as there are no constraints on the means $\mu_k(x)$, we just represent them with the network output activations:

$$\mu_k(s) = a_k^{\mu} \tag{4.5}$$

The mixture density network, which contains the parameters denoted as w, is trained to minimize the negative logarithm of the conditional likelihood of correct driving commands a_i^* :

$$\mathcal{L}(\boldsymbol{w}) = -\sum_{n=1}^{N} \log \left\{ \sum_{k=1}^{K} \pi_k(s_n, \boldsymbol{w}) \mathcal{N}(a_n^* | \mu_k(a_n, \boldsymbol{w}), \sigma_k(a_n, \boldsymbol{w})) \right\}$$
(4.6)

The model architecture for the mixture density network is shown in Fig. 4.3

Then we can train this mixture network with the stochastic gradient descent (SGD) method.

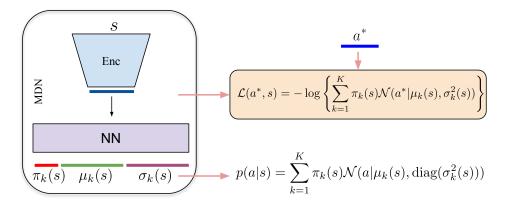


Figure 4.3: Model architecture for a mixture density network.

4.1.2 Inference

In order to let the model drive the vehicle, we need to generate the driving action a. One method is to find the most likely solutions based on the conditional density function $p(a|s_n)$, which can be predicted by a trained mixture density network given the sensor inputs s. However, this method requires expensive numerical iteration, which is not suitable for real-time driving tasks. Another method, one of the simplest, is to calculate the conditional average of driving commands. This is given by:

$$\mathbb{E}[a|s] = \sum_{i=1}^{K} \pi_k(s) \mu_k(s)$$

However, as we discussed at the beginning of this chapter, the conditional distribution for the driving model is supposed to be a multimodal distribution. We cannot directly use the average of valid driving actions because that average is not necessarily itself a valid driving action and may lead to a dangerous situation.

A simple alternative method is to first select the component with the largest mixing coefficient, then select the mean of the selected component.

$$\hat{a} = \mu(s, \hat{k})$$
, where $\hat{k} = \arg\max_{k} \pi_k(s)$

This is clearly suboptimal, but can be computed efficiently during the test time.

Limitations

Before moving on to the proposed variational mixture density network, it is worth discussing the limitation of the mixture density networks. We observe that the presence of the summation over the number of components k appears inside the logarithm in the MDN loss function 4.6. By maximizing the logarithm likelihood of conditional probability p(a|s), one component of the GMM may 'collapse' on a particular sample, which causes the logarithm likelihood to go to infinity (Bishop, 2006, p.434). In addition, there is an issue of numerical instability when training a mixture density network, for example arithmetic underflow, which has been discussed by many researchers (Almahairi, 2014; Bonnett, 2016). A number of tricks such as gradient clipping and variance clipping need to be used to stabilize the training process (Guillaumes, 2017). Conversely, instead of directly maximizing the logarithm likelihood, we can maximize the variational lower bound on the logarithm likelihood. By doing so, we will show that the logarithm function acts directly on the Gaussian, which avoids the numerical instability issue.

4.2 Variational mixture density networks

Before introducing our proposed variational mixture density network, we first discuss the variational inference framework.

4.2.1 Variational inference framework

For any given choice of a distribution q(z) defined over the latent variable z, the following decomposition of p(a|s) holds:

$$\log p(a|s) = \mathcal{L}_{\text{ELBO}}(q) + D_{\text{KL}}(q(z)||p(z|s,a))$$
(4.7)

where we have:

$$\mathcal{L}_{\text{ELBO}}(q) = \sum_{z} q(z) \log\{\frac{p(a, z|s)}{q(z)}\}$$
(4.8)

$$D_{KL}(q||p) = -\sum_{z} q(z) \log\{\frac{p(z|s,a)}{q(z)}\}$$
 (4.9)

 $\mathcal{L}_{\text{ELBO}}$ is often called the variational lower bound or evidence lower bound. Since the Kullback–Leibler divergence $D_{\text{KL}}(q||p) \geq 0$, it follows that $\log p(a|s) \geq \mathcal{L}_{\text{ELBO}}$. Maximizing the variational variational lower bound $\mathcal{L}_{\text{ELBO}}$ will allow us to maximize the likelihood function. The variational lower bound $\mathcal{L}_{\text{ELBO}}$ can be further decomposed as:

$$\mathcal{L}_{\text{ELBO}} = \sum_{z} q(z) \log \left\{ \frac{p(a, z|s)}{q(z)} \right\}$$

$$= \mathbb{E}_{q(z)} [\log p(a|s, z)] - D_{\text{KL}}(q(z)||p(z|s))$$

$$= \sum_{z} q(z) \log p(a|s, z) - D_{\text{KL}}(q(z)||p(z|s))$$

Relative to the likelihood in the MDN loss function Eq. 4.6, the summation over the components k and the logarithm have been interchanged. Now we can use the neural network to govern the parameters of the prior distribution q(z|s), the approximated posterior distribution q(z), and the conditional distribution p(a|s,z).

