Norwegian University of
Science and Technology

End-to-end learning and sensor fusion with deep convolutional networks for steering an off-road unmanned ground vehicle

Johann Dirdal

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Kristin Ytterstad Pettersen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

Current autonomous driving policies based on deep learning are mostly learned from images of roads within rural and city areas. In this master thesis more unconventional settings are considered, such as off-road and forest terrain. Specifically, the goal is to train an off-road vehicle to make autonomous steering predictions based on input from a single camera and LiDAR scanner. To achieve this, we propose a fusion model comprising two convolutional networks and a fully-connected network. The convolutional nets are trained on images and LiDAR, respectively, whereas the fully-connected net is trained on combined features from each of these networks. Our experimental results show that fusing image and LiDAR information yields more accurate steering predictions on our dataset, than considering each data source separately. Results also show that training our networks on LiDAR and images individually produces similar root mean squared error (RMSE), and that better generalization is achieved by increasing the number of LiDAR features for training.

As a secondary task, we a propose a proof-of-concept verification model for steering trustability. This model utilizes segmented images from a separately trained segmentation network, and, using projective geometry, can determine if the path generated from a given steering angle is valid or not. Combining this model with our fusion network above, steering angle predictions from this network can be accepted or discarded online. Experiments on a small test set show promising results, but additional experimentation is needed to confirm validity.

Sammendrag

Autonome kjøreregler basert på dyp læring er for det meste lært fra bilder av vei i land- og byområder. I denne masteroppgaven legges det heller vekt på ukonvensjonelle kjøreomgivelser som offroad og skogsterreng. Målet er å trene et terrengkjøretøy til å ta autonome styrebeslutninger basert bilder og LiDAR. For å oppnå dette foreslår vi en fusjonsmodell som består av to konvolusjonsnett og et fullt-forbundet nettverk. Konvolusjonsnettverkene er opplært på henholdsvis bilder og LiDAR, mens det fullt forbundne nettet er trent på kombinerte features fra hver av konvolusjonsnettverkene. Våre eksperimentelle resultater viser at fusjonering av bilde og LiDAR informasjon gir mer nøyaktige styringsbeslutninger på vårt datasett enn å vurdere hver datakilde separat. Resultatene viser også at opplæring på LiDAR og bilder individuelt produserer tilsvarende effektivverdi (RMS), og at bedre generalisering er oppnådd ved å øke antallet LiDAR kjennemerker i treningen.

I tillegg foreslår vi en verifikasjonsmodell for vurdering av styrepålitelighet. Denne modellen benytter segmenterte bilder fra et separat opplært segmenteringsnettverk, sammen med projektiv geometri, til å avgjøre om kjørebanen fra en gitt styringsvinkel er gyldig eller ikke. Kombinerer vi denne modellen med fusjonsnettverket ovenfor, kan predikterte styrevinkler fra dette nettverket aksepteres eller kastes online. Eksperimenter på et lite prøveset viser gode resultater, men ytterligere forsøk er nødvendig for å bekrefte gyldigheten.

Preface

This master thesis concludes a five year journey at the Norwegian University of Science and Technology (NTNU). I am grateful to this institution and my supervisors, Professor Kristin Y. Pettersen (NTNU) and Senior Scientist Narada Warakagoda (FFI), for presenting me with the opportunity to pursue a thesis within the fascinating research field of deep learning. The relevant background for the work that shall be presented can be found in Sec. 1.3. The reader is assumed to have basic knowledge on machine learning and neural networks, however, an appendix presenting an overview and discussion of the most essential components for deep learning is included.

First, I would like to thank the Norwegian Defense and Research Establishment (FFI) for providing me with the necessary resources to complete this task, and, Marius Thoresen, for answering any questions regarding the dataset used. Second, I would like to thank my supervisor, Narada Warakagoda, for his invaluable input and guidance throughout this thesis and for answering all of my deep learning questions, no matter how stupid. Finally, I would like to thank Erlend Faxvaag Johnsen for his continuous collaboration and contributions on this project.

Kjeller, *Johann A. Dirdal*
11th of June 2018

Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

1 Introduction **1**

1.1 Motivation 1
1.2 Objective, scope and contributions 2
1.3 Background 3
1.4 Outline 4

2 Literature review **5**

2.1 Autonomous driving paradigms 5
2.2 End-to-end steering angle prediction 7
2.3 Improving learning with LiDAR 8

3 Theory **9**

3.1 Convolutional networks 9
3.1.1 Motivation 10
3.1.2 Convolution 12
3.1.3 Pooling 14

3.1.4 Regularization 15
3.1.5 The VGG net 16
3.2 Transfer learning 18
3.3 Pixel-wise segmentation 18
3.3.1 Fully convolutional networks (FCN) 19
3.4 Perspective projection geometry 21
3.4.1 Pinhole camera model 21
3.4.2 Depth from stereo 23
3.5 LiDAR 25
3.5.1 Generating LiDAR images 25

4 Methods **27**
4.1 Generating datasets 27
4.1.1 Data collection 28
4.1.2 Data processing 29
4.1.3 Data labeling 31
4.2 Steering network 33
4.2.1 Overview 33
4.2.2 Architecture 36
4.2.3 Training 38
4.2.4 Evaluation 39
4.3 Segmentation network 40
4.3.1 Overview 40
4.3.2 Architecture 42
4.3.3 Training 44
4.3.4 Evaluation 44
4.4 Path verification 45
4.4.1 Overview 45
4.4.2 Steering angle verification 46
4.4.3 Evaluation 49

5 Results and discussion **51**

5.1 Steering network 51
5.1.1 Results 51
5.1.2 Discussion 56
5.2 Segmentation network 59
5.2.1 Results 59
5.2.2 Discussion 60
5.3 Path verification 61
5.3.1 Results 61
5.3.2 Discussion 62

6 Conclusion **65**

A Neural network basics **67**
A.1 The structure of neural networks 68
A.2 Activation functions 70
A.2.1 Sigmoid unit 70
A.2.2 Softmax unit 71
A.2.3 Rectified linear units 72
A.3 Maximum likelihood estimation 73
A.4 Loss functions 74
A.4.1 Mean squared error (MSE) 74
A.4.2 Binary cross-entropy 75
A.4.3 Categorical cross-entropy 76
A.5 Training a neural network 77
A.5.1 Gradient based optimization 77
A.5.2 Back-propagation algorithm 79
A.5.3 Batch normalization 80

References **83**

List of Figures

2.1 Autonomous driving paradigms. Mediated perception approaches decompose the driving scene into several objects relevant for driving, and use these objects for making driving decisions. Behavior reflex approaches learn a direct mapping from driving scene to driving action. 6

3.1 Example digit from the MNIST dataset [34]. Left and right images are identical except that pixels on the right have been randomly permuted. Applying the same permutation on all instances in the MNIST dataset and training a fully-connected network on it, will yield identical classification performance as on the original dataset. 10

3.2 (*Left*) In a fully-connected layer, each output node is affected by every input node (shown in gray). (*Center*) Sparse connectivity. (*Right*) Parameter sharing. 11

3.3 Convolution operation. A 3×3 matrix is multiplied element-wise with each corresponding image-pixel to produce a single output value. . . 12

3.4 3D convolution process. 14

3.5 2×2 Max pooling layer with stride 2. The maximum value in 2×2 image regions is returned after each stride. The result is a subsampled image. 15

3.6 VGG-16 network 17

3.7 Transformation/convolutionaliztion of fully-connected layers into convolutional layers. The classification network outputs class scores, whereas the FCN outputs a heatmap. The image was taken from [36]. 19

3.8 FCN structure. Image taken from [36]. 20

3.9 Segmentation detail improves when fusing information from lower pooling layers. Leftmost image shows the result of upsampling from only the final prediction layer with stride 32. The other images shows results when combining outputs from lower layers with decreasing pixel stride. Image taken from [36]. 20

3.10 Transforming a 3D world coordinate to the image plane of a pinhole camera. First, an extrinsic transformation transforms the world coordinate $\mathbf{p}^w$ to the camera coordinate system. Then, an intrinsic transformation projects this coordinate onto the image plane. 21

3.11 Inferring depth information using two identical parallel cameras. A 3D point is mapped onto the image plane of each camera. Using simple geometry, an expression for Z can be found. 24

3.12 Data collection process. For each rotational step, 32 lasers emit light and the reflected beams are processed by the scanner. 26

3.13 LiDAR data structure produced after a *single* horizontal scan (N_s = # of horizontal steps). The first channel contains distances to any surrounding objects, while the second channel contains the intensities of the reflected beams. 26

4.1 Vechicle used by FFI to collect data. 28

4.2 Examples of the road environment. 28

4.3 Some predefined colormaps. 30

4.4 Processed camera and LiDAR image. 30

4.5 Data collection and processing pipeline. 31

4.6 Labeling instances in the segmentation dataset. 32

4.7 (Stage 1) Only camera images and steering angles are used for training. We extract features from the fifth pooling layer of a pre-trained VGG-16 model and use this as input to our CNN. The CNN outputs steering predictions which are compared to the ground truths, and appropriate weight adjustments are made to the Image network for a fixed number of iterations. 34
4.8 (Stage 2) Only LiDAR and steering angles are used for training. We extract features from the fifth pooling layer of a pre-trained VGG-16 model and use this as input to our CNN. The CNN outputs steering predictions which are compared to the ground truths, and appropriate weight adjustments are made to the LiDAR network for a fixed number of iterations. 34
4.9 (Stage 3) Late fusion. The trained models from Figs. 4.7 and 4.8 are combined, but with the final prediction layers removed such that each network outputs key features instead. The features are combined by late fusion, and used as input to a separate fusion network. Comparison with ground truth value and appropriate weight adjustment is the same as in Figs. 4.7 and 4.8. 35
4.10 After training the model in Fig. 4.9, it can be used to make online steering predictions, which can be used as references by the vehicle control system. 36
4.11 The segmentation network is trained by feeding it examples of the road environment along with the corresponding labels. Features from the 3rd, 4th and 5th pooling layers from a pre-trained VGG-16 network are used as input the FCN. Upsampling the result and yields a segmented image with same dimensions as the original input image. Weight updates based on the loss are performed a fixed number of iterations. 41
4.12 After the model in Fig. 4.11 is trained, it can be used to make segmentations online. 42
4.13 The upsampling process performed in the upsampling block of Figs. 4.11 and 4.12. Figure not to scale. 43

4.14 Trustability system. 46
4.15 Steering angle verification process. First, the segmented road image is transformed by a perspective transformation into a bird's eye perspective. Then, the trajectory of the vehicle is determined by drawing the given steering angle (shown in green) on the transformed image. We assume the vehicle is always in the road center, and therefore draw the angle from the bottom center. Finally, the steering angle is accepted if the trajectory lies within the road boundary for the distance threshold *d*. 47

5.1 Absolute error between predicted and ground truth steering angle on all 28404 data instances in the validation set, for the respective networks. The two highest errors from the Image and LiDAR networks are marked Top 1 and Top 2, respectively. 52
5.2 Column (a) and (b) show the bar plots and data instances corresponding to the Top 1 and Top 2 image errors from Fig. 5.1, respectively. Color mapped images are shown below the bar plots. LiDAR images consists of distance (top) and intensity (bottom) channels. 53
5.3 Column (a) and (b) show the bar plots and data instances corresponding to the Top 1 and Top 2 LiDAR errors from Fig. 5.1, respectively. Color mapped images are shown below the bar plots. LiDAR images consists of distance (top) and intensity (bottom) channels. 54
5.4 Learning curves produced by the respective Image, LiDAR and Fusion networks. The curves show the progression of training and validation set errors as the number of training epochs increases. 55
5.5 Some segmentation results from the validation set. 60
5.6 Verification process on a single instance from the test set. The drawn steering angle (shown in red) is outside the road boundary for the distance threshold, which is chosen as the image height, and therefore not accepted. 62

A.1 A simple feedforward neural network consisting of three layers. . . . 69
A.2 Logistic sigmoid. 71

A.3 Tanh. 71
A.4 Rectified linear unit. 72
A.5 Leaky rectified linear unit. 72
A.6 Batch normalization algorithm. Image taken from [25]. 80

List of Tables

1.1 Comparison of driving datasets (inspired by [56]). 2

4.1 CNN architecture with image input. All Conv. layers use ReLU activation. 37

4.2 CNN architecture with LiDAR input. All Conv. layers use ReLU activation. 37

4.3 Fusion network architecture. All fully-connected layers use ReLU activation. 38

4.4 The number of data instances used for training and validation. We use the same training and validation sets for all networks. 39

4.5 FCN architecture. 43

4.6 Upsampling architecture. 43

4.7 Number of images used for training and validation in the respective datasets. 44

4.8 Evaluation metrics. 45

4.9 Structure of the confusion matrix. Diagonal elements show correct predictions, while off-diagonal elements show false predictions (TP=True Positives, FN=False Negatives, FP=False Positives, TN=True Negatives). 49

5.1 Performance on the validation set by the Image, LiDAR and Fusion networks presented in Sec. 4.2. Values are based on the metrics discussed in Sec. 4.2.4. Zero and Mean represent blind predictions of the steering angle. The former always predicts a zero angle, while the latter always predicts the mean angle based on the ground truth values. (RMSE = Root Mean Squared Error, S.A = Steering Accuracy). 52
5.2 Top 1 and Top 2 Image and LiDAR errors from the respective bar plots in Figs. 5.2 and 5.3 shown numerically. 55
5.3 Results from other steering models in the literature (reported from [10]) compared to our model, which we call FusionNet (RMSE=Root Mean Squared Error). 58
5.4 Results on the validation set. Values are based on the metrics introduced in Sec. 4.3.4 (IoU = Intersection over Union, FW = Frequency Weighted). 59
5.5 Confusion matrix (see Sec. 4.4.3) showing the output results from our verification model (see Fig. 4.14) compared to the actual class labels, as defined in Sec. 4.4.3. We use a test set containing only *true* data instances (270). In this case, TP=225, TN=45, FP=0 and TN=0. 61

Chapter 1

Introduction

1.1 Motivation

Today, deep learning based autonomous driving systems are mainly trained on images of roads within rural and city areas. This master thesis is motivated by the fact that current deep learning approaches do not address autonomous driving in unconventional settings, such settings include off-road terrain which may, e.g., be of military application. However, training a deep learning system on images of off-road terrain is more challenging when compared to city and rural environments. The images often lack distinct road boundaries as well as having variable road texture, making it difficult for a deep learning system to learn meaningful features. This motivates the use of additional sensors to help detect road boundaries and avoid potential obstacles. Although combining information from different sensors to improve driving is not new, it is relatively unexplored in the context of deep learning when applied to direct steering prediction. There are three main groups of sensor systems used for autonomous driving today: camera, radar and LiDAR (short for Light Detection And Ranging). LiDAR is a widely adopted sensing technology that can produce accurate

3D-point clouds of the surrounding environment. In this work, we use readings from camera and LiDAR sensor systems, along with measured wheel angles, as input data for learning.

Safety is paramount when it comes to evaluating the success of an autonomous driving system. For trustability reasons, it is important that a deep learning system learns to detect features that make sense from a human standpoint. Evidence has shown that deep learning approaches do learn to recognize relevant driving-related objects [2, 4, 10]. To show that this is indeed the case for our work, we design and implement a separate path verification system to decide if the given steering predictions are appropriate or not.

1.2 Objective, scope and contributions

The main objective of this research is end-to-end steering angle prediction for an off-road vehicle using convolutional networks. By "end-to-end" we mean a direct mapping from input to actuation is learned. It is worth stressing, however, that a complete autonomous driving model is not intended; the task in question merely involves steering. The technical contributions that this work offers are three-fold and are as follows.

First, the driving setting introduced by our dataset differs significantly from other conventional driving datasets (see Table 1.1). Specifically, the environment is made up

Table 1.1: Comparison of driving datasets (inspired by [56]).

Dataset	**Setting**	**Type**	**Diversity**
KITTI	city, rural area, highway	real	one city, one weather condition, daytime
Cityscape	city	real	German cities, multiple weather conditions, daytime
Comma.ai	mostly highway	real	highway, N.A., daytime and night
Oxford	city	real	one city (Oxford), multiple weather conditions, daytime
Princeton Torcs	highway	synthesis	N.A.
GTA	city, highway	synthesis	N.A.
BDDV	city, rural area, highway	real	multiple cities, multiple weather conditions, daytime and night
FFI (ours)	**dirt-roads, forest**	real	multiple weather conditions, dawn, daytime and dusk

of isolated dirt-roads and other natural terrain (see Fig. 4.2), in many cases without distinct road boundaries. We argue that learning from this dataset poses a greater challenge than learning from other conventional driving datasets, which usually have well-defined road boundaries and road texture.

Second, we propose an end-to-end learning approach similar to [1], but instead of training solely on images and steering angles, we train our convolutional network on *fused* sensor input comprising image and *LiDAR* information together with time-synchronized steering commands. Since many industrial autonomous driving systems include LiDAR as an essential sensory component, we believe that combining images of the road environment along with corresponding LiDAR readings can improve learning. To our knowledge, this approach has only been applied once to end-to-end steering prediction, but in a different driving setting [8].

Third, we introduce a separate path verification model with the purpose of establishing end-to-end steering trustability. To do this, we implement an individual segmentation network taking images of the road environment as input, and producing images in which each image-pixel is categorized as either *road* or *not-road* as output. Then, a separate decision model based on projective geometry uses these segmented images and decides if the given steering angle is trustable or not.

1.3 Background

The template code used for training and evaluating our convolutional networks were taken from the TensorFlow-Slim image classification model library[1]. The library contains a lightweight high-level API called TF-Slim, which is used for defining, training and evaluating complex models. This API makes data processing, in addition to training- and testing models, significantly easier. The code, however, is actually intended for image classification, whereas the task in question involves regression. For this reason, appropriate adjustments had to be made to make the code compatible

[1] https://github.com/tensorflow/models/tree/master/research/slim

with regression.

The network architecture used for segmentation was taken from the following github page[2], and adapted to the segmentation task in question. In particular, the number of segmentation classes, epochs and batch sizes, were altered.

The dataset used for learning contains data that has been collected in regular intervals from 2015 to 2017 by an Unmanned Ground Vehicle (UGV) in association with the Norwegian Defense Research Establishment (FFI). It is worth pointing out that the data was not originally intended for deep learning applications when it was collected, and therefore requires some processing. The data collection and processing steps are explained in detail in Sec. 4.1.

1.4 Outline

The organization of this master thesis is as follows. Chapter 2 gives a literature review of the current deep learning based approaches for autonomous driving, providing context for the task in question. Chapter 3 discusses the underlying theory for the tasks performed in Chapter 4. Chapter 4 introduces the general methodology. First, the method for generating the appropriate datasets is described. Then, architecture design, training and evaluation of our proposed network models is discussed. Finally, a model for path verification is presented along with a method for evaluating performance. In Chapter 5, the results on the training and test sets are presented, discussed, and recommendations for future work is given. In Chapter 6, conclusive remarks and further comments on future work are made.

[2] https://github.com/maxritter/SDC-Semantic-Segmentation

Chapter 2

Literature review

The main objective of this chapter is to review the literature on current deep learning based approaches for autonomous steering. To achieve this goal, we start by discussing the two main paradigms for autonomous driving today. Then, we narrow the scope by considering end-to-end steering prediction and present the relevant literature on the topic. Finally, the current use of LiDAR data for driving applications in the context of deep learning is discussed.

2.1 Autonomous driving paradigms

To date, vision-based autonomous driving systems are mainly dominated by two paradigms: *mediated perception approaches* and *behavior reflex approaches* (see Fig. 2.1).

