Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Audun Wigum Arbo
Even Dalen

Domain-Independent Perception for Autonomous Driving

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2020

Audun Wigum Arbo
Even Dalen

Domain-Independent Perception for Autonomous Driving

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The idea of vehicles driving autonomously has been an ongoing topic of research ever since the inception of artificial intelligence. With the big resurgence of neural networks in the 2010s, new approaches using end-to-end imitation learning began to surface. In contrast to classical autonomous driving methods, the end-to-end neural network learns to drive by imitating an expert driver, learning from the recorded driving data alone.

In this thesis, we build on several promising results from the end-to-end autonomous driving research field, and implement a neural network for driving both in simulation and in the real world. Other authors often use raw RGB images to give their driving system an understanding of the surrounding environment, but that approach does not support transfer of the system to the real world. We investigate techniques for being able to train a model in a simulator, and then deploy it to the real world, in some cases without any additional fine-tuning. We approach this problem of domain transfer by creating a two-part architecture: a perception model, which creates generalized semantic segmentation and depth maps; and a driving model, which learns to drive from the more abstracted outputs of the perception model.

Our first experiment explores different design choices for the perception model, finding that a MobileNet + SegNet model gives the best predictions relative to computational costs. Furthermore, we find that multi-task learning with segmentation and depth predictions further improves the perception model. In the second experiment, we test multiple perception models in a simulated driving environment, to learn how different perception models affect real driving. We evaluate seven models in an autonomous driving simulator, and find that predicting both segmentation and depth leads to the best driving performance.

Finally, we deploy our model to a real-world environment — using a small four-wheeled vehicle in a closed track mimicking real roads. We show that our perception model abstracts away the visual differences between a simulator and the real world, and allows for a driving model only trained in simulation to perform simple driving in real-life as well.

Sammendrag

Idéen om at et kjørtetøy kan kjøre helt autonomt har vært et aktivt forskningsspørsmål i mange år — og har vært forsket på helt siden man begynte å studere kunstig intelligens. Med nevrale nettverks økende popularitet på 2010-tallet, ble det samtidig publisert mye nytt og spennnende om bruk av ende-til-ende-nettverk for imitasjonslæring. I motsetning til klassiske metoder for autonom kjøring, lærer ende-til-ende-nettverkene å kjøre ved å imitere en ekspert-sjåfør; dette gjøres ved å kun se på opptak av kjøredata.

I denne masteroppgaven bygger vi på flere lovende resultater innen ende-til-ende-kjøring, og vi implementerer et nevralt nettverk som kan kjøre både i simulator og i virkeligheten. I relatert arbeid har det ofte blitt brukt rene RGB-bilder for å gi kjøresystemet sitt en forståelse av omgivelsene, men denne framgangsmåten støtter ikke overføring av kjøresystemet fra simulator til virkelighet, eller omvendt. Vi utforsker teknikker som kan åpne opp for å trene en modell i simulator for så å bruke den på ekte kjøretøy, i gitte tilfeller uten å trenge noen finjusteringer i det hele tatt. Vi prøver å løse dette domene-overføringsproblemet ved å lage en todelt arkitektur. Den første delen er en persepsjonsmodell som lager generaliserte segmenterings- og dybdeprediksjoner, og den andre er en kjøremodell som lærer å kjøre fra det mer abstrakte resultatet fra persepsjonsmodellen.

I vårt første eksperiment prøver vi ut forskjellige designvalg for persepsjonsmodellen, og vi finner ut at en MobileNet + SegNet-modell gir den beste kombinasjonen av høy treffsikkerhet og lite komputasjonelt arbeid. Videre konkluderer vi med at en persepsjonsmodell som både lærer segmenterings- og dybdeprediksjon yter best. I vårt andre eksperiment tester vi ulike persepsjonsmodeller ved hjelp av vår kjøremodell, dette for å forstå hvordan forskjellige persepsjonsmodeller påvirker reell kjøring. Vi evaluerer syv kjøremodeller i en simulator for autonom kjøring, og konkluderer med at de som tar nytte av både segmenterings- og dybdedata kjører best.

Til slutt prøver vi ut vår kjøremodell i virkeligheten, der vi kjører et lite firehjulet kjøretøy i en lukket veibane. Vi viser at vår persepsjonsmodell abstraherer bort de visuelle forskjellene mellom en simulator og den virkelige verden, og gjør det mulig for en kjøremodell trent kun i simulering til å utføre enkel kjøring i virkeligheten.

Preface

This Master's thesis in Computer Science is a part of the research conducted within the NTNU Autonomous Perception Laboratory (NAPLab)[1] research group at the Norwegian University of Science and Technology (NTNU).

We would like to thank our supervisor Frank Lindseth for giving access to the required equipment and resources needed to complete the project, as well as his invaluable feedback during the exploration, experimentation, and writing phases of this thesis.

Audun Wigum Arbo, Even Dalen

Trondheim, June 14, 2020

[1] `https://www.ntnu.edu/web/ntnu-autonomous-perception/naplab`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the thesis. This includes the background and motivation for conducting this project, the overall goals and research questions, our contributions, and finally an outline of the thesis.