4.2.2 Variational mixture density networks

In our proposed variational mixture density network, we use a discrete variable $Z \in \{1, \dots, K\}$ as the latent variable. We then use a neural network $\alpha(s)$ to model the conditional prior over the latent variable:

$$p(z|s) = \text{Categorical}(\alpha_1(s), \dots, \alpha_K(s)),$$

and use another neural network $\beta(s,a)$ to approximate a conditional posterior over the latent variable:

$$q(z|s, a) = \text{Categorical}(\beta_1(s, a), \dots, \beta_K(s, a)).$$

The likelihood function is then defined as

$$p(a|s, z) = \mathcal{N}(a|\mu(s, z), \operatorname{diag}(\sigma^2(s, z))).$$

We can then write the variational lower bound to be the marginal, conditional log-probability of a given s as

$$\mathcal{L}(s, a) = \mathbb{E}_{p(z|s, a)}[\log p(a|s, z)] - D_{\text{KL}}(q(z|s, a) || p(z|s))$$

If we fix the variances in the likelihood function to be constant $\sigma^2(s,z)=$ Const, then the mean $\mu(s,z)$ can be viewed as our primary policy for predicting driving commands $\pi_{\theta}(s,z=k)=\mu(s,z=k)$ and the variational lower bound can be rewritten as:

$$\mathcal{L}(s,a) = \mathbb{E}_{p(z|s,a)}[-\|\pi_{\theta}(s,z) - a\|^{2}] - \delta D_{\text{KL}}(q(z|s,a)\|p(z|s))$$
(4.10)

The first term represents the expectation of prediction loss over the latent variable z. The hyper-parameter δ represents the trade-off between minimizing action prediction error and fitting the prior.

4.2.3 Inference for driving

Based on the method of inferring the latent variable $Z \in \{1, ..., K\}$, we have two strategies by which to apply a trained variational mixture density network to driving. First, we can select

the latent code k by maximizing the learned prior $\hat{k} = \arg\max_k \alpha(s,z=k)$, then predict the driving action using $\hat{a} = \pi_{\theta}(s,z=\hat{k})$ at each time step t. Alternatively, we can manually set the latent variable z at test time to achieve different driving behaviors. This second strategy comes from our intuition for the latent variable. The latent variable z can be viewed as a high-level driving command such as "change into the left lane."

4.2.4 Model architecture

We use three convolutional neural networks $\alpha(s)$, $\beta(s)$, $\pi(x,z)$ to model the three parts p(z|s), p(z|s,a), and $\mu(s,z)$. The frames from sensor inputs are first encoded via a feed-forward convolutional neural network, which is shared across all three parts of the model. The model architecture is shown in Fig. 4.4.

4.2.5 Discussion

The mixture density network can also be viewed as a model with a latent variable. One can think of mixture density as modelling a process in which first a component k is selected according to the multinomial distribution $p(k) = \{\pi_1, ..., \pi_K\}$, and then a sample is drawn from the corresponding component density $p(a|\mu_k(s), sigma_k(s))$. The component density $p(a|\mu_k(s), sigma_k(s))$ can be viewed as a conditional distribution p(a|s, k). Thus, the marginal probability p(a|s) is given by $p(a|s) = \sum_{k=1}^K p(a|s, k)p(k)$. We can think of the component k a latent variable: only action a is observed, and the information in k is missing.

Unlike in MDN, our proposed VMDN model addresses the difficulty in maximizing the likelihood function p(a|s) by maximizing the variational lower bound. By doing so, we can have the logarithm appear inside the summation over the components k. The VMDN loss function can

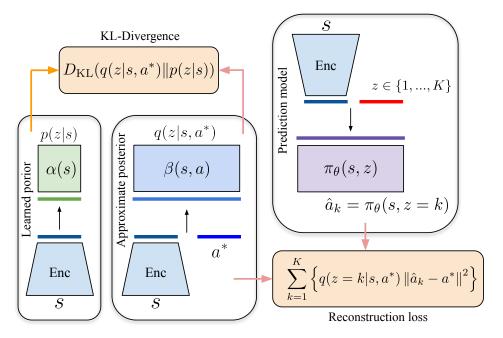


Figure 4.4: Model architecture for the variational mixture density network. The sensor input encoder is shared across all three parts. We use one hot encoding for the latent variable z as the input of the policy network $\pi(s, z)$.

be explained by two terms: the expectation of prediction loss and the KL-divergence term, which prevents z from just copying the information of the ground truth actions a.

4.3 Experiments

We evaluate our proposed model in the racing game TORCS, described in Chapter 3. For this evaluation, we alter the reference policy to perform lane changes randomly; in other words, the car driven by the controller is tasked with lane following, and randomly changes lanes while doing so. In this scenario, there are three different driving tasks: 1) following the lane, 2) changing lane to the left, and 3) changing lane to the right. Our goal is to have VMDN capture these

three separate modes from expert demonstrations without any additional supervision signals. We follow the same data collection process as discussed in Chapter 3. The model inputs $x_{T:0}$ are the past T-frame images captured by a front-facing camera mounted on the racing car in the game at time step t_0 . The model outputs $y_{0:P}$ are the P-frame future driving commands. The encoder is a five-layer CNN that outputs a flattened 64-dimensional vector as the sensor feature. In the prediction model, the latent variable encoded as one hot vector is first fed to a network with three fully-connected layers, then is concatenated with the sensor feature.

We train all models with the ADAM optimizer and learning rate $\eta=0.0002$. For the variational mixture density work, we set $\delta=1e-3$.

4.3.1 Mean square error of driving commands

We first evaluate the performance of the deterministic model using Gaussian regression, mixture density network (MDN) and proposed variational mixture density network (VMDN) by comparing the mean square error (MSE) between the ground-truth driving commands and model predictions. In the case of mixture density network, we have two inference strategies:

- 1. Greedy inference: $\hat{a} = \mu_{\hat{k}}(s)$, where $\hat{k} = \arg\max_k \pi_k(s)$.
- 2. Best MSE inference: $\hat{a} = \mu_{\hat{k}}(s)$, where $\hat{k} = \arg\min_{k} \text{MSE}(\mu_{k}(s), a^{*})$, a^{*} is the ground-truth action given by the expert demonstrations.

