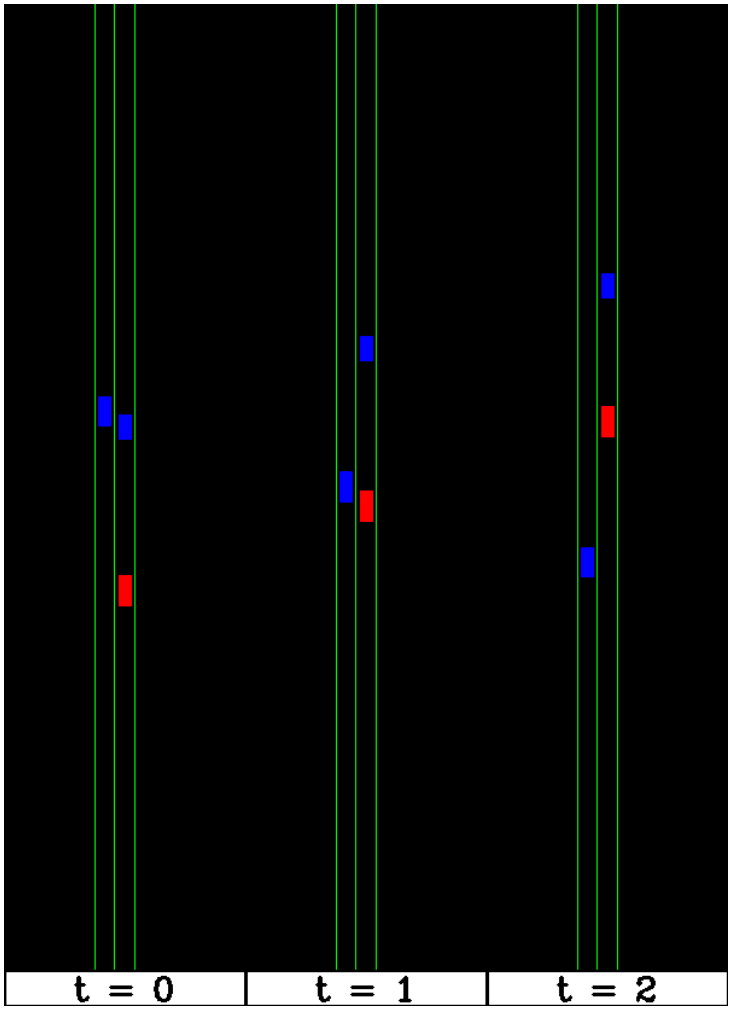


Imitation learning for autonomous driving using high-level data

Master's thesis in Computer Science – Algorithms, Languages and Logic

HAI DINH

ZEESHAN UL HASSAN DAR

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

Imitation learning for autonomous driving using high-level data

Hai Dinh
Zeeshan Ul Hassan Dar

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

Imitation Learning for Autonomous Driving using High-Level Data
HAI DINH, ZEESHAN UL HASSAN DAR

Supervisor: Christopher Innocenti, Zenuity AB
Supervisor: Vilhelm Frändberg, Zenuity AB
Supervisor: Lennart Svensson, Department of Electrical Engineering
Examiner: Lennart Svensson, Department of Electrical Engineering

Master's Thesis 2020
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example sequence of top-down views showing how the red ego-vehicle moves over time, with respect to the lane markings and other vehicles.

Typeset in LaTeX
Gothenburg, Sweden 2020

Imitation learning for autonomous driving using high-level data
HAI DINH
ZEESHAN UL HASSAN DAR
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Data-driven approaches using deep learning have become increasingly popular within the field of autonomous driving. One such approach is called imitation learning, where the autonomous agent learns by simply observing demonstrations from a human expert driver. This form of supervised learning will be used in our thesis for training the DecisionNet—a neural network responsible for predicting the future trajectory of the autonomous vehicle.

To simplify the task, sequences of 2D top-down views (TDVs) will be used as high-level data representation for both the input and output of the DecisionNet. The spatio-temporal information in the TDVs will then be processed by ConvLSTM, serving as the backbone for the DecisionNet. A CNN-based network, called ENet, will also be used to efficiently increase the capacity of the model. In the two-phase variant of the DecisionNet, the entire network is organized into an encoder-decoder architecture, where the encoding phase encodes information from the past, and the decoding phase unrolls the hidden states to give predictions into the future.

One difficulty that arises is the time-horizon problem, which prevents the model from being able to see beyond the present timestep. To solve this, we transform the two-phase variant into the one-phase variant, by connecting the output of one timestep as input to the next timestep. This is to allow the DecisionNet to predict the future trajectory, while observing the situations on the road at the same time.

In our dataset of mostly high-way scenarios, it is rare to find a scenario where the vehicle has to decelerate behind another vehicle. To address this issue, we use data augmentation to generate more data for these cases, thus increasing the robustness, and also allowing the models generalize well for these deceleration scenarios.

Finally, evaluation is done both by manual visual inspection, and by comparing the output predictions directly against the ground-truth data. Our results show that the one-phase ENet-ConvLSTM-ENet architecture, trained with data augmentation, gives the best performance on most scenarios we want to tackle.

Acknowledgements

We would like to first thank our supervisors at Zenuity, Christopher Innocenti and Vilhelm Frändberg, for offering us the opportunity to write this thesis within your group at Zenuity. Thank you for all your support and enthusiasm during the project. It has been a pleasure listening to all your wisdom and your tremendous knowledge about deep learning. Thank you, Vilhelm, for encouraging us to always be more ambitious with our ideas. Thank you, Christopher, for always giving us helpful advice on how to properly implement our network in Tensorflow.

We would also like to extend our thanks to Lennart Svensson for agreeing to be our academic supervisor and examiner. Without your support and your help in outlining the project, this work would never have been made possible.

Finally, we would like to thank our fellow thesis students at Zenuity, Viktor Skantze, Petter Hansegård, Oscar Almér, and Emil Andersson, for all the valuable discussions that we have had. You guys are good friends and fun to be around, making each day that we come to work a very enjoyable day.

Hai Dinh, Zeeshan Ul Hassan Dar, Gothenburg, June 2020

Contents

List of Figures xi

List of Tables xiii

Acronyms xv

1 Introduction 1
- 1.1 System design for autonomous driving . . . 1
  - 1.1.1 End-to-end paradigm . . . 1
  - 1.1.2 Modular pipeline paradigm . . . 2
- 1.2 Imitation learning . . . 4
  - 1.2.1 Inverse reinforcement learning approach . . . 4
  - 1.2.2 Behavior cloning approach . . . 5
- 1.3 Objectives . . . 5
- 1.4 Thesis outline . . . 6
- 1.5 Additional resources . . . 6

2 Data pipeline 7
- 2.1 Data preprocessing . . . 8
- 2.2 Top down view . . . 10
  - 2.2.1 Input . . . 11
  - 2.2.2 Output . . . 14
- 2.3 Coordinate systems . . . 14
  - 2.3.1 UTM map projection . . . 16
  - 2.3.2 Conversions to Ref coordinate system . . . 18
- 2.4 Data augmentation . . . 20
  - 2.4.1 Augmentation of deceleration scenarios . . . 21

3 Model design 25
- 3.1 Popular network architectures . . . 25
  - 3.1.1 FFNN . . . 25
  - 3.1.2 CNN . . . 27
  - 3.1.3 ENet . . . 28
  - 3.1.4 LSTM . . . 29
  - 3.1.5 ConvLSTM . . . 31
- 3.2 DecisionNet architecture . . . 31

3.2.1 SNet-ConvLSTM-SNet . . . 32
3.2.2 ENet-ConvLSTM-ENet . . . 34
3.2.3 The time-horizon problem . . . 35

4 Training and evaluation **37**
4.1 Training of DecisonNet . . . 37
4.1.1 Loss function . . . 37
4.1.2 Gradient descent . . . 38
4.1.3 Data split . . . 39
4.1.4 Models to be trained . . . 40
4.2 Evaluation strategies . . . 40
4.2.1 Objective quantitative evaluations . . . 40
4.2.2 Evaluations by visual inspection . . . 42
4.3 Results . . . 43

5 Discussion **45**
5.1 $\mathcal{M}_1$ versus $\mathcal{M}_2$ . . . 45
5.2 $\mathcal{M}_3$ versus $\mathcal{M}_4$ . . . 46
5.3 $\mathcal{M}_4$ versus $\mathcal{M}_5$ . . . 46

6 Conclusion **49**

7 Future work **51**
7.1 Full system deployment . . . 51
7.2 Adding more features and scenarios . . . 51
7.3 Using of High Definition maps . . . 51
7.4 Going beyond imitation loss . . . 52
7.5 Conditional imitation learning . . . 52

Bibliography **53**

A Conversions of Coordinates **I**

List of Figures

1.1 End-to-end paradigm to autonomous driving 1
1.2 Modular pipeline paradigm to autonomous driving 3

2.1 Overview of the data preprocessing steps 8
2.2 Images fetched from Zenuity's image repository showing the typical types of road contained in our filtered data. 9
2.3 Example of the top-down view at a particular time instance, where the ego-vehicle is moving horizontally towards the right of the image, along y-direction. 12
2.4 Examples of noisy data with missing lanes. 13
2.5 The UTM grid with 60 zones horizontally and 20 latitude zone bands vertically. Image taken from the public domain on Wikipedia Commons [46]. 16
2.6 Illustration of the difference between the Grid North and the True North in an arbitrary UTM zone. Image drawn not to scale. 17
2.7 Plot of two arbitrary Cartesian coordinate systems, α and β. 18
2.8 Illustration of the GPS (g), the Local (l), and the Ref (r) coordinate system, plotted from the perspective of the flattened GPS coordinate system. 19
2.9 Data augmentation by rotations of the top-down view. 20
2.10 Illustration of an arbitrary driving trajectory (left) that has been shortened to create a deceleration scenario (right). Points C and D are chosen to mark the section in which the vehicle should decelerate (i.e., with a non-zero deceleration rate). The points along each trajectory are equally spaced in time. 21

3.1 Example of a deep FFNN with biases. 26
3.2 Plots of some popular activation functions. 27
3.3 Two-phase SNet-ConvLSTM-SNet architecture, shown for a sequence of t_p past frames and t_f future frames. Downward arrows represent flow of spatial information. Rightward arrows represent flow of temporal information. Components with the same name also share the same training parameters. 32

3.4 Two-phase ENet-ConvLSTM-ENet architecture, shown for a sequence of t_p past frames and t_f future frames. Downward arrows represent flow of spatial information. Rightward arrows represent flow of temporal information. Components with the same name also share the same training parameters. 35
3.5 One-phase variant of DecisionNet, in which the network is allowed to dynamically observe the situation on the roads, while simultaneously predicting the future ego-trajectory. $\bigoplus$ denotes the concatenation of the 1-channel output with the 2 last channels of the input top-down views, thus creating a 3-channel input that can be fed again into the network. Only one layer of ConvLSTM is used here. 36

4.1 Illustration of how the positional deviation d_t is computed in order to figure out the MPD score. Image drawn not to scale. 41

A.1 Illustration of two arbitrary Cartesian coordinate systems, α and β. . I
A.2 Illustration of linear transformations used to convert from α to β. . . I

List of Tables

2.1 List of parameters used for the generation of a driving sequence. . . . 10

3.1 Overview of the SNet encoder, where downsampling of the resolution is done by non-unity strided convolutions. 33
3.2 Overview of SNet decoder, where upsampling of the resolution is done by nearest-neighbor interpolations. The last Conv2D layer is only used for reshaping the final output and fine-tuning the details. 33
3.3 ConvLSTM layers, serving as backbone for the DecisionNet. The layers are concatenated along the time dimension, for a sequence of t_p past TDVs and t_f future TDVs. Number of learnable parameters can be computed with Equation 3.9. 34

4.1 Descriptions of 5 different models for the DecisionNet. 40
4.2 Pixel-wise MSE scores, obtained when evaluating the 5 models on the test set of 1000 samples. The lower the score, the better the model. . 43
4.3 MPD scores (in meters), obtained when evaluating the 5 models on the test set of 1000 samples. The lower the score, the smaller the deviations are between the predicted positions and the ground-truth positions. 43
4.4 Results showing the performance of the 5 models on the 4 categories of extracted scenarios mentioned in Section 4.2.2. For each of the categories, this table shows the average number of future timesteps that the models can successfully predict. The higher the number, the better the model. Note that the maximum number of future timesteps available is $t_f = 25$. 43
4.5 Results showing the performance of the 5 models on the 2 deceleration scenarios that were synthesized from scratch, as explained in Section 4.2.2. Here, we use a binary criteria, where success (✓) means that the ego-vehicle has managed to decelerate to avoid a collision, and failure (✗) means the opposite. 43

Acronyms

AD Autonomous Driving.

ADAS Advanced Driver Assistance Systems.

CNN Convolutional Neural Network.

ConvLSTM Convolutional LSTM.

DL Deep Learning.

DNN Deep Neural Network.

ENet Efficient Network.

FFNN Feed-Forward Neural Network.

GD Gradient Descent.

GPS Global Positioning System.

GPU Graphics Processing Unit.

IL Imitation Learning.

IMU Inertial Measurement Unit.

IRL Inverse Reinforcement Learning.

LiDAR Light Detection And Ranging.

LSTM Long Short-Term Memory.

MPD Mean Positional Deviations.

MSE Mean Squared Errors.

PReLU Parametric ReLU.

ReLU Rectified Linear Unit.

ResNet Residual Network.

RGB Red, Green, Blue.

RMSProp Root Mean Square Propagation.

SNet Simple Network.

TDV Top-Down View.

UTM Universal Transverse Mercator.

1

Introduction

Within the last few years, autonomous driving has become one of the hottest topics in the field of artificial intelligence. However, despite lots of investments coming from both industry and academia, it still largely remains an unsolved problem. Recent advances in deep machine learning have opened up many new doors to make fully-autonomous cars become a reality. To contribute to this exciting progress, the main goal of this thesis is to investigate how to perform imitation learning on high-level perception data, that would allow an autonomous driving agent to learn directly from a human expert driver.

The thesis was performed in the Deep Learning Team at Zenuity, which is a company that focuses on providing new solutions for Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD), thus shaping the future of the automotive industry. The company has provided us with necessary training data as well as crucial computing resources, in order to make this work possible.

1.1 System design for autonomous driving

An autonomous driving system is typically very complex, and thus must have good design choices to bring out the best performance, while still maintaining a level of scalability that would allow the system to handle real-world scenarios. Focusing mainly on utilizing deep neural works, the latest research in this area have proposed a number of approaches, many of which can broadly be classified into two main categories: an end-to-end approach, or a modular-pipeline approach. This section will discuss them in detail, including both their advantages and disadvantages.

1.1.1 End-to-end paradigm

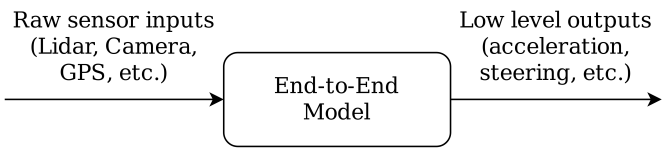


Figure 1.1: End-to-end paradigm to autonomous driving

The current state of the art in autonomous driving relies on a set of sensors (such as LiDAR, Cameras, GPS, etc.) to collect the raw data that can capture exactly what is happening around the vehicle. Using these raw sensor data, we can then build an end-to-end learning algorithm where a Deep Neural Network (DNN) can be used to directly predict the low-level control outputs such as the steering angles or the brake pressures, without ever having to extract any hand-crafted features from the raw sensor data. Such an end-to-end framework has been demonstrated to produce good results in imitating the human expert driver [4, 8, 9, 17, 27, 47].

The fact that end-to-end models can be trained without the need for any hand-crafted features gives it a major advantage over other approaches, as it requires less manual effort from the developers. For example, there is no need to manually label camera images with features such as lane markings or bounding boxes of surrounding cars. Such labeling work is not only time-consuming but can also be prone to human mistakes. Instead, the end-to-end paradigm lets the data speak for itself, allowing the model to automatically detect features that it deems to be important. As a result, this can lead to better performance and a smaller system [4].

Despite of these advantages, there is one major drawback that might make it difficult to use the end-to-end paradigm in practice. This is due to its lack of modularity—which, in turn, makes it hard for the developers to unit test, debug, and verify the system to the customers, insurance companies, and law enforcement. If the system produces an erroneous behavior, it would rather be difficult to pinpoint exactly which parts of the model that are responsible. Recent research into *interpretable* end-to-end models, such as the ones presented in [5, 19], partially solves this problem, by attempting to identify potential regions on the input images that the models used to make certain driving decisions. However, this is still not enough to put complete trust in the system, as already pointed out by [23].

1.1.2 Modular pipeline paradigm

As the name suggests, the modular pipeline paradigm organizes the overall system architecture into a set of modules. This is to tackle the inherent lack of modularity in the end-to-end paradigm.

In the recent research, Müller et al. [26] argues that the end-to-end approach is difficult to scale into realistic urban driving due to the black-box nature of the end-to-end models, as well as because of the need for a huge amount of training data required to cover full diversity of different driving scenarios. Instead, their approach is to design the system based on modularity and abstraction, which can give more scalability, and will also allow them to transfer the driving policies trained in simulation environment into a reality. A similar idea has also been presented by Bansal et al. [3], as well as by many other previous works [42, 43, 53]; while Chen et al. [7] and Sauer et al. [37] have chosen to explore the *direct perception* approach, which is often described as the combination between the end-to-end paradigm and the modular pipeline paradigm.

As shown in Figure 1.2, the key idea of the modular pipeline is to organize the architecture into three major stages: the perception of the surroundings, the decision making stage to output a driving policy, and finally a controller that can produce low-level outputs (such as acceleration, steering, etc.) to actually control the vehicle. Each of these stages can either be learned from data with machine learning or hand-designed using mathematical models and various traditional techniques [32]. For the scope of this thesis, the main focus will be on the decision module, where the goal is to develop a deep neural network (called the DecisionNet) that can be trained to produce the desired driving behaviors.