1.1 Background and motivation

In the recent years, autonomous vehicles have become an increasingly popular research domain. Improvements in sensor technology, machine learning and other relevant fields have made fully autonomous vehicles probable in the near future. The traditional approach to autonomous vehicles has been the modular approach, where the driving task is divided into several sub-tasks, such as perception, localization and planning. As each module in the vehicle will have to be fine-tuned individually, and handle a vast amount of driving situations, the scalability of the modular approach has been questioned.

A competing approach to modularity is the end-to-end approach, where a single module is responsible for the entire driving task. The module starts from a set of input sensors such as radar, camera and GPS and directly outputs driving commands, similar to how humans drive. End-to-end driving requires large amounts of data in order to handle most scenarios, and simulation has therefore been explored to reduce the amount of real-world data required.

A simulated driving environment such as CARLA [10] can be used to gather data in a variety of weather- and light conditions, as well as validate end-to-end models in a reproducible way. However, as simulated environments differ significantly from real-world, the training done in simulation does not necessarily transfer to real-world environments. Therefore, a domain-independent perception and motor control representation would be useful to take advantage of simulated environments, both for data generation and model validation.

During this master's thesis, we will explore ways to allow end-to-end driving models to be transferred from simulation to the real world. The main focus will be on abstracting away perception using higher level image representations such as semantic segmentation and depth estimation. To reduce the need to gather real-world data, we will use public datasets to generate models that work in real-world environments. We will further use a combination of simulation and self-collected real-world data to fine-tune the models and optimize for our environments. The models will be tested and validated both in simulation and the real world, in order to verify that the results we achieve in simulation are representative for the real world as well.

In order to validate that our models transfer adequately to the real-world domain, we will use a downsized vehicle, called SPURV Research. The vehicle uses a single forward facing camera and accepts desired speed and turning angle as controls. It will be driven on a representative test road including lane lines, intersections and more.

1.2 Goals and research questions

The overall goal of this thesis is to make an autonomous driving system that utilizes both simulation and real-world data to drive; both to reduce the amount of real-world data needed, and to be able to learn scenarios that are expensive, dangerous, or rare in real-world driving. The models created should be evaluated both in simulation and in the real world, to validate that the simulation results are representative for real-world performance.

Subgoal 1 Develop a driving model able to generalize the perception task from a simulated to a real-world domain using only public image datasets and simulation data.

Subgoal 2 Validate driving results from simulation in a real-world environment.

To help guide our research, we formulate a set of research questions (RQs) related to our goals. These questions will be revisited and addressed in the discussion, specifically in Section 5.3.

RQ 1 Can an end-to-end model learn to drive from the segmentation and optionally depth maps of a pre-trained perception model?

RQ 2 Can a perception model trained on real-world public datasets generalize to simulated environments?

RQ 3 Can an end-to-end driving model trained exclusively on simulated driving data successfully drive in the real world?

1.3 Contributions

This thesis' main contribution is a suggested architecture for improving the domain-independence of end-to-end driving models that uses a separate perception model. We show that dividing end-to-end driving into a perception task and driving task opens up for utilizing public image datasets; reducing the amount of real-world driving data required when training an end-to-end driving model. Our work within domain-independent simulated driving is presented in a paper currently under review for the Colour and Visual Computing Symposium 2020 (CVCS 2020)[1], which can be read in full in Appendix A.

Summarized, these are our main contributions:

1. **A perception model for extracting a domain-independent scene understanding.** The model combines segmentation and depth prediction as a multi-task learning model.
2. **Evaluation of multiple perception models in simulation.** We both quantitatively and qualitatively evaluate three approaches for a perception model: using no perception model, prediction of segmentation, and prediction of both segmentation and depth. Moreover, we determine how different types of segmentation and depth training data affects the performance of both perception and driving.

[1]`https://www.cvcs.no/`

3. **Demonstrating the real-world viability of an autonomous driving system trained in simulation.** We demonstrate that our models trained in simulation can be transferred to the real world.

1.4 Report structure

This section provides a brief overview of the structure of this thesis.

Chapter 1: Introduction Introduces the work done — its goals, research questions, and contributions.

Chapter 2: Background and Related Work Covers the background knowledge relevant for the work done in this thesis, introduces relevant hardware and software, and presents related work.

Chapter 3: Methodology Covers the various methods used, including data collection and processing, machine learning architectures, and our SPURV pipeline.

Chapter 4: Experiments and Results Contains the three main experiments conducted during this project. We show setup, results, and related discussion for each of the experiments.

Chapter 5: Discussion Discusses the work done in the master's project, including experimental results, consistency to related work, research questions and some reflection from our point of view.

Chapter 6: Conclusion and Future Work Presents a summary of the thesis' contributions, which findings are of most interest, and how these findings can be used in future work.

Chapter 2

Background and Related Work

This chapter introduces the background for our thesis. First, we explain the relevant theory from the fields of neural networks, computer vision, and autonomous driving. Later, we go through the technology used to implement and test our systems. The remainder of the chapter is focused on providing a brief history of end-to-end driving, starting from the earliest approaches to state-of-the-art results published while we were conducting our own research.