In the case of variational mixture density network, we have three inference strategies:

- 1. Greedy inference: $\hat{a} = \pi(s, \hat{z})$, where $\hat{z} = \arg\max_k \alpha_k(s)$
- 2. Best MSE inference: $\hat{a} = \pi(s, \hat{z})$, where $\hat{z} = \arg\min_k \text{MSE}(G(s, z_k), a^*)$

3. Max posterior inference: $\hat{a} = \pi(s, \hat{z})$, where $\hat{z} = \arg\max_k \beta_k(s, a)$

When comparing these networks with the baseline model Gaussian regression, we only use the greedy inference MSE results. Tab. 4.1 shows that both MDN and VMDN outperform the baseline model and that VMDN performs best overall. We additionally used the best MSE inference to compare model performance when choosing the best k.

Inference	Deterministic	MDN	VMDN
Greedy	0.0124	0.0097	0.0021
Best MSE	-	0.0064	0.0010
Max posterior	-	-	0.0011

Table 4.1: Mean square error of steering determined through different inference strategies.

4.3.2 Driving trajectory visualization

To show that our proposed model can learn the underlying latent structure of expert demonstrations, we determine driving trajectories in the context of different latent codes. We calculate the driving trajectory by simulating the actions that would occur if we executed the sequence of predicted diving commands $a_k = \pi(s, z_k)$.

In Fig. 4.6, we show that our proposed model can distinguish the three driving tasks. The prior distribution given by $\alpha(s)$ represents confidence in the corresponding latent code.

4.3.3 In game evaluation

We let the trained VMDN model drive the racing car in TORCS. When manually setting the latent variable $z \in \{1, 2, 3\}$, the car can achieve all three driving tasks. Fig. 4.7 shows the sensor

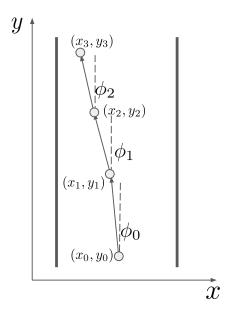


Figure 4.5: Driving trajectory: For each time step, we assume the moving distance in the y direction is a constant d_y . ϕ_t denotes the angle between the lane direction and heading of the vehicle at time step t. The position of the vehicle coordinate (x_t, y_t) is calculated by $y_t = y_{t-1} + d_y$ and $x_t = x_{t-1} + d_y \tan \phi_{t-1}$.

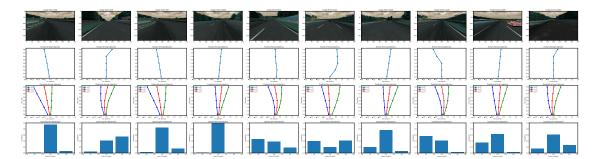


Figure 4.6: Visualization of various driving trajectories based on the different latent variables z_k . The top row shows the image frames of the sensor input. The second row shows the corresponding human driving trajectories. The third row shows the various driving trajectories based on different latent codes. The last row shows the learned prior distribution p(z|s). This figure empirically illustrates that the three dimensional latent variable $z_{1:3}$ can be viewed as a set of high-level driving commands, which are lane following, left lane changing, right lane changing.

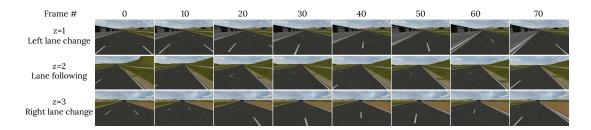


Figure 4.7: Screenshot of sensor inputs for VMDN evaluation in TORCS, illustrating that the driving behavior of the car driven by the trained model can be controlled by the latent variable.

inputs for the VMDN in the three driving tasks when given different latent variables.

4.4 Summary

In this chapter, we introduce a variational mixture density model with a categorical latent variable to address the uncertainty issues of driving actions for learning an end-to-end driving model. Learning these latent variables does not require any direct supervision signals beyond expert demonstrations. Our experimental results in the TORCS simulator show that VMDN can automatically distinguish certain human driving tasks such as lane following and lane changing. The latent variable can be used as a high-level driving command to perform different driving tasks.

5

Driving Simulation for End-to-end Autonomous Driving

With the boom of researches in autonomous driving, there is also a growing need for driving simulators to train models and evaluate the driving performances. It is true that an ideal evaluation is to let the trained vehicle drive on the real roads. However, it is impractical for most research

groups as testing in a physical environment requires significant funds and human resources to build the infrastructure and manage the requisite hundreds of cars. As a result, many researchers choose to use simulation tools when developing and evaluating algorithms in the field of autonomous driving. There are some distinct advantages to using simulation tools for autonomous driving:

- Cheap and unlimited simulated data Driving simulators like racing games can generate unlimited quantities of synthetic data extremely quickly and without much engineering overhead.
 Multiple simulators can also be run simultaneously, allowing for parallelization that is not usually feasible in a real-world environment.
- Low risk and rapid prototyping This is especially important for autonomous driving, as it is necessary to evaluate risk by testing extensively in a simulator before letting a model drive on the road. Companies take the safety of on-road test driving very seriously, and require drivers to be trained to operate an autonomous vehicle. In a simulator, we can quickly try new models without worrying about damage to the environment or a vehicle.
- Reproducible experiments In the real world, it is impossible to reproduce a particular experiment, as the environment changes in ways we cannot control. In a simulator, we can easily run experiments under identical driving conditions to debug or compare different algorithms.