Mediated perception approaches [52] decompose the driving scene into several driving-related objects and use these objects to infer driving decisions. Such objects include lanes, traffic signs, traffic lights, cars, pedestrians, etc., and are usually detected

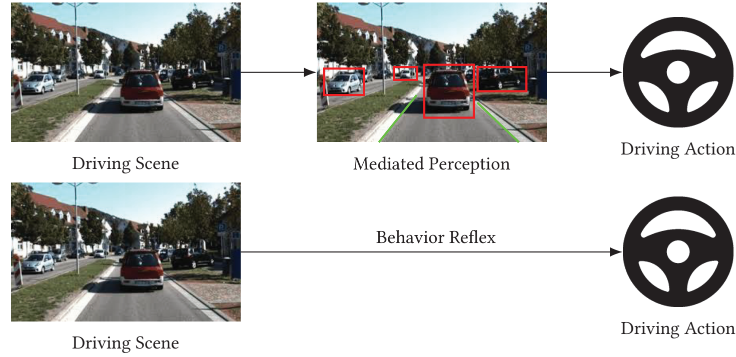


Figure 2.1: Autonomous driving paradigms. Mediated perception approaches decompose the driving scene into several objects relevant for driving, and use these objects for making driving decisions. Behavior reflex approaches learn a direct mapping from driving scene to driving action.

and recognized separately using various perception models [60, 5, 38, 17]. However, with the recent advances in deep learning research and computer hardware, many driving-related objects can be detected and recognized simultaneously [7]. Today, most industrial autonomous driving systems, including the state-of-the-art, use a mediated perception approach [10, 4]. Despite its success, the level of total scene understanding imposed may in some cases add unnecessary complexity to the task [4]. Indeed, only a small portion of detected objects is needed in many situations to make adequate driving decisions. Chen et. al [4] address this issue and show that convolutional networks can be used to directly estimate the most important affordance indicators for driving. Such indicators include heading angle, distance to lane markings, and distances to cars in current and adjacent lanes. These indicators are then fed into pre-defined steering and velocity models to compute the appropriate driving commands.

In contrast, behavior reflex approaches [1, 56, 10] learn a direct mapping from driving scene to driving action. The model, usually a neural network, is trained on data collected by a human driver and learns to emulate human maneuvers. This eliminates the need for hand-coding driving rules and instead creates a system that learns from

observation alone. Despite the simplicity and elegance of this idea, it does present some important challenges [4, 56]. First, drivers will make different driving decisions when meeting similar or same situations (e.g., deciding whether to follow or pass a car) [4]. Presenting a neural network with similar input, but with differing target actions, will cause confusion and can stall learning. Second, the learned model is vehicle-specific. There are no guarantees that the trained model will work correspondingly well across different automobiles since the actuation systems they employ are often different. Xu et. al [56] address this issue and show that a convolutional network can produce a direct mapping from image input to a discrete set of driving actions (such as go straight, turn-left, turn-right, etc.), independent of the vehicle's actuation system.

2.2 End-to-end steering angle prediction

In the late 1980s, Dean A. Pomerleau developed the pioneering work of ALVINN (Autonomous Land Vehicle In a Neural Network) [42, 43], which is considered to be the earliest attempt to map road pixels to steering angles using neural networks [1, 56, 10]. This approach falls under the behavior reflex category, and compared to more modern deep networks, the network used was mostly made up of shallow, fully-connected, layers. Despite its simple structure, it proved successful in a number of basic driving situations. Inspired by this success, NVIDIA recently demonstrated that convolution networks, combined with general purpose GPUs, are capable of performing more complex steering tasks [1]. In particular, they show that it is indeed possible for an end-to-end system to learn to drive in traffic on local roads with and without lane-markings. A limitation of this approach, however, addressed by [56, 10], is that steering predictions are made independently on each video frame, thus contradicting driving as a stateful process. Chi et. al [10] proposed using a convolutional-LSTM (Long Short Term Memory [23]) architecture to improve steering predictions by taking into account historical vehicle states. Their results do in fact show that leveraging earlier driving states leads to improved steering.

2.3 Improving learning with LiDAR

The use of LiDAR scanners in autonomous vehicles is attractive for two reasons. First, it can produce direct distance measurements of nearby objects, which in turn can be used by various controllers and planners for driving [55]. Second, it is robust under different lighting conditions; a problem most cameras suffer from. These properties are appealing also for deep learning systems and several approaches have attempted to benefit from this technology. Wu et. al [55] use a transformed 3D LiDAR point cloud as input to a convolutional network in order to segment interesting road-objects such as cars, pedestrians, cyclists, etc. Specifically, the task aims to isolate objects and predict their respective categories using LiDAR input only. Their results show high accuracy in addition to fast and stable runtime. Other approaches using convolutional nets together with LiDAR for detection/segmentation include [35, 3, 59]. Despite the increased interest for LiDAR, only one approach, to our knowledge, has attempted fusing it with other sensor data to improve end-to-end steering. Chen and Wang et. al [8] propose their own LiDAR-Video driving dataset and show that extra depth information does indeed have a positive impact on driving. Their approach, however, differs from ours in three areas. First, the dataset proposed comprises conventional driving settings, similar to those in Table 1.1, whereas ours is unique in that we consider off-road terrain. Second, the experimental framework used for learning is different. Specifically, they adopt Resnet [22], Inception-v4 [51] and NVIDIA [1] architectures for extracting features, whereas we use VGG [49]. Third, their strategy for preprocessing LiDAR readings differs from ours. In particular, they adopt two techniques known as Point Clouds Mapping [57] and PointNet [44] to create powerful depth representations, whereas we simply reorganize the point cloud data into an image with two feature channels (more details in Sec. 3.5).

Chapter 3

Theory

In this chapter we present the underlying theory relevant to the steering, segmentation and path verification models in Chapter 4. Specifically, we consider *convolutional networks*, and the more specific *fully-convolutional networks* for segmentation, *transfer learning*, *projective geometry*, and finally, a simple strategy for preprocessing LiDAR data. The reader is advised to consider reading Appendix A before studying this chapter if unfamiliar with neural networks and deep learning.

3.1 Convolutional networks

In 2012 convolutional networks (also known as convolutional neural networks, or CNNs) [33] won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)[1] by a wide margin, bringing down the (at the time) state-of-the-art error from 26.1% to 15.1% [30], and in many ways led to the rise of deep learning. Since then, convolutional networks have shown great success in a number of different image classification tasks [49, 50]. As we shall see, CNNs play a crucial role in the development of the steering-

[1] ILSVRC is held each year and is the largest contest in object recognition.

and segmentation networks presented in Chapter 4. It is therefore worthwhile to study the underlying theory behind convolutional nets in order to get a grasp of the operations involved.

3.1.1 Motivation

Although convolutional nets have experienced a great deal of success in tasks such as *voice recognition* and *natural language processing*, they are mainly popularized for their performance on complex visual recognition tasks. These problems usually require a vast amount of image data in order to give adequate results. Given a large enough training set, a fully-connected network could in principle yield a good solution to tasks of this nature. However, such networks present two important issues when applied to visual perception: (1) they have large memory requirements, and (2) they ignore any structure in the input features, as will now be discussed.

Images are high-dimensional inputs, where each pixel is treated as a feature. This means that a 100×100 input image would be treated as a 10 000 feature vector in the input layer of a fully-connected network. Recall that in a traditional neural network each output node is connected to every input node with a separate parameter describing the interaction. If the first hidden layer comprises, say, 100 nodes, then this would require 1 000 000 parameters in just this layer.

Figure 3.1: Example digit from the MNIST dataset [34]. Left and right images are identical except that pixels on the right have been randomly permuted. Applying the same permutation on all instances in the MNIST dataset and training a fully-connected network on it, will yield identical classification performance as on the original dataset.

Fully-connected networks ignore a key property of images, namely that nearby pixels are more strongly correlated than more distant pixels. Consequently, networks of this type do not take into account the local spatial layout of features in the data. Once a fully-connected net has learned to recognize a pattern in one location, it can only recognize that pattern in that particular location (see Fig. 3.1).

The shortcomings above are addressed and incorporated into convolutional networks through three mechanisms: (1) *sparse interactions*, (2) *parameter sharing*, and (3) *equivariant representations*. Mechanisms (1) and (2) are illustrated graphically in Fig. 3.2. The figure shows the same number of input and output units in all three examples but with differing configurations in terms of connectivity and the number of weights (not shown in left and center). With sparse connectivity each output node is only affected by the nodes in a local region of the input. The units influencing the output z_3 are known as the *receptive field* of z_3 (shown in gray). With parameter sharing, the same parameters are used at all input locations (shown as x_1, x_2, etc.), which is illustrated by using the same color for each parameter. Fewer parameters reduces the memory requirements imposed by the network. Fewer connections also

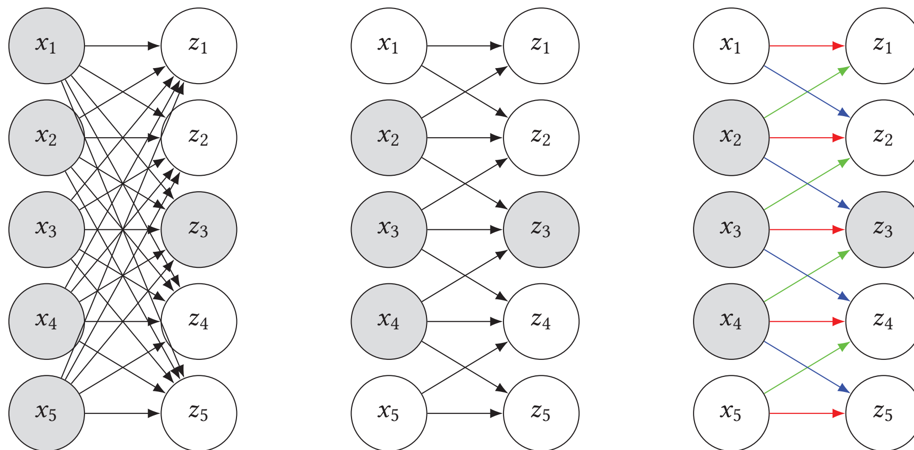


Figure 3.2: (*Left*) In a fully-connected layer, each output node is affected by every input node (shown in gray). (*Center*) Sparse connectivity. (*Right*) Parameter sharing.

reduce the number of operations needed to compute the output. Most importantly, sparse connectivity and parameter sharing enables the network to recognize the same patterns in different subregions of the image. This architecture allows the network to concentrate on low-level features in the initial hidden layers, then assemble them into higher-level features in the subsequent layers, which can then be used to detect complex patterns in the image.

Convolution (discussed in the next section) is said to be equivariant to translation. This means that if some input is translated, that is, shifted by some amount of pixels, the convolution output will be translated in the same way. Mathematically, a function f is equivariant to some transformation g if $f(g(x)) = g(f(x))$. This property is important as it means that convolution can detect the same features (e.g., edges) at different locations. It is worth stressing that convolution is not naturally equivariant to other transformations such as scaling or rotation.

3.1.2 Convolution

Fig. 3.3 shows a 5×5 input image being convolved with a 3×3 kernel matrix (shown in blue) to produce a 5×5 *feature map*. Studying the figure closely, we can see mechanisms (1) and (2) in play. All output units have a 3×3 local receptive field defined by the size

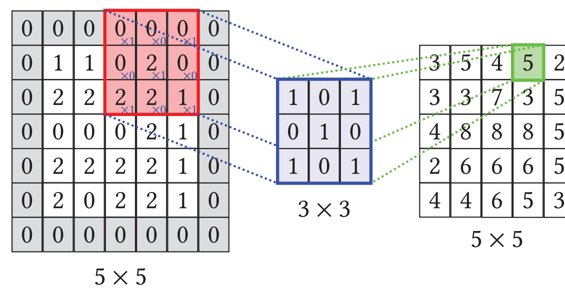


Figure 3.3: Convolution operation. A 3×3 matrix is multiplied element-wise with each corresponding image-pixel to produce a single output value.

of the kernel matrix, making interactions with the input units sparse. In addition, the parameters associated with each output node are the same; the same kernel matrix is applied to all subregions of the image. Each element in the kernel is multiplied with the corresponding element in the local input region, and the products are summed to produce the output. The kernel is then shifted horizontally to compute the next output and continues in this manner until it reaches the edge. At this point, the kernel returns to start but is shifted vertically downwards to compute the outputs on the next row, and so on. The amount of pixels the kernel moves after each computation is called the *stride*, and is not necessarily the same in both horizontal and vertical directions. In the above example, the stride was set equal to 1 in both directions, which is common whenever we want the feature map to have same dimensions as the input. In order to process the edge pixels of the input, it is customary to coat the image with extra rows and columns. This process is known as *padding*, and various strategies exist for deciding the row- and column values. In Fig. 3.3 we have employed what is known as *zero-padding* (shown in gray), which, as the name suggests, involves coating the border with zeroes.

The discussion so far has, for simplicity, considered the input layer as 2D images with one color channel and convolutional layers as a single feature map. In practice, however, it is common to train a convolutional net on images comprising three color channels (red, green and blue) in addition to using multiple kernels for each input layer. Altering the image dimensions from 2D to 3D implies that the depth in the local receptive field of each output unit be extended accordingly. This means that the kernel matrix needs to be extended to 3 dimensions as well. Fig. 3.4 gives a graphical illustration of the 3D convolution process. Each output unit, in each feature map of the first convolutional layer, is computed by processing 3D volumes of pixel values within the local receptive field of the output. More specifically, the parameters in the first plane of the kernel matrix are multiplied with the corresponding pixel values in the first color channel, the parameters in the second plane with the second color channel, and so on, before summing the results to produce the output. It is common to simultaneously apply multiple kernels to the input in order to detect different features. This results in stacks of different feature maps (recall that each feature map

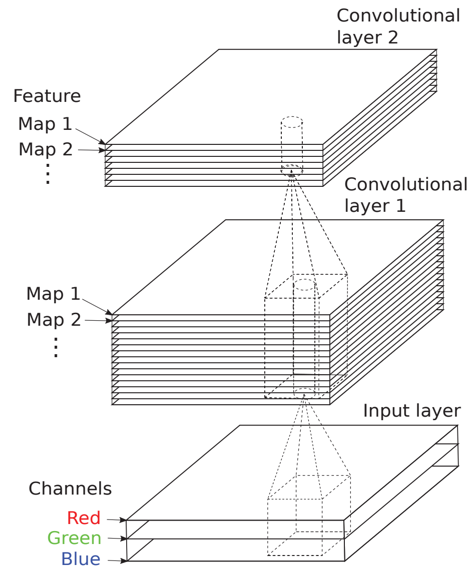


Figure 3.4: 3D convolution process.

is associated with a single kernel). The outputs in the next convolutional layer can then be computed by processing 3D volumes of pixel values in the feature maps within the local receptive field of each output unit. Combining features in this way enables detection of more complex image patterns.

3.1.3 Pooling

In order to reduce computational load, memory usage and the number of parameters in a network it is customary to subsample (i.e., shrink) the input using a pooling operation. We usually distinguish between three types of pooling functions; *average pooling*, *L2-norm pooling* and *max pooling*. Max pooling [62], which is the most common type of pooling layer, is shown in Fig. 3.5. Just like convolution, each output unit is connected to local regions of the input, in this case, by a 2×2 local receptive field (shown in different colors). However, in contrast to convolution, pooling does not have any

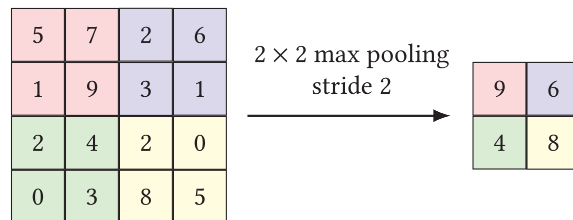


Figure 3.5: 2×2 Max pooling layer with stride 2. The maximum value in 2×2 image regions is returned after each stride. The result is a subsampled image.

weights associated with its nodes; it uses an aggregation function to aggregate the inputs in each local region. With max pooling, the maximum value in each region is returned. To make the output smaller, it is necessary to use a stride greater than 1 (given that the input has been padded accordingly).

In addition to the already mentioned attributes, pooling possesses a property known as *translation invariance*. For small translations of the input, the output response from pooling is (approximately) the same as before the image was shifted. Invariance to local translation can be useful whenever we need to know whether a feature is present rather than exactly where it is (e.g., detecting the presence of eyes for facial recognition).

Lastly, we mention that pooling can be used to handle tasks with variable-sized inputs. In the case of classification, the classification layer, which is fully-connected, requires fix-sized inputs. By varying the stride between pooling regions in the final pooling layer, the classification layer will in general receive the same number of inputs, independent of input dimensions.

3.1.4 Regularization

To prevent our convolutional networks in Sec. 4.2 from overfitting the training data, we consider four regularization techniques—*weight-decay*, *Dropout*, *Batch Normalization*,

and *early stopping*. With weight-decay a penalty is added to the loss function, thereby restricting the size of how large the network parameters are allowed to be. In this way, the network is prevented from assigning arbitrarily large parameter values in order to fit the training data. With Dropout, a set of arbitrary nodes are "dropped" at each training stage. This creates different networks at each stage, and prevents specific nodes from overfitting the training data. Batch Normalization integrates normalization as part of the model architecture and has proven to have a regularizing effect [25], sometimes eliminating the need for Dropout. Finally, early stopping involves monitoring the validation error as the number of training epochs go by, and stop training as soon as the error reaches a minimum.

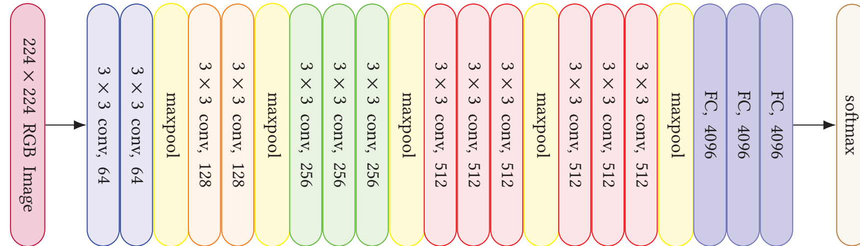
We use weight-decay and Dropout when training the VGG-16 network (discussed in Sec. 3.1.5). In addition, we use Batch Normalization when training the Image and LiDAR networks introduced in Sec. 4.2. Early stopping is not considered since the learning curves presented in Sec. 5.1 indicate that overfitting is not an issue in our trained models.

3.1.5 The VGG net

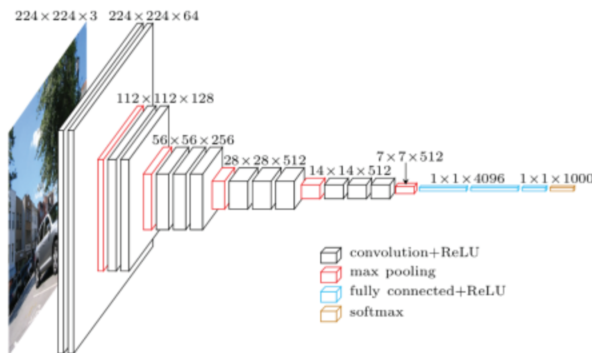
In 2015 the Visual Geometry Group[2] (VGG) published a paper [49] investigating the effect of depth in a convolutional network on the ILSVRC dataset. The dataset, commonly referred to as ImageNet [13], consists of millions of images used for object category classification and has a total of 1000 different categories. The VGG team found that by steadily increasing depth (i.e., the number of convolution layers in the network), top-1 and top-5 classification errors continued to decrease. The VGGNet, as it is called, exists with various depth configurations (11, 13, 16 and 19 weight layers), with the largest having best performance on ImageNet. Convolutional nets of this size are feasible in terms of computation and memory because of the small kernels involved (all kernels have a receptive field of 3×3 or smaller).

Fig. 3.6a gives graphical illustration of the network configuration for the VGG-16

[2] http://www.robots.ox.ac.uk/~vgg/



(a) VGG-16 layer description and configuration.



(b) VGG-16 layer configuration in 3D with input/output dimensions. Image taken from [31].