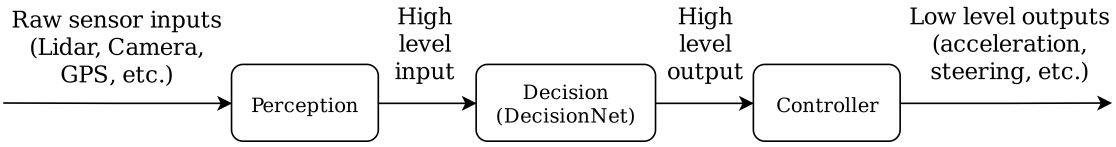


Figure 1.2: Modular pipeline paradigm to autonomous driving

An important aspect of the modular pipeline is the interface between modules, and how data is passed between them. The key idea is to introduce abstractions, by using high-level data representations within the pipeline. For example, the input to the DecisionNet can contain relevant features (such as lane markings, positions of surrounding cars, traffic lights, etc.), which have been extracted from the raw data by the perception module. The output, on the other hand, should be able to encode the driving policy generated by the DecisionNet. Exactly how these high-level data representations are implemented will be explained in more detail in Chapter 2.

Using high-level representation for the input helps encapsulate the DecisionNet from being directly exposed to the raw sensor input, which can be heavily influenced by the changing weather and illumination conditions on the road. Likewise, by using high-level representation for the output, the DecisionNet will not need to care about the exact dynamics of the vehicle. Such encapsulation of the DecisionNet means that the decision module will not suffer from the problem of the domain transfer. In other words, a learned model of the DecisionNet—which has been trained on data collected during the day-time—can also be used to drive many different types of vehicles during the night-time, or even within a simulated environment. Such characteristics can be important when scaling the system into realistic urban driving.

One downside of this architecture is that it can lead to the accumulation of errors throughout the pipeline. In other words, errors coming from the perception module can propagate onto the decision module, and then to the low-level controller. Thus, each part of the system might require time-consuming engineering to be done in order to minimize the errors. However, this can also be considered as an added benefit, because the DecisionNet will now have to adapt to the imperfections and the noise characteristics of the perception module, thus making it more robust.

Finally, as will be shown in Chapter 2.4, the high-level data representation makes it much easier to perform data augmentation in order to generate interesting scenarios

for training and testing the DecisionNet. Doing such data augmentation directly on the raw sensor data would have been too difficult and computationally expensive.

1.2 Imitation learning

Imitation learning (IL), as the name suggests, is a machine learning approach where an autonomous agent tries to acquire certain behaviors by simply imitating a human expert. This was originally inspired by the work in robotics [2], and has recently gained a lot of attention in the field of autonomous driving, either in the form of the *inverse reinforcement learning* approach, or the *behavior cloning* approach.

To a certain extent, IL is very similar to the supervised learning approach that many are familiar with. However, instead of having to manually label all the training data, the learning process of IL only requires demonstration from a human expert. This makes this approach quite intuitive for autonomous driving, since demonstration is exactly how we would teach a young adult how to drive.

1.2.1 Inverse reinforcement learning approach

In traditional reinforcement learning (RL), an agent typically learns how to behave optimally through trials and errors. The learning process often requires a *reward* function that acts a feedback signal, telling the agent whether it is performing well or badly. The agent then uses this reward function to come up with the best driving policy that maximizes the reward. Unfortunately, it is often very hard to define a good reward function for autonomous driving, since we would need to take many factors into consideration (e.g., maintaining a safe following distance, staying away from any pedestrians, not changing lane so often, etc.). This is one of the reasons why it is difficult to apply RL in practice for the task of driving.

To overcome this limitation of RL, Abbeel & Ng [1] proposed inverse reinforcement learning (IRL) as a way to iteratively learn the reward function that can best model the behavior of the human expert driver. IRL can often be formulated as a linear or quadratic program, and thus can be solved very efficiently [30]. Once the reward function is known, then it is just a matter of finding the best driving policy that maximizes the reward, using the standard traditional RL algorithm.

Due to safety reasons, the training of such RL algorithm must be done in a simulated environment, because otherwise, it would be too dangerous and costly for the agent to make mistakes on the real roads. Unfortunately, creating a simulation that can capture all the features and dynamics of the real world can be extremely difficult, if not impossible. This makes it very hard to transfer the expert's driving policy from the real world into the simulation, which poses a great challenge for training to be performed with imitation learning via IRL. Due to this problem of domain transfer, we have decided to opt for an imitation learning approach that lies closer to the behavior cloning approach presented below.

1.2.2 Behavior cloning approach

Compared to IRL, behavior cloning approach is much more straight-forward. Simply put, the autonomous agent will aim to clone the expert's behavior by mapping the input directly to an output, typically with the help of a neural network. This allows the system to be trained end-to-end, with the raw sensor input being mapped directly to the low-level steering controls, as shown in [4, 9, 34].

There is one inherent problem with pure behavior cloning, however. Since the data collected demonstrates only the desired behaviors, the agent will never be taught how to recover from the mistakes that it makes during training. For example, consider a scenario where the agent is driving too fast that it is about to crash into another car in front. Since the human demonstrator would never make such a mistake in the first place, no data will be available to tell the agent to slow down in that situation. This makes it quite difficult to generalize the model towards scenarios that are either too rare, or even missing from the collected data.

One way to address this problem of pure behavior cloning is to use an online learning approach with an interactive human demonstrator [35]. The main idea is to incorporate the demonstrator as part of the training loop itself, such that the agent can ask for feedback whenever it makes a mistake. This essentially means that more data are collected even during training. The only downside of this method is that the expert has to be available at all time, which can be very expensive.

Recently, researchers at Waymo have proposed data augmentation as a much cheaper and easier way to tackle the problem of pure behavior cloning [3]. Their approach is to introduce small perturbations in the trajectory of the human expert driver. These perturbations essentially generate more training data that will allow the agent to correct its behavior, in case it deviates slightly from the desired trajectory. As will be shown later in Chapter 2.4, similar approach will be used in our thesis, but with more focus on deceleration scenarios, as these scenarios are quite rare in our dataset.

1.3 Objectives

Our main goal is to design and develop a deep neural network for the DecisionNet as part of the modular pipeline shown in Figure 1.2. This network is to be trained and evaluated via the imitation learning approach, using only the high-level data representation. To further limit the scope of the thesis, only high-way scenarios will be considered, in which the model is expected to output basic safe driving behaviors, including lane following and not crashing into other cars. With these objective in minds, the thesis aims to answer the following questions:

1. What are the challenges in designing a high-level data representation that can accurately capture the information from the surrounding environment, as well as being able to encode the driving decisions output from the DecisionNet?

2. What are the challenges in designing an architecture for the DecisionNet that can process both the spatial and temporal information at the same time?

3. To what extent can imitation learning be used to learn about safe driving behaviors and the interactions between vehicles (such as in the case when a vehicle has to decelerate behind a slow car in front)? Will the behavior cloning approach be sufficient, or will we have to go beyond pure imitation?

4. How should the evaluation be done for a model trained with imitation learning? What are the possible methods of quantifying the model's performance?

1.4 Thesis outline

In Chapter 2, the focus will be on the data pipeline, in which we will describe how the raw sensor data is processed from start to finish, all the way until the data gets fed into the DecisionNet. Here, we introduce many important concepts (such as the top-down view, driving sequence, etc.) that will be used throughout the thesis.

In Chapter 3, we will first cover some basic DNN architectures that already exist in the literature. These background knowledge will then be used to design our own network architecture for the DecisionNet. Lastly, we will introduce the so-called *time-horizon problem*, and then explain how to modify network to tackle this issue.

In Chapter 4, we will discuss the process of training 5 different DecisionNet models, and also propose different strategies for evaluating these models. The results of the evaluation will be displayed in this chapter, including both the quantitative and qualitative measurements on various driving scenarios.

In Chapter 5, we will attempt to explain the behaviors of each of the trained models, and then compare their performances against each other. This will hopefully give us an insight into the advantages and disadvantages of our chosen approach.

Finally, in chapters 6 and 7, we will draw our final conclusions, and then discuss briefly about what more we can do in the future work.

1.5 Additional resources

The thesis comes with additional materials and resources that show the results of the training, as well as short video clips that demonstrate different concepts presented in the thesis. These additional materials can be accessed at our project website [10]:

https://sites.google.com/view/imitationlearningthesisproject/

2

Data pipeline

As mentioned in Section 1.2, imitation learning is essentially a data-driven approach, where the goal of the agent is to imitate the behaviors of the expert driver. In order to best capture these behaviors, it is important to set up a good data pipeline that is able to correctly collect and preprocess the data, thus accurately reflecting the ego-vehicle's surroundings as well as the driving decisions of the expert driver. Having a good data pipeline also allows us to partly automate the process of generating the data in an efficient manner, thus giving the agent access to a large pool of training data that can be used to steer the agent towards the desired driving behaviors.

In addition to assuring the correctness and efficiency of the data generation, the data pipeline is also the place where the representation for the high-level input and output would be defined. This is quite essential for the realization of the modular pipeline approach, presented earlier in Figure 1.2. The data representation that is chosen should at least fulfill the following three criteria:

1. The high-level input should be able to capture both the spatial and temporal information from the situations on the roads in a compact and simple way. This would hopefully make it easier for the learning model to pick out the features that are important for making the desired driving decisions.
2. Both the high-level input and high-level output should be independent of the vehicle models and the environments in which the driving takes place. This would allow us to transfer the learned skills and reuse the trained model for other types of vehicles in other types of environments as well.
3. The data representation should allow for some flexibility, such that one can easily add new features to the representation itself, in order to capture a greater detail of what is really happening on the road. This is very helpful if the system is to be further developed and then deployed in the real world.

In Section 2.2, we introduce the concept of the top-down view and argue for why this particular data representation would satisfy all the criteria listed above. But before going into those details, Section 2.1 will give an overview of the preprocessing steps in the data pipeline, thus explaining how the data provided by Zenuity is utilized. Section 2.3 will focus on the coordinate systems used for plotting the top down view, which will also lead to a rather interesting insight into what's known as a *driving sequence*. Finally, Section 2.4 will describe several data augmentation techniques that can be used to improve the generalization of the learning model.

2.1 Data preprocessing

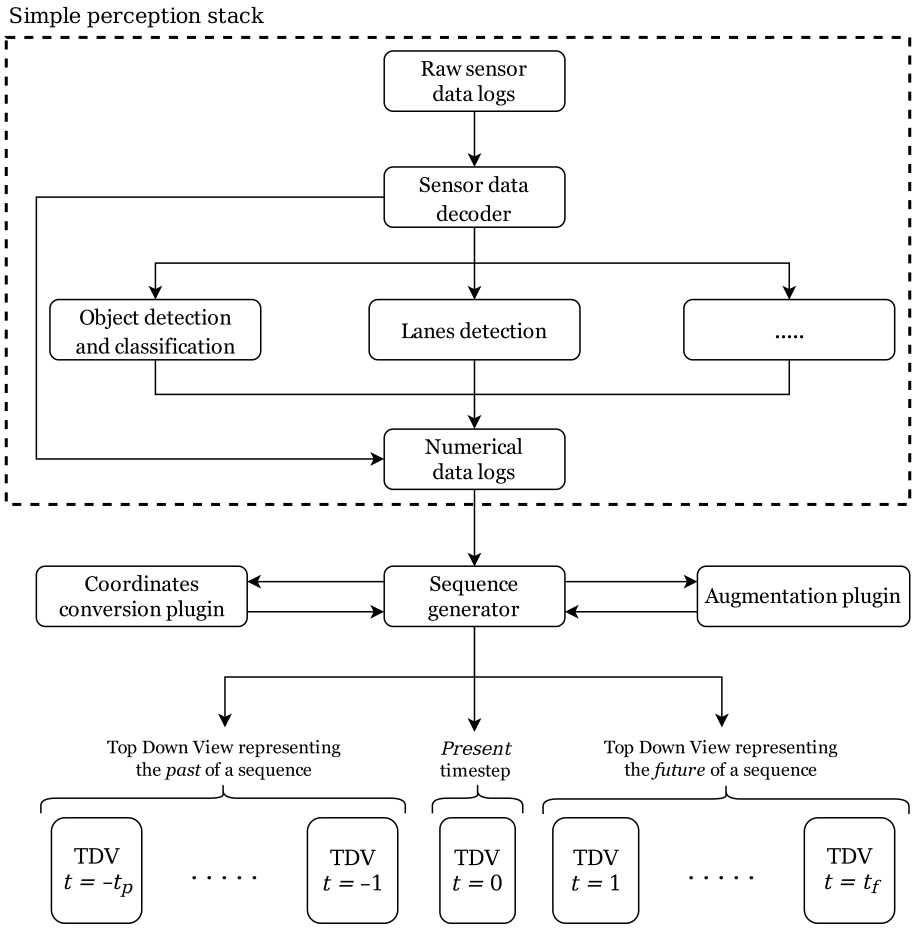


Figure 2.1: Overview of the data preprocessing steps

An autonomous vehicle must be equipped with a set of data sensors in order to dynamically collect data from the changing environment. The sensors serve two main purposes. First, they allow the vehicle to gather information about all objects within a certain range of the surroundings, such as lane markings, road barriers, pedestrians, other vehicles, street signs, etc. Secondly, the sensors allow the vehicle to know about its own motions within the environment. This is also known as the *ego-motions*. Combining these information together will give a clear image of what is happening on the road at every single timestep. In this project, the data that is used comes directly from the following three sensors:

- Lidar: Relies on laser beams to create data point clouds that can give a 3D representation of the surrounding. Lidar can accurately measure the positions and motions of objects in vicinity, and thus can be used in many perception tasks, such as object tracking and object detection.
- Camera: Produces colored images that capture the surrounding environment in a 2D representation. The dense information stored in the camera's pixels can reveal a lot about the objects on the road, which is very useful for many computer vision tasks such as object classification or image segmentation.

- GPS-enabled IMU: A collection of out-of-the-box algorithms and hardware equipment, that are used to accurately measure the ego-motions, including the GPS positions, velocity, acceleration, and orientation of the ego-vehicle.

As can be seen from Figure 2.1, the raw data from the sensors will be decoded and processed by a relatively simple perception stack, which is responsible for running many sensor fusion and computer vision algorithms to solve various perception tasks in autonomous driving. Since the scope of the thesis focuses only on the usage of the high-level data, this perception stack can simply be thought of as a black box that summarizes the ego-motions and the perception data into a set of numerical data logs—which is also where the desired high-level features can be extracted from. These logs contain information such as: the longitude and latitude GPS positions of the ego-vehicle over time, the bounding boxes of objects and obstacles on the road together with their classification probabilities, the positions of the lane markings and road barriers, among many other things.

Since the scope of the thesis has been narrowed down to only deal with high-way scenarios, it is necessary to filter out all samples that are related to driving in densely-populated urban areas with lots of pedestrians. Such filtering work has been done by manually inspecting the camera images, as well as by automated computer scripts that can filter out data based on the GPS locations. Figure 2.2 below shows the typical high-way scenarios in the filtered dataset, which actually amounts to approximately 150 hours of continuous driving. Also, in order to increase the diversity of the dataset, it is important for the filtering process to gather data that was collected from different times of the day, in different weather conditions, and even in different countries. The diversity would help the learning model to be more robust and thus generalize better to unseen data.

Figure 2.2: Images fetched from Zenuity's image repository showing the typical types of road contained in our filtered data.

The numerical data logs from the perception stack will be further processed by the latter parts of the preprocessing pipeline, which we have written and tailored to our specific needs. These numerical data logs are to be consumed directly by the sequence generator, a module responsible for generating a single *driving sequence* at a time. The sequence generator is also connected to two other plugins for converting between different coordinate systems and for the data augmentation purposes. These

will be explained in further details in Section 2.3 and 2.4, respectively. As shown in Figure 2.1, a driving sequence is composed of multiple consecutive top down views that can be divided into the past, the present, and the future timesteps. The idea is for the learning model to consume the top down views from the past and the present as inputs, and then predict an output that can be compared to the ground truth data in the future timesteps. The length of a driving sequence is simply the number of timesteps in the sequence, and should be dynamically adjustable during both training and testing. However, for the sake of convenience, the number of the past and the future timesteps are fixed at $t_p = 14$ and $t_f = 25$, respectively, as shown in Table 2.1. Similarly, the sampling frequency has been chosen to be 10 Hertz, making the time difference between two consecutive timesteps to be 0.1 seconds.

Configurable Parameters	**Default Values**
Visibility in x-direction	40 meters (ranging from -20 to 20)
Visibility in y-direction	160 meters (ranging from -60 to 100)
Image resolution	0.2 meters per pixel
Sampling frequency	10 Hertz (with 1 timestep = 0.1 seconds)
Number of past frames (t_p)	14
Number of future frames (t_f)	25

Table 2.1: List of parameters used for the generation of a driving sequence.

2.2 Top down view

The usage of the top-down view as a high-level representation of the data is not a new idea, and has in fact been explored by the team at Waymo Research in their recent related work [3]. Before going into the details of why using the top-down view is so beneficial, let us start with some concrete definition.

While it is certainly convenient to think of the top-down view as if you are looking down at the road from above, it should not be confused with the idea of taking a photograph over the road and representing the data simply as an RGB image. It is much more precise to define the top-down view mathematically as a 3D tensor of dimension (H, W, C), where H and W are the height and width of the top-down view, and C is the number of channels. Unlike an RGB image which always must have exactly 3 channels, the top-down view is much more flexible, since the number of channels, as well as what kind of information to be stored in each channel, can vary across different implementations. In fact, the only requirement is that the information from one channel has to be independent from the information stored in another channel. This is because it would allow the top-down view to be extended with more features, simply by adding new channels on top of existing ones. As will be discussed later, in our chosen implementation, three pieces of information will be represented in the top-down view—namely, the bounding box of the ego-vehicle, the detected lane markings, and the bounding boxes of other vehicles—all of which can be represented within a channel, independently from one another.