In this chapter, we will first discuss the driving simulators that have been widely used in academic researches, and then we will introduce the framework of the simulator SEEL we designed to support training and evaluation of the end-to-end autonomous driving systems in video games. Finally, we will demonstrate one example that utilizes the simulator to train and test a model to drive a truck following a navigation map in a video game. By using the navigation map as an

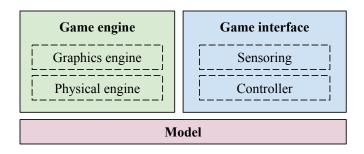


Figure 5.1: Schematic of the simulator used in autonomous driving research.

additional input, the trained model can make turns at intersections or exit highways according to the map instructions.

5.1 Simulator for autonomous driving research

The driving simulator used in most autonomous driving researches is a driving game engine with all necessary interfaces for autonomous driving. This game engine is usually built upon many software modules that render the scene to be driven through and simulate driving by taking commands from device inputs. Interfaces are used to capture sensor inputs, obtain model outputs, and to apply those outputs as device inputs in order to control the vehicle in the game engine.

The extant driving simulators used in research have been developed either by hacking or modifying a particular computer game or building an engine from scratch. When using a computer game platform, most such are racing games because their driving scenarios are relatively simple and the games are easily modified. For instance, the racing-style simulators *Super Tux Kar*, *The Open Racing Car Simulator (TORCS)*, and *World Rally Championship 6 (WRC6)* are widely used as simulators in imitation learning and reinforcement learning (Ross et al., 2010; Chen et al., 2015; Zhang and Cho, 2017; Perot et al., 2017). However, while these racing-style simulators



Figure 5.2: Screenshots of representative racing simulators: Super Tux Kar (top-left), TORCS (top-right), WRC6 (bottom-left), and Udacity's Self-Driving Car Simulator (bottom-right).

have the advantage of simplicity, they are far from approximating real driving scenarios; they are characterized by non-realistic rendering, monotonous landscapes, and limited driving tasks. Making modifications to these aspects is a game-specific and time-consuming process. For example, in order to modify an open-source game like TORCS to serve as a driving simulator, we generally need to go through the whole design document to understand the code framework, then dive into the source code to hack specific parts. Sometimes it is necessary to manually generate data for rendering, such as textures, tracks, and terrains. This time-consuming work must be done for every game to be used as a driving simulator.

In contrast, simulators built from scratch, such as *CARLA: An Open Urban Driving Simulator* (Dosovitskiy et al., 2017) and *Udacity's Self-Driving Car Simulator* (Udacity, 2017), are designed

from the ground up for autonomous driving research. Therefore, the interfaces for training and evaluating models in the game are well defined. However, these simulators are still game-specific and offer limited driving environment content. Building a realistic game simulator from scratch is usually not economically feasible for research groups. Instead of doing so, we can use existing realistic driving games such as $Grand\ Theft\ Auto\ V^*$ and $American\ Truck\ Simulator^\dagger$ as the simulator engine and build the necessary interface for autonomous driving research. One such universal interface is SEEL (Simulator for End-to-end Learning), which was implemented in Python and designed to unite the game engine with a general interface for sensor capture and controller simulation. SEEL does not rely on any particular game engine, which gives researchers the freedom to work with any video games that run under Linux, including non-driving games.

5.2 Framework of the simulator

SEEL provides a simple interface between a video game and a model that needs to interact with the video game. Its implementation includes functions to capture sensor inputs from a screenshot of the video game and to generate control signals and emulate virtual devices such as a joystick for controlling the game. We implemented SEEL in a multi-processed environment in which modules including a game, player, controller, and recorder were sub-processes of SEEL, allowing us to run SEEL alongside a video game in real time. Communication between these processes occurs via *queue* data structures so that the program itself is free from the burden of synchronization. Data such as sensor inputs, control signals, and other customized information such as the speed of the truck are collected at each time step as the state of the simulator. We can generate a data set by

^{*}http://www.rockstargames.com/V

[†]http://americantrucksimulator.com

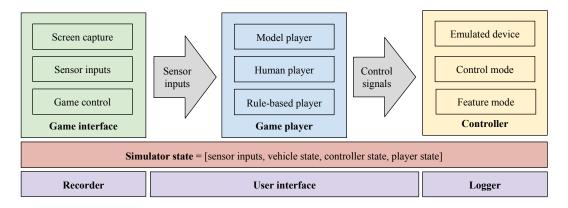


Figure 5.3: Block diagram of the simulator framework. The simulator state, including sensor inputs and control signals, is shared across all modules and updated at each time step.

recording this state data frame by frame and displaying them in the user interface of the simulator.

Game module. The game module is a game resource interface that can be accessed by a model through SEEL. Game resources include information from game windows, sensor inputs, and game utilities such as pause and resume game. Sensor inputs are generated from screenshots of the game windows and displayed in the SEEL user interface. One challenge in the development of the game module was reducing the latency of screen capturing and producing sensor inputs while running at a stable framerate. The framerate of the sequence of sensor inputs in the simulation must match that used to train the model, making it critical to run the simulator in real time with multi-frame sensor inputs.

Player module. The player module contains different virtual players who generate play commands such as steering angles, thereby playing games while observing the sensor inputs. These play commands are generated based on either model outputs (model player) or physical device outputs (human player).

Controller module. The controller module is designed to emulate a standard game control device

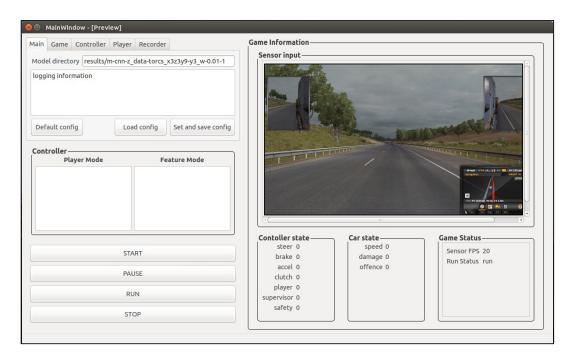


Figure 5.4: Simulator interface.

such as a keyboard, mouse, joystick, or steering wheel. It generates control signals from player commands by following a customized control strategy. Using the controller module, we can switch between model and human players to control, or we can even generate control signals by mixing driving commands from all players according to a pre-defined strategy. This allows us to efficiently perform imitation learning, in which we need to switch between different policies frequently.