Figure 3.6: VGG-16 network

net. The network comprises 16 weight layers; 13 convolutional- and 3 Fully-Connected (FC) layers (recall that pooling does not have any parameters). As we can see, the number of feature maps in each convolution layer is increased by a power of 2, from 64 to 512, after each pooling operation. Max pooling is performed over a 2×2 pixel window, with stride 2. Fig. 3.6b shows the same layer pipeline as in Fig. 3.6a but in 3D, in addition to showing the input/output dimensions of each layer. We see that the spatial dimensions of the input image are initially fixed and gradually subsampled to produce classification scores.

3.2 Transfer learning

Training a convolutional network from scratch is uncommon due to the lack of sufficiently sized datasets [8, 10, 55]. In general, training a complex model on a relatively small dataset is more likely to produce an overfitted model. To circumvent this, we use already pre-trained VGG-16 layers as part of the steering and segmentation networks presented in Sec. 4.2 and 4.3, and fine-tune the weights of these layers when training on our data. This method is known as *transfer learning* and, as the name suggests, involves transferring knowledge from a previously learned task to a new one. Different strategies for using transfer learning exists and depends on the dataset size in addition to how similar the data is to the original in the transferred layers. A general guideline is to use features from lower layers if the data is very different from the original dataset, and use features from higher layers if the data is similar [58]. The reason is that features in the earlier stages tend to be more generic than later on. We experiment with different pre-trained VGG-16 layers for the fusion network in Sec. 4.2, and find (somewhat surprisingly) that features from higher layers produce the best results.

3.3 Pixel-wise segmentation

The path verification model introduced in Sec. 4.4 uses segmented images of the road to perform steering angle verification. These images are produced from a separate segmentation network, as we shall see in Sec. 4.3. In this section, some of the most important components of this network are studied.

Pixel-wise segmentation is the task of assigning each pixel in an image to a specific category. In the application of self-driving cars, images of the road environment can contain a multitude of categories—traffic signs, roads, pedestrians and sidewalks to name a few. Prior approaches to pixel-wise segmentation include Texton Forests [47], Random Forest based classifiers [48, 46] and Convolutional Networks [15, 41, 21, 20]. In this section we shall introduce a different paradigm known as *Fully Convolutional*

Networks (FCN) [36] which achieves state-of-the-art performance and exceeds the aforementioned approaches on computational speed and efficiency.

3.3.1 Fully convolutional networks (FCN)

Although convolutional networks have experienced a great deal of success in image classification, they have not shown the same degree of success in segmentation, which is mainly due to their inherent structure. A convolution network generally has three operations: *convolution*, *activation* and *pooling*. The convolution and activation operations are important for detecting useful features, whereas pooling, in addition to subsampling, introduces invariance to local translation of the input. However, for tasks such as pixel-wise prediction, which relies on the preservation of the location of features, invariance to translation poses some difficulty. Previous efforts using CNNs also rely on a technique known as *patchwise training* [11, 15, 41], which involves training on sub-images or patches instead of whole-images. Fully convolutional networks address the structural shortcomings of CNNs by re-architecting and fine-tuning existing classification networks to pixel-wise prediction. Re-architection involves discarding the final classification layer of the classifier network and "convolutionalizing" the fully-connected layers. This is achieved by simply viewing fully-connected layers as

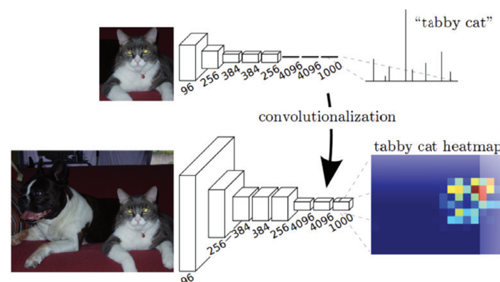


Figure 3.7: Transformation/convolutionaliztion of fully-connected layers into convolutional layers. The classification network outputs class scores, whereas the FCN outputs a heatmap. The image was taken from [36].

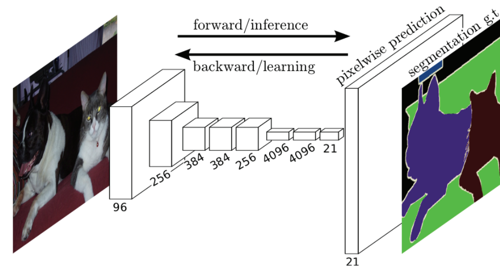


Figure 3.8: FCN structure. Image taken from [36].

convolutions with kernels covering their entire input region. Fig. 3.7 gives a graphical depiction of this transformation process. As the figure shows, the dimensions of the output from the FCN has been reduced due to subsampling. To transform the subsampled output to a segmented image with the same dimensions as the original image, we can use interpolation, a process known as *upsampling*. However, upsampling is typically performed in-network meaning that the interpolation-parameters are not fixed but learned. Fig. 3.8 shows the overall structure of the fully convolution network. A problem with this structure is that upsampling at the final prediction layer limits the scale of detail in the upsampled output. Ref. [36] defines a skip architecture which fuses information from lower layers to improve segmentation detail. More specifically, the skip architecture uses the output from earlier pooling layers thereby preserving local

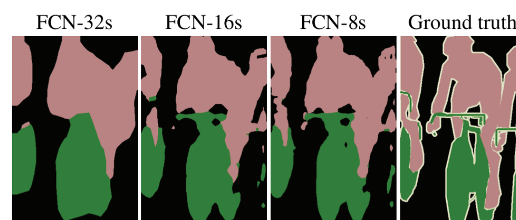


Figure 3.9: Segmentation detail improves when fusing information from lower pooling layers. Leftmost image shows the result of upsampling from only the final prediction layer with stride 32. The other images shows results when combining outputs from lower layers with decreasing pixel stride. Image taken from [36].

features and allows for upsampling with smaller strides. Fig. 3.9 shows segmentation results when using information from lower layers with decreasing stride.

3.4 Perspective projection geometry

In Sec. 4.4, we design and implement a path verification model to assign trust to the predicted steering angles generated from the trained steering network. This section presents some of the underlying theory behind this model, and will serve as aid when discussing limitations later on.

3.4.1 Pinhole camera model

The pinhole camera model is a mathematical model describing the process in which rays of light are mapped onto the image plane. More specifically, the model describes the transformations needed to take a 3D point from some object in the world coordinate

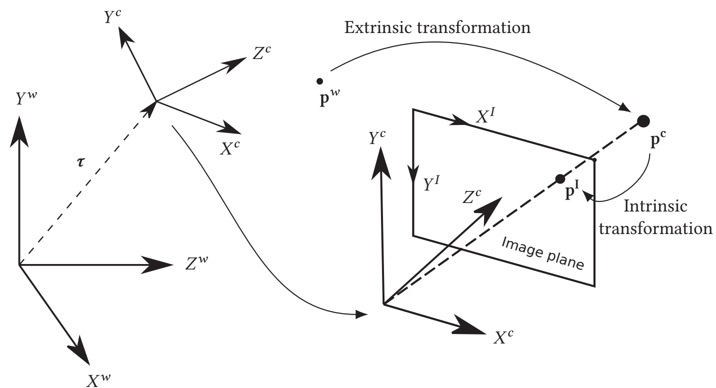


Figure 3.10: Transforming a 3D world coordinate to the image plane of a pinhole camera. First, an extrinsic transformation transforms the world coordinate $\mathbf{p}^w$ to the camera coordinate system. Then, an intrinsic transformation projects this coordinate onto the image plane.

system and project it onto the 2D image plane of an *ideal* pinhole camera. The overall process can be explained in two steps. First, the 3D world point is transformed from the world frame to the camera frame. Then, the point is transformed from the camera frame to the image plane through a second transformation. Fig. 3.10 illustrates both processes. Let (X^w, Y^w, Z^w) and (X^c, Y^c, Z^c) be the coordinate systems of the world and camera frames, respectively, and let $\mathbf{p}^w$ be the coordinate of some object in the world frame. The transformation from world to camera can be mathematically described by:

$$\mathbf{p}^c = \mathbf{R}_w^c \mathbf{p}^w + \boldsymbol{\tau}, \tag{3.1}$$

where $\mathbf{R}_w^c$ is a 3×3 rotation matrix between the world and camera frame and τ is a 3×1 translation vector. The set $\{\mathbf{R}_w^c, \tau\}$ is called the *extrinsic parameters* and describe the position and orientation of the camera in the world. Before defining the transformation from camera frame to image plane, some terminology is necessary. The origin of the (X^c, Y^c, Z^c) coordinate system is called the *optical center*, and defines pinhole of the camera. Furthermore, the point at which the Z^c axis, or *optical axis*, strikes the image plane is known as the *principal point*. The distance between the optical center and principal point is called the *focal length*. With these quantities defined, the camera coordinate $\mathbf{p}^c = [x, y, z]^T$ can be projected onto the image plane by the following transformation:

$$x_p = f\frac{x}{z}, \; y_p = f\frac{y}{z}$$

where f denotes the focal length. We have for simplicity assumed that the focal length parameters are the same in the x- and y-directions. Most imaging systems consider $(0, 0)$ to be at the top-left corner of the image plane, rather than the optical center. To cope with this, we add offset parameters δ_x and δ_y so that the transformation becomes

$$\begin{aligned} x_p &= f\frac{x}{z} + \delta_x, \\ y_p &= f\frac{y}{z} + \delta_y. \end{aligned} \tag{3.2}$$

The set $\{f, \delta_x, \delta_y\}$ is known as the *intrinsic* parameters, and is camera specific. Notice that the transformation in (3.2) is nonlinear due to the division by z. Multipying both sides of (3.2) with z and expressing the system in homogeneous coordinates, yields

$$z \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f & 0 & \delta_x & 0 \\ 0 & f & \delta_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{Intrinsic matrix}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Letting $\mathbf{P}^I$ denote the left-hand side, $\mathbf{K}$ the intrinsic matrix and $\mathbf{P}^c$ the homogeneous camera coordinate, we get the linear system:

$$\mathbf{P}^I = \mathbf{K}\mathbf{P}^c. \tag{3.3}$$

Re-writing the extrinsic transformation in (3.1) in a similar manner, gives

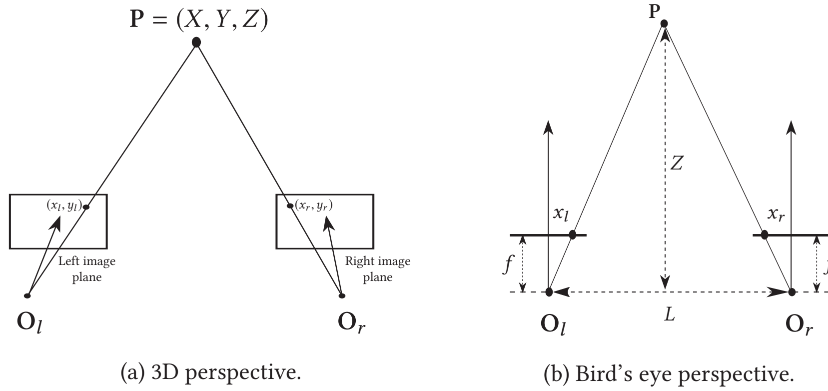
$$\mathbf{P}^c = [\mathbf{R}_w^c \ \boldsymbol{\tau}]\ \mathbf{P}^w, \tag{3.4}$$

where $\mathbf{P}^c = [\mathbf{p}^c, 1]^T$ and $\mathbf{P}^w = [\mathbf{p}^w, 1]^T$. Finally, substituting (3.4) into (3.3), we get the general transformation from 3D world to 2D image plane:

$$\mathbf{P}^I = \mathbf{K}\,[\mathbf{R}_w^c \ \boldsymbol{\tau}]\ \mathbf{P}^w. \tag{3.5}$$

3.4.2 Depth from stereo

The real world depth of a 2D image point can be estimated from a stereo-pair image. In other words, given two images depicting the same scene, the depth of a pair of corresponding points in each image can be estimated given certain camera parameters. In the subsequent derivations we assume (1) identically calibrated cameras, meaning the intrinsic parameters are the same, and (2) the cameras are placed in parallel. A 3D



(a) 3D perspective.

(b) Bird's eye perspective.

Figure 3.11: Inferring depth information using two identical parallel cameras. A 3D point is mapped onto the image plane of each camera. Using simple geometry, an expression for Z can be found.

view of the setup is illustrated in Fig. 3.11a. In particular, we have two pinhole cameras (left and right) indicated by their respective optical centers, $\mathbf{O}_l$ and $\mathbf{O}_r$. The optical axis from each center strikes the corresponding image plane at the origin. A world coordinate denoted $\mathbf{P}$ is projected onto the image plane of each camera resulting in image-plane coordinates (x_l, y_l) and (x_r, y_r), defined relative to the origin of the image plane. Given the assumptions outlined above, it can be shown that $y_l = y_r$. Fig. 3.11b shows a bird's eye perspective of the setup in Fig. 3.11a. We introduce parameters f, denoting the focal length of each camera, and L, representing the baseline distance between the cameras. Using similar triangles, we have that

$$\frac{Z}{L} = \frac{Z - f}{L - x_l + x_r}.$$

After some rearranging, the depth Z of a 2D image point can be expressed as

$$Z = f\frac{L}{x_l - x_r}, \tag{3.6}$$

where the difference, $(x_l - x_r)$, is known as the disparity between the corresponding image points.

3.5 LiDAR

LiDAR (also known as Lidar, LADAR or LIDAR) is a remote sensing technology that uses light to infer the relative position of surrounding objects. Typically, rapid pulses of light are emitted from a laser instrument and a sensor measures the time taken for the reflected light to return. In ground-based LiDAR systems, the LiDAR instrument is often mounted on a tripod in which it can rotate 360° around. By emitting pulses of light and recording the corresponding time delays while rotating, the instrument can build a complex map (known as a point-cloud) of the surrounding environment. Most companies active in the self-driving car community, such as Uber, Waymo, Baidu, etc., include LiDAR as an essential sensory component in their vehicles. In this section, we deviate from the conventional point-cloud based approaches, and instead look at how LiDAR data can be exploited to operate directly with deep learning systems.

3.5.1 Generating LiDAR images

The test vehicle used for data collection in this thesis is equipped with a Velodyne HDL-32E LiDAR sensor[3]. The instrument contains 32 lasers, has a measurement range of 1-70 m, and has a vertical and horizontal field of view (FOV) of 40° and 360°, respectively. The data collection process is illustrated in Fig. 3.12.

The scanner rotates horizontally and at each horizontal step, 32 vertical lasers emit light and the reflected rays with corresponding time delays are processed by the instrument. The processed data from each laser contains two important pieces of information, namely the distance to any obstacle in the surrounding environment, and the intensity of the reflected beam. This information can be conveniently expressed in

[3]For more information see: http://velodynelidar.com/hdl-32e.html

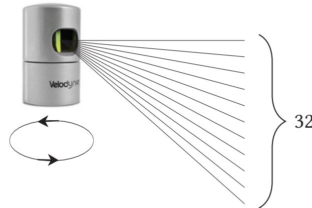


Figure 3.12: Data collection process. For each rotational step, 32 lasers emit light and the reflected beams are processed by the scanner.

a $32 \times N_s \times 2$ matrix (see Fig. 3.13), where N_s represents the number of horizontal steps needed to complete a single turn, and the first and second channel represents respective distance and intensity measurements from each laser after one cycle. Since we are more interested in the events taking place in front rather than behind the vehicle, only the data associated with a $120°$ arc of the horizontal FOV is considered.

By reinterpreting LiDAR readings as multi-dimensional arrays with various distance and intensity features, we can use this data to train a deep neural network.

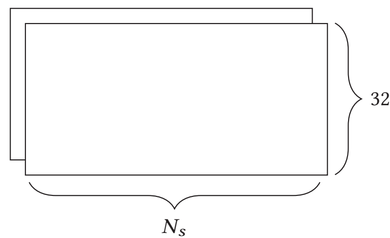


Figure 3.13: LiDAR data structure produced after a *single* horizontal scan (N_s = # of horizontal steps). The first channel contains distances to any surrounding objects, while the second channel contains the intensities of the reflected beams.

Chapter 4

Methods

In this chapter, the general methodology is introduced. First, the strategy used for generating the appropriate datasets is described. This process involves collecting, processing and labeling data. Then, we describe the design, training and evaluation of our proposed networks. Finally, a method for performing path verification is described along with an appropriate evaluation metric.

4.1 Generating datasets

The primary goal of this section is to describe the underlying process involved in generating the datasets used for end-to-end training of the steering- and road segmentation networks. The data (comprised of images, LiDAR and steering angles) was obtained by a test vehicle owned by the Norwegian Defense Research Establishment (FFI). The vehicle, shown in Fig. 4.1, is mostly operated in the proximity of a small village named Sessvollmoen in the municipality of Ullensaker, Norway. In this area the driving environment is primarily made up of dirt roads and other natural terrain (see Fig. 4.2). It is worth stressing that the data provided by FFI was not originally

intended for deep learning applications when it was collcted, and therefore requires some processing.

Figure 4.1: Vechicle used by FFI to collect data.

Figure 4.2: Examples of the road environment.

4.1.1 Data collection

The vehicle is equipped with a left, right and center camera in addition to a LiDAR scanner. Multiple cameras enable depth perception of the environment, which is useful for measuring distances to various objects. In this task, however, the left and right cameras are mainly used to provide examples of off-center road images. The reasons for this will become clear in Sec. 4.1.3.

The vehicle employs an operating system known as *Robot Operating System*[1], or ROS for short. In this operating system, each sensory device (camera, LiDAR, etc.) records information and sends it to a file known as a *bag*, which stores the serialized data. More specifically, a bag stores ROS message data published by various ROS *nodes*. Nodes are processes that perform some computation and can communicate with other nodes by sending messages.

[1]Documentation and other resources can be found at: http://wiki.ros.org/.

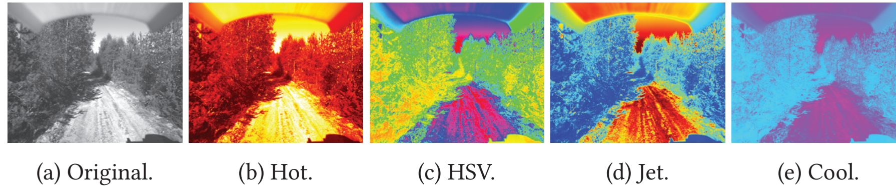
The dataset provided by FFI consists roughly of 1.4 TB of raw data amounting to over 150 ROS bags and over 400 000 images in total. Since the dataset was not originally intended for deep learning applications many of the bags lack measurements of specific quantities (such as steering angle and LiDAR) thus making them irrelevant. Additionally, many of the bags contain images which are poor in terms of learning quality. For instance, some video sequences consist merely of images of the vehicle remaining stationary for several minutes. We filter out such sequences as they can stall learning.

4.1.2 Data processing

The sensory equipment in the vehicle operate at different frequencies. For instance, the cameras capture images at 6 Hz, the LiDAR scanner rotates between 5-20 Hz, and the steering-sensor records angles at 50 Hz. Consequently, image, LiDAR and steering angle measurements are not time-synchronized which poses a problem since each camera image and LiDAR reading should be associated with a steering angle at which the information was recorded. Additionally, the time at which each image, LiDAR reading and angle were recorded, and the time at which they were written to memory, are different. The image-delay was estimated to be less than 50 ms using a tool called *rqt_bag*. Since the steering data is significantly smaller than the image data, it is reasonable to assume that the delay for this quantity is much less than 50 ms, and therefore negligible. Moreover, we only have have access to the time at which the steering information was saved and *not* recorded. For this reason, we choose to operate with save-time for the image data as well.

To synchronize the image, LiDAR and steering angle data, we simply select the steering angle and LiDAR reading which is closest in time to the timestamp of a given image, and use the angle as a corresponding label. This method works well provided that the difference in time between initial recordings of sensory devices is small. For the ROS bags that we have manually inspected, this appears to be the case.