Since a channel in the top-down view is simply a 2D grid of pixels, capturing the spatial information from the road is quite straight-forward. This grid of pixels is a direct representation of an x-y Cartesian coordinates system that has the origin located at the position of the ego-vehicle at the *present* timestep. In other words, at time $t = 0$, the ego-vehicle is always located at the coordinates $(x_0, y_0) = (0, 0)$, and the direction of motion is towards the positive y-axis. Since the focus of the thesis is mostly on high-way scenarios, the ability to see far ahead is much more important than the ability to see sideways. Therefore, at the present timestep, the visibility in the y-direction is chosen to be 160 meters, much higher than the visibility in the x-direction, which is only 40 meters. With these configurations, the top-down view will be able to cover at least 4 seconds of driving time, assuming that the vehicle is travelling straight along the y-axis at the maximum speed of 40 m/s.

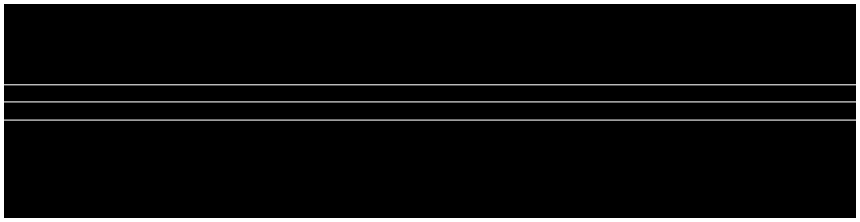
One important thing to note here is that there is always some loss of information when converting the 3D information from the real-world into the 2D representation of the top-down view. For example, information about the steepness of the road will naturally be excluded from the top-down view. However, it is possible to argue that the effect of this is minuscule for the task at hand, especially considering that the top-down view is always parallel to the surface of the road, and that it only covers a short travelling distance within a short period of time. Besides, it is always possible to simply add a new channel to represent the steepness of the road, in the case that this information proves to be important later on.

Lastly, as a final remark, it should be emphasized that the top-down view is not only capable of capturing the spatial information, but it can also track how the situation on the road changes over time, thus capturing the temporal information as well. This is done by simply stacking consecutive top-down views on top of each other, which is mathematically equivalent to stacking multiple 3D tensors into a single 4D tensor of dimension (T, H, W, C), where T denotes the number of timesteps available.

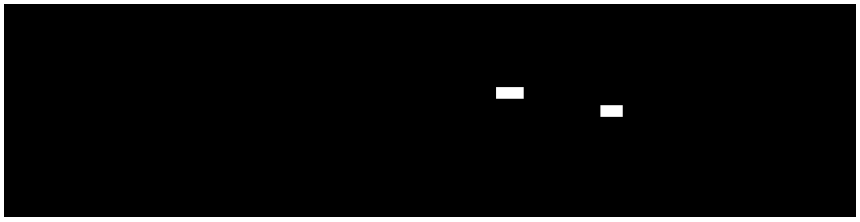
2.2.1 Input

As shown in Figure 2.3, the top-down view that we have chosen for the model input consists of three channels. The first channel contains the bounding box of the ego-vehicle, which encodes both the position and the heading of the vehicle at some particular time instance. Note that it is not necessary to explicitly encode the speed of the vehicle as part of the top-down view, since the speed can easily be deduced mathematically just by looking at how the top-down view changes over time in a driving sequence. The second channel contains lines that represent the lane markings on the road. These lane markings are assumed to be 20 centimeters wide, which is equivalent to 1 pixel under the current resolution specified in Table 2.1. Finally, the third channel contains the bounding boxes of all dynamic objects on the road, which are basically objects that can move by themselves, such as other cars, trucks, motorcycles, pedestrians, etc.

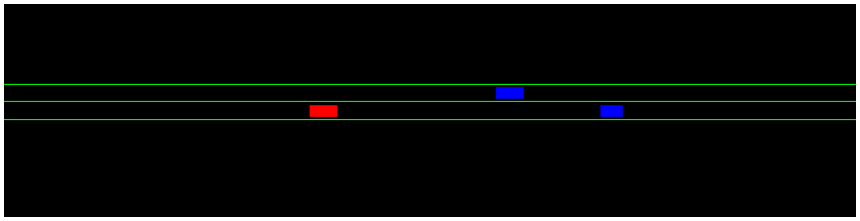
(a) Channel 1: The bounding box of the ego-vehicle



(b) Channel 2: The detected lane markings



(c) Channel 3: The bounding boxes of other vehicles

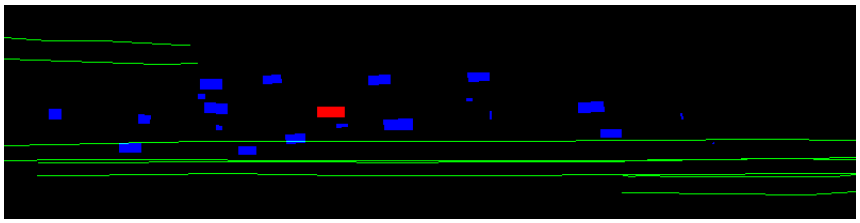


(d) All channels combined into a colored image for visualization

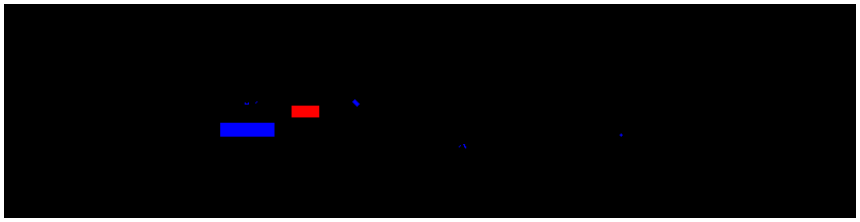
Figure 2.3: Example of the top-down view at a particular time instance, where the ego-vehicle is moving horizontally towards the right of the image, along y-direction.

Since the input should contain the spatial information from both the past and the present, the total number of timesteps in the input sequence must be $t_p + 1$ (see Figure 2.1). Height and width of the top-down view can be calculated by taking the visibility and dividing it by the resolution. Using the values presented in Table 2.1 in these calculations would give us $(T, H, W, C) = (15, 800, 200, 3)$ as the dimension of the 4D tensor that represents the input sequence. The content of this 4D tensor should also be normalized by scaling all values between the range $[0, 1]$. In fact, one can think of each channel in the input sequence simply as an occupancy grid, where 1 means the pixel is occupied, and 0 means the opposite. This also implies that if a pixel is occupied by two channels at the same time, then some sort of a collision has occurred. If it is a collision between a vehicle and a lane, it might be alright as long as a lane switch is allowed at that point. On the other hand, a collision between a vehicle and another vehicle is disastrous and an unwanted behavior.

While much of the attention has been focused on the simplicity of the top-down view, using only three channels in the input sequence certainly has some limitations. For example, since street signs and traffic lights are not part of our implementation of the top-down view, it is impossible for the model to learn when to stop and wait for the red light, or when to slow down to obey the speed limit. To overcome this limitation in the future work, the top-down view must be extended with new channels that can encode information from the street signs and the traffic lights, in order to capture a more complete picture of the driving environment. Another limitation of our approach has to do with the second channel, which appears to be too generic, and does not distinguish between different types of lane markings. In the real world, lane markings can either be solid or dashed lines, and can even have different colors. These different types of lane markings are there to convey different traffic rules regarding whether a vehicle is allowed to cross and switch to another lane. If this information is omitted from the input sequence, it can lead to bad driving behavior from the learning model. Therefore, in the future work, the second channel must be modified such that it is more capable of capturing these different types of lane markings, and thus allowing the model to obey these traffic rules.



(a) Missing lane markings for parts of the top-down view



(b) Missing lane markings for the entire view altogether

Figure 2.4: Examples of noisy data with missing lanes.

As a final remark, it is worth noting that real data is never perfect, and may suffer from different kinds of noises that could lead to poor estimations from the perception modules. Figure 2.4 shows some examples of such noisy data, where in the first case, the lanes are only missing in the areas surrounding the ego-vehicle, but in the second case, the lanes are completely missing from the top-down view. Even for a human, it is quite difficult to navigate, given such a noisy input. However, this can always happen in the real world, regardless of how advanced the perception modules are made to be. To prepare for such scenarios, it is important to increase the robustness by also including these noisy data samples into the training set. Even though it is not possible to properly evaluate the driving behavior on such data samples, including noisy data would hopefully allow the model to give some reasonable predictions, at least until other safety measures can be kicked in to drive the vehicle to safety.

2.2.2 Output

For the sake of simplicity and convenience, the top-down view will also be used to represent the high-level output of the learning model. However, unlike the input to the model, the output will only consist of one single channel, containing the predictions of where the bounding box of the ego-vehicle is located in the future timesteps. This is equivalent to the first channel of the input top-down view described previously. It is also mathematically equivalent to a 4D tensor of dimension $(25, 800, 200, 1)$, given that the desire is for the model to predict 25 timesteps into the future. As an interesting note, since the output channel can always be thought of as an occupancy grid of 1s and 0s, the learning task is essentially a classification problem, aiming to classify each pixel either as occupied or not occupied. With such an occupancy grid, the model can easily be evaluated, since any collision between the ego-vehicle and another object can quickly be spotted out. Given the similarity between the top-down view of the input and the output, another advantage of our approach is that the output at a future timestep can easily be fed back as the input to the model to predict the next timestep. This technique will actually be used in one of the learning models introduced later in Chapter 3.

By predicting the locations of the bounding box over time, the learning model essentially outputs the positions and the headings of the ego-vehicle in future timesteps. However, to actually control the vehicle, much more information must be provided, such as the throttle levels or the angles of the steering wheel. While it is certainly possible to train the model to also predict these quantities as well, it would make the problem much more complex and difficult. The reason is because the gas pressures and the steering angles are both mathematically linked to the positions and the headings, which means that the model would also have to learn about these mathematical relationships under the hood. Instead, a much better idea is to simply output the locations of the bounding box, thus giving us the future trajectory of the ego-vehicle. A controller can be written to directly consume this trajectory, and then derive other quantities and low-level controls such as steering, gas, speed, and acceleration. This low-level controller is part of the modular pipeline presented in Figure 1.2, but is beyond the scope of the thesis.

2.3 Coordinate systems

To correctly plot the top-down view, the sequence generator requires a plugin that can perform conversions between different measurement units and coordinate systems. More specifically, the data pipeline has to deal with these coordinate systems:

1. The GPS coordinate system
2. The Local coordinate system
3. The Reference coordinate system (or Ref for short)

The GPS coordinate system is used by the GPS device to pin-point the exact location

of the ego-vehicle. It follows the WGS-84 standard[1], which models the Earth as an ellipsoid and sets up a non-Cartesian coordinate system with the origin located at the center of the Earth. A particular point on the surface of the Earth can then be specified by measuring the angles that this point makes with the center of the Earth. The angles are usually given in the unit of Decimal Degrees (DD), with the range of $[-180°, 180°]$ for longitudes and $[-90°, 90°]$ for latitudes. Apart from the GPS position, the GPS device also reports the heading of the ego-vehicle, which is basically the angle between the forward direction of the vehicle and the True North[2].

The Local coordinate system, on the other hand, is a standard x-y Cartesian coordinate system that can be used to directly plot the top-down view. It has the origin being attached to the position of the ego-vehicle, and it is always oriented in such a way that the y-direction coincides with the forward direction of the vehicle. Since the vehicle is moving and changing its direction at all time, the Local coordinate system is essentially a *non-static* coordinate system, and thus cannot be used to track the motion of the ego-vehicle itself. It is largely needed because all of the information about the surrounding, including the lanes and objects, are tracked and measured with respect to the data sensors installed on-board.

Since the GPS coordinate system cannot be used to capture information about the surrounding, and the Local coordinate system cannot be used to track the ego-motion, a third coordinate system is needed to combine them both. For this purpose, the Reference coordinate system (or Ref for short) is introduced. Ref has the exact same properties as the Local coordinate system, except for the fact that the origin is fixed at the location of ego-vehicle at the *present* timestep of a driving sequence, making Ref a *static* coordinate system. The main difference between the two coordinate systems lies exactly in this aspect. From the point of view of the Local frame of reference, the lanes appear as if they are moving backwards relative to the static ego-vehicle. On the other hand, the Ref coordinate system directly shows that the ego-vehicle is moving forward relative to the static lanes. A side-by-side visualization can actually be found on our project website [10].

Considering that a static frame of reference is able to capture both the surrounding and the ego-motion at the same time, a driving sequence can actually be defined by specifying the static coordinate system that was used for plotting the top-down view. In fact, every time the static coordinate system is changed with a different origin and orientation, it is as if an entirely new driving sequence has been generated. Therefore, it is important to convert all information from the driving sequence into the static Ref coordinate system, which can be achieved by applying some linear transformations to a 2D vector space, as will be shown later. Before that can be done however, it is necessary to first transform the ellipsoidal WGS-84 coordinate system into a flat Cartesian plane, using algorithms such as the UTM map projection.

[1] World Geodetic System is a standard used for the GPS navigation system that many people are familiar with. The latest revision of the standard was given in 1984, hence the name WGS-84.

[2] True North is the direction along a meridian on the Earth's surface towards the North Pole.

2.3.1 UTM map projection

Map projections are algorithms used to project coordinates on the surface of the Earth onto a flat 2D map. Typically, this involves wrapping an imaginary shape around the globe, and then projecting the points directly on the surface of this shape. A popular choice for such a shape is a cylinder, as used in the Mercator map projection and its variants, including the Universal Transverse Mercator (UTM). Since the surface of the Earth is curved, projecting it onto a plane will always cause some distortions in the shapes of objects, as well as in the directions and distances between them. UTM map projection was actually designed to minimize the amount of distortions, with the aim of achieving high accuracy for small regions on the map. This is ideal for the data pipeline since a driving sequence only needs to cover a very short driving distance at a time.

UTM divides the longitudinal range of the globe into 60 zones, with each being flattened onto a plane separately. The narrow width of each zone, which only spans 6° in longitudes, allows the projection to be done with little amount of distortions. To make it easier for referring to an area on Earth, the zones are further divided into 20 latitude zone bands, resulting in the UTM grid as shown in Figure 2.5. For each of the 60 zones, the entire flattened area will be superimposed by a standard Cartesian coordinate system—oriented in such a way that the x-axis is at the equator, and the y-axis coincides with the central meridian of that particular zone. Since each zone is superimposed by a different coordinate system, moving across the boundary of two zones is problematic. In such a rare case, the simplest solution is to approximate and assume that the trajectory lies entirely within one of the two zones.

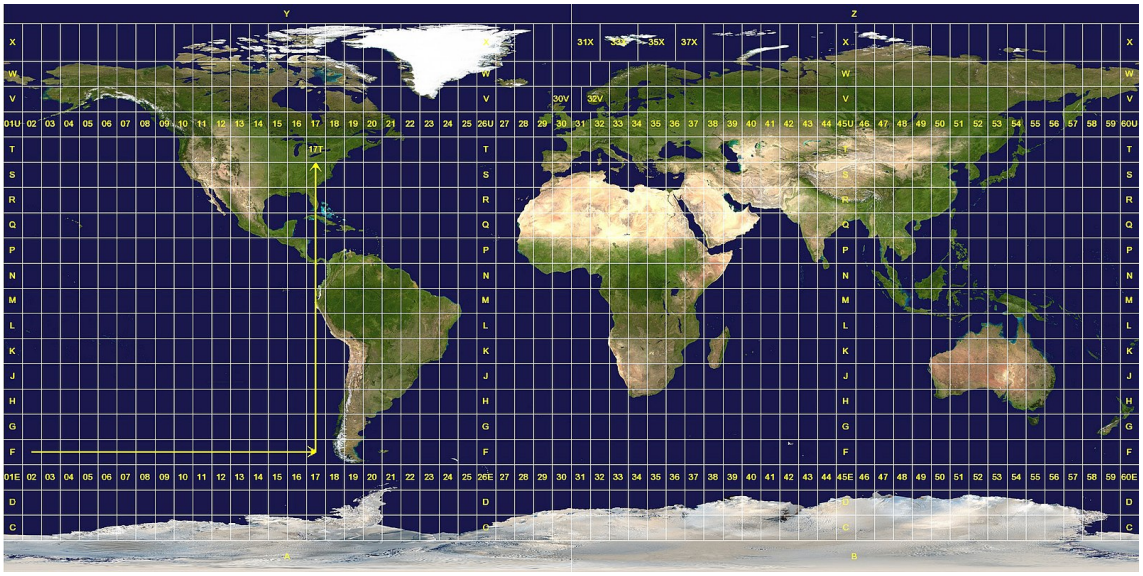


Figure 2.5: The UTM grid with 60 zones horizontally and 20 latitude zone bands vertically. Image taken from the public domain on Wikipedia Commons [46].

To complete the conversion from the WGS-84 standard to the UTM coordinate system, the headings of the ego-vehicle must also be converted as well. Since the curvature of the Earth has been flattened down onto a 2D plane, the headings

should now be measured with respect to the Grid North[3], instead of the True North as mentioned previously. The difference in angles between the two North directions is usually referred to as the *grid convergence*. As illustrated in Figure 2.6, the grid convergence can vary across different places on the map; and it is zero only at the equator and along the central meridian of a UTM zone, since those locations coincide exactly with the two axes of the Cartesian coordinate system that has been superimposed on that particular zone. In fact, exact calculations can be carried out by using simple trigonometry, as shown in [31]. Let λ and ϕ be the WGS-84 longitude and latitude of a point in a UTM zone, and λ_0 be the longitude of the central meridian of that zone. Then the grid convergence, γ, can be calculated as:

$$\gamma(\lambda, \phi) = \arctan(\tan(\lambda - \lambda_0) \sin(\phi)) \tag{2.1}$$

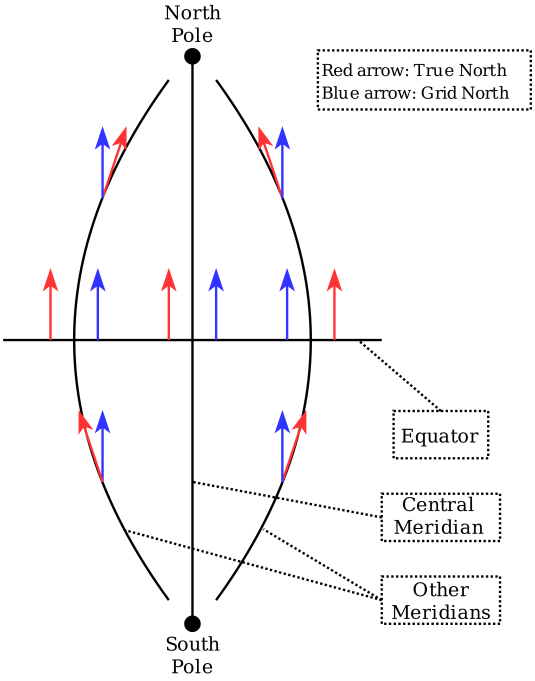


Figure 2.6: Illustration of the difference between the Grid North and the True North in an arbitrary UTM zone. Image drawn not to scale.