User interface. The user interface is designed to provide a convenient means for controlling the simulator, visualizing simulator state, and modifying the configuration.

5.3 End-to-end driving using navigation map

In this section, we start by discussing the limitations of previous driving systems and then introduce an end-to-end driving system that uses a navigation map. We use SEEL in conjunction with the video game *American Truck Simulator* [‡] to train and evaluate the proposed model.

5.3.1 Navigation commands for urban driving

The previous chapters described how to train a model to drive using end-to-end learning methods. However, these autonomous driving systems are limited to simple driving tasks such as lane following and lane changing, and the optimal actions were inferred from perceptual inputs alone. What happens if a car approaches an intersection? The sensor inputs from cameras are not sufficient to decide turns. High-level navigation commands such as *make a left turn at the next intersection* are needed to achieve fully autonomous driving, but are lacking in the existing implementations of an end-to-end learning system. In other words, the inputs for the end-to-end driving system need to include both perceptual inputs and navigation commands. How can these navigation commands be generated? One method is to first define a set of high-level navigation commands such as left turn, right turn, or go straight, then ask human to label them for all cases in which we need to make a choice. In short, this is like driving to the destination by following a list of driving directions. Such manual labeling is time-consuming, and sometimes complicated because it is challenging to generate simple navigation commands when we face complex intersections.

[‡]American Truck Simulator is a vehicle simulation game developed by the Czech company SCS Software. In the game, players drive trucks and deliver trailer-moved goods to a designated location to earn money. The game is set in an abridged 1:20 scale version of the western United States, which makes it a realistic environment for testing an autonomous driving system.

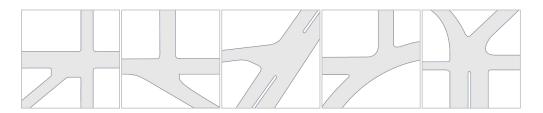


Figure 5.5: Samples of complex intersections based on actual intersections(of City Transportation Officials, 2019). These irregular intersections, which result from successive urban developments, make for difficult-to-describe navigation commands.

Instead of using a set of manually-defined navigation commands, we propose to use the navigation map itself as an additional input for the end-to-end learning system. A navigation map can usually be generated automatically with the vehicle's Global Positioning System (GPS) signal and dynamic routing service. Using the map naturally resolves the complex-intersection issue because geometry information is encoded in the street map.

5.3.2 Model architecture

In order to drive by following a navigation map, the model needs to take both sensor inputs and navigation map as inputs for the prediction of driving commands. The model is a feed-forward convolutional neural network, which contains two types of encoders corresponding to the video frames captured from cameras and the navigation maps:

1. Sensor-input encoder $E_c(X_{t-k:t})$: the sensor inputs $X_{t-k:t}$ contain a sequence of video frames captured from multiple cameras in American Truck Simulator. The encoder E_c , a feed-forward convolutional network, takes these frames as inputs for generating the feature map. The lower-level convolutional layers are shared across different camera inputs and different time steps.

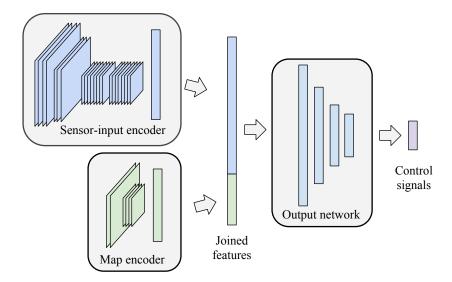


Figure 5.6: Model architecture for end-to-end learning using a navigation map.

2. Map encoder $E_m(M_t)$: M_t is the navigation map at time step t. Please note that, unlike with the multi-frame camera inputs, we only provide a navigation map for a single time step because M_t should contain enough navigating information. We also add some random noise to the time step t during training. We want the navigation map to provide rough guidance, like "make a left turn," instead of precise detail as in "turn the steering wheel two degrees to the left in 40 milliseconds." In other words, we want the driving system to be more reliant on the camera inputs. In reality, a GPS map is quite noisy, and furthermore it is not interesting to train a model to drive while relying heavily on a precise navigation map.

The outputs of the camera-frame encoder E_c and the map encoder E_m are concatenated and fed to the *output network*, which is a neural network with several fully-connected layers, in order to predict the driving commands a_t , including steering angle, brake, and acceleration. The details of the model architecture are shown in Fig. 5.6.

We defined the per-sample loss function as the mean square loss between the predicted driving

commands \hat{a} and human driving commands a.

$$\mathcal{L}(\hat{\boldsymbol{a}}, \boldsymbol{a}) = \|\hat{\boldsymbol{a}} - \boldsymbol{a}\|^2 \tag{5.1}$$

5.3.3 Experiments

Data collection

We use a Logitech G27 § to control the truck in ATS because it resembles the physical driving experience. The driving mode in the game is set to use an automatic transmission in order to simplify operation and remove the need for the model to learn to control both clutch and shifter for acceleration, as this is not the main interest of our work. We collected training data by manually driving the truck using G27 controller in ATS to randomly-selected destinations following the navigation map provided in-game. The setup for data collection is shown in Fig. 5.7

We collected data for a diverse set of traffic, lighting, surrounding environment, and weather conditions. Fig. 5.8 shows some samples from our data set. Most scenarios in the data set concern highway driving, but some feature urban driving, including traffic lights and all kinds of intersections.