2.1 Theory

2.1.1 Deep learning

2.1.1.1 Deep feedforward networks

The deep feedforward network is according to Goodfellow et al. [14] probably the most important deep learning model. The goal of such a model is to approximate a given function, from a set of datapoints to an output class or value.

Neural networks are loosely inspired by biological neurons where each unit, also called artificial neuron, has a set of input values and output values, and the output is determined by the total sum of input values. The network consists of a set of layers, where each layer has one or more artificial neurons. The first layer is called the input layer, and the input flows further through hidden layers in a feedforward manner. The final output is determined by the last layer called the output layer. An illustration of a feedforward network can be seen in Figure 2.1. By including feedback connections into the network, we get a network called Recurrent Neural Network (RNN), which is described further in Section 2.1.1.7.

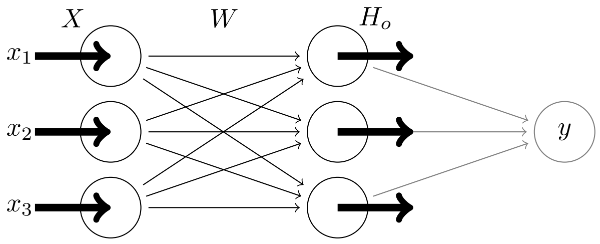


Figure 2.1: A fully connected feed forward network. The network has one input layer, one output layer and one hidden layer. The input starts at the first layer, and is propagated to all the nodes in the next layer. In the connections between each node, we find a trainable weight and bias, which determines how much the input of the first node should affect the latter node.

The computed output of a single node is based on its set of input values x_i, the connections weight w_i, and a bias value b_n. The computation for a node n with input connection i is described in Equation 2.1

$$f_{n,i}(x) = W_{n,i} \cdot x + b_n \tag{2.1}$$

We can also describe the entire network as a chain of functions, where each layer is represented as a link in the chain. The network's depth will then be equal to the length of the chain. A network of depth three could then be visualized by the following Equation 2.2.

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}))) \tag{2.2}$$

The goal of a neural network is to a approximate a function. In order to approximate almost any function, it is important to introduce nonlinearity to the network. Leshno et al. [32] described this in their paper from 1993. This is achieved by introducing a non-linear function into each node, called an activation function.

2.1.1.2 Activation functions

Activation functions are used for several reasons. One is to squash the output of a computational node into a given range, another is to introduce non-linearity to a neural network. The activation function is applied after the weighted sum in each computational node in the networks. Originally, the step function was the activation function used in neural networks, but now the activation function has been generalized to include functions such as Rectified Linear Unit (ReLU), sigmoid, and tanh.

ReLU was introduced by Hahnloser et al. [18] in 2000, and is possibly the most used activation function in modern deep neural networks. It is a relatively simple function that is a linear mapping for all input above 0, and 0 otherwise. This can be represented as the max function, seen in Equation 2.3. Another popular function is sigmoid, which maps all input values to the interval [0, 1]. Tanh is a variant of sigmoid, however scaled and shifted to map values to the interval [-1, 1]. Tanh can be seen in Equation 2.4, and is often preferred over sigmoid as it improves convergence, as shown by Lecun et al. [30].

$$f(x) = max(0, x) \tag{2.3}$$

$$tanh(x) = 2 \cdot \sigma(2x) - 1 = \frac{2}{1 + e^{-2x}} - 1 \tag{2.4}$$

2.1.1.3 Learning

A neural network is trained by tweaking the weights and biases until the model approximates the function mapping the training data to training labels satisfactory. An approximation is satisfactory when it has minimized the total error on the dataset, calculated using a loss function. The weights are first initialized to small random values; however, a lot of research has been done to optimize the weight initialization to improve convergence.

Through backpropagation, we adjust the weights and biases repeatedly in order to minimize a measure between the predicted label and actual label. The measure is called a loss function, and can for instance be Mean Squared Error (MSE). After doing a forward pass and getting a predicted label, the error is propagated backwards, first through the hidden layers all the way to the input layers. The error is used to calculate the gradient, which in turn allows us to optimally adjust the weights and biases to reduce the error. Figure 2.2 shows a visualization of how the error gets propagated backwards in a feed forward network.

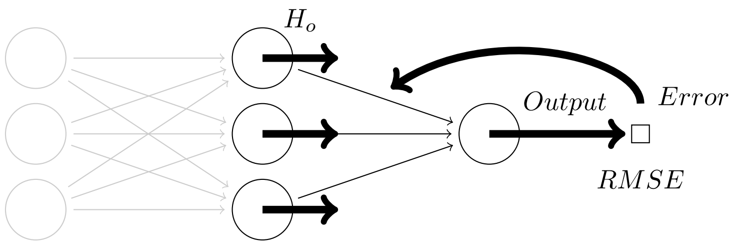


Figure 2.2: Backpropagation. Finding the error with root mean squared error and propagating it backwards.