Due to the narrow width of the UTM zones, the grid convergence is typically very small, and thus can often be ignored in many applications. However, this is not the case here, since even a tiny change in the heading angle can cause a large visible change in the top-down view. The reason is because in the Ref coordinate xsystem, the top-down view is plotted from the perspective of the ego-vehicle. This means whenever the ego-vehicle heads at a different direction, the entire top-down view will have to be rotated accordingly. Therefore, it is important to take the grid convergence into account when using UTM map projection.

[3]Grid North is the direction pointing northwards along the grid lines in UTM map projection.

2.3.2 Conversions to Ref coordinate system

As mentioned previously, in order to plot the top-down views for a driving sequence, both the ego-motions and the surrounding information must be converted to the static Ref coordinate system. Thanks to the UTM map projection, the conversions to Ref can now be done by applying simple linear transformations to different Cartesian coordinates. These conversions, while simple, can actually be confusing and error-prone, since the Cartesian coordinate systems can be located at different origins and have different orientations.

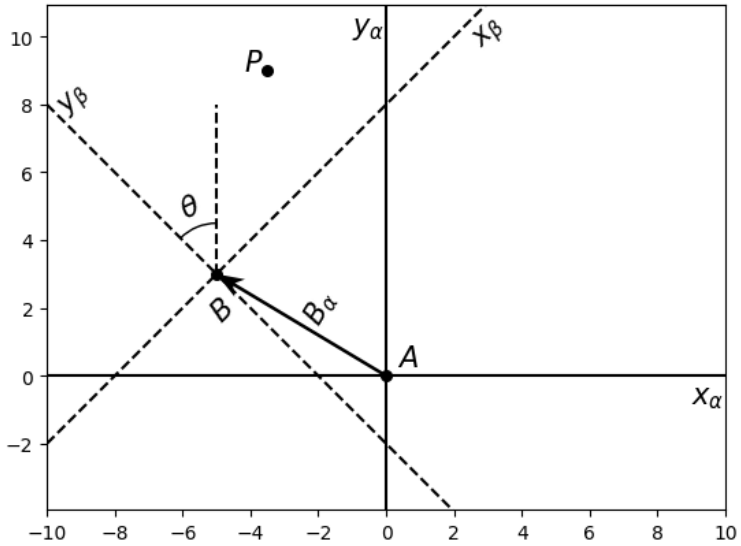


Figure 2.7: Plot of two arbitrary Cartesian coordinate systems, α and β.

Let α and β be two arbitrary x-y Cartesian coordinate systems, with the origins located respectively at point A and point B, as illustrated in Figure 2.7. Let's also denote θ as the angular difference between the y-axes of the two coordinate systems, and let B_α be the location of the origin B with respect to the α coordinate system. Now, assuming that both θ and B_α are given, then the coordinates of any random point P can be converted from α to β, by using a rotational matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$, as shown in the following equation:

$$\begin{gathered} P_\beta = \mathbf{R}(P_\alpha - \mathbf{t}) \\ \text{where:} \quad \mathbf{t} = B_\alpha \quad \text{and} \quad \mathbf{R} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \end{gathered} \tag{2.2}$$

The complete derivation of Equation 2.2 can be found in Appendix A. The strategy now is to use this generic equation to convert from GPS to Local coordinate system, and then apply the exact same technique to convert from Local to Ref coordinate system. To simplify the notations, let's denote each of these three coordinate systems with the subscripts g (GPS), l (Local), and r (Ref), such that:

- θ_{gl}, $\mathbf{R}_{gl}$, and $\mathbf{t}_{gl}$ specify the rotational matrix and translation vector used to transform from the GPS to the Local coordinate system.
- Similarly, θ_{lr}, $\mathbf{R}_{lr}$, and $\mathbf{t}_{lr}$ specify the rotational matrix and translation vector used to transform from the Local to the Ref coordinate system.

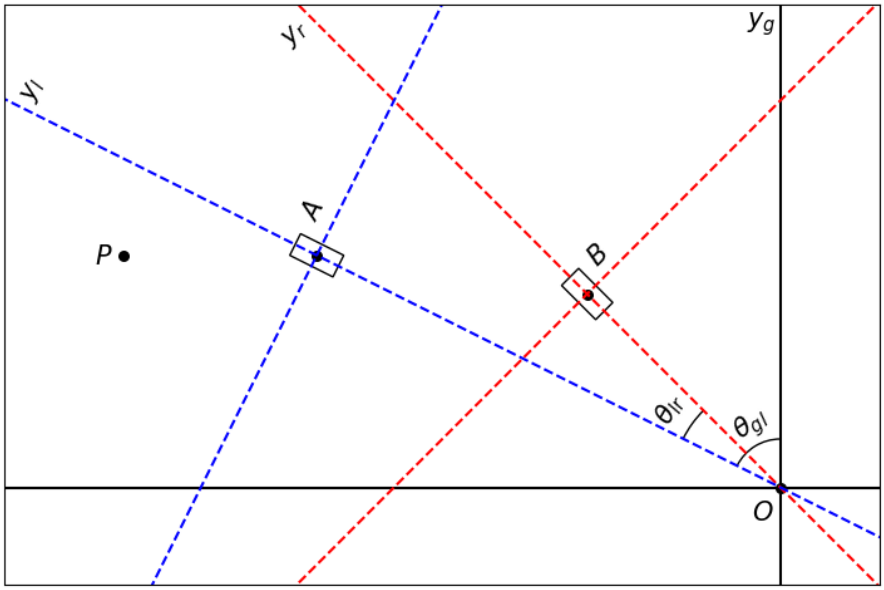


Figure 2.8: Illustration of the GPS (g), the Local (l), and the Ref (r) coordinate system, plotted from the perspective of the flattened GPS coordinate system.

Suppose that the origins of the Local and the Ref coordinate system are located at point A and point B, respectively, as illustrated in Figure 2.8. This means that:

- A_g is the GPS coordinates of the ego-vehicle at the timestep that the top-down view is being plotted, and θ_{gl} is the heading of the ego-vehicle at that timestep.
- B_g is the GPS coordinates of the ego-vehicle at the present timestep $t = 0$, and θ_{lr} is the difference in the heading angles between the two timesteps.

Once the values for the quantities $A_g, B_g, \theta_{gl}, \theta_{lr}$ are known, the generic Equation 2.2 can then be used to convert any arbitrary point P from the Local coordinate system to the Ref coordinate system as follows:

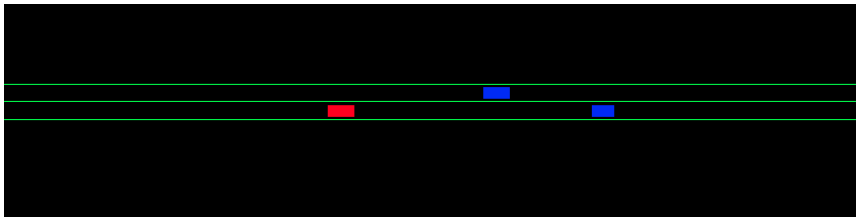
$$\begin{aligned} P_r &= \mathbf{R}_{lr}(P_l - \mathbf{t}_{lr}) = \begin{bmatrix} \cos(\theta_{lr}) & \sin(\theta_{lr}) \\ -\sin(\theta_{lr}) & \cos(\theta_{lr}) \end{bmatrix} (P_l - B_l) \\ \text{where:} \quad B_l &= \mathbf{R}_{gl}(B_g - \mathbf{t}_{gl}) = \begin{bmatrix} \cos(\theta_{gl}) & \sin(\theta_{gl}) \\ -\sin(\theta_{gl}) & \cos(\theta_{gl}) \end{bmatrix} (B_g - A_g) \end{aligned} \tag{2.3}$$

For a single top-down view, points along the lane markings and points around the bounding boxes of different objects, are all converted to the Ref coordinate system using the Equation 2.3 above. The same strategy can then be repeated for other top-down views as well, thus giving a complete driving sequence for training.

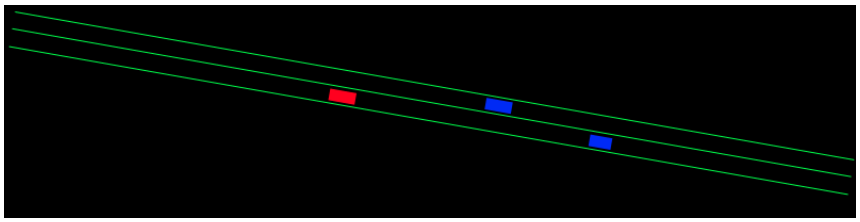
2.4 Data augmentation

Data augmentation is a set of techniques that are used to generate new data samples based on the existing ones, essentially enlarging the training dataset without actually having to collect more data. Having a larger training set can help the learning model to be more robust and generalize better towards unseen data. Two augmentation techniques are used for this project: (1) applying random rotations to the top-down views, and (2) synthesizing deceleration scenarios based on the collected data.

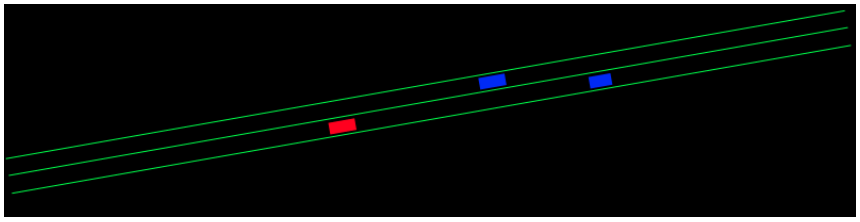
Since the focus of the thesis is on high-way scenarios, a large proportion of the data would only involve driving in a straight line along the y-direction of the top-down view. This might lead to the tendency of the model to always drive straight, regardless of whether of the lanes are straight or not. This is an example of an irrelevant pattern in the data, that can potentially trick the model into picking out the wrong features, instead of actually learning the underlying cause of the observed driving behavior. To tackle this, for each driving sequence in the training set, the top-down views will be rotated by a random angle in the range of $[-10°, 10°]$, as shown in Figure 2.9. Note that such rotations would violate the definition of the Ref coordinate system. However, since all of the top-down views in a sequence are rotated by the same angle, this still gives a valid driving sequence for training.



(a) Original top-down view without any rotation



(b) The same top-down view rotated by $-10°$



(c) The same top-down view rotated by $10°$

Figure 2.9: Data augmentation by rotations of the top-down view.

One benefit of using data augmentation is that it can also be used to synthesize driving scenarios that occur very rarely in the training dataset. This is particularly useful for imitation learning via behavioral cloning. As already mentioned in Section 1.2.2, if the autonomous agent ends up in a situation that has not been observed during training, the learning model would get stuck and even fail to predict anything at all. In our case, since much of the training data is about high-way scenarios where it is always possible to switch to a faster lane and perform an overtake, it is very rare to see a scenario where the ego-vehicle would have to decelerate for a slower car in front (in case overtaking is not desirable). Augmenting such scenarios into the training dataset would be hugely beneficial.

2.4.1 Augmentation of deceleration scenarios

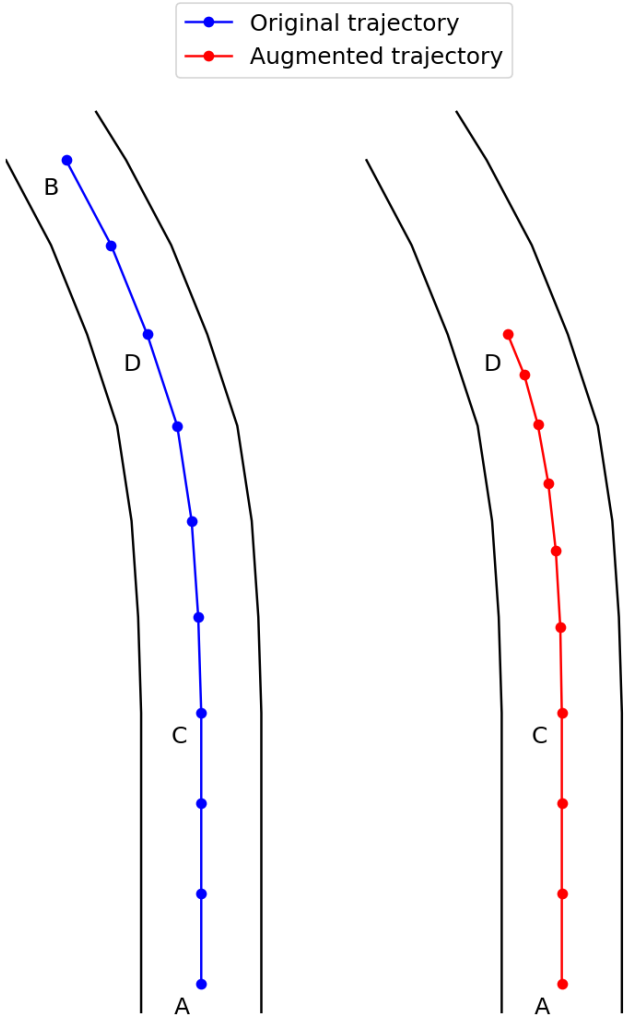


Figure 2.10: Illustration of an arbitrary driving trajectory (left) that has been shortened to create a deceleration scenario (right). Points C and D are chosen to mark the section in which the vehicle should decelerate (i.e., with a non-zero deceleration rate). The points along each trajectory are equally spaced in time.

Listed below is the set of criteria for the augmentation of deceleration scenarios:

- Augmentation should result in scenarios where the ego-vehicle moves at a constant speed for a while, and then decelerates to a slower speed. Note that it is important not to first accelerate and then decelerate, since it is not desirable to teach the learning model such bad behavior.

- Augmentation should strive for a uniform deceleration. While this might not be realistic in real life, it is still a good behavior and much simpler to simulate.

- As it does not make sense for the ego-vehicle to suddenly decelerate without any reasons, it is also necessary to synthesize an extra car that is driving slowly just in front of the ego-vehicle. Any vehicles from the original data that are within the path of this extra car must also be removed.

- The augmentation procedure should be fast and simple to compute. It should also take the curvature of the road into account, i.e., both the ego-vehicle and the extra car must not appear as if they are switching to a different lane.

Consider a trajectory from point A to B, as illustrated in Figure 2.10. To create a deceleration scenario based on this trajectory, the strategy is to simply shorten it, and then put all the points from the original trajectory to the new trajectory in such a way that the distances between the points decrease over time. The shortening is done by randomly picking two points (C and D) along the original trajectory, in order to mark the section where deceleration should take place.

Since the goal is to simulate uniform deceleration, the SUVAT equation of motions will be used; in particular, equations 2.4 and 2.5 below. Note that the two dimensions of the top-down view should be treated separately and independently from one another, which means the vector notations can be used. Here, t stands for the number of timesteps since the deceleration begins, $\mathbf{s}$ is the position of the vehicle at time t, $\mathbf{u}$ is the initial velocity at point C, $\mathbf{v}$ is the final velocity at point D, and $\mathbf{a}$ is the (non-zero) rate of deceleration.

$$\mathbf{s} = \mathbf{u}t + \frac{1}{2}\mathbf{a}t^2 \tag{2.4}$$

$$\mathbf{v} = \mathbf{u} + \mathbf{a}t \tag{2.5}$$

$$\text{where:} \quad \mathbf{s} = \begin{pmatrix} s_x \\ s_y \end{pmatrix}, \quad \mathbf{u} = \begin{pmatrix} u_x \\ u_y \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}, \quad \mathbf{a} = \begin{pmatrix} a_x \\ a_y \end{pmatrix} \neq \mathbf{0} \tag{2.6}$$

The main idea now is pick an appropriate deceleration rate $\mathbf{a}$, and then use it in the equation 2.4 to compute the position of every point along the new trajectory.

One naive method to pick $\mathbf{a}$ is to first choose the final point D, and then use its position to compute $\mathbf{a}$ via equation 2.4. However, this does not allow us to control the final velocity, since $\mathbf{v}$ is never part of the equation. Another alternative is to

choose the final velocity $\mathbf{v}$, and then use that to compute $\mathbf{a}$ via equation 2.5. This does not work either, since $\mathbf{s}$ is no longer part of the calculations, and thus it is not possible to control the final position of the ego-vehicle. A much better alternative is combine the two methods together. The key idea is to first pick a desired value for the final velocity, and then find the final position along the original trajectory that would best match this desired final velocity. (See Algorithm 1 below.)

Algorithm 1: Augmenting deceleration of the ego-vehicle

1. Randomly choose the initial point C. Then compute the corresponding initial velocity $\mathbf{u}$, and the total number of timesteps during deceleration.
2. Choose the final point D by doing the following:
   (a) Randomly pick a desired final velocity $\mathbf{v}'$ as some small ratio of $\mathbf{u}$.
   (b) Use $\mathbf{v}'$ to compute the desired deceleration rate $\mathbf{a}'$ via equation 2.5.
   (c) Use $\mathbf{a}'$ to compute the desired final position $\mathbf{s}'$ via equation 2.4.
   (d) Find a point along the original trajectory that is closest to the desired position $\mathbf{s}'$. Use the found point as the final point D.
3. Use the positions of the start and end points, C and D, to compute the actual deceleration rate $\mathbf{a}$ via equation 2.4.
4. Use the equation 2.4 again, but now with the actual deceleration rate $\mathbf{a}$, to compute positions of each and every point along the augmented trajectory.

While this algorithm might seem complicated at first glance, it only involves basic vector computations, and thus can be done very efficiently.