While driving, SEEL runs in the background to collect screenshots and control signals, which are the steering angle and acceleration captured from the G27 controller. At each time step, the sensor inputs consist of three RGB video frames captured from one front-facing camera and two rear-facing cameras. The front-facing camera covers the front-view vision context. The two rear-facing cameras mounted on the left and right side of the truck cover side vision context, and provide

[§]The Logitech G27 is an electronic steering wheel designed for a driving simulator game. It consists of a steering wheel, a set of stainless steel pedals for braking, acceleration, and clutch, and a shifter unit.



Figure 5.7: Setup for data collection in American Truck Simulator.

the necessary side view information when making turns or lane changes. The navigation map is provided by ATS. Fig. 5.9 shows a sample of the sensor inputs and navigation map.

Model training and results

The detailed model architecture is shown in Fig. 5.10. Our proposed model consists of three parts:

1) a sensor-input encoder, 2) a map encoder, and 3) a join net. The sensor-input encoder contains a camera shared net and a camera feature net. Within the camera shared net, the weights of the convolutional layers are shared across different camera inputs. Additionally, the camera shared net has two kinds of sharing layers. The convolutional layers are shared for all frames across different cameras and different time steps. After that, the frame features, the outputs of the frame net,

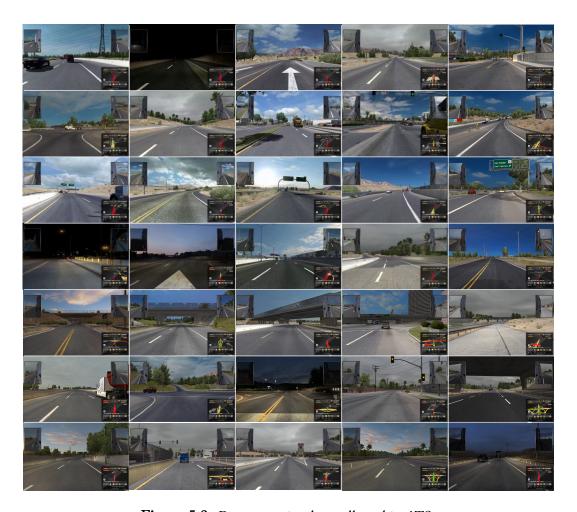


Figure 5.8: Representative data collected in ATS.

are concatenated along with different time steps and fed to the temporal net. The convolutional layers within the temporal net are shared across different camera inputs, and output temporal feature maps. All camera feature maps and the feature map generated from the map encoder are concatenated and fed to the join net. Finally, the outputs of the join net are the driving commands used to control the truck.

The model is trained using the ADAM optimizer (Kingma and Ba, 2014) with mini-batches of 40 samples and an initial learning rate of $\eta = 0.0002$. To evaluate the trained model, we use



Figure 5.9: Representative sensor inputs and navigation map from our dataset.

SEEL to let the trained model drive a truck in ATS while following the in-game navigation map. The model can successfully make correct turns at intersections and when exiting highways.

Difficulties Although the model can follow the navigation map to a certain degree, we observe that it is challenging to train a model to drive a truck in ATS. Some of the difficulties are as follows:

- The truck can crash and get stuck on objects that are not visible from the camera inputs. For example, the wheels of the truck can get stuck on the curb when making turns at intersections. This happens even for a human driver, and can be hard to avoid.
- The speed control for the truck is complicated. More specifically, when given only the camera inputs, it is difficult to train a model to control speed through the gas and brake pedals accurately. For example, in order to stop before an intersection or exit a highway, the truck needs to be slowed down in advance. The need to slow down is difficult to tell from the camera inputs. Even when keeping the truck driving at a certain speed, we need to continuously adjust the gas and brake pedal according to the truck, road, weather, and traffic conditions. As an alternative, we suggest training the model to predict the desired driving speed and let the vehicle controller regulate the gas and brake to adhere to the desired speed.
- It is difficult to train a model to enter the correct lane before or after making a turn at an

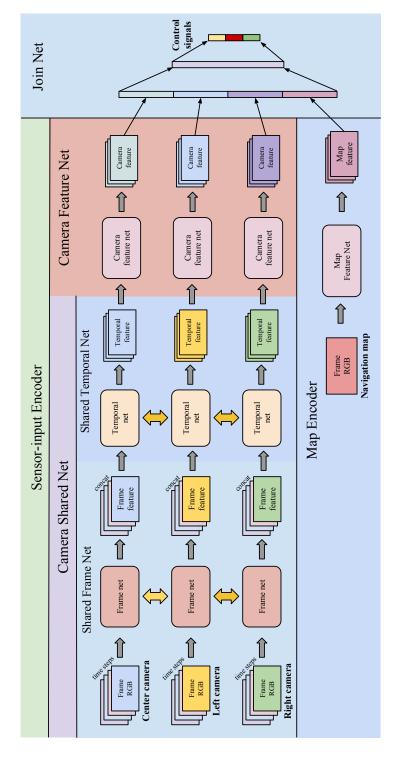


Figure 5.10: Our proposed model for autonomous driving using a navigation map.

intersection due to the limited camera view; there are no obvious clues such as lane markers at intersections.

5.4 Summary

In this chapter, we introduced SEEL as an interface to support the development, training, and evaluation of the end-to-end learning approaches for autonomous driving. By leveraging SEEL, we can train and test models via end-to-end learning to drive a truck while following a navigation map in American Truck Simulator. SEEL provided us with tools to capture and transform the sensor inputs and to send control signals to a video game by emulating physical devices in the Linux system. We hope that other researchers working on autonomous driving and artificial intelligence will build upon this work and benefit from using SEEL.