The images generated from the left and right cameras are in grayscale, whereas the

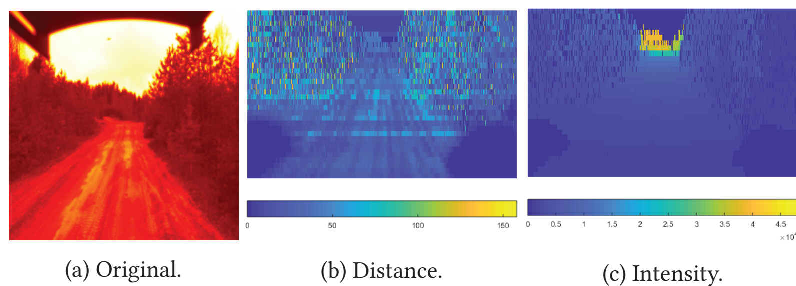


(a) Original. (b) Hot. (c) HSV. (d) Jet. (e) Cool.

Figure 4.3: Some predefined colormaps.

center camera produces color-images. This becomes problematic since the VGG-16 network, which we use as a pre-trained model (see Sec. 3.2), is originally trained on RGB images. To make this network compatible with our data, it is necessary to convert the grayscale images to RGB. This can be achieved by using predefined colormaps to alter the color scheme of the image. Fig. 4.3 shows some examples. It is worth mentioning that a colormapped image contains exactly the same amount of information as its grayscale counterpart, but differs in that it contains three channel dimensions instead of one. From experimentation we found that the Hot colormap array produced the most satisfactory results.

In Section 3.5, the general strategy used for constructing LiDAR image data was outlined. Since the LiDAR scanner operates at varying scanning frequencies (5-20 Hz) the number of horizontal steps varies for each scan, resulting in different image widths. To overcome this issue, we (1) determine the most frequent output width from all



(a) Original. (b) Distance. (c) Intensity.

Figure 4.4: Processed camera and LiDAR image.

LiDAR images generated, and (2) bilinearly interpolate all LiDAR images satisfying a certain length threshold to this output width. Fig. 4.4 shows an example of a processed camera image along with the corresponding LiDAR image. As discussed in Sec. 3.5, the LiDAR image contains two feature-channels, namely distance and intensity. The distance channel shows normalized distance measurements to objects in the front-facing part of the vehicle. Likewise, the intensity channel shows normalized intensity values of the reflected laser beams. A clear pattern is observed when comparing the processed LiDAR images to the original. In particular, we see that the road ahead is clearly visible with some noise in the form of trees and bushes on the sides.

The data collection and processing steps above are summarized in Fig. 4.5. The test vehicle records the desired quantities at different frequencies, which are then processed and time-synchronized into a data structure, which can be used for learning.

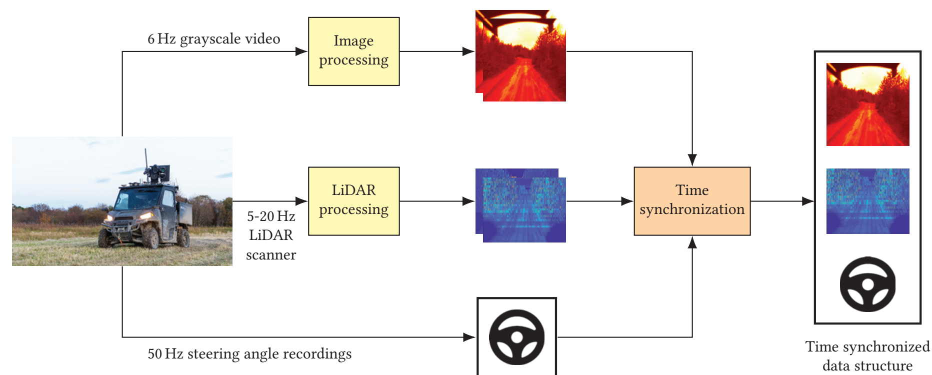


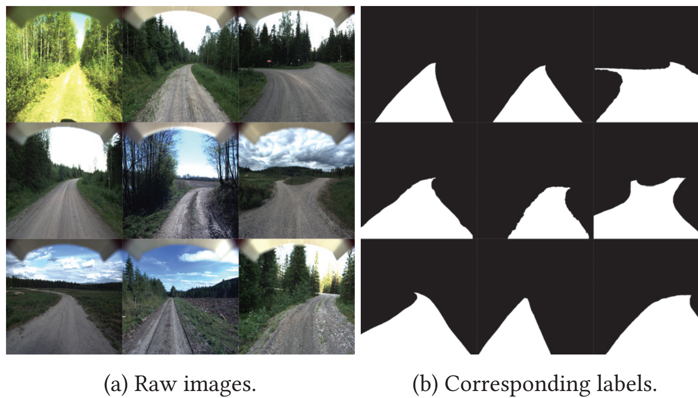
Figure 4.5: Data collection and processing pipeline.

4.1.3 Data labeling

In an end-to-end learning approach for driving, the network is taught to emulate human maneuvers. This involves providing the network with data exemplifying

correct driving behaviour. However, with this approach, the system never learns how to recover from mistakes. For example, if the vehicle finds itself away from the lane center, it should quickly adjust itself to return back to the center position. Training with data only from a human driver is therefore not sufficient and can cause the vehicle to drift [1]. To lessen the effect of this issue, we present our steering network with image examples simulating the vehicle in different positions from the road center, along with the appropriate steering adjustment. Motivated by [1], we use images from the left and right cameras on the vehicle as off-center examples. Inspired by [40], we add/subtract a small offset value to the ground truth steering angle associated with the vehicle's *actual* position, and use this as an appropriate label for the off-center images. This will cause the vehicle to slowly drift back towards the center of the road.

Recall from Sec. 3.3 that the task of a segmentation network is to assign each pixel in an image to a specific category. For road-segmentation, the categories are comprised of road and *not*-road. In order for the network to recognize these classes, it is necessary to train it on sample images in which the pixels have been labeled to their respective categories. We manually label each image in the dataset by blackening background pixels and whitening road pixels. Fig. 4.6 shows some examples.



(a) Raw images. (b) Corresponding labels.

Figure 4.6: Labeling instances in the segmentation dataset.

4.2 Steering network

Up to now, we have considered strategies for preprocessing and labeling data. This is perhaps one of the most important steps in building a successful deep learning application. Spending weeks training a model on the back of weak quality data will typically underperform a relatively simple model trained on high quality data. For this reason, a significant portion of time was spent on generating a quality dataset for learning.

In this section, the design, training and evaluation aspects of the steering model will be examined in detail. As mentioned earlier, the main goal is to combine feature information from images and LiDAR and use this to make steering predictions. Different strategies for accomplishing this can be found in the literature. The main techniques include *early fusion* [26], *late fusion* [26, 16, 9, 14, 39, 54] and *slow fusion* [26]. We adopt a late fusion approach as this is relatively easy to implement and has shown good results in practice [9, 39, 54].

4.2.1 Overview

Late fusion involves merging features from the final layers of two separately trained networks. For this reason, we divide the training process into three separate stages. Fig. 4.7 shows a block diagram of the first stage. In this stage, a convolutional network is trained to predict steering angles based solely on road images and time-synchronized steering commands. The goal here is to train the CNN to learn to detect features from the image input that are important for steering. To achieve this goal, it is beneficial to feed the network with vast amounts of data. We augment the training set by mirroring each image such that images of a left-turn become a right-turn, and vice versa. Note that the steering angle corresponding to the mirrored image is adjusted by inverting the sign. Images from this dataset are then selected at random, and fed into a pre-trained VGG network (discussed in Sec. 3.1.5). For reasons discussed in Sec. 3.2, we extract the features from the fifth max pooling layer of this network (denoted Pool5) and use

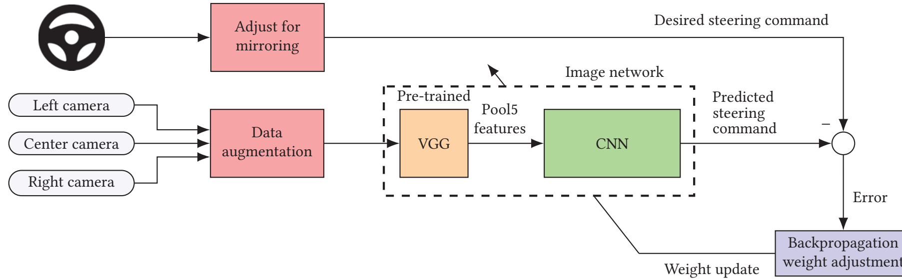


Figure 4.7: (Stage 1) Only camera images and steering angles are used for training. We extract features from the fifth pooling layer of a pre-trained VGG-16 model and use this as input to our CNN. The CNN outputs steering predictions which are compared to the ground truths, and appropriate weight adjustments are made to the Image network for a fixed number of iterations.

this as input to the CNN. The CNN produces an estimated steering angle which is compared to the ground truth angle. The error between these angles is then used to make a weight update to the network. After the weights have been updated, the process is repeated a fixed number of iterations.

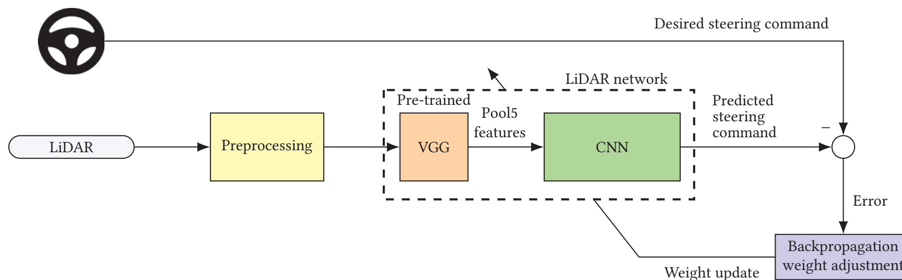


Figure 4.8: (Stage 2) Only LiDAR and steering angles are used for training. We extract features from the fifth pooling layer of a pre-trained VGG-16 model and use this as input to our CNN. The CNN outputs steering predictions which are compared to the ground truths, and appropriate weight adjustments are made to the LiDAR network for a fixed number of iterations.

Fig. 4.8 shows a block diagram of the second stage. This stage is similar to the former, except for two important changes. First, a separate convolutional network is used to extract features from LiDAR images instead of RGB-images. Second, the data augmentation block has been replaced by a preprocessing block. This is necessary to ensure that the output dimensions from preprocessing agree with the dimensions expected by the pre-trained VGG model. Although we use a separate CNN for this stage, it has the exact same architecture as the CNN used in stage 1. The architecture will be discussed in Sec. 4.2.2.

Fig. 4.9 shows a block diagram of the third and final stage. In this stage, the trained models from stages 1 and 2 are used together with a new network, which we call *fusion network*, to estimate steering commands. Notice that the CNNs above do not output steering angles, as in stages 1 and 2, but instead output features. We remove the final prediction layer from each CNN and concatenate the resulting image and LiDAR features into a single vector and use this as input to the fusion network. The process of updating the weights and biases in the fusion network is identical to the

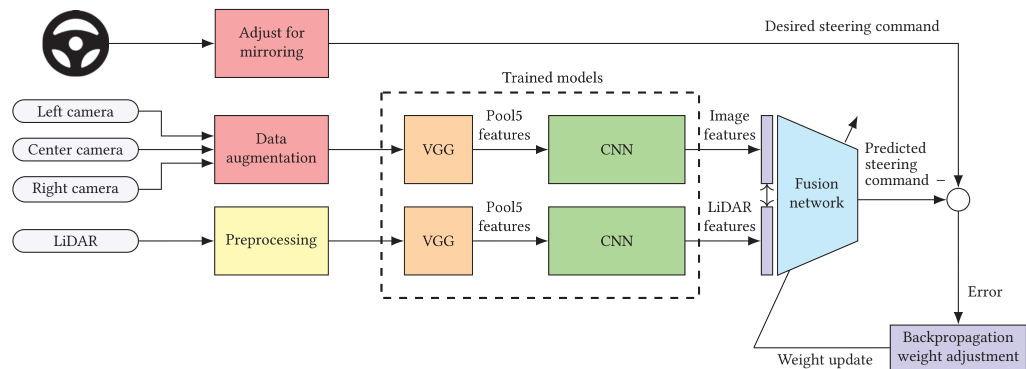


Figure 4.9: (Stage 3) Late fusion. The trained models from Figs. 4.7 and 4.8 are combined, but with the final prediction layers removed such that each network outputs key features instead. The features are combined by late fusion, and used as input to a separate fusion network. Comparison with ground truth value and appropriate weight adjustment is the same as in Figs. 4.7 and 4.8.

update-process discussed above for the Image and LiDAR networks.

Once the steering network has been trained, it can be used to make online steering predictions, as Fig. 4.10 illustrates. Time-synchronized measurements from the camera and LiDAR scanners are supplied as input to the trained steering network, which outputs a predicted steering angle. This angle is then used as a reference by the vehicle control system to steer the vehicle.

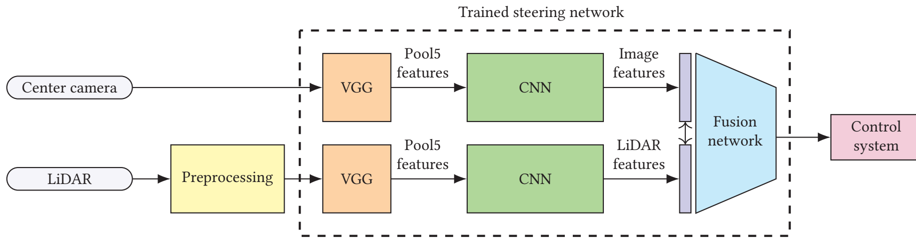


Figure 4.10: After training the model in Fig. 4.9, it can be used to make online steering predictions, which can be used as references by the vehicle control system.

4.2.2 Architecture

Tables 4.1 and 4.2 show the network configurations of the CNN blocks in Figs. 4.7 and 4.8, respectively. Notice that the layer types and arrangement in both tables are identical. The input dimensions differ because of different data used for training. Additionally, we use stride 4 in the max-pooling layers in Table 4.2 to reduce the number of parameters. The architecture below was inspired by [40], who also used a convolutional network for steering angle prediction, but in a conventional driving setting, and was able to achieve a root mean squared error (RMSE) of 0.0645.

As discussed in Sec. 3.1, the convolutional layers extract salient features and combine them to form more complex features. The batch normalization layers simply normalize the output from each convolution layer, while max-pooling subsamples the input to reduce computational load in addition to providing local invariance to translation.

Finally, a fully-connected layer takes a flattened vector as input and outputs the predicted steering angle.

Table 4.1: CNN architecture with image input. All Conv. layers use ReLU activation.

Layer	Layer type	Input dims	Output dims	#Filters	Filter dims	Stride
1	Conv2D	$7 \times 7 \times 512$	$7 \times 7 \times 256$	256	$3 \times 3 \times 512$	1
2	BatchNorm	$7 \times 7 \times 256$	$7 \times 7 \times 256$	-	-	-
3	Conv2D	$7 \times 7 \times 256$	$7 \times 7 \times 128$	128	$1 \times 1 \times 256$	1
4	BatchNorm	$7 \times 7 \times 128$	$7 \times 7 \times 128$	-	-	-
5	Conv2D	$7 \times 7 \times 128$	$7 \times 7 \times 256$	256	$3 \times 3 \times 128$	1
6	BatchNorm	$7 \times 7 \times 256$	$7 \times 7 \times 256$	-	-	-
7	Max-pool	$7 \times 7 \times 256$	$4 \times 4 \times 256$	-	2×2	2
8	Conv2D	$4 \times 4 \times 256$	$4 \times 4 \times 128$	128	$3 \times 3 \times 256$	1
9	BatchNorm	$4 \times 4 \times 128$	$4 \times 4 \times 128$	-	-	-
10	Max-pool	$4 \times 4 \times 128$	$2 \times 2 \times 128$	-	2×2	2
11	Conv2D	$2 \times 2 \times 128$	$2 \times 2 \times 256$	256	$3 \times 3 \times 128$	1
12	BatchNorm	$2 \times 2 \times 256$	$2 \times 2 \times 256$	-	-	-
13	Conv2D	$2 \times 2 \times 256$	$2 \times 2 \times 512$	512	$4 \times 4 \times 256$	1
14	BatchNorm	$2 \times 2 \times 512$	$2 \times 2 \times 512$	-	-	-
15	FC	2048	1	-	-	-

Table 4.2: CNN architecture with LiDAR input. All Conv. layers use ReLU activation.

Layer	Layer type	Input dims	Output dims	#Filters	Filter dims	Stride
1	Conv2D	$7 \times 22 \times 512$	$7 \times 22 \times 256$	256	$3 \times 3 \times 512$	1
2	BatchNorm	$7 \times 22 \times 256$	$7 \times 22 \times 256$	-	-	-
3	Conv2D	$7 \times 22 \times 256$	$7 \times 22 \times 128$	128	$1 \times 1 \times 256$	1
4	BatchNorm	$7 \times 22 \times 128$	$7 \times 22 \times 128$	-	-	-
5	Conv2D	$7 \times 22 \times 128$	$7 \times 22 \times 256$	256	$3 \times 3 \times 128$	1
6	BatchNorm	$7 \times 22 \times 256$	$7 \times 22 \times 256$	-	-	-
7	Max-pool	$7 \times 22 \times 256$	$2 \times 6 \times 256$	-	2×2	4
8	Conv2D	$2 \times 6 \times 256$	$2 \times 6 \times 128$	128	$3 \times 3 \times 256$	1
9	BatchNorm	$2 \times 6 \times 128$	$2 \times 6 \times 128$	-	-	-
10	Max-pool	$2 \times 6 \times 128$	$1 \times 2 \times 128$	-	2×2	4
11	Conv2D	$1 \times 2 \times 128$	$1 \times 2 \times 256$	256	$3 \times 3 \times 128$	1
12	BatchNorm	$1 \times 2 \times 256$	$1 \times 2 \times 256$	-	-	-
13	Conv2D	$1 \times 2 \times 256$	$1 \times 2 \times 512$	512	$4 \times 4 \times 256$	1
14	BatchNorm	$1 \times 2 \times 512$	$1 \times 2 \times 512$	-	-	-
15	FC	1024	1	-	-	-

The network configuration of the fusion network in Fig. 4.9 is shown in Table 4.3. The archiecture is comprised of two fully-connected layers. The first layer takes a concatenated vector of image and LiDAR features as input, which is reduced to size 100, while the final layer takes this as input and yields the final steering command. The output dimensions and number of layers were selected with the intention of keep the number of parameters low, due to overfitting concerns.

Table 4.3: Fusion network architecture. All fully-connected layers use ReLU activation.

Layer	Layer type	Input dims	Output dims	#Filters	Filter dims	Stride
1	FC	3072	100	-	-	-
2	FC	100	1	-	-	-

4.2.3 Training

We train the convolutional and fusion networks by minimizing a mean-squared loss using mini-batch gradient descent with momentum. A batch size of 40 and momentum of 0.9 was generally used. The learning rate was initially set to 0.00001, and after a fixed number of steps, decayed by a factor of 0.94 until a learning rate of 0.0000001 was reached. The numbers above were determined as a result of trial and error. The number of iterations used for training were calculated using the following formula:

$$\text{Iterations} = \frac{\#\text{Epochs} \times \text{Training set size}}{\text{Batch size} \times \#\text{GPUs}}. \tag{4.1}$$

One epoch represents a full pass through the training set, and the total number of epochs was set to 100 when training the individual networks. The training and validation set sizes are shown in Table 4.4. Notice that the networks use the same training and validation sets for learning. This is to ensure that the performance of each network can be compared on equal terms. We use a 80-20 dataset split (80% for training and 20% for validation), which is common. The experiments were carried out using two GeForce GTX TITAN X GPUs. Finally, weights in the convolutional and fusion

network layers were initialized using sampled values from a uniform distribution in the (0,1) range.