As the final step of the augmentation, an extra car of some random size will be augmented into the scene just in front of the ego-vehicle. This extra car will move at the same velocity as the final velocity of the ego-vehicle (i.e., the velocity at point D after the augmentation). All existing objects that are moving within the path of this extra car will also be removed.

3

Model design

This chapter focuses on the designs of different neural network architectures for the DecisionNet, which is the module responsible for making the driving decisions for the autonomous vehicle. As described in the previous chapter, the DecisionNet will receive from the data pipeline the top-down views that represent the past and present of the driving sequence as input, and then predict a series of top-down views representing the future trajectory of the ego-vehicle.

To deal with the past and the future separately, an encoder-decoder architecture will be used. The key idea is to have the encoding phase to encode information from the past (and also the present); while the decoding phase is for decoding the hidden states to give the predictions of the future. This will be referred to as the **two-phase** variant of the DecisionNet. Later on in the last section of the chapter, this two-phase variant will be modified a little bit to give us the **one-phase** variant, which will allow the model to both predict and observe the future at the same time, thus solving the so-called **time-horizon problem**.

The architectures for the DecisionNet are actually built upon many popular neural network architectures that have already existed in the literature. These network architectures will be described in detail in Section 3.1. Among these, the most important ones for the DecisionNet are CNN, ENet, and ConvLSTM.

3.1 Popular network architectures

3.1.1 FFNN

FFNN (or Feed Forward Neural Network) is the most basic form of a neural network architecture, which was designed based of the inspiration of the network of neurons in the human brains. However, instead of the biological neurons, FFNN is made up of artificial neurons, organized in multiple layers. Each artificial neuron can be thought of as a function, mapping the inputs that come from the neurons in the previous layer, to an output that can be connected to the neurons in the next layer.

In the context of FFNN, such function mapping can be expressed recursively as:

$$\mathbf{z}_l = \mathbf{W}\mathbf{a}_{l-1} + \mathbf{b}_{l-1} \tag{3.1}$$

$$\mathbf{a}_l = f(\mathbf{z}_l) \tag{3.2}$$

where $\mathbf{z}_l$ represents the linear combination of the inputs $\mathbf{a}_{l-1}$ from the previous layer; $\mathbf{a}_l$ is the output of a (typically non-linear) activation function applied on $\mathbf{z}_l$; and the weights $\mathbf{W}$ and biases $\mathbf{b}_{l-1}$ are learnable parameters of the network.

Let m and n be the number of neurons in layer $l-1$ and layer l, respectively. Then it can be deduced that $\mathbf{a}_l$, $\mathbf{b}_{l-1}$, and $\mathbf{z}_l$ are all n-by-1 vectors; $\mathbf{a}_{l-1}$ is an m-by-1 vector; and $\mathbf{W}$ is an n-by-m matrix.

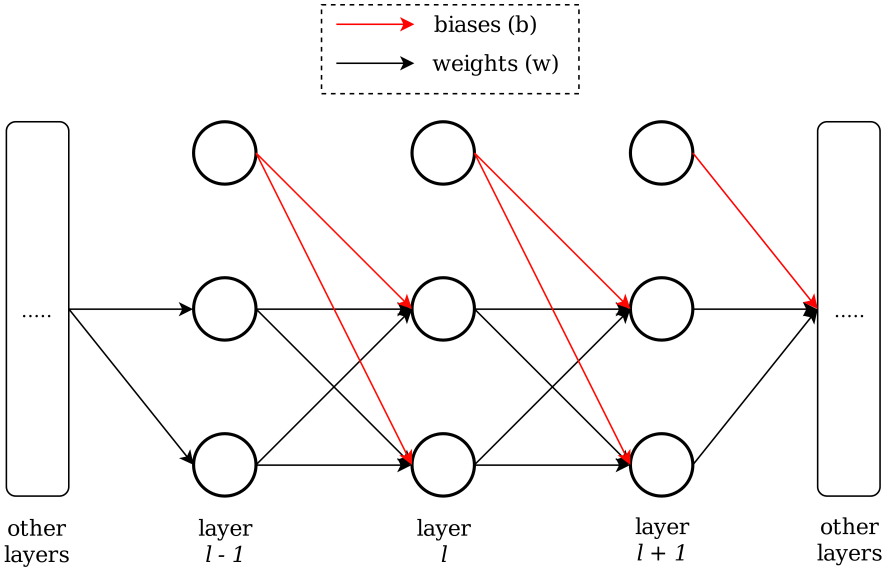


Figure 3.1: Example of a deep FFNN with biases.

Figure 3.1 shows an example of how the artificial neurons from different layers are connected to each other in a typical FFNN. The number of layers is called the depth of the network, which is essentially where the term "deep learning" originates from. The deeper the network, the more capable it is of learning more complex features from the data. This is due to the non-linearity introduced by the non-linear activation functions applied at each layer, as shown in Equation 3.2. In fact, it can easily be shown that without any activation function, the deep structure of the network would have no meaning, since all the layers can simply be collapsed into a single-layer neural network. Therefore, choosing the right non-linear activation functions and having a deep structure are usually the keys to success in deep learning.

There are many candidates for a good activation function. Some of the most popular ones include Sigmoid, Tanh, ReLU, and PReLU, which will be used later on in the thesis. Sigmoid and Tanh are often used as squashing functions in order to squash the values between some specific range. This range is $(0, 1)$ for Sigmoid, and $(-1, 1)$ for Tanh. On the other hand, ReLU function allows the value to go

up to positive infinity, and sets all negative values to zero. However, this often can lead to a phenomenon known as "dying" neurons [25, 38], because a negative value will immediately give zero gradient, thus stopping the learning process altogether in typical gradient descent algorithm used to optimize the network. Due to this reason, PReLU (or Parametric ReLU) was introduced as an alternative. This function is very similar to ReLU, but instead of setting all negative values to zero, it uses an extra learnable parameter α to avoid the problem with the zero gradient [14, 49].

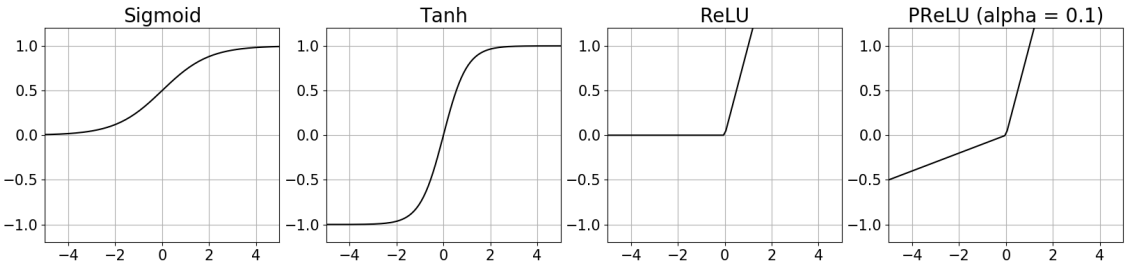


Figure 3.2: Plots of some popular activation functions.

Mathematically, the activation functions in Figure 3.2 can be expressed as follow:

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}} \tag{3.3}$$

$$\tanh(z) = \frac{\sinh z}{\cosh z} = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{3.4}$$

$$\text{relu}(z) = \left\{ \begin{array}{ll} 0, & \text{for } z \leq 0 \\ z, & \text{for } z > 0 \end{array} \right\} \tag{3.5}$$

$$\text{prelu}(z) = \left\{ \begin{array}{ll} \alpha z, & \text{for } z \leq 0 \\ z, & \text{for } z > 0 \end{array} \right\} \tag{3.6}$$

3.1.2 CNN

CNN (or Convolutional Neural Network), originally proposed by [22], is a type of neural networks most commonly used for learning features from inputs that are equivalent to 3D tensors of dimension $(H, W, C) = (\text{height}, \text{width}, \text{channels})$. In particular, CNN can be used for learning from RGB images, and thus has received a huge success within the field of computer vision [12, 13, 45]. Part of the reason for this success is due to the CNN's ability to learn important spatial features from the the images with much less parameters, especially when compared to FFNN.

A single CNN layer contains a number of so-called *kernels*, which are basically windows of relatively small size (typically 3 to 7 pixels on each dimension). These kernels hold the learnable parameters that can capture various spatial information from the input feature maps, which have been output from the previous layer. A convolution operation, denoted by $*$, will then slide each of these kernels across the width and the height dimensions of the input feature maps, matching the input with

the kernel, in order to create a new feature map to be passed on to the next layer (see the graphical illustration from [29]). This type of convolution is most commonly referred to as Conv2D (since the sliding operation is applied across 2 dimensions of the input), and can be expressed mathematically as:

$$\mathbf{Y} = f(\mathbf{W} * \mathbf{X} + \mathbf{b}) \tag{3.7}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are the input and output of the Conv2D layer, $\mathbf{W}$ is the set of kernels, and $\mathbf{b}$ contains a set of learnable biases (one bias for each kernel, typically).

To work out the dimensions of each quantity in Formula 3.7, let the input $\mathbf{X}$ be a 3D tensor of dimensions (H_x, W_x, C_x). Now suppose that there are N_k kernels in the layer, then $\mathbf{W}$ will be a 4D tensors of dimension (N_k, H_k, W_k, C_x). Note that each kernel must have exactly the same number of channels as the input, as Conv2D only allows the kernels to be slided across the width and height dimensions. Finally, the convolution operation should result in the output $\mathbf{Y}$ of size (H_y, W_y, N_k).

As usual, the activation function f should be applied pixel-wise across the feature map in order to introduce non-linearity into the network. With such activation applied at every layer, the key is to construct the final network by stacking multiple CNN layers on top of each other, resulting in a deep structure that would allow the network to learn more complex features from the input.

In the classical form of convolutions, the kernels are typically moved across the feature maps with a step size of one. However, in *non-unity strided* convolutions [21], the step size can be set to a value higher than one. This can be utilized as a downsampling technique [39], which reduces the computational resources of the network, thus allowing us to stack even more layers of CNN on top of each other to increase the depth of the network. The downsampled feature maps can then be upsampled again using various upsampling techniques, such as nearest-neighbor interpolation, where each pixel value is copied to other pixels in the nearest neighborhood. These techniques mentioned here will later be used in the thesis as an efficient way to downsample and upsample the top-down views.

3.1.3 ENet

ENet (or Efficient Net) is a neural network architecture that was originally proposed by [33] to be used for semantic segmentation of images in real-time system. However, it was later found that ENet could also be utilized for many other deep machine learning tasks as well, such as image classification as shown in [6], or monocular depth estimation as shown in [28]. This is no surprise, since the design of ENet was built primarily based on a variety of different types of convolutional layers, which have been shown to be very good at learning spatial features from the data. ENet is not only capable, but is also computationally efficient, and thus it would be beneficial to take advantage of such an architecture in order to efficiently extract the spatial information contained within the top-down views.

The efficient performance of ENet is the result of the combination of many state-of-the-art techniques already existing in the deep learning literature. In particular, ENet utilizes the so-called *bottleneck* architecture [15], which uses skip connections, just like in any typical residual neural networks (or ResNets). For each "bottleneck residual block", the idea is to have a 1×1 convolutional layer with fewer kernels to reduce the dimensionality of the input tensor when entering the block; and then have another 1×1 convolutional layer to expand (or restore) the dimensionality when exiting the block. Such changes in dimensionality mean smaller tensors, and thus cheaper computations for the bottleneck parts inside each block. Together with skip connections, these bottleneck blocks make it easier to efficiently stack more convolutional layers on top of each other to create a deeper network.

ENet is actually organized as an encoder-decoder architecture. Here, the authors choose to have more learning parameters in the encoder, instead of opting for a more symmetric architecture. This means the encoder will do most of the processing on the downsampled input, and the decoder only needs to upsample the output back to the original resolution and fine-tuning the details. Downsampling the resolution is crucial for reducing the computational cost; but it can also hurt the performance, and thus should only be deployed in the first few layers of the encoder, where the computations are often the most expensive. In ENet, downsampling in the encoder is achieved by both max-pooling and non-unity strided convolutions; and conversely, upsampling in the decoder is achieved by max-unpooling and transposed convolutions, as suggested by [52].

One extra benefit of downsampling the resolution is that it will give the network a wider receptive field. This is because as the resolution gets smaller, each neuron will be able to "perceive" a wider area of the input, thus taking more into account the visual context of the surrounding scenes. However, since aggressive downsampling can also hurt the performance, the authors of ENet found that it is much better to achieve wider receptive field using dilated convolutions [50]. The main idea of this technique is to perform convolutional operations on non-consecutive pixels of the input tensor, thus allowing each kernel to be applied on a wider area. In ENet, dilated convolutions are used inside 4 bottleneck blocks of the encoder.

Last but not least, asymmetric convolutions [41] are used in some bottleneck blocks as a way to further optimize the network. In these blocks, instead of convolving with symmetric kernels of size $n \times n$, the idea is to decompose these kernels into two layers of asymmetric kernels of size $n \times 1$, followed by kernels of size $1 \times n$. This reduces the number of learnable parameters from n^2 to $2n$, thus removing potential redundancy, and also giving a larger speedup. Moreover, as non-linear activations can also be added between these two layers, the network becomes more expressive.

3.1.4 LSTM

LSTM (or Long Short Term Memory), originally proposed by [16], is a type of a recurrent neural network that can be used for many sequence modelling problems.

It is capable of extracting temporal information, thus learning both the long-term and short-term dependencies of an input sequence. The key innovation of LSTM is the usage of memory cells, with the read-write access being controlled by a set of gates, that can be open or closed based on the network's learning experience. The mathematical description of such a memory cell can be summarized by the equations below, which follows the formulation given by [48], with some slight modifications.

$$\begin{aligned} i_t &= \text{sigmoid}(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ f_t &= \text{sigmoid}(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\ o_t &= \text{sigmoid}(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ \tilde{c_t} &= \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c_t} \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{3.8}$$

Here, $\odot$ denotes the Hadamard's element-wise multiplication. W's and b's represent the learnable weights and biases. x_t is the input of the current timestep. c_t and h_t are the cell states and the hidden states of the memory cell. Note that x_t, c_t, and h_t are all vectors of some particular dimensions that should be chosen depending on the problem being tackled. Finally, i_t, f_t, o_t represent the three control gates of the cell, namely, the *input* gate, the *forget* gate, and the *output* gate, respectively.

As clearly shown from the Equation 3.8, all three gates of the LSTM cell are activated by the sigmoid activation function, which outputs a value between 0 and 1, where 0 represents a closed gate, and 1 represents an open gate. Therefore, these gates control the flow of information within the LSTM network. The forget gate f_t controls how much information from the previous cell states should be discarded from the memory. The input gate i_t controls how much information from the current input x_t that should be accumulated to the memory. Lastly, the output gate controls what information to output for the current timestep. The LSTM network will try to learn the best values for these gates, using the set of weights W's and biases b's that are shared among all memory cells.

Multiple LSTM layers can also be stacked on top of one another to create a deeper structure, that would allow the network to learn more complex behaviors from the input sequences. In such an architecture, the outputs h_t of an LSTM layer will be passed on as the inputs x_t to the next layer.

As the final remark, encoder-decoder architecture is very popular in LSTM, as it allows the network to take the entire input sequence into account before predicting the output sequence. For example, in the work of Sutskever et al. [40] for machine translations, the encoder is used to only extract all information from the English sentence, while the translation to French is actually done in the decoder. Similar strategy will be used in our thesis, with the idea of encoding information from the past driving sequence, before predicting the future driving sequence in the decoder.

3.1.5 ConvLSTM

Even though LSTM is capable of learning the temporal information from the input sequence, one major drawback with LSTM is that it cannot be used to learn any spatial information, because the input x_t in Equations 3.8 are just simple vectors that contain no spatial dimensions. To tackle this issue, Xingjian et al. [48] introduces ConvLSTM (or Convolutional LSTM), which aims to combine the capabilities of both CNN and LSTM together into a single architecture, allowing it to be able to process both spatial and temporal information at the same time.

Mathematically, a ConvLSTM memory cell behaves exactly the same way as an LSTM cell, as clearly shown by the similarity between Equations 3.8 and Equations 3.9 below. But instead of using simple vectors, the inputs ($\mathcal{X}_t$), the gates ($\mathcal{I}_t$,, $\mathcal{F}_t$, $\mathcal{O}_t$), and the internal memory states ($\mathcal{C}_t$, $\mathcal{H}_t$) are all 3D tensors, thus allowing the cell to also capture the spatial information. Using $*$ and $\odot$ to denote the convolutional operation and the Hadamard's product, the mathematical operations performed by a ConvLSTM cell can be expressed as:

$$\begin{aligned}
\mathcal{I}_t &= \text{sigmoid}(W_{xi} * \mathcal{X}_t + W_{hi} * \mathcal{H}_{t-1} + b_i) \\
\mathcal{F}_t &= \text{sigmoid}(W_{xf} * \mathcal{X}_t + W_{hf} * \mathcal{H}_{t-1} + b_f) \\
\mathcal{O}_t &= \text{sigmoid}(W_{xo} * \mathcal{X}_t + W_{ho} * \mathcal{H}_{t-1} + b_o) \\
\tilde{\mathcal{C}}_t &= \tanh(W_{xc} * \mathcal{X}_t + W_{hc} * \mathcal{H}_{t-1} + b_c) \\
\mathcal{C}_t &= \mathcal{F}_t \odot \mathcal{C}_{t-1} + \mathcal{I}_t \odot \tilde{\mathcal{C}}_t \\
\mathcal{H}_t &= \mathcal{O}_t \odot \tanh(\mathcal{C}_t)
\end{aligned} \tag{3.9}$$

Like any other deep neural networks, ConvLSTM layers can also be stacked on top of each other, thus allowing the network to deeply process the spatial information for each timestep. However, it should be noted that stacking ConvLSTM can be very expensive computationally. Every time a new ConvLSTM layer is added, eight convolutional operations need to be performed for each timestep, as clearly shown by Equations 3.9. On the other hand, adding an extra CNN layer only requires an extra convolutional operation, which means CNN is much cheaper than ConvLSTM in terms of both computation time and memory consumption.

Nevertheless, with the ability to learn both spatial and temporal information at the same time, ConvLSTM proves to be very useful for many applications related to video processing, such as future frames predictions demonstrated in [11, 24, 44]. In a much similar fashion, one can think of the driving sequence as a video over the top-down views, and thus ConvLSTM will serve as the backbone for the DecisionNet.