2.1.1.4 Loss functions

Loss functions are as previously mentioned used as the function we want to minimize, in order to get the best possible function approximation. Several loss functions were used in this thesis, such as Root Mean Squared Error (RMSE). RMSE is useful for regression tasks, and can be seen in Equation 2.5. A specific loss for depth estimation is described in Section 2.1.5.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y})^2} \tag{2.5}$$

2.1.1.5 Optimizers

Optimizers serve a vital role in neural network learning, as they have the role of adjusting the weight values to minimize the loss function.

Stochastic Gradient Descent (SGD) calculates the loss gradient based on the weights, which is used to update the weights. SGD additionally uses a learning rate α - a number which specifies the magnitude of change when updating weights. A SGD weight update is shown in Equation 2.6.

$$w_{i,j}^{t+1} = w_{i,j}^{t} - \alpha * \frac{\partial L}{\partial w_{i,j}^{t+1}} \tag{2.6}$$

AdaDelta is was proposed by Zeiler [48] in 2012, and improves upon SGD without any noticeable computational overhead. It introduces separate and adaptable learning rates per dimension, and the authors claim that it requires no manual tuning of hyperparameters, including learning rate.

The Adam optimizer was introduced by Kingma and Ba [25] in 2014, and has later become one of the most commonly used optimizers. The authors describe the optimizer as both simple and computationally efficient, well fit for high-dimensional parameter spaces and large datasets. The way the Adam optimizer works is by first calculating an adaptive learning rate from the first (the mean) and second (the uncentered) moments of the gradient. Equation 2.7 shows the actual weight update of the optimizer. $\hat{m}^t$ and $\hat{v}^t$ are the first and second momentums, respectively. α is the specified learning rate, and ϵ is a constant with a recommended default value of $1e^{-8}$.

$$w^{t+1} = w^{t} - \frac{\alpha}{\sqrt{\hat{v}} + \epsilon} * \hat{m}^{t} \tag{2.7}$$

2.1.1.6 CNNs

Convolutional Neural Networks (CNNs) were first introduced by LeCun et al. [29] in 1998. They are neural networks specialized in processing grid-like data structures, such as images, where it can utilize spatial information in the images to reduce computation.

Typically, a convolutional layer consists of three stages. During the first stage, several convolutions are performed to generate a set of feature maps. Then the feature maps are typically sent through an activation function, and finally a pooling function is used to further modify the input, and often lower the resolution of the result. The three stages can be seen in Figure 2.3

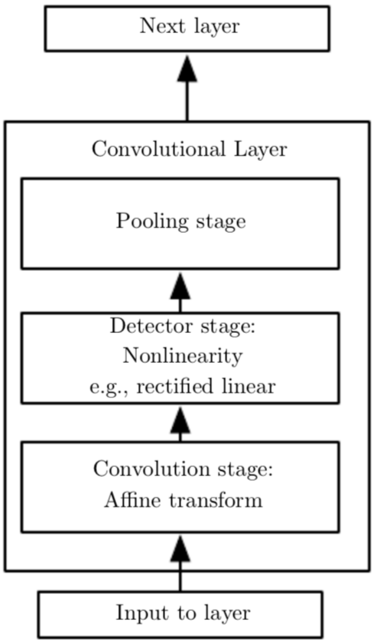


Figure 2.3: A typical convolution layer, divided into three different steps. Convolution operations on the input using kernels, non-linear activation function and finally pooling. Adapted from Goodfellow et al. [14].

The convolution step involves sliding kernels across the input and doing convolution operations for each step, generating a new feature map for each kernel. These kernels have weights, and can be trained to detect features such as edges. See Figure 2.4 for an illustrative description.

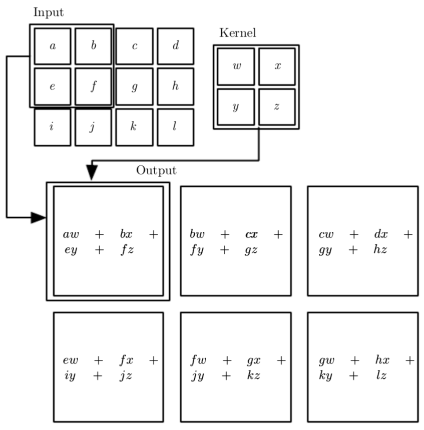


Figure 2.4: The first convolution stage. Feature maps are generated by sliding the kernels across the input while doing kernel operations. Adapted from Goodfellow et al. [14].

According to Goodfellow et al. [14], the use of convolutional layers is motivated by three main advantages: sparse interactions, parameter sharing, and equivariant representations. Oppositely to fully connected, sparse interaction mean that each input unit does not interact with each unit on the next layer. Kernels are often smaller size than the input, which means we can store fewer parameters using less computation. Overall, these advantages mean a more scalable and performant layer than fully connected layers.

2.1.1.7 RNNs

Recurrent neural networks (RNNs) are another family of neural network that are specialized in processing sequential data. They are often used in applications such as translation, speech recognition and image sequences. Using feedback connections, these networks can use information from previous time steps to determine the output of the current time step.