Table 4.4: The number of data instances used for training and validation. We use the same training and validation sets for all networks.

	Image CNN	**LiDAR CNN**	**Fusion network**
Training	113 613	113 613	113 613
Validation	28 404	28 404	28 404
Total	142 017	142 017	142 017

4.2.4 Evaluation

The performance on the validation set is measured using a normalized root mean squared error (RMSE). Specifically, the metric used is:

$$\text{RMSE} = \sqrt{\frac{\text{MSE}}{\text{MSV}}}, \tag{4.2}$$

where

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \quad \text{MSV} = \frac{1}{N}\sum_{i=1}^{N} y_i^2.$$

y_i denotes the ground truth steering angle, while $\hat{y}_i$ is the predicted value. We use normalized RMSE as opposed to just MSE, because this is a common metric used for evaluating regression tasks [10] and enables us to compare results with other literature. A problem with using MSE or RMSE for driving, however, is that any difference between predicted and true steering angles, will be accumulated, no matter how small. Since it is not uncommon for human drivers to have small biases when driving, we argue that predicted steering angles close to ground truth behavior should be considered equally valid. For this reason, we use an additional metric for measuring steering accuracy. In particular, a steering angle is considered correct if it is within a certain tolerance threshold of the ground truth angle. Chen and Wang et. al [8]

perform a similar task, and uses an accuracy metric for evaluating performance. In particular, they use a tolerance threshold of 6°. Motivated by this, we use the same accuracy metric, which can be mathematically expressed as

$$\text{Steering Accuracy} = \frac{1}{N}\sum_{i=1}^{N}[|y_i - \hat{y}_i| < 6]. \quad (4.3)$$

The brackets, known as Iverson Bracket[2], convert any logical proposition to 1 if satisfied, and 0 otherwise.

4.3 Segmentation network

In the previous section, a model for predicting steering angles based on image and LiDAR data was considered. To determine whether predictions on future data are trustworthy or not, we design and implement a separate path verification model in Sec. 4.4. This model uses predicted steering angles along with segmented images of the road environment to decide if the given steering angle is trustable or not.

In this section, the design, implementation and evaluation of a segmentation network will be examined in detail. The goal of this network is to produce segmented road images (i.e., images comprising only pixels of road and background) to be used by the path verification model in the next section. The components of this model are based on the theory presented in Sec. 3.3.

4.3.1 Overview

The main goal of the segmentation network is to produce segmented images of the road environment. To achieve this goal, it is necessary to train the network to recognize pixels of road vs. pixels of not-road. This is accomplished by presenting the network with images of the road environment along with the segmented counterparts. In this

[2] https://en.wikipedia.org/wiki/Iverson_bracket

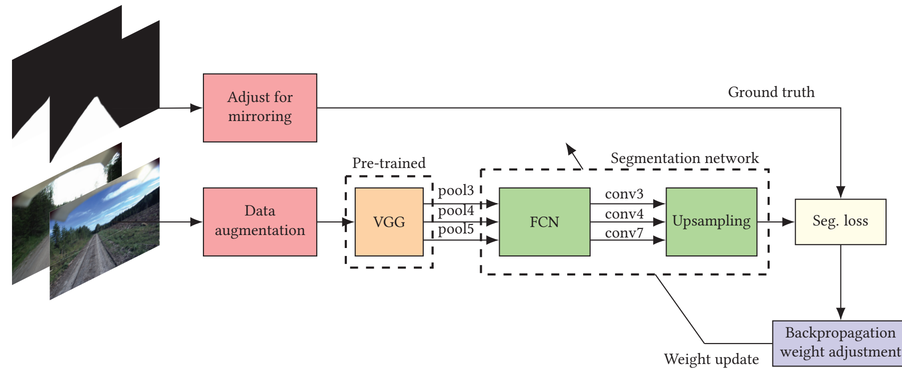


Figure 4.11: The segmentation network is trained by feeding it examples of the road environment along with the corresponding labels. Features from the 3rd, 4th and 5th pooling layers from a pre-trained VGG-16 network are used as input the FCN. Upsampling the result and yields a segmented image with same dimensions as the original input image. Weight updates based on the loss are performed a fixed number of iterations.

way, the network eventually learns to detect features that are characteristic of road and use this information to produce accurate segmentations.

Fig. 4.11 shows a block diagram of the strategy used for training the segmentation network. This strategy was inspired by [36], which demonstrates state-of-the-art results for several segmentation tasks. We utilize features from pre-trained VGG-16 layers as input to a fully convolutional network (FCN). Specifically, the outputs from the 3rd, 4th and 5th max-pooling layers from a pre-trained VGG-16 model are used. The FCN processes these inputs individually, and feeds them to an upsampling network. This network fuses the features together and interpolates the result to produce a segmented image, which is compared to the ground truth by means of a loss metric. A weight adjustment is issued to the FCN and upsampling networks, as these contain the learnable parameters. This process is then repeated a fixed number of iterations. Similar to the steering network, we augment the training set by mirroring each image. However, we also perform augmentation based on color by adding sampled values

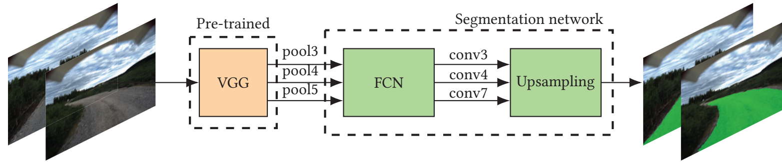


Figure 4.12: After the model in Fig. 4.11 is trained, it can be used to make segmentations online.

from a normal distribution to the hue and saturation of the original image.

Once the segmentation net has been trained, it can be used to segment raw images online, as Fig. 4.12 illustrates. As we shall see, the segmented road-images is used together with steering predictions to estimate the general path of the vehicle.

4.3.2 Architecture

Table 4.5 shows the network configuration of the FCN block in Fig. 4.11. Each convolutional layer processes the corresponding pre-trained pooling layers (indicated by the layer number) individually. This differs from the architectures introduced in Tables 4.1 and 4.2 where each output was sequentially fed as input to the next layer. Table 4.6 shows the network configuration of the *upsampling block* in Fig. 4.11. This configuration uses a sequence of transposed convolutions to upsample the input dimensions. Fig. 4.13 illustrates the process involved in turning the convolutional outputs from the FCN into a segmented image. The first layer simply upscales the output of the conv 7 layer. The output of this layer is summed element-wise with the output of the conv 4 layer, which is then fed as input to the second layer. This input is upsampled, and the same process is repeated until the ouput has the same spatial dimensions as the original image used for training (in our case we use images of size $256 \times 320 \times 3$). The motivation behind combining features in this manner was discussed in Sec. 3.3.1, and has proven to improve segmentation detail. The configurations below were taken

from the following github page[3] and adapted to our own data.

Table 4.5: FCN architecture.

Layer	Layer type	Input dims	Output dims	#Filters	Filter dims	Stride
3	Conv2D	$32 \times 40 \times 256$	$32 \times 40 \times 2$	2	$1 \times 1 \times 256$	1
4	Conv2D	$16 \times 20 \times 512$	$16 \times 20 \times 2$	2	$1 \times 1 \times 512$	1
7	Conv2D	$8 \times 10 \times 4096$	$8 \times 10 \times 2$	2	$1 \times 1 \times 4096$	1

Table 4.6: Upsampling architecture.

Layer	Layer type	Input dims	Output dims	#Filters	Filter dims	Stride
1	Conv2D_transpose	$8 \times 10 \times 2$	$16 \times 20 \times 2$	2	$4 \times 4 \times 2$	2
2	Conv2D_transpose	$16 \times 20 \times 2$	$32 \times 40 \times 2$	2	$4 \times 4 \times 2$	2
3	Conv2D_transpose	$32 \times 40 \times 2$	$256 \times 320 \times 2$	2	$16 \times 16 \times 2$	8

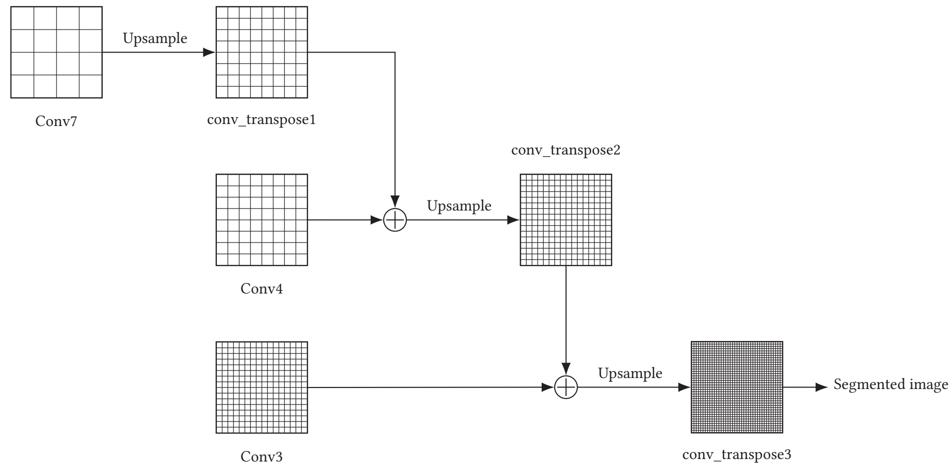


Figure 4.13: The upsampling process performed in the upsampling block of Figs. 4.11 and 4.12. Figure not to scale.

[3] https://github.com/maxritter/SDC-Semantic-Segmentation

4.3.3 Training

The segmentation network is trained by minimizing a cross entropy loss using an Adam optimizer [27], which is an extension to mini-batch gradient descent. A batch size of 20 was chosen for this task. The learning rate was initially set to 0.0001 and adaptively reduced by Adam during training. Again, the numbers above were chosen as a result of trial and error. The iterations used for training were determined from equation (4.1). A total of 50 epochs were used due to the size of the training set. Note that for pixel-wise prediction tasks, each pixel has a label meaning that an input image of size $256 \times 320 \times 3$ actually amounts to 245 760 labeled examples. Table 4.7 shows the training and validation set sizes used. The KITTI dataset [18] was used to pre-train the network for segmentation. The dataset contains camera images along with segmentations (and much more), recorded around Karlsruhe, Germany. After training on this dataset, we train the network on our own data (labeled FFI). The experiments were carried out on a single GeForce GTX TITAN X GPU. The network weights were initialized using sampled values from a normal distribution with zero mean and 0.01 standard deviation.

Table 4.7: Number of images used for training and validation in the respective datasets.

	KITTI	**FFI**
Training	398	400
Validation	290	100
Total	688	500

4.3.4 Evaluation

The segmentation performance on the validation set is measured using four evaluation metrics, which are displayed in Table 4.8. We use the same metrics from [36], as these are commonly used in segmentation and scene parsing applications. n_{ij} denotes the number of pixels of class i predicted to belong to class j, n_{cl} is the number of classes, and $t_i = \sum_j n_{ij}$ denotes the totalt number of pixels of class i. Pixel accuracy simply

Table 4.8: Evaluation metrics.

Pixel Accuracy	Mean Accuracy	Mean IoU	Frequency Weighted IoU
$\sum_i n_{ii} / \sum_i t_i$	$(1/n_{\text{cl}}) \sum_i n_{ii}/t_i$	$(1/n_{\text{cl}}) \sum_i n_{ii} / \left(t_i + \sum_j n_{ji} - n_{ii}\right)$	$\left(\sum_k t_k\right)^{-1} \sum_i t_i n_{ii} / \left(t_i + \sum_j n_{ji} - n_{ii}\right)$

measures the ratio of correctly predicted pixels for each class to the total number of pixels for each class. Mean accuracy is similar, but instead determines the pixel accuracy of each class individually and computes the average. As the name implies, intersection over union (IoU) computes the ratio of the number of intersecting pixels between a predicted image and a ground truth image and the union of pixels between these two, for some class i. Mean IoU just computes the mean of the IoUs for each class. Finally, frequency weighted IoU is similar to IoU, but weights the ratio based on the frequency of pixels in class i. For instance, a class with few pixels is weighted less than a class with a great number of pixels.

4.4 Path verification

Up to now, we have considered steering and segmentation as two separate networks performing individual tasks. In this section, we combine outputs from both networks to tell whether a given steering angle should be trusted or not. A general overview of the system is first presented, giving a basic idea of the task in question in addition to showing the components involved. Then, the verification process is described in detail, outlining and explaining the steps needed in order to accept or discard a given steering angle. Finally, a metric for evaluating performance is described.

4.4.1 Overview

Fig. 4.14 shows a block diagram of our trustability system. We incorporate trained steering and segmentation models, together with a verification system (treated as a black box for now), to decide whether a given steering angle should be trusted or not.

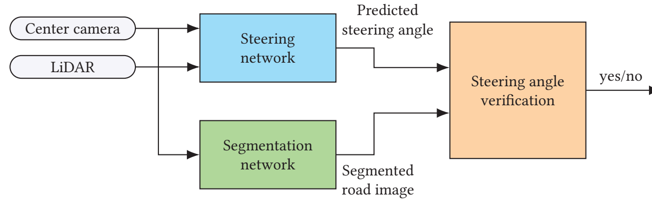


Figure 4.14: Trustability system.

Specifically, camera images and LiDAR measurements are fed into a steering network which outputs a predicted steering angle. The same camera image is supplied to a segmentation network, which produces a segmented road image. The outputs from these networks are then used as input to verification, and a binary decision is made.

4.4.2 Steering angle verification

The predicted steering angle is accepted if the vehicle, with a given speed, remains within the designated road boundary for a given distance threshold. Fig. 4.15 illustrates the general verification process, which involves three steps. First, we transform the segmented road image into a bird's eye view image, using a perspective transformation. Second, we use depth stereo vision to estimate the real world road length in the bird's eye view image, which can then be used to calculate pixel density. Third, with the computed pixel density, the distance the vehicle moves after 1 second can be converted to pixels, which is used as a verification threshold. Specifically, the steering angle is accepted if the vehicle remains within the road boundary for a distance threshold d. The details of the calibration steps 1 and 2 are described in the following paragraphs.

Perspective transformation. The transformation needed to take a segmented road image and produce a corresponding bird's eye view can be determined as follows. Let $\mathbf{P}^w$ be a 3D world coordinate that is mapped to the image plane coordinates $\mathbf{P}^{I_1}$

and $\mathbf{P}^{I_2}$, in two different cameras. From (3.5) we have that

$$\mathbf{P}^{I_1} = \mathbf{T}_1 \mathbf{P}^w$$
$$\mathbf{P}^{I_2} = \mathbf{T}_2 \mathbf{P}^w,$$

where $\mathbf{T}_i = \mathbf{K}_i \, [\mathbf{R}^c_{wi} \; \boldsymbol{\tau}_i]$. Assuming that $\mathbf{T}_1$ and $\mathbf{T}_2$ are invertible, we can rewrite and substitute the transformations above into the following expression

$$\mathbf{P}^{I_2} = \mathbf{D}\mathbf{P}^{I_1}, \tag{4.4}$$

where $\mathbf{D} = \mathbf{T}_2 \mathbf{T}_1^{-1}$, and is 4×4. Equation (4.4) denotes the transformation from a point in the image plane of camera 1 to the image plane of camera 2, for some arbitrary 3D world coordinate. If we let I_1 represent the image plane associated with the segmented road image, and let I_2 represent the image plane associated with the bird's eye view

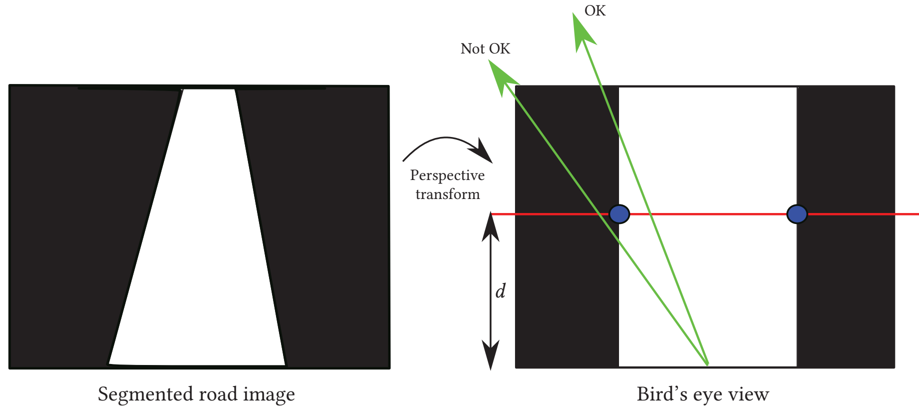


Figure 4.15: Steering angle verification process. First, the segmented road image is transformed by a perspective transformation into a bird's eye perspective. Then, the trajectory of the vehicle is determined by drawing the given steering angle (shown in green) on the transformed image. We assume the vehicle is always in the road center, and therefore draw the angle from the bottom center. Finally, the steering angle is accepted if the trajectory lies within the road boundary for the distance threshold d.

image, we can estimate the matrix **D** by finding 4 corresponding point pairs in each image, and solve the resulting system of equations for the unknowns in **D**. In particular, we select the four road corner points in each image as our corresponding point pairs, and solve the resulting system of equations below for the entries in **D**:

$$
\begin{aligned}
\mathbf{P}_1^{I_2} &= \mathbf{D}\mathbf{P}_1^{I_1} \\
\mathbf{P}_2^{I_2} &= \mathbf{D}\mathbf{P}_2^{I_1} \\
\mathbf{P}_3^{I_2} &= \mathbf{D}\mathbf{P}_3^{I_1} \\
\mathbf{P}_4^{I_2} &= \mathbf{D}\mathbf{P}_4^{I_1}.
\end{aligned}
$$

The OpenCV Library for Python has geometric image transformation functions for calculating the perspective transform **D**, thereby circumventing the need for a custom implementation.

Estimating pixel density. The real world pixel resolution (i.e., pixels per meter) can be determined by dividing the height in pixels of the bird's eye view image by the road length, assuming flat and straight road. The road length can be estimated from equation (3.6), which requires two parallel and identical cameras with known baseline and focal length in addition to two corresponding point pairs. As mentioned earlier, the test vehicle is equipped with left, right and center cameras, which we assume to be parallel and identical. The left and right cameras provide the needed stereo-image pair and selecting one of the corner points in which the road vanishes, in both images, as our corresponding point pair, the road length can be estimated. It is worth stressing, however, that the calibration process above can only be performed on images containing flat and straight road surfaces. Once the pixel density has been computed, the distance threshold in pixels can be calculated by multiplying the pixel density with the distance the vehicle travels in 1 second, or in other words, its speed. If this threshold sounds small, remember that the process is repeated 6 times per second (camera frequency is 6 Hz) and is done independently on each frame.

4.4.3 Evaluation

The verification model presented in the previous section can be viewed as a classifier since it outputs yes or no classes for each steering angle. A common way of evaluating the performance of a classifier is to study the *confusion matrix* (see Table 4.9), which presents a visual layout of the number of predicted vs. actual class labels. In this matrix, diagonal elements represent correct predictions, while off-diagonal elements represent false prediction. Specifically, we divide these elements into four categories: *true positives* (TP), *true negatives* (TN), *false positives* (FP) and *false negatives* (FN). Since it is rare in practice for a classifier to produce only true positives and true negatives, we need other metrics for evaluating classification performance. Equations (4.5), (4.6) and (4.7) present three common metrics based on the categories above.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{4.5}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{4.6}$$

$$\text{F}_1 = \frac{\text{TP}}{\text{TP} + \frac{\text{FN}+\text{FP}}{2}}. \tag{4.7}$$

The output of these metrics will be valued differently depending on the classification task in question. In our case, it is paramount that the verification system does not classify any incorrect steering angles as correct (false positives). For this reason, precision should be close to 100%, even if this means that acceptable steering angles

Table 4.9: Structure of the confusion matrix. Diagonal elements show correct predictions, while off-diagonal elements show false predictions (TP=True Positives, FN=False Negatives, FP=False Positives, TN=True Negatives).