6
Conclusion

During recent decades, researchers and industry leaders have mainly focused on developing autonomous driving systems using a mediated perception approach, which includes perceiving and analyzing the environment, localizing the ego-vehicle, and planning individual trajectories and maneuvers. Although significant progress has been made towards building a fully autonomous vehicle, many challenges remain unresolved. Notably, the mediated perception approach often

involves many manually-defined sub-tasks such as lane marker detection, traffic sign recognition, and tracking of cars and pedestrians (Geiger et al., 2013). These individual sub-tasks themselves remain open research questions, and their implementation can make autonomous driving systems overcomplicated. Certain aspects of the information obtained from these sub-tasks are necessary, such as the distance from the vehicle in front, whereas other information is useful but unnecessary, such as detailed semantic scene understanding or accurate depth map estimation. Furthermore, these individual sub-tasks defined in the medicated perception approach remain to be open research questions and can make the autonomous driving system overcomplicated. It is also standard for autonomous vehicle systems to rely on accurate GPS positioning and prior information such as a high-definition map. However, heavy reliance on prior information can lead a driving system to suffer when handling novel situations that arise from dynamic driving environments. For instance, autonomous driving vehicles cannot localize themselves when on bridges due to the limited number of distinguishing landmarks, and therefore are unable to drive over bridges without human intervention (Coren, 2018).

The end-to-end deep learning approach offers an alternative to continuously detecting and classifying objects and localizing the vehicle, as it directly maps sensor inputs to driving actions. In this thesis, we explored such approaches for autonomous driving.

In Chapter 2, we demonstrated the potential of the end-to-end learning framework by training a deep neural network to do lane following, and furthermore achieved long-distance lane following in a real-world environment without any human intervention. Our success consists of prior work, which has inspired an increasing number of studies in this field over the past three years (Xu et al., 2017; Codevilla et al., 2018; Chen et al., 2019).

In Chapter 3, we introduced SafeDAgger, a more general end-to-end learning approach for

autonomous driving via query-efficient imitation learning. By using a safety classifier, SafeDAgger queries the reference policy significantly less and therefore trains a primary policy more efficiently.

In Chapter 4, we introduced the use of variational mixture density networks (VMDNs) to model the uncertainty in driving actions. We showed that a VMDN could automatically distinguish specific driving tasks such as following a lane or making a lane change. The latent variable introduced in VMDN can be viewed as a high-level driving command. At test time, the latent variable can be manually set to achieve different driving behaviors.

In Chapter 5, we introduced SEEL, a general simulator for end-to-end learning that supports development, training, and evaluation for autonomous driving. SEEL provides tools for capturing the sensor inputs from driving video games and for control of the vehicle in-game by the trained model. We demonstrate that a deep neural network trained using end-to-end learning can drive a truck while following the navigation map in a video game.

6.1 Future work

Although many promising advances and technologies have been developed for autonomous driving, industrial leaders such as Waymo acknowledge that present autonomous vehicle systems are not yet robust to drive themselves under all conditions and that full autonomy may not be achieved very soon. We hope that the end-to-end learning approach provides an interesting and promising alternative to classical autonomous driving techniques. To conclude this thesis, there are several avenues of future work we would like to highlight.

6.1.1 Temporal information

In autonomous driving, it is essential to maintain the temporal consistency of driving actions. Learning a model of driving actions from individual video frames can produce inconsistent driving actions, which may lead to dangerous situations. It is also necessary for autonomous driving to predict and recognize the movements of surrounding objects such as pedestrians, which is especially critical in the urban driving environment. By leveraging the temporal structure of sensor input data, a model learned the motion of surrounding objects and predicted time-consistent driving actions to ensure a smooth autonomous driving experience (Chi and Mu, 2017). In the field of video prediction, there is much exciting research work being done that leverages the temporal structure of video data (Denton et al., 2017; Denton and Fergus, 2018). The development of end-to-end learning methods using temporal information may provide more robust and efficient driving systems.

6.1.2 Predicting the future

Predicting scene dynamics is an important research topic in computer vision, especially for autonomous driving. For instance, to achieve autonomous driving in urban scenarios, it is extremely helpful to have accurate models for the dynamic surrounding environment that facilitate future prediction and planning of the driving path ahead. There is some initial work in this area (Luc et al., 2017; Henaff et al., 2019). Since scene dynamics can be learned using unsupervised learning, this is a great way to utilize large amounts of unlabeled data such as video frames captured while driving, and would benefit end-to-end learning approaches.

6.1.3 Multi-task learning

The end-to-end learning approaches for autonomous driving introduced in this thesis predict driving commands such as steering angle, brake, and acceleration. While we focused on optimizing this single task, we may have ignored information that might help the system perform even better, specifically, information from the training signals of highly related tasks such as predicting vehicle speed or the presence of a car ahead within a certain distance. By sharing representations between related tasks, we can train the driving model to produce better and more reliable generalizations. Using multi-task learning for autonomous driving can combine the end-to-end learning approach with the commonly-used mediated perception approach.

Bibliography

The Open Racing Car Simulator, accessed May 12, 2016. URL http://torcs.sourceforge.net/.

Amjad Almahairi. Mixture density networks. https://amjadmahayri.wordpress.com/2014/04/30/mixture-density-networks/, 2014. [Online; accessed 23-April-2019].

American Automobile Association. Think you're in your car more? you're right. americans spend 70 billion hours behind the wheel. https://goo.gl/gdttYz, 2019. [Online; accessed 14-Mar-2019].