Long short-term memory unit (LSTM) was introduced by Hochreiter and Schmidhuber [21] in 1997, and is a computational unit often used in RNNs. Intuitively, the LSTM cell can capture important features early in the input sequence, and then remember the feature for several iterations. This way, it can capture long-term dependencies in the data. This is further described by Chung et al. [5]

2.1.2 Image classification

One of the first computer vision tasks that utilized CNNs was image classification. In this task, a model is given an image and predicts its class from a set of predefined classes. An example would be that it is given a picture showing a cat, with the correct prediction being the class "cat". In the next subsections, we will describe a few of the most common models (backbones) for image classification relevant to this thesis.

2.1.2.1 VGG (2014)

VGG is a CNN architecture for image classification introduced by Simonyan and Zisserman [42] in 2014. VGG was one of the earlier architectures that gave significantly improved results on the ImageNet [40] dataset. Perhaps the most important contribution from VGG was the demonstration that the network's depth was heavily correlated to its ImageNet score. This opened for further work on deep networks for improved performance. An important predecessor to VGG was AlexNet [27], which is a CNN from 2012 that revolutionized image recognition tasks, by performing 10.8 percentage points better than the follow-up in the ImageNet competition.

2.1.2.2 ResNet (2015)

ResNet was introduced by He et al. [20] in 2015 and is another CNN with outstanding performance in computer vision tasks. It improves upon previous approaches such as VGG by adding residual blocks in order to increase network depth while minimizing the computational cost. ResNet uses shortcut connections to allow the network to skip unneeded layers, which in turn allows the network to have a large number of layers. Figure 2.5 show a visualization of these skip connections, compared to both a plain network and VGG.

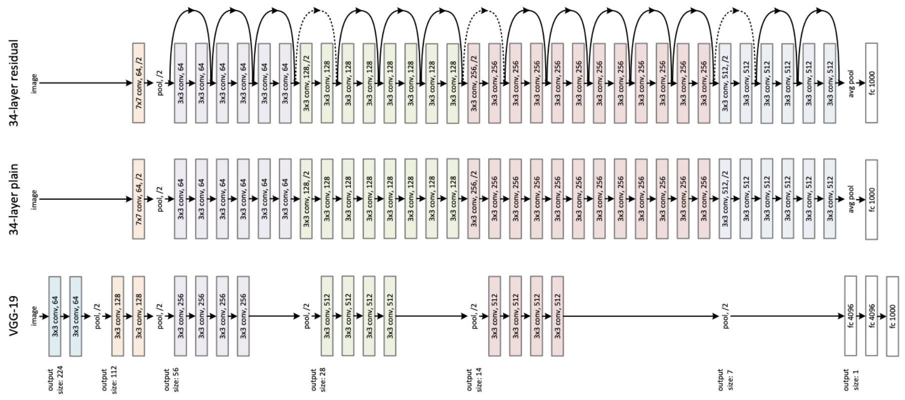


Figure 2.5: ResNet architecture. A visualization of ResNet compared to a plain network without skip connections and VGG. Adapted from He et al. [20].

2.1.2.3 MobileNets (2017)

MobileNets were introduces by Howard et al. [22] in 2017, and was a counter to the continued push for higher complexity models with marginally increased accuracy. As neural networks were used more in real-world settings, performance in memory and processing was of increasing importance. MobileNets are a class of networks created for mobile and embedded vision applications. They use depthwise separable convolutions to build lightweight neural networks. As part of the paper, Howard et al. also include two hyper-parameters that allow the user to find the perfect tradeoff between resource usage and accuracy for their use case. They also include a set of comparisons between MobileNets and previous conventional networks, and show that MobileNets give comparable performance with drastically increased efficiency.

MobileNets use a type of convolution called Depthwise Separable Convolution, initially introduced by Sifre and Mallat [41] in 2014. The convolution process works by first doing a single 2D convolution for each depth channel (Separable Convolution), and then doing a convolution using a 1x1 filter with the desired depth value (Pointwise Convolution). Dividing this process into two steps — compared to a single step for regular convolutions — reduces the number of operations required drastically, and is the way MobileNets are able to reduce the computational requirements while keeping most of the accuracy.

2.1.3 Object detection

Object detection is a computer vision task focused on localizing and identifying a set of objects in an image. It takes image classification a step further, and typically predicts bounding boxes with corresponding class labels of what it can localize and identify in an image. Neural network-based object detection generally utilize a so-called backbone to extract features, often based on one of the architectures described in Section 2.1.2.