		Predicted	
		Yes	No
Actual	Yes	TP	FN
	No	FP	TN

get rejected (low recall). The F_1 score calculates the *harmonic mean* of precision and recall, and favors classifiers that have similar precision and recall.

To evaluate classification performance of our verification system, we need to define the *actual* class labels. We label any predicted steering angle as true if it satisfies the following condition

$$|\text{predicted steering angle} - \text{ground truth angle}| < 6^\circ,$$

and false otherwise. The tolerance threshold of 6° was inspired from [8].

Chapter 5

Results and discussion

In this chapter, we analyze the performance of the proposed models in Chapter 4. For each model, we (1) present evaluation results on the validation set, and (2) give a discussion involving: (i) most important findings, (ii) explanations, (iii) limitations, and (iv) recommendations for future research.

5.1 Steering network

5.1.1 Results

Table 5.1 shows the evaluation results based on the metrics discussed in Sec. 4.2.4 for the Image CNN, LiDAR CNN and Fusion network. In the following discussion, we refer to these networks simply as Image network, LiDAR network and Fusion network. Fig. 5.1 shows the absolute error between the predictions from the Image, LiDAR and fusion networks and ground truth values, for all 28404 instances in the validation set. Figs. 5.2 and 5.3 show the bar plots and data instances corresponding to the marked Top 1 and Top 2 errors in Fig. 5.1. In addition, Table 5.2 shows the numerical errors

Table 5.1: Performance on the validation set by the Image, LiDAR and Fusion networks presented in Sec. 4.2. Values are based on the metrics discussed in Sec. 4.2.4. Zero and Mean represent blind predictions of the steering angle. The former always predicts a zero angle, while the latter always predicts the mean angle based on the ground truth values. (RMSE = Root Mean Squared Error, S.A = Steering Accuracy).

Metric	Zero	Mean	Image	LiDAR	Fusion
RMSE	1.00	0.99	0.22	0.22	**0.19**
S.A	0.82	0.82	0.98	0.98	**0.99**

from these bar plots. Finally, 5.4 shows the respective learning curves produced by the Image, LiDAR and fusion models from Sec. 4.2. These results shall now be discussed in the following section.

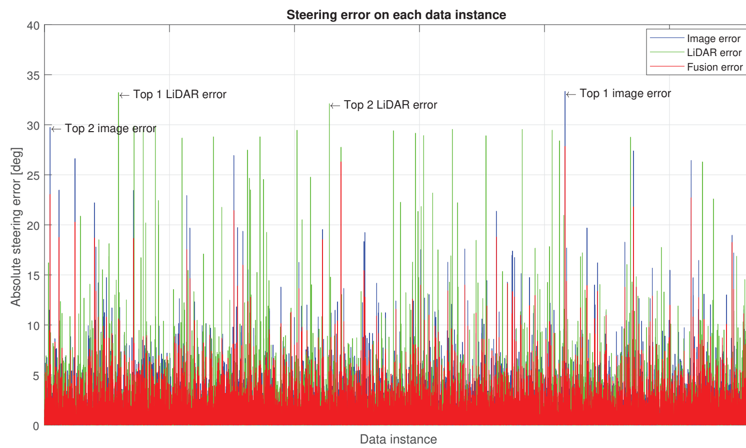


Figure 5.1: Absolute error between predicted and ground truth steering angle on all 28404 data instances in the validation set, for the respective networks. The two highest errors from the Image and LiDAR networks are marked Top 1 and Top 2, respectively.

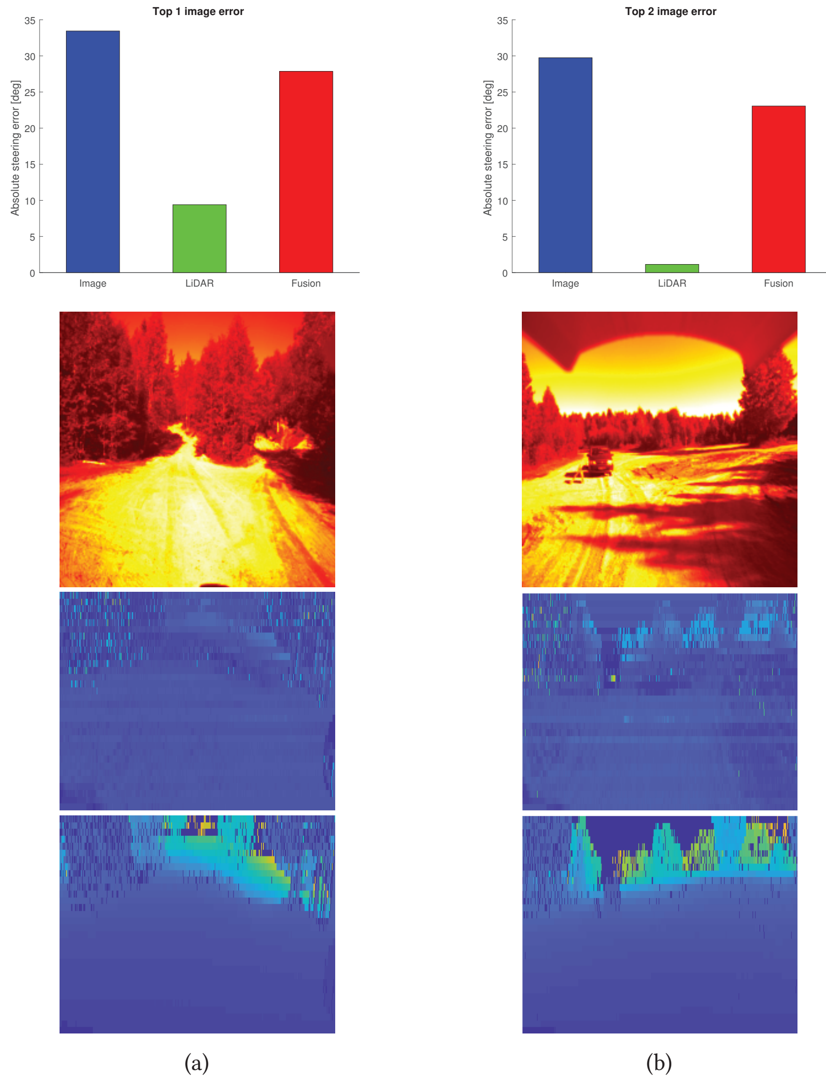
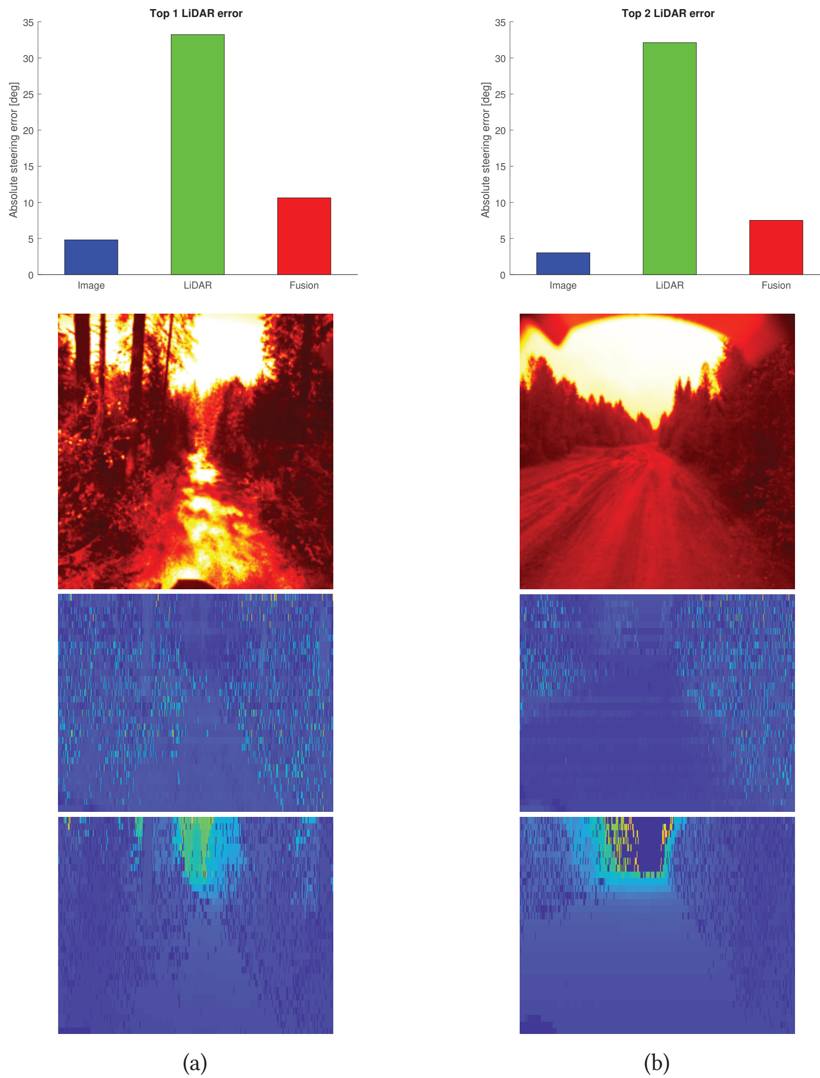


Figure 5.2: Column (a) and (b) show the bar plots and data instances corresponding to the Top 1 and Top 2 image errors from Fig. 5.1, respectively. Color mapped images are shown below the bar plots. LiDAR images consists of distance (top) and intensity (bottom) channels.

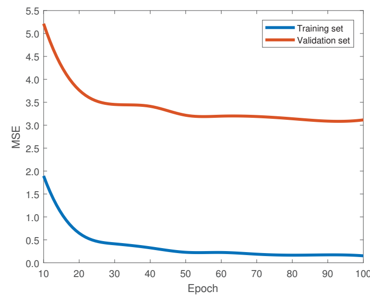


(a) (b)

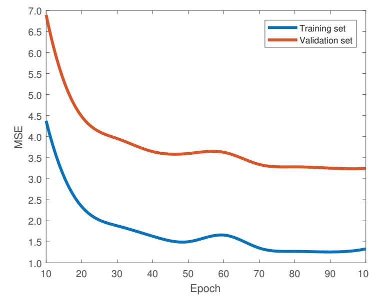
Figure 5.3: Column (a) and (b) show the bar plots and data instances corresponding to the Top 1 and Top 2 LiDAR errors from Fig. 5.1, respectively. Color mapped images are shown below the bar plots. LiDAR images consists of distance (top) and intensity (bottom) channels.

Table 5.2: Top 1 and Top 2 Image and LiDAR errors from the respective bar plots in Figs. 5.2 and 5.3 shown numerically.

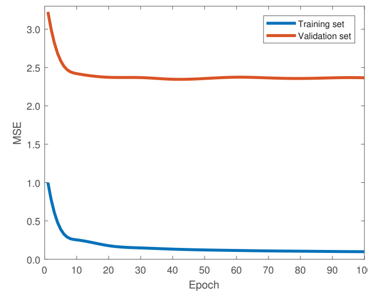
Column	Image error	LiDAR error	Fusion error
Fig. 5.2a	33.45	9.40	27.86
Fig. 5.2b	29.75	1.13	23.06
Fig. 5.3a	4.80	33.23	10.62
Fig. 5.3b	3.02	32.12	7.51



(a) Image.



(b) LiDAR.



(c) Fusion.

Figure 5.4: Learning curves produced by the respective Image, LiDAR and Fusion networks. The curves show the progression of training and validation set errors as the number of training epochs increases.

5.1.2 Discussion

The main goal of this master thesis is end-to-end steering angle prediction for an off-road vehicle using convolutional networks, and to evaluate the potential improvement of using fused image and LiDAR data for steering. Studying Table 5.1, we see that training on fused image and LiDAR features produces better generalization on this dataset, than training on each data source separately. This result agrees with Chen and Wang et. al [8], who also use fused image and LiDAR input for driving, but in a different setting, and show that extra depth information does indeed improve learning.

To understand why fusion outperforms the individual networks on this dataset, it is worth examining some data examples where each network has made particularly bad predictions. Studying the plots in Figs. 5.2 and 5.3, as well as the corresponding errors in Table 5.2, closely, the following is observed: when images generalize poorly, LiDAR does better, and vice versa. This result indicates that the respective networks have learned to recognize different features from the input data. Examining the plots in Figs. 5.2 and 5.3 in addition to Table 5.2 again, we see from the fusion error that combining image and LiDAR features leads to improvement when compared to the worst performing data source. However, from the plots it is clear that fusion performs better when LiDAR generalizes poorly than when images generalize poorly. It appears that fusion places a higher emphasis on image features than LiDAR features when making predictions. This is not surprising since we employ twice as many image features as LiDAR features when training the fusion network in Sec. 4.2 (see final layer in Tables 4.1 and 4.2).

After studying the processed camera and LiDAR images in Fig. 4.4, it makes intuitive sense that regular camera images should contain more information about the road environment than LiDAR, which is why we utilize more image features for training our fusion model. However, judging from the results in Table 5.1, LiDAR shows comparable generalization performance to that of images on this dataset. This result suggests that LiDAR may be underrated compared to images, and that better generalization results may be achieved by placing a higher emphasis on LiDAR features for learning. We

increase the number of LiDAR features with 512 (reducing the stride from 4 to 2 in first max-pooling layer in Table 4.2) and train the same fusion model on the extended feature input. With this modification, we achieve an RMSE of 0.18, thus confirming better generalization.

In addition to the bar plots in Figs. 5.2 and 5.3, we also provide the actual image and LiDAR instances corresponding to the Top 1 and Top 2 errors. The exact reasons as to why the Image network performs badly on the instances in Figs. 5.2a and 5.2b, and why LiDAR performs better, is difficult to state with certainty. Studying Figs. 5.2a and 5.2b closely, and comparing them with Figs. 5.3a and 5.3b, we observe more sun glare in the former, than the latter. As mentioned before, LiDAR scanners are invariant to lighting conditions, which can explain why LiDAR images generalize better in these cases. Studying the LiDAR instances in Figs. 5.3a and 5.3b, and comparing them to the LiDAR instances in Figs. 5.2a and 5.2b, we observe more noise in the former, than the latter. Recall from Sec. 4.1.2 that we bilinearly interpolate all LiDAR images less than or greater than the most frequent output width. If the original LiDAR width is significantly smaller than the target output width, interpolation can explain the noise observed in Fig. 5.3.

Fig. 5.4 presents the learning curves produced by the respective image, LiDAR and fusion networks. These curves are useful for evaluating the general learning trends and, in particular, to detect if the models are overfitting the data. This is typically identified in the curves by the training error getting smaller in addtion to the validation error getting larger. Studying Figs. 5.4a, 5.4b and 5.4c closely, we see that this is not the case. The validation errors have stabilized and appear to be more or less constant, thereby ruling out overfitting as an issue in our trained models.

Table 5.3 compares our RMSE result from Table 5.1 with other steering models in the literature. As demonstrated above, RMSE can be reduced further by utilizing additional LiDAR features for learning. From Table 5.3, we see that our proposed fusion model is, in terms of RMSE, close in proximity to PilotNet [1], which is the network proposed by NVIDIA. NVIDIA showed that this network is capable of driving in traffic on local roads with or without lane markings and on highways [1]. Although the dataset used

Table 5.3: Results from other steering models in the literature (reported from [10]) compared to our model, which we call FusionNet (RMSE=Root Mean Squared Error).

	FusionNet	PilotNet	AlexNet	VGG-16	ST-Conv+ConvLSTM+LSTM
RMSE	0.19	0.16	0.13	0.10	0.06

by NVIDIA differs significantly from ours, the reported RMSE can be used as an overall indication of good driving performance. We therefore reason that the reported RMSE result on our dataset is an indication of good steering performance.

There are several reasons as to why our fusion model does not equal or surpass the models in Table 5.3. First and foremost, we use a dataset significantly different from the one used in the models presented. In particular, the models use a dataset representing a conventional driving setting with well-defined road boundaries and road texture. This is not always the case for our data, and makes the task in question more challenging. Second, a significant portion of poor image and LiDAR examples were filtered out during preprocessing, thereby reducing the size of the effective training and validation sets. In fact, the models above use a dataset comprising almost twice the amount of images as we use. Third, due to grayscale images from the left and right cameras, our model was not trained on *real* RGB images, which typically contain more information.

In addition to the reasons above, we list two further potential improvements for future research. First, in this thesis we have treated steering angle prediction as a regression problem, where the goal is to predict the exact real valued angle for each video frame. However, as discussed in Sec. 4.2.4, human drivers usually have small biases when driving, meaning that several steering angles can be considered valid for steering. For this reason, the task in question can be redefined as a classification problem, where the range of possible steering angles are divided into bins, e.g., 1 degree apart, and the goal is to classify them correctly. Classification is generally viewed as an easier task than regression, and could be more suitable for steering angle prediction. Second, our steering model makes steering predictions independently on each video frame. This is somewhat unintuitive since driving is widely considered a stateful

process (drive straight, turn left, turn right, etc.). Several approaches address this issue [10, 8, 56], and have adopted a recurrent network structure to account for historical vehicle states when making future predictions. In particular, these works have included recurrent units, such as LSTM and Conv-LSTM, at specific network layers, which have shown superior driving performance (see right side of Table 5.3). Converting our steering model into a recurrent network would be an interesting application for future inquiry.

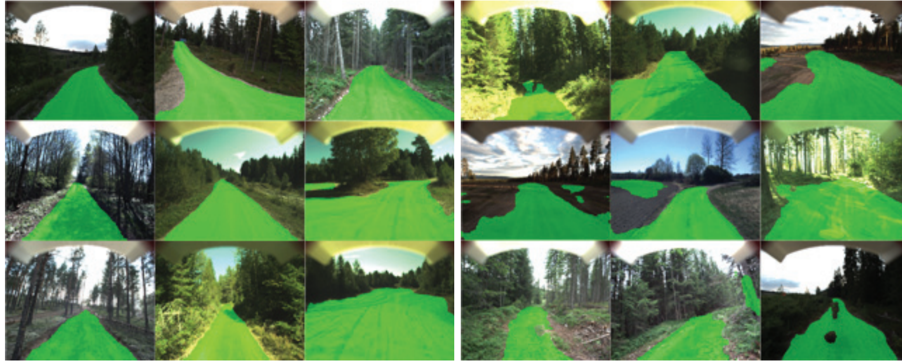
5.2 Segmentation network

5.2.1 Results

Table 5.4 shows the accuracy and intersection over union results on validation, using the evaluation metrics from Table 4.8. The numbers indicate that the segmentation network has, for the most part, learned to recognize pixels of road. Fig. 5.5 presents positive and negative segmentation results from the validation set. It is clear that for certain images, segmentation can be improved. The results are discussed in detail in the next section.

Table 5.4: Results on the validation set. Values are based on the metrics introduced in Sec. 4.3.4 (IoU = Intersection over Union, FW = Frequency Weighted).

Pixel Accuracy	**Mean Accuracy**	**Mean IoU**	**FW IoU**
0.966	0.936	0.917	0.906



(a) Positive results. (b) Negative results.

Figure 5.5: Some segmentation results from the validation set.

5.2.2 Discussion

The goal of the segmentation network is to provide accurate segmentations of the road environment to the path verification model. Since our verification model operates under the assumption that the images from segmentation are correctly segmented, it is important to ensure that output from this network is acceptable. Judging from the results presented in Table 5.4, it is clear that segmentation performance is well within tolerance. Despite the low error values, it is worth pointing out that higher errors are expected on examples with significant differences from those seen during training. Such examples may include images of snowy and icy roads, glaring sunlight, wet roads, etc. Nevertheless, it is not necessary to train the segmentation network on all types of road conditions in order to demonstrate path verification.