3.2 DecisionNet architecture

This section focuses on the architectural design of the DecisionNet—the module responsible for processing the top-down views (TDVs) from the past and the present,

in order to predict the next TDVs representing the future trajectory of the ego-vehicle. As mentioned previously, all proposed architectures will have ConvLSTM layers at the core, such that the networks can process both spatial and temporal information at the same time. Since ConvLSTMs are computationally expensive, the idea is to not stack them to create a deep structure. Instead, several layers of CNNs will be used to increase model capacity, while keeping the consumption of computational resources as minimal as possible. This results in a "sandwich" architecture, where ConvLSTMs are wrapped around by CNN layers.

As a starting point, we design a very simple network consisting of standard Conv2D layers, to be used for wrapping around the ConvLSTMs. Our simple network will be referred to as SNet, and will be composed of an encoder and a decoder, both of which will be presented in Section 3.2.1. As will be shown later, we will eventually replace SNet with ENet, in order to improve the performance of the final network.

In our "sandwich" architecture, each frame of the input driving sequence will first be spatially processed by SNet/ENet encoder to generate the corresponding hidden representations. Downsampling the resolution will be done in this stage to increase computational efficiency, and also widen the receptive field. These hidden representations are then passed on to the ConvLSTMs, which will unroll in the temporal dimension all the way into the future. Finally, the outputs from the ConvLSTMs will be fed into the SNet/ENet decoder to upsample the resolution and generate one-channel TDVs that actually present the final outputs of the DecisionNet.

3.2.1 SNet-ConvLSTM-SNet

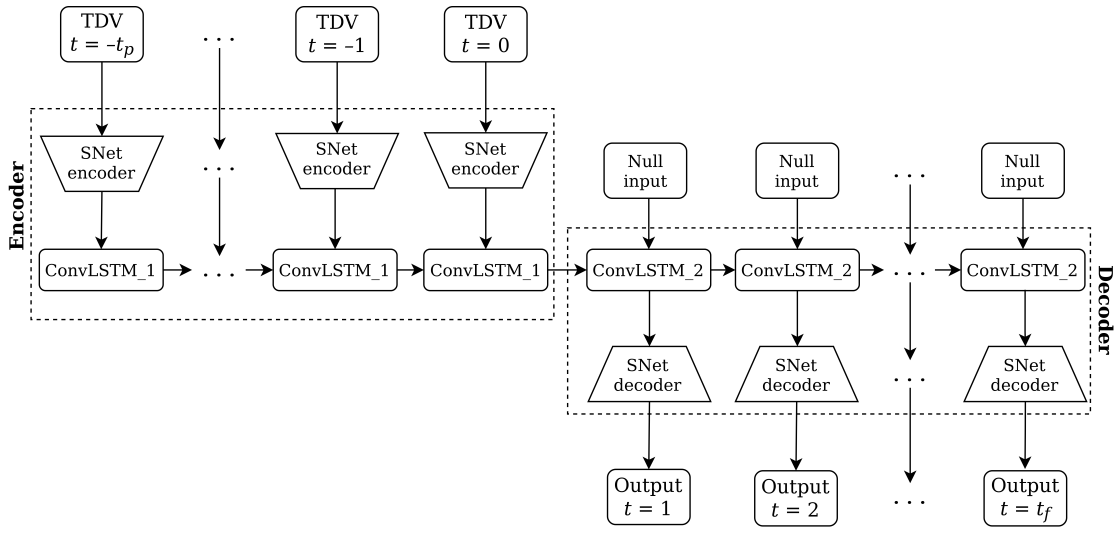


Figure 3.3: Two-phase SNet-ConvLSTM-SNet architecture, shown for a sequence of t_p past frames and t_f future frames. Downward arrows represent flow of spatial information. Rightward arrows represent flow of temporal information. Components with the same name also share the same training parameters.

Shown in Figure 3.3 above is the encoder and decoder of our SNet-ConvLSTM-SNet sandwich architecture, which has been unrolled along the time dimension of

the driving sequence. This type of architecture should, from now on, be referred to as the **two-phase variant** of the DecisionNet, since it splits a single forward pass through the network into two separate phases. The encoding phase encodes information from the top-down views that come from the past and the present. The decoding phase will then unroll the hidden representations, and output the top-down views representing the future trajectory of the ego-vehicle.

Layers	**Kernel Shape**	**Downsampling Factors**	**Output Shape**	**# Params**
input	—	—	$(800, 200, 3)$	—
conv2d_1	$(3, 3, 3)$	$(2, 2, 1)$	$(400, 100, 16)$	448
conv2d_2	$(3, 3, 16)$	$(2, 2, 1)$	$(200, 50, 32)$	4640
conv2d_3	$(3, 3, 32)$	$(2, 2, 1)$	$(100, 25, 64)$	18496
conv2d_4	$(3, 3, 64)$	$(1, 1, 1)$	$(100, 25, 128)$	73856

Table 3.1: Overview of the SNet encoder, where downsampling of the resolution is done by non-unity strided convolutions.

Layers	**Kernel Shape**	**Upsampling Factor**	**Output Shape**	**# Params**
input	—	—	$(100, 25, 128)$	—
conv2d_5	$(3, 3, 128)$	$(1, 1, 1)$	$(100, 25, 128)$	147584
conv2d_6	$(3, 3, 128)$	$(1, 1, 1)$	$(100, 25, 64)$	73792
upsampling2d_1	—	$(2, 2, 1)$	$(200, 50, 64)$	0
conv2d_7	$(3, 3, 64)$	$(1, 1, 1)$	$(200, 50, 32)$	18464
upsampling2d_2	—	$(2, 2, 1)$	$(400, 100, 32)$	0
conv2d_8	$(3, 3, 32)$	$(1, 1, 1)$	$(400, 100, 16)$	4624
upsampling2d_3	—	$(2, 2, 1)$	$(800, 200, 16)$	0
conv2d_9	$(3, 3, 16)$	$(1, 1, 1)$	$(800, 200, 8)$	1160
conv2d_10	$(1, 1, 8)$	$(1, 1, 1)$	$(800, 200, 1)$	9

Table 3.2: Overview of SNet decoder, where upsampling of the resolution is done by nearest-neighbor interpolations. The last Conv2D layer is only used for reshaping the final output and fine-tuning the details.

SNet architecture is mostly made up of relatively simple and standard Conv2D layers, as clearly shown by Tables 3.1 and 3.2. The encoder and decoder also appear to be quite symmetrical; each having 4 to 6 layers deep, with ReLU activation functions being inserted in between to obtain non-linearity. Downsampling is done in the encoder using 3 layers of stride-2 convolutions, shrinking the original resolution of 800×200 down to only 100×25. This allows us to add more kernels, thus increasing the number of learnable parameters and capacity of the model. The resolution is then restored in the decoder using 3 layers of nearest-neighbor interpolations.

Sitting between the SNet encoder and decoder are two thin layers of ConvLSTM, designed to have the properties as described in Table 3.3. These ConvLSTM layers will not be stacked on top of each other, but instead will be concatenated along the time dimension, passing the internal memory states ($\mathcal{H}_t$ and $\mathcal{C}_t$ in Equation 3.9) from the first layer into the second layer, thus allowing the network to predict all the way into the future.

With the second ConvLSTM layer already being initialized by the internal memory states from the first layer, there is no need for the decoder to receive any extra inputs. Implementation-wise, this will be equivalent to passing the *null inputs* into the ConvLSTM. These null inputs are basically tensors of only zeros, which will zero out the gradients during a typical gradient descent algorithm, thus allowing the network to ignore these inputs completely.

Layers	**Kernel Shape**	**Input Shape**	**Output Shape**	**# Params**
convLSTM_1	$(3, 3, 128)$	$(t_p + 1, 100, 25, 128)$	—	1180160
convLSTM_2	$(3, 3, 128)$	—	$(t_f, 100, 25, 128)$	1180160

Table 3.3: ConvLSTM layers, serving as backbone for the DecisionNet. The layers are concatenated along the time dimension, for a sequence of t_p past TDVs and t_f future TDVs. Number of learnable parameters can be computed with Equation 3.9.

Looking at the sandwich architecture as a whole, it might seem like we have put too little effort on extracting the temporal information from the data, considering that there are only 2 layers of ConvLSTM. This, by no means, implies that processing the spatial dimensions is more important than the time dimension. If anything, it actually reflects the intertwined relationship between space and time, because learning more about the spatial dimensions also helps the network to learn and predict how the top-down views are changing over time. This is also the idea behind ConvLSTM, as the purpose of ConvLSTM is learn the spatio-temporal information, instead of treating space and time as separate entities.

3.2.2 ENet-ConvLSTM-ENet

While SNet is a great starting point for DecisionNet, it does not great performance. This is not surprising, as SNet is relatively simple and shallow, compared to many state-of-the-art deep neural networks already existing in the literature. However, thanks to the usage of the sandwich architecture presented earlier, it is quite a trivial task to replace SNet with a much more powerful network such as ENet, thus creating the ENet-ConvLSTM-ENet architecture as can be seen in Figure 3.4. ENet helps boost the performance, while consuming the computational resources as efficiently as possible. One advantage is that ENet has already been carefully designed by the authors of the original paper [48], thus saving us the troubles of fine-tuning the network ourselves, which can be a very tedious process.

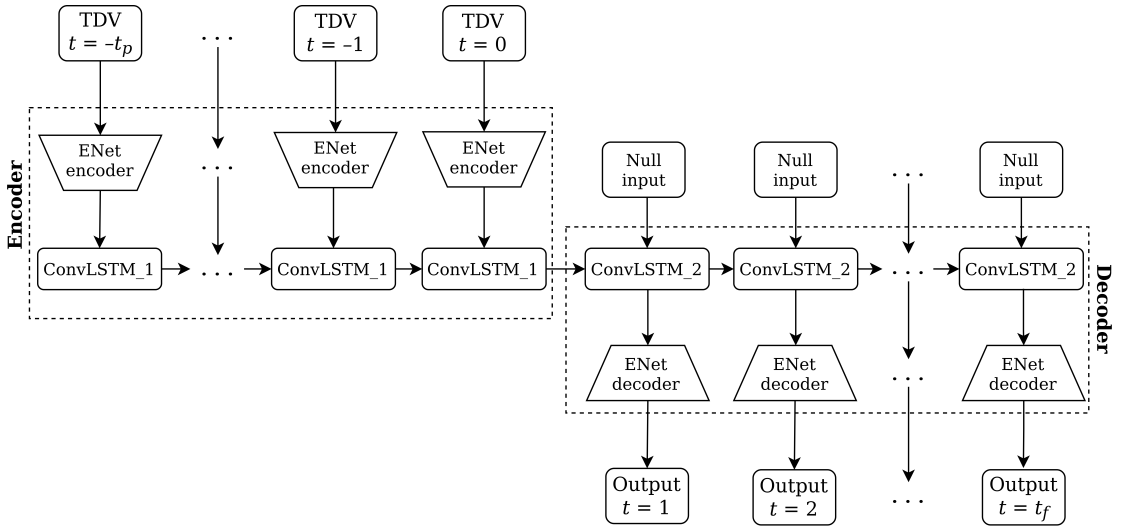


Figure 3.4: Two-phase ENet-ConvLSTM-ENet architecture, shown for a sequence of t_p past frames and t_f future frames. Downward arrows represent flow of spatial information. Rightward arrows represent flow of temporal information. Components with the same name also share the same training parameters.

Unlike SNet, which is only a few layers deep, ENet has 27 bottleneck blocks that are stacked on top of each other [33]; and within each block are also several layers of convolutions. Such depth allows ENet to pack lots of non-linearities internally, thus allowing the network to learn more complex spatial features than SNet. As previously described in Section 3.1.3, ENet uses many optimization techniques to maintain this depth efficiently in terms of computations. These techniques include the usage of 1×1 convolutions to reduce the number of kernels; max-pooling and non-unity strided convolutions for downsampling the resolutions; dilated convolutions to expand the receptive field with smaller kernels; and finally, asymmetric convolutions for substantially reducing the number of learnable parameters.

3.2.3 The time-horizon problem

One major drawback of using the two-phase variant of the DecisionNet, in which the encoder and decoder are run separately from one another, is that the network will not be able to see how the situation on the road changes over time in the future timesteps. This is called the time-horizon problem, since the network is not permitted to see anything beyond the present timestep.

For most scenarios, time-horizon problem is not an issue, because with good training, the network should be able to indirectly predict the future motions of surrounding cars, and then adjust the trajectory of the ego-vehicle accordingly. But in some cases, such accurate predictions are hard to achieve, or even impossible. Consider a scenario where a car right in front of the ego-vehicle is moving at a constant speed, and then suddenly decides to decelerate in the future. Since the network can only see the past and the present, it will predict the speed of that car to be constant at all time, and thus will not prepare to decelerate to avoid a collision.

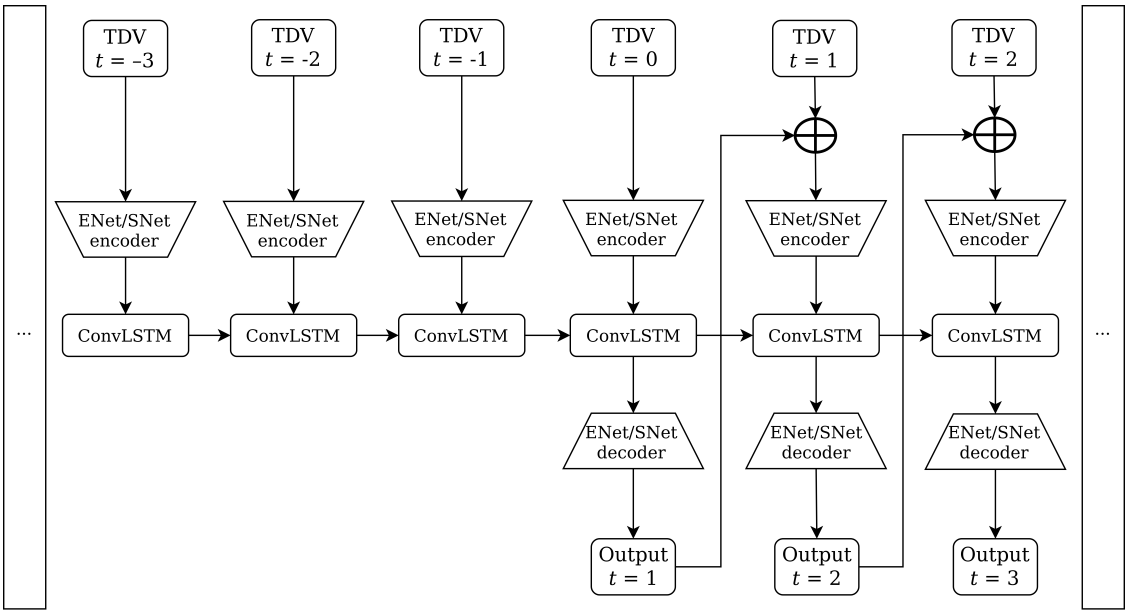


Figure 3.5: One-phase variant of DecisionNet, in which the network is allowed to dynamically observe the situation on the roads, while simultaneously predicting the future ego-trajectory. $\bigoplus$ denotes the concatenation of the 1-channel output with the 2 last channels of the input top-down views, thus creating a 3-channel input that can be fed again into the network. Only one layer of ConvLSTM is used here.

The only solution to the time-horizon problem is to allow the network to also observe the situation on the roads at the same time as it is trying to predict the future ego-trajectory. This is the idea behind the **one-phase variant** of the DecisionNet. In this setup, there will no longer be an encoder and a decoder, but instead, the entire forward pass will run in one single phase. The output at a future timestep t will be used to drive the vehicle towards the predicted position. And then the entire scenes at timestep t will be captured as a top-down view, which can then be fed back into the network to produce the next output for timestep $t + 1$.

In the one-phase variant, the idea of feeding the output of one timestep as input to the next timestep is partially inspired by Sutskever et al., in their LSTM architecture designed for machine translations [40]. It was easy to apply this technique in their LSTM model because the inputs and outputs are essentially the same, with both being one-hot vector representations of words in the same vocabulary. However, this is much harder to do for the DecisionNet, since its inputs and outputs are TDVs containing different numbers of channels. Thus, as illustrated in Figure 3.5, the trick is to concatenate the 1-channel output from the network (which represents the future position of the ego-vehicle), with the last 2 channels from the ground-truth data (which represent the lanes and positions of other cars), in order to obtain a 3-channel input that can be fed into the network again. Initially, it might seem like the network is cheating by looking at the ground-truth data. But if analyzed carefully, this is conceptually the same as if the ego-vehicle is being driven by a model that can dynamically observe the situation on the roads in real-time.

4

Training and evaluation

This chapter focuses on the training and the evaluation of the DecisionNet.

Section 4.1 starts with the optimization strategy for the DecisionNet, describing the choice of the loss function as well as how the gradient descent algorithm should be carried out. Next, we will go through the strategy for random data sampling, thus showing how the entire dataset is split into the train/validation/test sets. Finally, 5 different models for the DecisionNet will be chosen for training. These models are all based on the architectures previously shown in Chapter 3.

Section 4.2 outlines different evaluation strategies, which can be performed either quantitatively or qualitatively. Quantitative methods will be applied on the test set of 1000 samples, in order to give a more objective and relatively unbiased evaluation of the models. Qualitative evaluations, on the other hand, will be carried out by visual inspections. The idea is to hand-pick a small set of relevant driving scenarios, and then manually check how well the models perform on these scenarios.

Finally, all the results will be summarized in Section 4.3. Please also take a look at our project website [10] for more extensive results.

4.1 Training of DecisonNet

4.1.1 Loss function

In supervised learning, the training of a neural network often requires a good loss function, which typically is used to quantify the accuracy of the model with respect to some ground-truth data. In the context of imitation learning, this is also known as the *imitation loss*, since it measures how good the model is at imitating the human expert driver. The imitation loss will be optimized during training, such that the average loss decreases over time as training goes on, as this would imply that the model is getting better, and the predictions are getting closer to the ground truth.