2.1.4 Segmentation

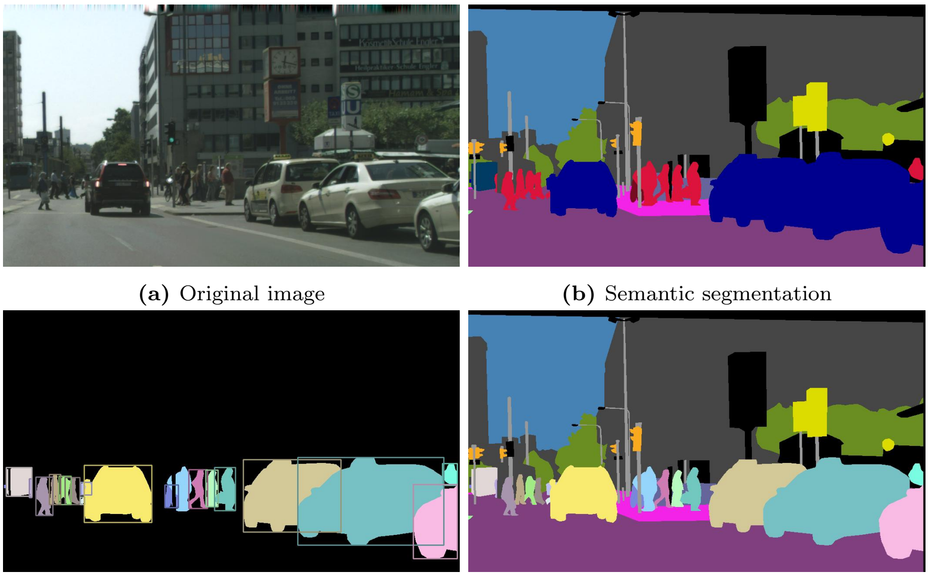
Segmentation is a computer vision task with the goal of classifying each pixel of an image as one of n possible classes. This can be interpreted as object localization and detection, only with pixel sized bounding boxes and no overlap. The segmentation task is often divided into two approaches: semantic segmentation and instance segmentation. Semantic segmentation only classifies what class each pixel belongs to, and two people in a picture will both be classified as the same "person" class. Instance segmentation however, cares about the instances in the image. That is, it should separate two people from each other, labelling them as two distinct people. A downside of instance segmentation implementaions is that they often cannot classify background classes, such as "sky" or "road" very well.

A third segmentation approach, panoptic segmentation, proposed in 2018 by Kirillov et al. [26], combines the best from both segmentation approaches, and can both classify every pixel on the screen, while additionally separating them into instances. Figure 2.6 shows an example of ground truth data for each of these tasks.

In this thesis, we focus on the applications of semantic segmentation, as it is able to give a good scene understanding while at the same time having lower computational cost compared to panoptic segmentation. In the next section, we present a few networks used for semantic segmentation.

2.1.4.1 Fully Convolutional Networks (FCNs) (2015)

Long et al. [33] proposed fully convolutional networks (FCNs), a class of neural networks where all layers are convolutional. They utilized recent improvements in CNNs with dense output layers (e.g. for object detection) to make several FCN-



(a) Original image **(b)** Semantic segmentation

(c) Instance segmentation **(d)** Panoptic segmentation

Figure 2.6: Illustration of various segmentation tasks. Adapted from Kirillov et al. [26].

.

based networks. Here, they replaced the final dense layers with deconvolutions — layers that can learn spatial upsampling — resulting in a pixel-wise segmentation. Figure 2.7 illustrates a basic FCN network.

The authors of the paper observed that using an output stride of 32 resulted in the segmentation outputs being overly coarse. They addressed this problem by adding skip connections between earlier layers and the output layer, where the earlier layers had a lower stride. The original model without skips were named FCN-32, while the models using one and two skips were named FCN-16 and FCN-8, respectively. FCN-8 performed best of the models, but any additional skip additions led to diminishing returns.

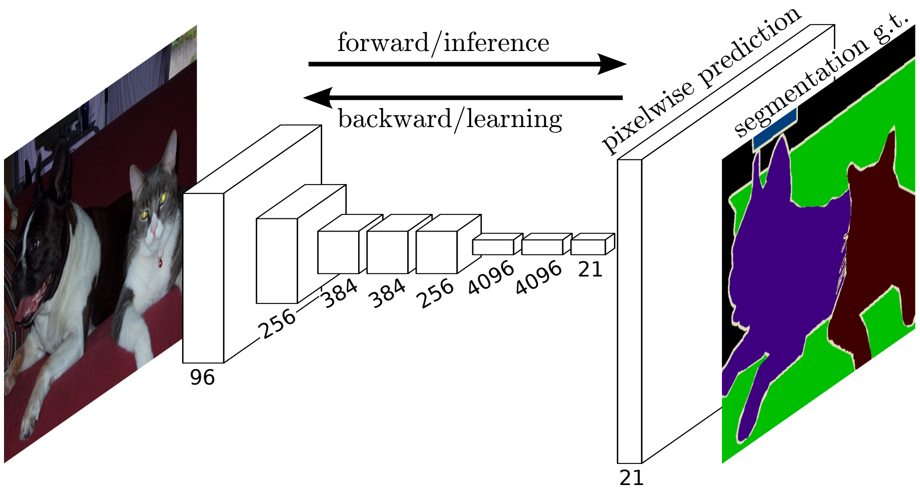


Figure 2.7: A basic FCN network based on AlexNet. Adapted from Long et al. [33].

2.1.4.2 SegNet (2015)

SegNet was introduced by Badrinarayanan et al. [2] in 2015. The network is a part of the FCN class, and is in many ways similar to the FCN networks proposed by Long et al. [33]. The architecture itself consists of an encoder and a decoder network. The encoder network is based on the first 13 layers of VGG16, which allows SegNet to start training with well-chosen pretrained weights.