Although segmentation is generally satisfying (see Fig. 5.5a), there are instances where segmentation can be improved (see Fig. 5.5b). The segmentation results in these examples are mainly caused by poor road boundaries and variable road texture. For instance, some roads contain pixels of grass and stone, which are labeled as background in other images. These sort of inconsistencies can be considered a source of noise in the network, and can stall learning. A statistical modeling method known as Conditional

Random Fields (CRFs) have shown widespread success in improving accuracy for pixel labeling tasks [29, 32]. In particular, CRFs are appealing as they leverage context more than other methods when making predictions. This is useful in our case as neighboring pixels give a good indication of whether, e.g., grass or stone should be considered part of the road. Some approaches have utilized CRFs as a separate post-processing step to enhance segmentation detail from a CNN [6], while other approaches have approximated CRFs as recurrent neural networks (RNN), achieving top results [61]. Increasing the training set size to cover more examples where inconsistencies, such as grass and stone, are present, could also lead to improved segmentation detail. We leave these potential improvements as fruitful areas for future inquiry.

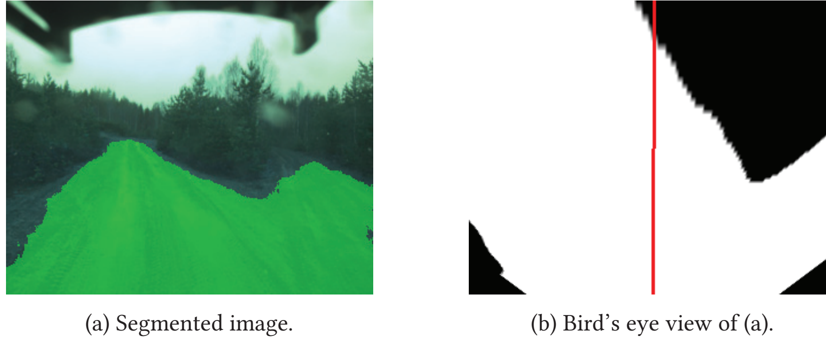
5.3 Path verification

5.3.1 Results

Table 5.5 shows the confusion matrix (see Sec. 4.4.3) of the results from our verification model on a test set containing 270 data instances. The matrix shows the yes/no predictions made from verification, and how many of these were *actually* correct. Fig. 5.6 illustrates the verification process and result on a single test instance. In particular, Fig. 5.6a shows the output from the segmentation network, and Fig. 5.6b shows the bird's eye perspective of this image along with the predicted steering angle.

Table 5.5: Confusion matrix (see Sec. 4.4.3) showing the output results from our verification model (see Fig. 4.14) compared to the actual class labels, as defined in Sec. 4.4.3. We use a test set containing only *true* data instances (270). In this case, TP=225, TN=45, FP=0 and TN=0.

		Predicted	
		Yes	No
Actual	Yes	225	45
	No	0	0



(a) Segmented image. (b) Bird's eye view of (a).

Figure 5.6: Verification process on a single instance from the test set. The drawn steering angle (shown in red) is outside the road boundary for the distance threshold, which is chosen as the image height, and therefore not accepted.

5.3.2 Discussion

The goal of path verification is to determine if online steering predictions from the steering model can be trusted or not. A perfect verification model can therefore be used as an evaluation metric for measuring steering performance online. This can be useful for detecting weaknesses in the model and can thereby identify which areas of the model are subject to improvement. Designing and implementing a verification model that is free of error is, however, outside the scope of this thesis. The goal here is to demonstrate a proof of concept, namely to show that our verification system is capable of reasonable decision making. Studying Table 5.5 closely, and applying the metrics in (4.5), (4.6) and (4.7), we get: $precision = 1.00$, $recall = 0.83$ and $F_1 = 0.91$. For safety reasons, the verification model should not produce any false positives and should therefore have a precision close to 100%. From the results above, it clear that this is indeed the case. However, since the dataset does not contain any false steering angles, it is difficult to state with certainty whether this precision value is accurate or not. Further experimentation was not possible due to limited test data and time. From Table 5.5 (and the recall value above), we see that the verification model rejects some steering angles which are actually appropriate. To understand why the model

makes these decisions, it is necessary to examine the ground truth steering angles in the test set. Inspection reveals that the majority of angles are close to zero, meaning that most of them will be represented as vertical line segments in the bird's eye view image (see Fig. 5.6b). A fundamental assumption of our verification model is that the vehicle is assumed to be in the road center for each video frame, meaning that the predicted steering angle is always drawn from the bottom center in bird's eye image. In reality, this is not always true and can explain why some of the actually valid steering predictions get rejected. For instance, assuming the vehicle position is slightly to the left of the road center in Fig. 5.6b, the drawn steering angle will be within the road boundary of the distance threshold.

Verification can potentially be improved by modifying some of the assumptions made in Sec. 4.4.2. In particular, the cameras on the test vehicle are not parallel and identical, and the vehicle is not always in the center of the road. The former can be handled by using a more general depth estimation algorithm, such as using triangulation based on different camera orientations. The issues related to the latter can be addressed by generating curved vehicle paths, instead of using straight line segments. This can be achieved by using either a simple bicycle model [28], or a more advanced Ackermann model [53]. In addition to experimenting with more test data, we leave such improvements for future work.

Chapter 6

Conclusion

In this thesis, end-to-end steering angle prediction using deep convolutional networks in an off-road setting was studied. Our experimental results show an RMSE of 0.19, close in value to what NVIDIA achieved [1], indicating that autonomous steering based on deep learning is possible off-road. In particular, we demonstrate that fusing camera and LiDAR information together leads to a smaller RMSE, than training on each data source separately. Intuitively, we expect images to contain more salient features for learning, than LiDAR. However, experimental results show that training on LiDAR and images individually yields similar RMSE, which suggests that LiDAR may perhaps be somewhat underestimated in the research community. To support this, we conduct a small experiment showing that our fusion model achieves better generalization when trained on an increased number of LiDAR features. Due to limitations in data, data quality and time, this result was not pursued further, but still represents an interesting topic for future inquiry.

Finally, we demonstrated a proof-of-concept verification model for steering. This model is based on projective geometry and utilizes segmented images from a separate segmentation network to determine steering trustability. Our experimental results

are promising on a small test set, yielding *precision*, *recall* and F_1 values of 1.00, 0.83 and 0.91, respectively. For future research we suggest further experimentation with more test data, in addition to using a more accurate depth estimation technique and utilizing curved steering paths instead of straight line segments.

Appendix A

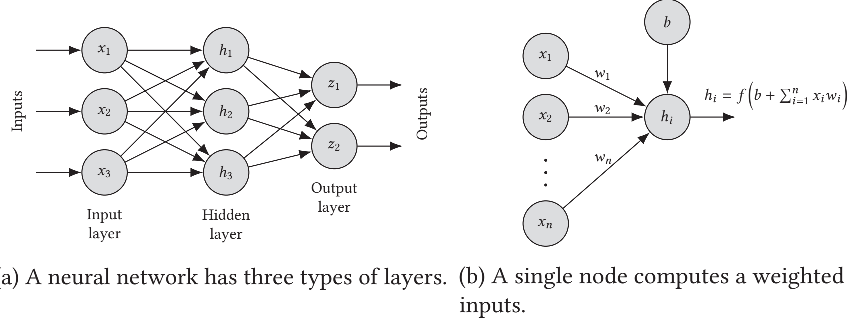
Neural network basics

The universal approximation theorem [24, 12] states that simple neural networks can be used to represent a variety of functions when given appropriate parameters. For instance, if we have labeled training data of the form $(\mathbf{x}, y)$ and assume there exists a function f^* such that $y = f^*(\mathbf{x})$, then by the universal approximation theorem there should exist a neural network $f_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$ that can approximate the target function f^*. Determining the optimal set of parameters that makes $f_{\boldsymbol{\theta}} \approx f^*$ is one of the main challenges of a neural network, a problem we attempt to address in this appendix. The goal of this appendix is not to give a comprehensive study of neural networks, but rather give a broad overview and discussion of the most essential components involved in learning $\boldsymbol{\theta}$.

A.1 The structure of neural networks

In this section, the general structure of a simple kind of neural network called feed-forward neural networks, also known as multilayer perceptrons (MLPs), is presented. These types of networks consists of multiple *layers*, where each layer is comprised of a vector of *nodes* (also called units). Fig. A.1a shows a simple feedforward neural network consisting of three layers. The nodes in each layer are connected to each node in the next layer by *edges*, represented by lines in the figure. A neural network contains three types of layers; input, hidden and output layers. The input layer maintains the input data that is to be processed by the network. This could for instance be a vector of features in an image. The top node would correspond to the first feature in the vector, the second node to the second feature etc. In general, the data in each node is passed along its edges to nodes in the next layer. Fig. A.1b shows how each node in the hidden layer processes data from the input layer. Each edge between connecting nodes has a quantity called a *weight* that is multiplied with the data on the edge. The sum of all contributions from the incoming edges constitutes the input of the node. The bias b is not included in Fig. A.1a, but is common in most feedforward networks.

The hidden layer performs the most crucial function—finding patterns in the data. For instance, if the objective is to detect cats and dogs in an image, the hidden layer(s) will find prominent features (shape, intensities, etc.) that are characteristic for each of the animals. Whenever a cat is present in an image, the node corresponding to a detected cat-feature will typically output a high value indicating the presence of a cat. It is worth pointing out that the task above would be handled better by a convolutional neural network (CNN) than a feedforward network (CNNs are discussed in Sec. 3.1). As Fig. A.1b illustrates, the input to a node is a weighted sum that is mapped to an output by a mapping f called an *activation* function. Each layer, execpt the input layer, has an activation function associated with it. There are several choices of activation functions, which will be discussed in the next section. The output layer outputs the result of the network, which can be a probability or real value depending on the task in question. For instance, if only a cat is present in an image the first node in the



(a) A neural network has three types of layers. (b) A single node computes a weighted sum of inputs.

Figure A.1: A simple feedforward neural network consisting of three layers.

output layer will have a high value (probability of cat) and the second node a low value (probability of dog). The network in Fig. A.1 is known as a shallow neural network since it only consist of a single hidden layer. Deep neural networks, on the other hand, are characterized by multiple hidden layers.

The structure presented above provides an easy way to visualize neural networks. An alternative way to represent these networks is as a series of matrix multiplications. Consider the network in Fig. A.1a which has three inputs $\{x_1, x_2, x_3\}$, three hidden outputs $\{h_1, h_2, h_3\}$ and two outputs $\{z_1, z_2\}$. If we merge these values into vectors $\mathbf{x} \in \mathbb{R}^3$, $\mathbf{h} \in \mathbb{R}^3$, $\mathbf{z} \in \mathbb{R}^2$ and denote each weight between node i and j in layer l by $w_{ij}^{(l)}$, the ouput of the network can be computed as follows

$$\begin{aligned} \mathbf{z} &= g(W^{(2)}\mathbf{h}) \\ \mathbf{h} &= f(W^{(1)}\mathbf{x}) \end{aligned} \tag{A.1}$$

where

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} & w_{33}^{(1)} \end{bmatrix}, \quad W^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \end{bmatrix} \tag{A.2}$$

and denotes all the weights in layers 1 and 2, respectively (the input layer is considered layer 0). The activation functions f and g are applied element-wise to the vectors. Different choices of activation functions are discussed next.

A.2 Activation functions

In the previous section we saw that each node maps input to output through a mapping called the activation function, and that each layer has an activation associated with it. It is common practice to consider separate classes of functions for hidden and output units. In Secs. A.2.1 and A.2.2 common activation functions for output units are presented, while Sec. A.2.3 presents the default choice for hidden units.

A.2.1 Sigmoid unit

We mentioned earlier that the objective of the hidden layers is to find a set of features that describe the data. The role of the output layer is to transform the set of features provided by the hidden layers to a number that can tell us how well the network has performed. If the task in question requires us to predict the value of a binary variable the number should be a probability. Such tasks could involve predicting whether or not a cat is present in an image, or the probability of a patient having a heart attack given his/hers medical record. For binary classification problems it is common to use a logistic sigmoid function, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{A.3}$$

The logistic sigmoid function is plotted in Fig. A.2 and maps x to the range $(0, 1)$.

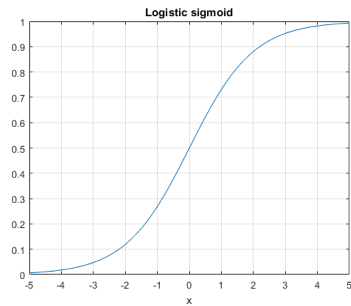


Figure A.2: Logistic sigmoid.

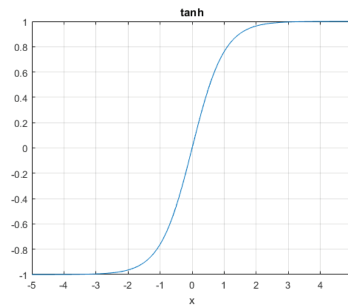


Figure A.3: Tanh.

The derivative of (A.3) is

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)), \tag{A.4}$$

which will come in hand later when we discuss the backpropagation algorithm in Sec. A.5.2. The hyperbolic tangent function is similar to the logistic sigmoid ("s"-shaped) but maps inputs to the range $(-1, 1)$, as illustrated in Fig. A.3.

A.2.2 Softmax unit

The softmax activation function can be used to represent a categorical distribution over K possible classes. It can be viewed as a generalization of the sigmoid, which was used to represent a probability distribution of a binary variable. The output is instead treated as a discrete variable representing the probability of each class K. For instance, the output layer from the cat and dog example in the previous section would typically employ a softmax function. The softmax function of a real value x_i is defined as

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \tag{A.5}$$

which is simply a normalized exponential function. The function transforms a vector $\mathbf{x} \in \mathbb{R}^n$ of arbitrary real values to a vector $\mathrm{softmax}(\mathbf{x}) \in \mathbb{R}^n$ of real values in the range (0,1) that all add up to 1.

A.2.3 Rectified linear units

In general, there are no guided principles for the choice of hidden unit activations [19]. While there are many possible choices, including the aforementioned, predicting in advance which will work better is challenging. Rectified linear units (ReLUs) have proven sucessful in practice, which have made them the default choice in many neural networks [19]. The rectified linear unit uses the activation function

$$g(x) = \max\{0, x\}. \tag{A.6}$$

It is similar to a linear unit except that it outputs zero in half of its domain (see Fig. A.4). This makes the gradient non-zero whenever the unit is active and zero whenever the unit is inactive. As we will see in Sec. A.5, this property is useful when training neural networks. Notice, however, that (A.6) is not differentiable at $x = 0$. Most software implementations overcome this issue by returning one of the one-sided derivatives instead of raising an error. In cases where it is desirable for training to continue on

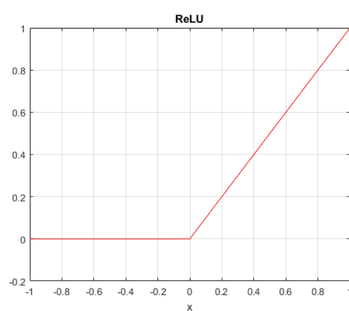


Figure A.4: Rectified linear unit.

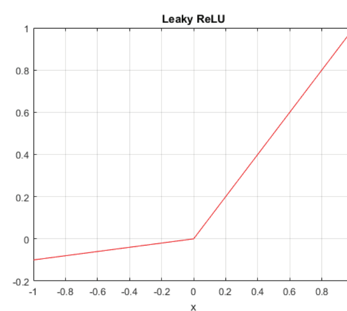


Figure A.5: Leaky rectified linear unit.

inactive units, a leaky ReLU function may be used instead. The Leaky ReLU [37] employs a small gradient α whenever $x < 0$ and is given by

$$g(x) = \begin{cases} x, & x > 0 \\ \alpha x, & \text{otherwise.} \end{cases} \tag{A.7}$$

Fig. A.5 shows a plot of the Leaky ReLU function for $\alpha = 0.1$.

A.3 Maximum likelihood estimation

In this section we introduce the principle of maximum likelihood which will be used to motivate the various loss functions presented in the Sec. A.4. Maximum likelihood estimation (MLE) finds the parameters of a statistical model that maximize the likelihood of a given set of observations. Consider a set of i.i.d.[1] observations $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$ sampled from an unknown data distribution $p_{\text{data}}(\mathbf{x})$. Suppose that $p_{\text{data}}(\mathbf{x})$ belongs to a parametric family of distributions denoted by $p_{\text{model}}(\mathbf{x}|\boldsymbol{\theta})$, with $\boldsymbol{\theta}$ being a vector of parameters for this family. Then the ML estimate for $\boldsymbol{\theta}$ can be defined as

$$\hat{\boldsymbol{\theta}}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p_{\text{model}}(\mathbf{x}^{(i)}|\boldsymbol{\theta}). \tag{A.8}$$

This product is, however, prone to numerical underflow[2]. To overcome this problem, we utilize the fact that the solution to (A.8) is the same as the solution to

$$\hat{\boldsymbol{\theta}}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \log \sum_{i=1}^{N} p_{\text{model}}(\mathbf{x}^{(i)}|\boldsymbol{\theta}). \tag{A.9}$$

The logarithm is an increasing function and consequently doesn't alter the solution. We can interpret the ML estimate in (A.9) as the parameter estimate that minimizes

[1] independent and identically distributed.
[2] Numbers near zero are rounded to zero.

the dissimilarity between the *empirical* data distribution $\hat{p}_{\text{data}}(\mathbf{x})$, defined by the set of observations, and the model distribution.

A.4 Loss functions

It was mentioned earlier that the output layer returns the output or result of the network. How do we know if the result is any good? This question can be answered by studying the loss (also called cost) function of the network. The loss function measures the performance of the network by comparing the output to the *desired* value. The desired value $\mathbf{y}^{(i)}$ is provided together with the input data $\mathbf{x}^{(i)}$ as a pair $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$. The set $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$ is known as labeled training data and is the basis of supervised learning. The shape of $\mathbf{y}$ depends on the task in question. For scalar outputs such as binary classification then $y \in \{0, 1\}$, while for linear regression, $y \in \mathbb{R}$. The importance of the loss function will be highlighted in Sec. A.5, where it will play a crucial role in training neural networks. In this section, some common loss metrics used to measure performance are introduced. As we shall see, the functions are not arbitrary, but derived based on the principle of maximum likelihood.

A.4.1 Mean squared error (MSE)

The goal of a neural network is to approximate a target function $f^*(\mathbf{x})$ by a function $f_{\boldsymbol{\theta}}(\mathbf{x})$, where $\boldsymbol{\theta}$ denotes the parameters (weights and biases) of the network. However, due to noise in the data, the relationship between $\mathbf{x}$ and $\mathbf{y}$ is not necessarily deterministic (the same $\mathbf{x}$ could produce different $\mathbf{y}$). For this reason, it becomes more natural for the neural network to approximate a target distribution $p(\mathbf{y}|\mathbf{x})$ rather than a deterministic function. To do this we need to make an assumption about what kind of probability distribution $p(\mathbf{y}|\mathbf{x})$ is. Once that assumption is made, MLE can be used to find the parameters of the model.