Since both the inputs and outputs of the DecisionNet are top-down views containing the bounding box of the ego-vehicle, it makes sense to directly compare the model

predictions, pixel by pixel, against the ground truth. Thus, the mean-square-error loss (or MSE for short) would be a good candidate for the imitation loss.

Let $W \times H$ be the resolution of the top-down view, and t_f be the total number of future timesteps to be predicted. Let $B_k(x, y)$ be the pixel that is located at the spatial coordinates (x, y) of the predicted output at timestep k. Similarly, let $B_k^{gt}(x, y)$ be the corresponding pixel in the ground truth data, fetched from the first channel that contains the bounding box of the ego-vehicle. Then the MSE loss can be computed mathematically as:

$$L_{\text{MSE}} = \frac{1}{t_f W H} \sum_{k=1}^{t_f} \sum_{x=1}^{W} \sum_{y=1}^{H} \Big(B_k(x, y) - B_k^{gt}(x, y) \Big)^2 \tag{4.1}$$

4.1.2 Gradient descent

Gradient descent (GD), which often goes hand in hand with back-propagation, is one of the most popular algorithms for optimizing a neural network. Training typically begins with the random initialization of all learnable parameters. Next, we perform a forward pass through the network to compute the loss with respect to the ground truth, such that during back-propagation, the loss can then be propagated backwards through all the layers in order to compute the partial derivatives of the loss with respect to each of the parameters. These partial derivatives represent the steepest slopes (or gradients) that can used for updating the parameters, thus guiding the algorithm to descend towards a lower loss. This process should be repeated until the loss function reaches convergence at some (hopefully global) minimum.

Let g_k be the gradient with respect to the parameter θ during the iteration k. Then a parameter update in the GD algorithm can be expressed mathematically as:

$$\theta_k = \theta_{k-1} - \alpha \cdot g_k \tag{4.2}$$

Here, the constant learning rate (α) must to be chosen very carefully. If the rate is too high, then there might a risk of over-shooting through the minimum, thus preventing the algorithm from ever reaching a convergence. On the other hand, a learning rate that is too low can substantially slow down the optimization, since less ground of the loss function's hyperspace can be explored.

To avoid the tedious process of tuning α, one idea is to train the DecisionNet with an *adaptive learning rate* method, that can adapt α to each individual parameter as the training goes on. One such method is called Adam [20], which is actually a combination of Momentum and RMSProp [36]. The main idea of Adam is to adapt individual learning rates according to the running averages of both the past gradients and the squares of the past gradients. These mathematical operations are summarized in Equation 4.3, which is just a modification of Equation 4.2 above.

$$
\begin{aligned}
m_k &= \frac{\beta_1 \cdot m_{k-1} + (1-\beta_1) \cdot g_k}{1-{\beta_1}^k} \\
v_k &= \frac{\beta_2 \cdot v_{k-1} + (1-\beta_2) \cdot {g_k}^2}{1-{\beta_2}^k} \\
\theta_k &= \theta_{k-1} - \frac{\alpha}{\sqrt{v_k}+\varepsilon} \cdot m_k
\end{aligned}
\tag{4.3}
$$

Following the recommendation from the original authors, the hyper-parameters are set with the values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$. The default learning rate α is set to be 10^{-4}. Note that this is only the initial value, as the rate will eventually be adapted to each individual parameter during training. Parameter updates will be done in mini-batches, with each mini-batch containing between 4 to 8 data samples, as this is the maximum size that can fit in the memory of our GPUs.

4.1.3 Data split

As mentioned in Chapter 2, the collected data of high-way scenarios amounts to approximately 150 hours of continuous driving. This data will be randomly shuffled, and then split into the train/validation/test sets at the ratio of 90:5:5, thus giving us 135 hours for the train set, and 7.5 hours each for the validation and test set.

In machine learning, an *epoch* usually refers to a training loop that passes through all the data samples in the train set. With 135 hours, the train set corresponds to more than 120000 non-overlapping 4-second driving sequences, and can even grow larger if overlapping is used. Thus, going through each and every training example would be infeasible, as that would take too long. Instead, for each loop, it is preferable to randomly sample a small subset from the train set. This, of course, means that less data will be used for each loop. But if the network is trained for long enough, the entire train dataset can still potentially be observed. So for each training loop, the idea is to configure the data pipeline such that it can produce:

- 4000 driving sequences, randomly sampled from the train set. As these samples are different for every loop, the average training loss can fluctuate over time, but should still generally decrease in a downward trend.

- 1000 driving sequences, randomly sampled from validation set. Note that the random sampling should only be done once, in order to keep the same samples for all the training loops. Otherwise, it would be difficult to compare which set of parameters actually gives the best generalization towards unseen data.

The set of parameters that gives the lowest loss on validation set will be saved for later evaluations. In order to save some time, only 1000 driving sequences will be used for testing the models. These sequences are randomly sampled from the test set, and hopefully can give a good representative for all scenarios we want to achieve.

4.1.4 Models to be trained

Shown in Table 4.1 are 5 different models that have been chosen for training. These models will be evaluated and compared against each other, in order to give an insight into the pros and cons of different architectures and training methods.

Models	Architecture	Variant	Data Augmentation	Past Motion Dropout
$\mathcal{M}_1$	SNet-ConvLSTM-SNet	Two-phase	No	No
$\mathcal{M}_2$	ENet-ConvLSTM-ENet	Two-phase	No	No
$\mathcal{M}_3$	ENet-ConvLSTM-ENet	Two-phase	No	Yes
$\mathcal{M}_4$	ENet-ConvLSTM-ENet	Two-phase	Yes	No
$\mathcal{M}_5$	ENet-ConvLSTM-ENet	One-phase	Yes	No

Table 4.1: Descriptions of 5 different models for the DecisionNet.

To make sure the output only contains pixels with values in the range of $(0, 1)$, the last layer of all models will be activated by ReluClip, which is similar to the ReLU function in Equation 3.5, but with all values larger than 1 being capped at 1. This appears to work quite well when training with the MSE loss presented earlier.

Data augmentation will only be performed for $\mathcal{M}_4$ and $\mathcal{M}_5$, using the strategies described in Chapter 2.4. The top-down views will be rotated at some random angles between $(-10°, 10°)$ for *all* driving sequences in the train set. On the other hand, only 10% of all training examples will be augmented with deceleration scenarios; anything more than 10% will risk overfitting the network, causing the DecisionNet to decelerate too often, even in undesireable situations. Note the data augmentation will only be performed during training, and not during evaluation of the models.

Finally, it would also be interesting see if *past motion dropout* [3] can replace data augmentation as a way to tackle deceleration scenarios. The main idea of this is to hide the motions of the ego-vehicle from the past timesteps for 50% of the data, thus forcing the network to pay more attention to the surroundings, instead of relying too much on the past motions. Our hope is that this can teach model $\mathcal{M}_3$ to decelerate for a slow car, without the need to even perform data augmentation.

4.2 Evaluation strategies

4.2.1 Objective quantitative evaluations

Designing an objective evaluation strategy can be challenging, as it is difficult to agree on what is considered to be the best driving behavior. The imitation learning approach addresses this issue by simply assuming that the human expert driver always gives the best driving behavior, and thus can act as the ground truth. With

this in mind, we came up with two different metrics (MSE and MPD), that can be used to objectively quantify how well the models can imitate the human expert driver. These metrics will be computed on an unseen test set of 1000 samples.

The MSE metrics is calculated in exactly the same way as the MSE loss shown earlier in Equation 4.1. It measures how well the model can imitate at the pixel level, and thus also indirectly taking into account the shape and the orientation of the bounding box of the ego-vehicle. However, one big problem with MSE is that it does not have any unit of measurement, which means that it can be difficult to interpret and visualize what a particular MSE score actually shows.

To address the problem with MSE, we introduce Mean Positional Deviations (or MPD for short), which measures the average distance in meters of how far the predicted positions deviate from the ground truth. For example, an MPD score of 2 meters means that on average, the model predicts the ego-vehicle to be 2 meters away from where it is expected to be. The idea is to compute the deviation d_t for each future timestep t, and then take the average of all these deviations.

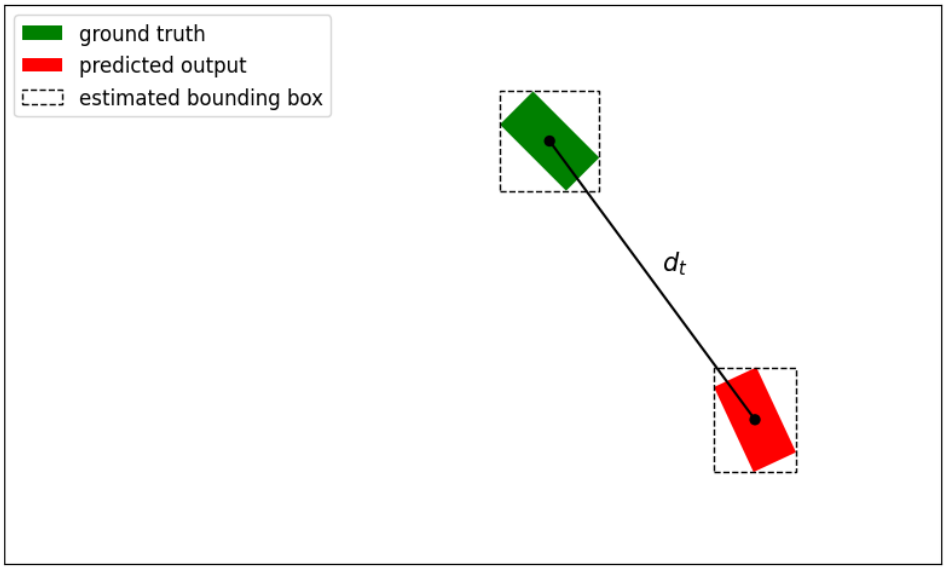


Figure 4.1: Illustration of how the positional deviation d_t is computed in order to figure out the MPD score. Image drawn not to scale.

To make it easier to extract the position of the ego-vehicle from the top-down view, a simple idea is to fit an *estimated* bounding box around the *actual* bounding box of the vehicle, as shown in Figure 4.1. This works because both estimated and actual bounding box have the same center that represents the position of the vehicle.

In the case of inadequate training, it is possible that the model might predict the vehicle to be in multiple places at once. Such predictions will result in a larger estimated bounding box, and can potentially lead to a higher MPD score. And if the model does not manage to predict anything at all, then we assume the vehicle has stayed at exactly the same place, which will also lead to a higher MPD score.

4.2.2 Evaluations by visual inspection

While MSE and MPD can give a very objective evaluation, these metrics can only measure how good a model is at imitating the human expert driver. This is a huge limitation, since not being able to imitate well does not automatically mean that the model is bad. One way to address this concern is to extract a small set of scenarios that can be used to manually evaluate the models by visual inspection. This set will consist of 100 scenarios, which are divided into 4 different categories:

- *Straight-lane scenarios*, where the lanes appear to be straight on the TDVs.
- *Curved-lane scenarios*, where the lanes appear to be curved on the TDVs.
- *Missing-lane scenarios*, where the lanes are either half-missing or completely missing from the TDVs. For these scenarios, we cannot expect the models to give the same behavior as the ground truth. But it can be interesting to see if the models can predict anything, or will fail completely due to missing lanes.
- *Slow-speed scenarios*, where the ego-vehicle appears to travel very slowly.

For each of the scenarios above, the main idea is to manually count how many timesteps into the future that the model can successfully predict with a high level of confidence. More concretely, a good prediction would mean that the model does not output any erroneous behavior (such as decelerating unexpectedly, or crossing to another lane for no reasons, etc.). The predicted bounding box of the ego-vehicle should also be relatively sharp and clear on the TDVs.

As mentioned in Chapter 2.4, deceleration scenarios are particularly interesting to test our models on. But since these scenarios are extremely rare in the dataset, finding them would be very time-consuming. Instead, it might be much easier to simply synthesize these scenarios from scratch. This is exactly the benefit of using the top-down view for high-level data representation, since synthesizing such a scenario would only require some simple plotting tools or libraries. Two deceleration scenarios will be synthesized, each aiming at testing different things:

- *Deceleration for a slow car*: In the past timesteps of this synthesized scenario, the ego-vehicle is set to travel at a very high speed. On the same lane, there will an extra car in front that will move at a much slower speed. In this case, the DecisionNet is expected to decelerate to avoid collision with the slow car.
- *Deceleration for a decelerated car*: Similar to the first synthesized scenario, the ego-vehicle is also set to travel at a very high speed. However, in this case, the extra car in front will move at a high speed in the past timesteps, and will then decelerate to a much slower speed in the future timesteps. The expectation here is that the DecisionNet should still be able to decelerate to avoid collision with this decelerated car in front.

4.3 Results

Video clips, that show the outputs of the models for each of the driving scenarios listed in Table 4.4 and Table 4.5, can be found on our project website [10].

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$
MSE scores ($\times 10^{-4}$)	6.55	3.12	2.98	2.76	2.42

Table 4.2: Pixel-wise MSE scores, obtained when evaluating the 5 models on the test set of 1000 samples. The lower the score, the better the model.

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$
MPD scores (meters)	5.09	0.62	0.60	0.46	0.41

Table 4.3: MPD scores (in meters), obtained when evaluating the 5 models on the test set of 1000 samples. The lower the score, the smaller the deviations are between the predicted positions and the ground-truth positions.

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$
Straight-lane scenarios	8.2	24.6	24.6	24.6	24.9
Curved-lane scenarios	11.1	18.9	19.9	21.8	23.4
Missing-lane scenarios	17.2	17.8	17.8	20.1	21.3
Slow-speed scenarios	22.8	20.7	22.0	23.8	24.7

Table 4.4: Results showing the performance of the 5 models on the 4 categories of extracted scenarios mentioned in Section 4.2.2. For each of the categories, this table shows the average number of future timesteps that the models can successfully predict. The higher the number, the better the model. Note that the maximum number of future timesteps available is $t_f = 25$.

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$
Decelerate for a *slow* car	✗	✗	✗	✓	✓
Decelerate for a *decelerated* car	✗	✗	✗	✗	✓

Table 4.5: Results showing the performance of the 5 models on the 2 deceleration scenarios that were synthesized from scratch, as explained in Section 4.2.2. Here, we use a binary criteria, where success (✓) means that the ego-vehicle has managed to decelerate to avoid a collision, and failure (✗) means the opposite.

5

Discussion

Looking at the results from Table 4.4, most models seem to perform very well on both straight-lane and slow-speed scenarios. This is no surprise, because in a dataset that contains mostly of high-way scenarios, the drivers will most likely find themselves in situations where they either drive very fast along straight lanes, or very slowly in a queue of busy traffic. On the other hand, curved-lane and missing-lane scenarios seem to be a bit more difficult for the models, possibly because these scenarios are relatively less abundant in our dataset.

Nevertheless, the performance of all models on the missing-lane scenarios seems to be a lot better than our expectations. Model $\mathcal{M}_5$ even managed to successfully predict, on average, 21 out of 25 timesteps into the future. This shows the importance of also including noisy data when training the models, as that would help the DecisionNet to be more robust towards the imperfections of the data pipeline.

All in all, the results from Chapter 4.3 show that $\mathcal{M}_1$ consistently performs worse than other models, while $\mathcal{M}_5$ gives the best performance. For the rest of this chapter, the the focus will be on comparing the models directly against each other in order to explain why we have such difference in the performance.

5.1 $\mathcal{M}_1$ versus $\mathcal{M}_2$

Although $\mathcal{M}_1$ and $\mathcal{M}_2$ have exactly the same method of training, the performance of $\mathcal{M}_1$ on the test set is much worse compared to $\mathcal{M}_2$. The MPD score of $\mathcal{M}_1$ is 5.09 meters, which is quite unacceptable. On the other hand, $\mathcal{M}_2$ only deviates 0.62 meters from the ground truth on average. This huge gain in performance is thanks to the switch in the network architecture from SNet over to ENet.

Compared to our custom-made SNet, the ENet architecture is much deeper in the number of hidden layers, thus packing more non-linearity that would allow the network to learn more complex features from the input sequence. In addition to the deep architecture, ENet also uses dilated convolutions to further expand the receptive field, allowing the network to capture more surrounding context from the input, while also covering a larger area of the resolution in the output.

There is one peculiar observation, however. While $\mathcal{M}_1$ seems to fail on straight-lane scenarios, it surprisingly performs very well on slow-speed scenarios. A theory is that this could be due to the narrow receptive field of SNet. In straight-lane scenarios, the ego-vehicle often travels very fast across the entire resolution of the top-down views. This means that the small receptive field of SNet will make it harder for the ConvLSTM to unroll its hidden states in order to reach the upper portion of the top-down views. On the other hand, in slow-speed scenarios, most of the predictions for the positions of the ego-vehicle will fall into the middle portion of the resolution, thus allowing $\mathcal{M}_1$ to predict far into the future. However, this is only a hypothesis, and more investigations will need to be done in order to confirm this.

As mentioned, switching from SNet over to ENet has given $\mathcal{M}_2$ a huge gain in performance over $\mathcal{M}_1$. Unfortunately, however, this is not enough, as $\mathcal{M}_2$ still fails on the deceleration scenarios shown in Table 4.5. This is a clear indication that the training of the DecisionNet has to go beyond the pure behavior cloning approach, which is exactly why later models $\mathcal{M}_3$ and $\mathcal{M}_4$ were introduced.

5.2 $\mathcal{M}_3$ versus $\mathcal{M}_4$

The first attempt at tackling the deceleration scenarios was to use past motion dropout as in model $\mathcal{M}_3$. As mentioned in previous chapters, the main idea behind past motion dropout is to hide the past motion of the ego-vehicle. This is to force the DecisionNet to pay more attention to the surroundings, such that when it sees a slow-moving car in front, it would decelerate to avoid a collision. However, that did not work as well as we had hoped. The reason is simply because while past motion drop helps improve the general performance of the model, it cannot be used to help the model learn about scenarios that are rarely seen during training.

In order to obtain more deceleration scenarios for the train dataset, $\mathcal{M}_4$ uses the strategy described in Chapter 2.4 to augment the deceleration scenario based on the real data collected. Such augmentation allows $\mathcal{M}_4$ to successfully decelerate for a slow car, without ever having to collect more data.