The main difference between Long et al. [33]'s models and SegNet is that SegNet does not use skips, but rather a more lightweight approach, to achieve the same goal. Instead of fusing feature maps as in Long et al. [33], SegNet saves the pixel indices used in its encoder's pooling layers, and uses these to reconstruct the same pixels in its decoder's upscaling. Figure 2.8 shows how the pooling indices are applied when upscaling the feature maps in the decoder.

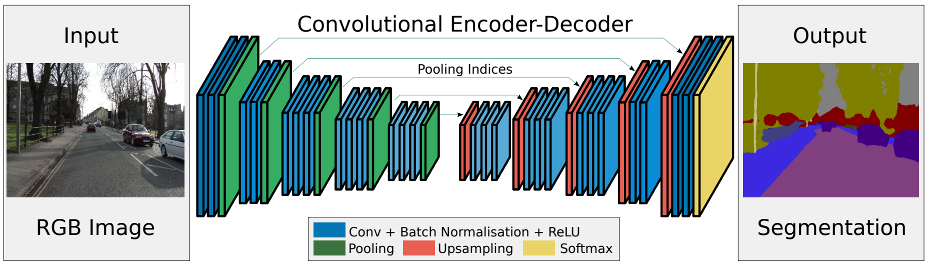


Figure 2.8: The SegNet architecture. Adapted from Badrinarayanan et al. [2]

2.1.4.3 U-Net (2015)

Ronneberger et al. [39] proposed a segmentation network for application in medical imaging, U-Net. Its name comes from its U-shape, shown in Figure 2.9, and is an FCN network. The architecture is similar to SegNet, but concatenates the encoder's feature maps to the corresponding deconvoluted feature maps of the decoder. When concatenating, each encoder feature map is cropped to match the dimensions of the decoder feature map it is being concatenated with.

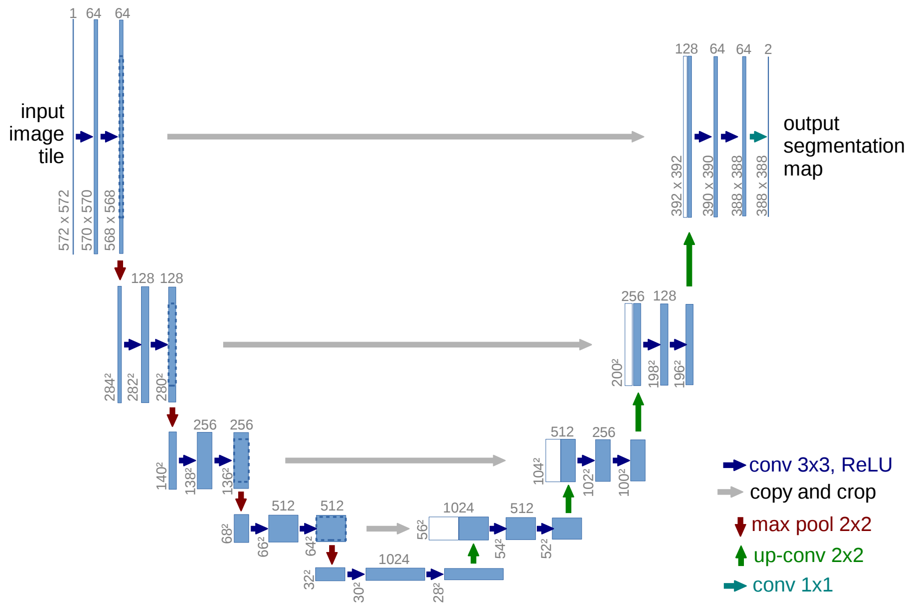


Figure 2.9: The U-Net architecture. Adapted from Ronneberger et al. [39]

2.1.4.4 Pyramid Scene Parsing Network (PSPNet) (2017)

The Pyramid Scene Parsing Network (PSPNet), introduced by Zhao et al. [49], employ a novel network module, the pyramid pooling module, and won the ImageNet scene parsing challenge of 2016. An overview of the network can be found in Figure 2.10.

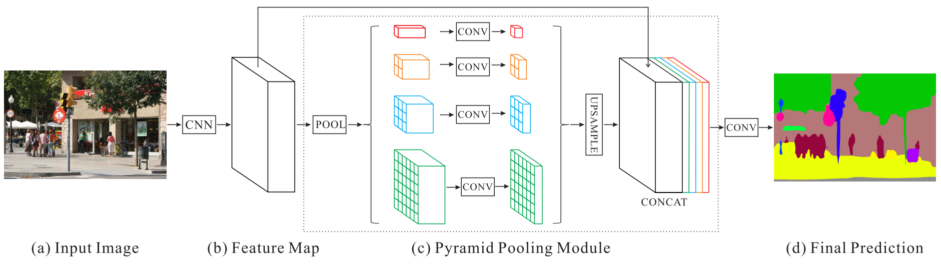


Figure 2.10: The PSPNet architecture. It takes an input image (a), which is first run through a CNN to get a feature map (b). The feature map is then sent to the pyramid pooling module (c), which splits the map into four sub-region representations with their own convolutions. The feature maps of each of the four modules are then upsampled by bilinear interpolation, and concatenated depth-wise with the original feature map from (b). Lastly, the concatenated feature map is sent through a final convolution, giving the final segmentation prediction (d). Adapted from Zhao et al. [49]

PSPNet addresses the issue of previous semantic segmentation networks not utilizing the global context of an image when making local predictions. An example mentioned in their paper is the classification of a pillow lying on a bed with a duvet of the same color and texture, see Figure 2.11. FCN was shown to struggle with separating the pillow from the blanket, as it only sees the local context, the texture. PSPNet's pyramid pooling module is able to understand the global context of the image, e.g. that the pillow is on the top-end of the bed, and that they are both in a bed in a room. This enables PSPNet to successfully segment the pillow.