Consider a set of labeled training examples $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$

where $y^{(i)} \in \mathbb{R}$. Further let $\hat{y}_{\boldsymbol{\theta}}^{(i)}$ be the the scalar output of a neural network with parameters $\boldsymbol{\theta}$ for a training example $(\mathbf{x}^{(i)}, y^{(i)})$. Assuming a normal distribution, $p(y|\mathbf{x}) = \mathcal{N}_y(\hat{y}_{\boldsymbol{\theta}}(\mathbf{x}), \sigma^2)$, where $\hat{y}_{\boldsymbol{\theta}}$ and σ^2 is the mean and variance, respectively, we can use (A.9) to compute the parameter estimates. The conditional log-likelihood is given by

$$\log \sum_{i=1}^{N} p_{\boldsymbol{\theta}}(y^{(i)}|\mathbf{x}^{(i)}) = -N \log \sigma - \frac{N}{2} \log(2\pi) - \sum_{i=1}^{N} \frac{(y^{(i)} - \hat{y}_{\boldsymbol{\theta}}^{(i)})^2}{2\sigma^2}, \tag{A.10}$$

where the right hand side was obtained by taking the natural logarithm of the probability density function of $\mathcal{N}_y(\hat{y}_{\boldsymbol{\theta}}(\mathbf{x}), \sigma^2)$. Comparing (A.10) with the mean squared error,

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)})^2, \tag{A.11}$$

we see that maximizing (A.10) will produce the same parameter estimate as minimizing (A.11) w.r.t $\boldsymbol{\theta}$. This result justifies the use of MSE as a loss function for real valued data. If $\mathbf{y} \in \mathbb{R}^n$ we would have to use a multivariate Gaussian distribution instead, but the result would nonetheless be the same.

A.4.2 Binary cross-entropy

We consider the case where y is a binary variable $y \in \{0, 1\}$. In such cases it is natural to model the target distribution as a Bernoulli distribution, namely $p(y|\mathbf{x}) = \text{Bern}_y[\hat{y}_{\boldsymbol{\theta}}(\mathbf{x})]$. The probability density function of a Bernoulli distribution with data point $(\mathbf{x}^{(i)}, y^{(i)})$ is

$$p_{\boldsymbol{\theta}}(y^{(i)}|\mathbf{x}^{(i)}) = [\hat{y}_{\boldsymbol{\theta}}^{(i)}(\mathbf{x}^{(i)})]^{y^{(i)}} (1 - \hat{y}_{\boldsymbol{\theta}}^{(i)}(\mathbf{x}^{(i)}))^{1-y^{(i)}}, \tag{A.12}$$

where $\hat{y}_{\boldsymbol{\theta}}$ is the output of the neural network with parameters $\boldsymbol{\theta}$. For notational simplicity we omit the dependence on $\mathbf{x}$ in the output. Assuming the training examples are i.i.d., we can use (A.9) to compute the parameter estimates. The conditional log-

likelihood is given by

$$\log \sum_{i=1}^{N} p_{\theta}(y^{(i)}|\mathbf{x}^{(i)}) = \sum_{i=1}^{N} y^{(i)} \log \hat{y}_{\theta}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}_{\theta}^{(i)}). \tag{A.13}$$

Comparing (A.13) with the binary cross entropy,

$$H_{\theta}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} y^{(i)} \log \hat{y}_{\theta}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}_{\theta}^{(i)}), \tag{A.14}$$

we see that maximizing (A.13) will produce the same parameter estimate as minimizing (A.14) w.r.t $\boldsymbol{\theta}$. This justifies the use of binary cross entropy as a loss function for a binary variable y.

A.4.3 Categorical cross-entropy

In the case of multiclass classification problems, the input $\mathbf{x}$ can be assigned to one of K mutually exclusive classes. The label y is therefore a discrete variable such that $y \in \{1, 2, \ldots, K\}$. The categorical distribution generalizes the Bernoulli distribution and is natural to consider in this case. The target distribution can be written as $p(y|\mathbf{x}) = \mathrm{Cat}_y[\hat{y}_{\theta 1}(\mathbf{x}), \hat{y}_{\theta 2}(\mathbf{x}), \ldots, \hat{y}_{\theta K}(\mathbf{x})]$, where $\hat{y}_{\theta j}$ represents the value in the j-th output node of the network. The probability density function is given by

$$p_{\theta}(y_1^{(i)}, y_2^{(i)}, \ldots, y_K^{(i)}|\mathbf{x}^{(i)}) = \prod_{j=1}^{K} (\hat{y}_{\theta j}^{(i)})^{y_j^{(i)}}, \tag{A.15}$$

where $y_j^{(i)} \in \{0, 1\}$ and $\sum_j^K \hat{y}_{\theta j}^{(i)} = 1$. We have for notational simplicity omitted writing the dependence of the input data $\mathbf{x}$. Assuming the training examples are i.i.d., we can use (A.9) to compute the parameter estimates. The conditional log-likelihood is given by

$$\log \sum_{i=1}^{N} p_{\theta}(y_1^{(i)}, \ldots, y_K^{(i)}|\mathbf{x}^{(i)}) = \sum_{i=1}^{N} \sum_{j=1}^{K} y_j^{(i)} \log \hat{y}_{\theta j}^{(i)}. \tag{A.16}$$

Comparing (A.16) with the categorical cross entropy,

$$L_{\boldsymbol{\theta}}(y_1, \ldots, y_K, \hat{y}) = -\sum_{i=1}^{N}\sum_{j=1}^{K} y_j^{(i)} \log \hat{y}_{\boldsymbol{\theta} j}^{(i)}, \tag{A.17}$$

we see that maximizing (A.16) will produce the same parameter estimate as minimizing (A.17) w.r.t $\boldsymbol{\theta}$. This justifies the use of categorical cross entropy as a loss function for a discrete variable y.

A.5 Training a neural network

The loss functions presented in the previous section provide information about the distance between the output of the network and the desired value. The next step is to figure out how this distance can be minimized by increasing/decreasing various weights and biases in the network. We call the procedure of minimizing a loss function by adjusting the parameters of the network for *training*. Training a neural network is in general a nonconvex optimization problem meaning we cannot expect a solution of parameters that globally minimizes the loss. In this section, we shall introduce an important algorithm for training called stochastic gradient descent (SGD), which is an extension of gradient descent. As we shall see, the gradient in this algorithm is computed by another important algorithm called backpropagation. Finally, a popular technique for speeding up the optimization is considered.

A.5.1 Gradient based optimization

Gradient descent is a simple optimization algorithm which can be used to train neural networks. It iteratively updates the parameters in the network by following a small step in the direction of the negative gradient of the loss function. The update rule is given by

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_n), \tag{A.18}$$

where $\eta > 0$ denotes the *learning rate* and J represents the function that is to be minimized. The loss functions discussed in Sec. A.4 have a common property; they all decompose as a sum over training examples[3]. If we express the quantities inside the summation as a per-example loss function, $L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta})$, then the overall loss can be written as

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}). \tag{A.19}$$

Taking the gradient of (A.19) at a fixed point, $\boldsymbol{\theta}_n$, we get

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_n) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}_n). \tag{A.20}$$

A significant drawback of (A.20) is that for each weight update in (A.18) the entire training set must be used to compute the gradient. For training sets comprising billions of examples, this computation can be quite expensive. The stochastic gradient descent (SGD) algorithm overcomes this problem by using a small set of training examples in each iteration. We call this set a *minibatch* and the size is typically chosen to be a relatively small number ranging from one to a few hundred [19]. The estimate of the gradient in (A.20) with batch size m is formed as

$$\nabla_{\boldsymbol{\theta}} \hat{J}(\boldsymbol{\theta}_n) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}), \tag{A.21}$$

and replaces (A.20) in (A.18). The gradient estimate in (A.21) is unbiased if the training examples are drawn i.i.d. from the data-generating distribution. In practice, however, training a neural network usually proceeds over several passes through the training set (we call these passes *epochs*) meaning that the same examples are reused. Only in the first epoch will the gradient be unbiased, while the other epochs will have an element of bias. At each epoch it is common to shuffle the training examples such that each minibatch of examples is random. Despite some of the gradients being biased, SGD is still capable of finding a very low value of the cost function.

[3] In fact, all loss functions derived from the principle of maximum likelihood have this property.

A.5.2 Back-propagation algorithm

In the previous section we looked at an efficient algorithm for training a neural network. The stochastic gradient descent optimization method finds the average gradient over all the examples in the minibatch and uses this to update the parameters of the network. Notice, however, that it was never mentioned how the gradient of the per-example loss function is computed. Finding an analytical expression for the gradient is a straightforward task for many software packages, but numerically evaluating the derivatives is usually computationally expensive. The back-propagation algorithm [45] provides a computationally efficient way of computing the gradient of the loss. The equations of the back-propagation algorithm in component form:

$$\frac{\partial L}{\partial w_{ij}^{l}} = \delta_j^l a_i^{l-1} \tag{A.22}$$

$$\frac{\partial L}{\partial b_j^{l}} = \delta_j^l \tag{A.23}$$

$$\delta_j^L = \frac{\partial L}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \tag{A.24}$$

$$\delta_j^l = \sum_i w_{ij}^{l+1}\delta_i^{l+1}\sigma'(z_i^l) \tag{A.25}$$

$$a_j^l = \sigma(z_j^l), \quad z_j^l = \sum_i w_{ij}^l a_i^{l-1} + b_i^l \tag{A.26}$$

The quantities are defined as follows: L is the per-example loss function, w_{ij}^l denotes the weights between node i and j in layer l, $\delta_j^l = \frac{\partial L}{\partial z_j^l}$ and is called the *error*, a_j^l denotes the output of unit j in layer l, a_j^L is the output of the network (the same as $\hat{y}$), z_j^l represents the weighted sum of inputs to node j in layer l, σ is an arbitrary activation function and b_j^l is a bias term. The back-propagation procedure is summarized in Algorithm 1.

For each training example information is propagated forward in the network, which allows us to obtain numerical values for the input and output of each unit. The error

Algorithm 1 Back-propagation algorithm

1: Apply the input vector $\mathbf{x}^{(i)}$ to the network and forward propagate through the network by computing (A.26) for each $l = 2, 3, \ldots, L$.
2: Evaluate (A.24) for all j output units.
3: For each $l = L-1, L-2, \ldots, 2$ compute (A.25) for each hidden unit j in l.
4: Use (A.22) and (A.23) to evaluate the required derivatives.

δ_j^L is evaluated for each output unit j in the output layer L (not to be confused with the loss function). The errors δ_j^l for each hidden unit is back-propagated until (A.22) or (A.23) (depending on which is desired) can be computed.

A.5.3 Batch normalization

Training a neural network is complicated by the fact that the inputs to a layer are affected by the parameters of all the preceding layers. For instance, any small change in one weight will impact the inputs to the subsequent layers. Formally, we say that the distribution of each layer's input is changed. This forces higher layers to adapt to the drift in inputs, thus slowing down training. We define the *Internal Covariate Shift* as the change in distribution of a network inputs/activations due to a change

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure A.6: Batch normalization algorithm. Image taken from [25].

in network parameters during training. Batch normalization (BN) [25] provides a way of reducing the internal covariate shift thus speeding up training. The method consists of two stages. The first stage involves transforming the inputs to have zero mean and unit variance, also called whitening. The second stage applies scaling and shifting to allow the normalized inputs to be reversed back to their original form when needed. Fig. A.6 presents the batch normalization algorithm. For a particular activation x, the mini-batch B comprises a set of m values of this activation. The mean and variance of this mini-batch is calculated and applied to each activation in the mini-batch resulting in a normalized value. The transformation with learned parameters γ and β allow the normalized inputs to be scaled and shifted whenever needed. For instance, $\gamma = \sqrt{\sigma_B^2 + \epsilon}$ and $\beta = \mu_B$ recovers the original activations. The ϵ is a constant and provides numerical stability to the variance.

References

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. URL http://arxiv.org/abs/1604.07316.

[2] 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. 04 2017.

[3] Luca Caltagirone, Samuel Scheidegger, Lennart Svensson, and Mattias Wahde. Fast lidar-based road detection using fully convolutional neural networks. *CoRR*, abs/1703.03613, 2017. URL http://arxiv.org/abs/1703.03613.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730, Dec 2015. doi: 10.1109/ICCV.2015.312.

[5] L. Chen, Q. Li, M. Li, and Q. Mao. Traffic sign detection and recognition for intelligent vehicle. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 908–913, June 2011. doi: 10.1109/IVS.2011.5940543.

[6] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic image segmentation with deep convolutional nets and

fully connected crfs. *CoRR*, abs/1412.7062, 2014. URL http://arxiv.org/abs/1412.7062.

[7] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. *CoRR*, abs/1611.07759, 2016. URL http://arxiv.org/abs/1611.07759.

[8] Yiping Chen, Jingkang Wang, Jonathan Li, Cewu Lu, Zhipeng Luo, Han Xue, and Cheng Wang. Lidar-video driving dataset: Learning driving policies effectively. 2018. URL http://dspace.xmu.edu.cn/handle/2288/161007.

[9] Guilhem Chéron, Ivan Laptev, and Cordelia Schmid. P-CNN: pose-based CNN features for action recognition. *CoRR*, abs/1506.03607, 2015. URL http://arxiv.org/abs/1506.03607.

[10] Lu Chi and Yadong Mu. Deep steering: Learning end-to-end driving model from spatial and temporal visual cues. *CoRR*, abs/1708.03798, 2017. URL http://arxiv.org/abs/1708.03798.

[11] Dan Ciresan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2843–2851. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4741-deep-neural-networks-segment-neuronal-membranes-in-electron-microscopy-images.pdf.

[12] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL https://doi.org/10.1007/BF02551274.

[13] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009. doi: 10.1109/CVPR.2009.5206848.

[14] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. *CoRR*, abs/1411.4389, 2014. URL http://arxiv.org/abs/1411.4389.

[15] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1915–1929, Aug 2013. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.231.

[16] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Convolutional two-stream network fusion for video action recognition. *CoRR*, abs/1604.06573, 2016. URL http://arxiv.org/abs/1604.06573.

[17] Li-Chen Fu and Cheng-Yi Liu. Computer vision based object detection and recognition for vehicle driving. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 3, pages 2634–2641 vol.3, 2001. doi: 10.1109/ROBOT.2001.933020.

[18] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2016.

[20] Saurabh Gupta, Ross B. Girshick, Pablo Arbelaez, and Jitendra Malik. Learning rich features from RGB-D images for object detection and segmentation. *CoRR*, abs/1407.5736, 2014. URL http://arxiv.org/abs/1407.5736.

[21] Bharath Hariharan, Pablo Arbelaez, Ross B. Girshick, and Jitendra Malik. Simultaneous detection and segmentation. *CoRR*, abs/1407.1808, 2014. URL http://arxiv.org/abs/1407.1808.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. 9:1735–80, 12 1997.

[24] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551 – 560, 1990. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(90)90005-6. URL http://www.sciencedirect.com/science/article/pii/0893608090900056.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[26] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, June 2014. doi: 10.1109/CVPR.2014.223.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[28] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1094–1099, June 2015. doi: 10.1109/IVS.2015.7225830.

[29] Philipp Krähenbühl and Vladlen Koltun. Efficient inference in fully connected crfs with gaussian edge potentials. *CoRR*, abs/1210.5644, 2012. URL http://arxiv.org/abs/1210.5644.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

[31] T L I Sugata and C K Yang. Leaf app: Leaf recognition with deep convolutional neural networks. 273:012004, 11 2017.

[32] L'ubor Ladicky, Chris Russell, Pushmeet Kohli, and Philip Torr. Associative hierarchical crfs for object class image segmentation, 11 2009.

[33] Yann Lecun. *Generalization and network design strategies*. Elsevier, 1989.

[34] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[35] Bo Li, Tianlei Zhang, and Tian Xia. Vehicle detection from 3d lidar using fully convolutional network. *CoRR*, abs/1608.07916, 2016. URL http://arxiv.org/abs/1608.07916.

[36] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014. URL http://arxiv.org/abs/1411.4038.

[37] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. 2013.

[38] Guo Mu, Zhang Xinyu, Li Deyi, Zhang Tianlei, and An Lifeng. Traffic light detection and recognition for autonomous vehicles. *The Journal of China Universities of Posts and Telecommunications*, 22(1):50 – 56, 2015. ISSN 1005-8885. doi: https://doi.org/10.1016/S1005-8885(15)60624-0. URL http://www.sciencedirect.com/science/article/pii/S1005888515606240.

[39] Joe Yue-Hei Ng, Matthew J. Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. *CoRR*, abs/1503.08909, 2015. URL http://arxiv.org/abs/1503.08909.

[40] Øyvind Kjeldstad Grimnes. End-to-end steering angle prediction and object detection using convolutional neural networks. Master's thesis, Norwegian University Of Science and Technology, Høgskoleringen 1, 2017.

[41] Pedro H. O. Pinheiro and Ronan Collobert. Recurrent convolutional neural networks for scene parsing. *CoRR*, abs/1306.2795, 2013. URL http://arxiv.org/abs/1306.2795.

[42] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 305–313. Morgan-Kaufmann, 1989. URL http://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf.

[43] Dean A. Pomerleau. *Neural Network Perception for Mobile Robot Guidance*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393732.

[44] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016. URL http://arxiv.org/abs/1612.00593.

[45] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6. URL http://dl.acm.org/citation.cfm?id=65669.104451.

[46] Florian Schroff, Antonio Criminisi, and Andrew Zisserman. Object class segmentation using random forests. In *Proc. British Machine Vision Conference (BMVC)*, January 2008. URL https://www.microsoft.com/en-us/research/publication/object-class-segmentation-using-random-forests/.

[47] J. Shotton, M. Johnson, and R. Cipolla. Semantic texton forests for image categorization and segmentation. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008. doi: 10.1109/CVPR.2008.4587503.

[48] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from single depth images. In *CVPR 2011*, pages 1297–1304, June 2011. doi: 10.1109/CVPR.2011.5995316.

[49] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

[51] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. URL http://arxiv.org/abs/1602.07261.

[52] S. Ullman. Against direct perception. *Behavioral and Brain Sciences*, 3(3):373–381, 1980. doi: 10.1017/S0140525X0000546X.

[53] A. J. Weinstein and K. L. Moore. Pose estimation of ackerman steering vehicles for outdoors autonomous navigation. In *2010 IEEE International Conference on Industrial Technology*, pages 579–584, March 2010. doi: 10.1109/ICIT.2010.5472738.

[54] Philippe Weinzaepfel, Zaïd Harchaoui, and Cordelia Schmid. Learning to track for spatio-temporal action localization. *CoRR*, abs/1506.01929, 2015. URL http://arxiv.org/abs/1506.01929.

[55] Bichen Wu, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. Squeezeseg: Convolutional neural nets with recurrent CRF for real-time road-object segmentation from 3d lidar point cloud. *CoRR*, abs/1710.07368, 2017. URL http://arxiv.org/abs/1710.07368.

[56] Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrell. End-to-end learning of driving models from large-scale video datasets. *CoRR*, abs/1612.01079, 2016. URL http://arxiv.org/abs/1612.01079.

[57] Bisheng Yang, Zheng Wei, Qingquan Li, and Jonathan Li. Automated extraction of street-scene objects from mobile lidar point clouds. *International Journal of*

Remote Sensing, 33(18):5839–5861, 2012. doi: 10.1080/01431161.2012.674229. URL https://doi.org/10.1080/01431161.2012.674229.

[58] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014. URL http://arxiv.org/abs/1411.1792.

[59] Mohammed Yousefhussien, David J. Kelbe, Emmett J. Ientilucci, and Carl Salvaggio. A fully convolutional network for semantic labeling of 3d point clouds. *CoRR*, abs/1710.01408, 2017. URL http://arxiv.org/abs/1710.01408.

[60] K. Zheng, G. Sun, Q. Fan, X. Wang, Q. Zhang, and L. Jia. A lane recognition system based on priority. In *2013 IEEE Global High Tech Congress on Electronics*, pages 183–186, Nov 2013. doi: 10.1109/GHTCE.2013.6767269.

[61] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip H. S. Torr. Conditional random fields as recurrent neural networks. *CoRR*, abs/1502.03240, 2015. URL http://arxiv.org/abs/1502.03240.

[62] Y. T. Zhou and R. Chellappa. Computation of optical flow using a neural network. In *IEEE 1988 International Conference on Neural Networks*, pages 71–78 vol.2, July 1988. doi: 10.1109/ICNN.1988.23914.