Incidentally, augmentation also seems to help $\mathcal{M}_4$ achieve a better performance on other scenarios as well, and not just deceleration scenarios. The MSE and MPD scores of $\mathcal{M}_4$ are a bit higher than those of the previous models. The same thing is observed for its performance on the extracted scenarios. This indicates that the rotation of the top-down views must have helped the model to be better at following lanes, while the augmentation of deceleration scenarios also helped the model to pay more attention to the surrounding cars, thus boosting the overall performance of $\mathcal{M}_4$.

5.3 $\mathcal{M}_4$ versus $\mathcal{M}_5$

The last scenario we would like to tackle is the deceleration of the ego-vehicle to avoid crashing into a decelerated car. Unfortunately, $\mathcal{M}_4$ did not manage to give a

good prediction for this synthesized scenario. This is actually due to the limitation of the two-phase variant of the DecisionNet, which is caused by the time-horizon problem already described in Chapter 3.2.3. Since the model is not allowed to see the situation on the roads beyond the present time step, there is no way for the network to be able to predict that the car in front will decelerate in the future.

The limitation of the time-horizon problem can be addressed by using the one-phase variant of the DecisionNet, as in model $\mathcal{M}_5$. In this architecture, the model is also allowed to observe the surrounding scenes in the future timesteps, simultaneously as it is trying to predict the new positions for the ego-vehicle.

As can be seen from the results, $\mathcal{M}_5$ does not only perform well on the synthesized scenarios, it also gives better performance than $\mathcal{M}_4$ on other test scenarios as well. This is no surprise, since $\mathcal{M}_5$ has access to more information about the future timesteps, and thus can actively adjust the driving trajectory depending on what it sees in the future. The only downside here is that instead of simply consuming the null inputs as in the two-phase network in Figure 3.4, the one-phase variant in Figure 3.5 will also have to process the inputs from the future timesteps as well, thus increasing the overall computation time and memory consumption.

6

Conclusion

This chapter aims to answer all the questions listed in Chapter 1.3, by summarizing some of the most important findings in our thesis project:

- Using top-down views (TDVs) as high-level data representation makes it easier to accurately capture the spatio-temporal information from the surrounding environment. It is also quite straight-forward for the DecisionNet to encode its driving decisions as part of the TDVs. This simplifies the problem a lot, and also allows the output from one timestep to be re-connected as an input to the next timestep, as we have done in the one-phase variant of the DecisionNet.

- A big challenge with plotting the TDVs is how to choose the correct coordinate system. In Chapter 2.3, we have shown that careful calculations must be done in order to convert the GPS and the Local coordinates into the Ref coordinate system. Since the Ref coordinate system plots the the TDVs with respect to the position of the ego-vehicle at the present timestep, this allows the DecisionNet to learn how the ego-vehicle moves across the TDVs over time.

- ConvLSTM can be used as a way to simultaneously process both the spatial and temporal information of the TDVs. The capability of ConvLSTM can further be enhanced by adding more CNN layers as part of our custom-made SNet architecture. However, as it turns out, SNet seems to be too shallow in terms of the number of layers. Hence, it much better to replace it ENet, an off-the-shelf architecture that is much deeper and more efficient.

- The time-horizon problem introduces a limitation that prevents the network from seeing far beyond into the future. This even makes it impossible for the DecisionNet to successfully predict for the synthesized scenario where the ego-vehicle has to decelerate behind a decelerated car. Fortunately, a simple way to solve this is to convert the encoder-decoder two-phase architecture into the one-phase variant of the DecisionNet. The one-phase variant allows the model to observe the situations on the road, simultaneously as it is trying to predict the future trajectory of the ego-vehicle.

- With 150 hours of data collected from the demonstrations of the human expert driver, our findings show that imitation learning has given the models quite a good performance on many high-way scenarios. However, it turns out that the pure behavior cloning approach is not sufficient for tackling deceleration scenarios that occur very rarely in our dataset. To address this problem, data augmentation can be used to generate more data for these scenarios, as this will allow us to teach the DecisionNet to slow down to avoid a collision with the car in front. Note that such data augmentation would be impossible to perform on the raw sensor data. This explains why using TDVs as high-level data representation is so advantageous for imitation learning.

- Thanks to imitation learning, the performance of DecisionNet can be evaluated by simply comparing its output directly against the ground-truth data from the human expert driver. This is exactly the purpose of the MSE and MPD scores, which can objectively quantify how well the models were able to imitate the expert for the scenarios contained in the test set.

- One important thing to remember is that not being able to imitate well does not automatically imply that the model is bad. For this reason, it is very beneficial to also include evaluation by visual inspection, which would allow us to manually judge the performance of the models ourselves. This type of evaluation can either be done on a small set of scenarios extracted from the dataset, or even on the scenarios that have been synthesized completely from scratch. Note that without the usage of the TDVs, synthesizing a new scenario would be extremely difficult, if not impossible.

7

Future work

This chapter focuses on identifying some areas where later research can build upon.

7.1 Full system deployment

Our thesis has mostly been focusing on the development of the DecisionNet and its interface when connecting to other modules. However, the modular pipeline in Figure 1.2 cannot be completed without a proper controller.

Therefore, part of the future work should focus on developing a controller that can consume the driving decisions from the DecisionNet, and then convert them into low-level controls that can actually drive the ego-vehicle. This would also allow the performance of the models to be tested in a simulated environment, or even on an actual test car, using a closed-loop evaluation.

7.2 Adding more features and scenarios

Including only the lane markings and the bounding boxes of the vehicles greatly hinders what the DecisionNet can actually imitate. Adding additional road features (such as traffic lights, speed limits, street signs, etc.) can help the DecisionNet to make more informed decisions on what driving actions to take. These new features can be added simply as new channels in the input top-down views.

As of now, the DecisionNet is mostly limited to high-way scenarios, since those are the only ones used for training. However, the future work should focus on tackling more challenging scenarios that involve driving on busy streets of urban cities, as it would be interesting to see how imitation learning can help the models to learn about the complex interactions between different vehicles on the road.

7.3 Using of High Definition maps

One of the biggest challenges with the generations of the top-down views is how to deal with noisy data coming from various on-board sensors. As previously shown in

Chapter 2.2.1, noisy data can result in scenarios where lanes are completely missing from the top-down views, making it difficult even for humans to navigate. To address this limitation of the noisy on-board sensors, High Definition (HD) maps can be used to equip the models with more precise information from the surroundings.

HD maps are actually pre-built maps containing highly precise digital models of the road networks, with the error margin of only a couple of centimeters [18]. Such high level of precision would help create more accurate top-down views, making it easier to train and evaluate the DecisionNet. Note that there are challenges of working with HD maps, too; and one particular problem is how to accurately self-localize the ego-vehicle within this map environment [51]. All of these challenges must be overcome if HD maps are to be considered for the future work.

7.4 Going beyond imitation loss

As mentioned in Chapter 4.1.1, the training of the DecisionNet has been relying solely on the pixel-wise MSE loss. This loss allows the optimization algorithm to compare the output top-down views directly against the ground-truth data, thus measuring how well the models can imitate the human expert driver.

However, the authors of the Chauffeurnet paper [3] have argued that there are benefits of going beyond the imitation loss. They have added, for examples, the collision loss (which punishes the model in case of collisions), the on-road loss (which prevents the vehicle from going off the road), and many others. These losses make it easier to train the DecisionNet, since the models will now have an extra feedback coming from the environment, instead of solely relying on the imitation loss to imitate the behavior. Note that these losses can also be used as evaluation metrics for the models as well, and thus should be considered for the future work.

7.5 Conditional imitation learning

So far, the DecisionNet has been outputting the driving decisions based solely on the perceptual inputs. However, this would never work in practice, because driving always requires some sort of intentions. For example, the action to switch lane to overtake another car should only be performed if it is intentional to do so, because alternatively, the vehicle would just need to slow down to stay in the same lane.

To take driving intentions into consideration, one way is to allow the DecisionNet to also receive an extra input that represents the low-level commands, such as turn left or turn right, speed up or slow down, etc. This is exactly the idea behind conditional imitation learning [9]. Note that these low-level commands can come either from a human passenger, or from an automatic route-planning module.

Bibliography

[1] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[2] Paul Bakker and Yasuo Kuniyoshi. Robot see, robot do: An overview of robot imitation. In *AISB96 Workshop on Learning in Robots and Animals*, pages 3–11, 1996.

[3] Mayank Bansal, Alex Krizhevsky, and Abhijit Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *arXiv preprint arXiv:1812.03079*, 2018.

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

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

[6] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[7] 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.

[8] Zhilu Chen and Xinming Huang. End-to-end learning for lane keeping of self-driving cars. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1856–1860. IEEE, 2017.

[9] Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9. IEEE, 2018.

[10] Hai Dinh and Zeeshan Ul Hassan Dar. Project website with additional resources and materials for the thesis. URL https://sites.google.com/view/imitationlearningthesisproject/.

[11] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. In *Advances in neural information processing systems*, pages 64–72, 2016.

[12] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*, 2017.

[13] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187: 27–48, 2016.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[17] Christopher Innocenti, Henrik Lindén, Ghazaleh Panahandeh, Lennart Svensson, and Nasser Mohammadiha. Imitation learning for vision-based lane keeping assistance. *arXiv preprint arXiv:1709.03853*, 2017.

[18] Jialin Jiao. Machine learning assisted high-definition map creation. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 367–373. IEEE, 2018.

[19] Jinkyu Kim and John Canny. Interpretable learning for self-driving cars by visualizing causal attention. In *Proceedings of the IEEE international conference on computer vision*, pages 2942–2950, 2017.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Chen Kong and Simon Lucey. Take it in your stride: Do we need striding in cnns? *arXiv preprint arXiv:1712.02502*, 2017.

[22] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10): 1995, 1995.

[23] Zhihao Li, Toshiyuki Motoyoshi, Kazuma Sasaki, Tetsuya Ogata, and Shigeki Sugano. Rethinking self-driving: Multi-task knowledge for better generalization and accident explanation ability. *arXiv preprint arXiv:1809.11100*, 2018.

[24] William Lotter, Gabriel Kreiman, and David Cox. Deep predictive coding networks for video prediction and unsupervised learning. *arXiv preprint arXiv:1605.08104*, 2016.

[25] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.

[26] Matthias Müller, Alexey Dosovitskiy, Bernard Ghanem, and Vladlen Koltun. Driving policy transfer via modularity and abstraction. *arXiv preprint arXiv:1804.09364*, 2018.

[27] 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.

[28] Davy Neven, Bert De Brabandere, Stamatios Georgoulis, Marc Proesmans, and Luc Van Gool. Fast scene understanding for autonomous driving. *arXiv preprint arXiv:1708.02550*, 2017.

[29] Andrew Ng. C4w1l06 convolutions over volumes [video lecture from deeplearning.ai]. URL https://www.youtube.com/watch?v=KTB_OFoAQcc. [Online; Accessed August 31, 2019].

[30] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

[31] Peter Osborne. *The Mercator Projections: The normal and Tranverse Mercator Projections on the Sphere and the Ellipsoid with full derivations of all formulae*. 2013. doi: 10.5281/zenodo.35392. URL http://doi.org/10.5281/zenodo.35392.

[32] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016.

[33] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *arXiv preprint arXiv:1606.02147*, 2016.

[34] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[35] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[36] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[37] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments. *arXiv preprint arXiv:1806.06498*, 2018.

[38] Yeonjong Shin and George Em Karniadakis. Trainability and data-dependent initialization of over-parameterized relu neural networks. *arXiv preprint arXiv:1907.09696*, 2019.

[39] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[41] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[42] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[43] Simon Ulbrich, Andreas Reschka, Jens Rieken, Susanne Ernst, Gerrit Bagschik, Frank Dierkes, Marcus Nolte, and Markus Maurer. Towards a functional system architecture for automated vehicles. *arXiv preprint arXiv:1703.08557*, 2017.

[44] Ruben Villegas, Jimei Yang, Seunghoon Hong, Xunyu Lin, and Honglak Lee. Decomposing motion and content for natural video sequence prediction. *arXiv preprint arXiv:1706.08033*, 2017.

[45] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[46] Wikipedia, the free encyclopedia. Utm zones. URL https://commons.wikimedia.org/wiki/File:Utm-zones.jpg. [Online; accessed August 31, 2019].

[47] Yi Xiao, Felipe Codevilla, Akhil Gurram, Onay Urfalioglu, and Antonio M López. Multimodal end-to-end autonomous driving. *arXiv preprint arXiv:1906.03199*, 2019.

[48] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810, 2015.

[49] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[50] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[51] Andi Zang, Zichen Li, David Doria, and Goce Trajcevski. Accurate vehicle self-localization in high definition map dataset. In *Proceedings of the 1st ACM SIGSPATIAL Workshop on High-Precision Maps and Intelligent Applications for Autonomous Vehicles*, pages 1–8, 2017.

[52] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[53] Julius Ziegler, Philipp Bender, Markus Schreiber, Henning Lategahn, Tobias Strauss, Christoph Stiller, Thao Dang, Uwe Franke, Nils Appenrodt, Christoph G Keller, et al. Making bertha drive—an autonomous journey on a historic route. *IEEE Intelligent transportation systems magazine*, 6(2):8–20, 2014.

A

Conversions of Coordinates

Problem Description:

Given two arbitrary x-y Cartesian coordinate systems, α and β, as illustrated in Figure A.1. Suppose that their origins and orientations are known. Derive a formula that can convert any arbitrary point P from α to β.

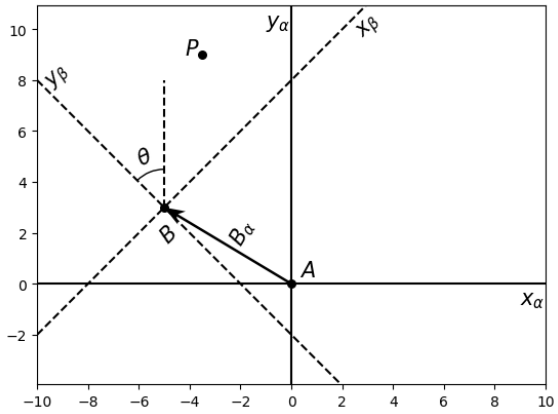
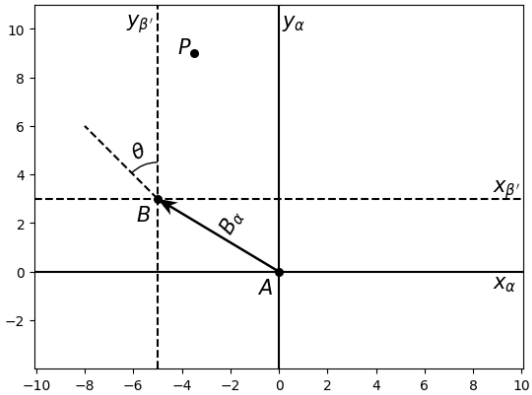
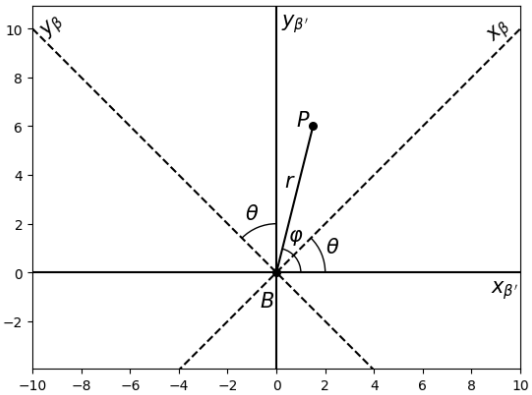


Figure A.1: Illustration of two arbitrary Cartesian coordinate systems, α and β.



(a) Translation from α to β'



(b) Rotation from β' to β

Figure A.2: Illustration of linear transformations used to convert from α to β.

Solution:

The main idea is to introduce an intermediate coordinate system, called β'. Let this coordinate system be centered at the same origin as β, but has the exact same orientation as α. The advantage of using such intermediate coordinate system is that the entire conversion from α to β can then be broken down into two smaller steps of linear transformations, as shown in Figure A.2.

In the first step, translation can be applied on the entire α coordinate system, in order to convert it to β'. This is possible because both α and β' have the same orientation. In the second step, the entire β' coordinate system can then be rotated about its own origin, thus completing the conversion to the targeted β coordinates.

List of some notations:

- Let A and B denote the two origins of α and β, respectively. Note that the origin B is shared between both β and β'.
- Let B_α be the vector that describes the location of the origin B with respect to the α coordinate system.
- Finally, let θ be the angular difference between the orientations of α and β.

Since B_α is known, translation from α to β' can be done by the following equation:

$$P_{\beta'} = P_\alpha - B_\alpha \tag{A.1}$$

Let r denote the distance between the origin B and the point P. Then according to Figure A.2b, both P_β and $P_{\beta'}$ can be written as polar coordinates as follows:

$$P_{\beta'} = \begin{bmatrix} r\cos\varphi \\ r\sin\varphi \end{bmatrix} \quad \text{and} \quad P_\beta = \begin{bmatrix} r\cos(\varphi-\theta) \\ r\sin(\varphi-\theta) \end{bmatrix} \tag{A.2}$$

Expanding P_β further would give:

$$\begin{aligned} P_\beta &= \begin{bmatrix} r\cos(\varphi-\theta) \\ r\sin(\varphi-\theta) \end{bmatrix} \\ &= \begin{bmatrix} r\cos\varphi\cos\theta + r\sin\varphi\sin\theta \\ r\sin\varphi\cos\theta + r\cos\varphi\sin\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} r\cos\varphi \\ r\sin\varphi \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} P_{\beta'} \end{aligned} \tag{A.3}$$

Equation A.1 and A.3 can be combined together to give the complete formula for transforming any point P, from the α to the β coordinate system:

$$\begin{gathered} P_\beta = \mathbf{R}P_{\beta'} = \mathbf{R}(P_\alpha - \mathbf{t}) \\ \text{where:} \quad \mathbf{t} = B_\alpha \quad \text{and} \quad \mathbf{R} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \end{gathered} \tag{A.4}$$