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

Christopher M Bishop. Mixture density networks. 1994.

Christopher M Bishop. Pattern recognition and machine learning. springer, 2006.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Urs Muller Mathew Monfort, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

Christopher Bonnett. Mixture density networks for galaxy distance determination in tensorflow. http://cbonnett.github.io/MDN.html, 2016.

Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

Kai-wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daume, and John Langford. Learning to search better than your teacher. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2058–2066, 2015.

Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

- Kevin Chen, Juan Pablo de Vicente, Gabriel Sepulveda, Fei Xia, Alvaro Soto, Marynel Vazquez, and Silvio Savarese. A behavioral approach to visual navigation with graph localization networks. *arXiv preprint arXiv:1903.00445*, 2019.
- Hong Cheng. Autonomous intelligent vehicles: theory, algorithms, and implementation. Springer Science & Business Media, 2011.
- Lu Chi and Yadong Mu. Deep steering: Learning end-to-end driving model from spatial and temporal visual cues. *arXiv preprint arXiv:1708.03798*, 2017.
- Adam Coates, Pieter Abbeel, and Andrew Y Ng. Learning for control from multiple demonstrations. In *Proceedings of the 25th international conference on Machine learning*, pages 144–151. ACM, 2008.
- Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. Endto-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9. IEEE, 2018.
- Hal Daumé Iii, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine learning*, 75(3):297–325, 2009.
- Emily Denton and Rob Fergus. Stochastic video generation with a learned prior. *arXiv preprint* arXiv:1802.07687, 2018.
- Emily L Denton et al. Unsupervised learning of disentangled representations from video. In *Advances in Neural Information Processing Systems*, pages 4414–4423, 2017.
- Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.
- Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.
- Axel Brando Guillaumes. Mixture density networks for distribution and uncertainty estimation. Master's thesis, University of Barcelona, 2017.
- Mikael Henaff, Alfredo Canziani, and Yann LeCun. Model-predictive policy learning with uncertainty regularization for driving in dense traffic. *arXiv preprint arXiv:1901.02705*, 2019.
- National highway traffic safety administration. Risky driving. https://www.nhtsa.gov/, 2019. [Online; accessed 04-Mar-2019].

- T Jochem and D Pomerleau. No hands across america official press release. *Carnegie Mellon University*, 1995.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.
- Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2013.
- Hoang M Le, Peter Carr, Yisong Yue, and Patrick Lucey. Data-driven ghosting using deep imitation learning. 2017.
- John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, et al. A perception-driven autonomous urban vehicle. *Journal of Field Robotics*, 25(10):727–774, 2008.
- Daniele Loiacono, Julian Togelius, Pier Luca Lanzi, Leonard Kinnaird-Heether, Simon M Lucas, Matt Simmerson, Diego Perez, Robert G Reynolds, and Yago Saez. The wcci 2008 simulated car racing competition. In *CIG*, pages 119–126. Citeseer, 2008.
- Pauline Luc, Natalia Neverova, Camille Couprie, Jakob Verbeek, and Yann LeCun. Predicting deeper into the future of semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 648–657, 2017.
- Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L Cun. Off-road obstacle avoidance through end-to-end learning. In *Advances in neural information processing systems*, pages 739–746, 2006.
- Gordon Frank Newell. Nonlinear effects in the dynamics of car following. *Operations research*, 9 (2):209–229, 1961.
- National Association of City Transportation Officials. Urban Street Design Guide. https://nacto.org/publication/urban-street-design-guide/intersections/complex-intersections/, 2019. [Online; accessed 23-Feb-2019].
- World Health Organization. Road traffic injuries. https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries/, 2019. [Online; accessed 04-Mar-2019].
- Etienne Perot, Maximilian Jaritz, Marin Toromanoff, and Raoul De Charette. End-to-end driving in a realistic racing game with deep reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 3–4, 2017.

- Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. Technical report, DTIC Document, 1989.
- Dean A Pomerleau. Progress in neural network-based vision for autonomous robot driving. In *Intelligent Vehicles' 92 Symposium., Proceedings of the*, pages 391–396. IEEE, 1992.
- Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive noregret learning. arXiv preprint arXiv:1406.5979, 2014.
- Stéphane Ross, Geoffrey J Gordon, and J Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. arXiv preprint arXiv:1011.0686, 2010.
- Susan Ross, Narek Melik-Barkhudarov, Kumar Shaurya Shankar, Andreas Wendel, Debabrata Dey, J Andrew Bagnell, and Martial Hebert. Learning monocular reactive uav control in cluttered natural environments. In *Robotics and Automation (ICRA)*, 2013 IEEE International Conference on, pages 1765–1772. IEEE, 2013.
- Christopher Rouff and Mike Hinchey. *Experience from the DARPA urban challenge*. Springer Publishing Company, Incorporated, 2011.
- Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 28. MIT press, 1998.
- Udacity. Udacity's self-driving car simulator. https://github.com/udacity/self-driving-car-sim, 2017.
- Rebecca A Williamson, Vikram K Jaswal, and Andrew N Meltzoff. Learning the rules: Observation and imitation of a sorting strategy by 36-month-old children. *Developmental psychology*, 46(1): 57, 2010.
- Bernhard Wymann. Torcs manual installation and robot tutorial. *Manuscript, Available from:* http://torcs. sourceforge. net/index. php, 2006.
- Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrell. End-to-end learning of driving models from large-scale video datasets. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2174–2182, 2017.
- Wojciech Zaremba and Ilya Sutskever. Learning to execute. arXiv preprint arXiv:1410.4615, 2014.
- Jiakai Zhang and Kyunghyun Cho. Query-efficient imitation learning for end-to-end simulated driving. In AAAI, pages 2891–2897, 2017.