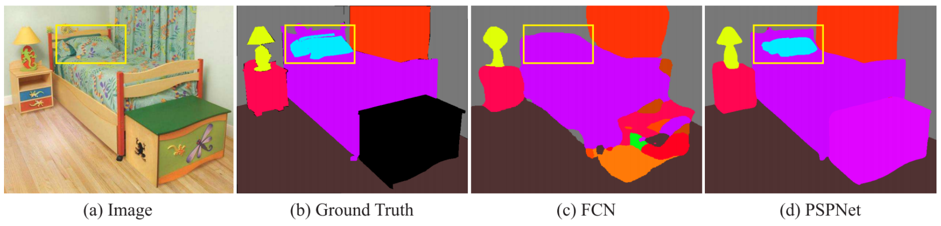


Figure 2.11: Comparison of FCN and PSPNet. The predictions of FCN (c) and PSPNet (d) for an image of a bed (a) with ground-truth (b) is shown. PSPNet can account for not only the texture on the pillow itself, but the context from the rest of the image as well. Adapted from Zhao et al. [49]

2.1.4.5 Intersection over Union (IoU)

A common metric when evaluating segmentation models is Intersection over Union (IoU), also referred to as the Jaccard index. IoU is calculated by taking the area of intersection for each segmentation class, and dividing it by the area of union for each class. The equation for IoU can be seen in Equation 2.8, where TP, FP and FN is true positive, false positive and false negative respectively. IoU is often used as it accounts for true positives, false positives and false negatives in the result.

$$IoU = \frac{TP}{FP + TP + FN} \tag{2.8}$$

For segmentation, IoU can also be weighted based on the different classes to predict. Common metrics are Mean IoU, calculated by finding the mean of each class-wise IoU. Frequency-based IoU is another metric, where the Mean IoU is weighted based on the number of occurrences (frequency) of the class in the given image.

2.1.5 Depth estimation

Depth estimation takes an image as an input, and aims to estimate each of its pixel's distance to the camera that captured the image. This allows one to understand the geometry of a scene, without the use of LIDARs, stereo cameras, or other depth-measuring sensors. Depth images can be generated by having several cameras, or estimated by a single camera, called monocular depth estimation.

A well-performing monocular depth estimation model called Monodepth2 was introduced by Godard et al. [13] in 2019. This model is trained with a technique called self-supervised monocular training, where a model learns by using a synchronized stereo image pair or monocular video to generate labeled ground truth estimation, which is further used to optimize a perception model. This is further described by Godard et al.

2.1.5.1 Accuracy within threshold

In order to measure estimated depth, a common accuracy measure used is *accuracy within threshold*. This metric is used in several papers [28, 4] when evaluating single image depth estimations. Accuracy within threshold is calculated by first finding the accuracy (δ) as seen in Equation 2.9. A pixel is then counted as a match if δ is above a given threshold, where we used $th = 1.25, th = 1.25^2$ and $th = 1.25^3$, adapted from the papers previously referenced.

$$\delta = max(\frac{d_{gt}}{d_p}, \frac{d_p}{d_{gt}}), \delta > th \tag{2.9}$$

2.1.5.2 Loss functions for depth estimation

Loss functions can be customized to a given problem domain, in order to optimize model performance. Alhashim and Wonka [1] describe such a loss function for depth estimation. The goal of this loss function is according to Alhashim and Wonka to balance between reconstructing the original image while also penalizing important features in the image's depth changes. This typically corresponds to edges of objects.

The first loss function, defined in Equation 2.10, is the point-wise loss between the actual depth map and the estimation. This corresponds to the goal of reproducing the original image.

$$L_{depth}(y, \hat{y}) = \frac{1}{n} \sum_{p}^{n} |y_p - \hat{y}_p| \tag{2.10}$$

The second loss function, defined in Equation 2.11, is based on the image's depth gradient, meaning the image's changes in depth, and handles penalizing important object boundaries.

$$L_{grad}(y, \hat{y}) = \frac{1}{n} \sum_{p}^{n} |g_x(y, \hat{y})| + |g_y(y, \hat{y})| \tag{2.11}$$

In this context, g_x and g_y is a function for computing the difference in gradient between y and $\hat{y}$ in x and y direction, respectively.

The final function is called Structural Similarity (SSIM), which tries to capture the difference in images on a higher level using groups of pixels, as opposed to a pixel-wise loss function. We represent SSIM as $L_{SSIM}(y, \hat{y})$. Finally, we combine Equation 2.10, 2.11, and $L_{SSIM}(y, \hat{y})$ to create the final depth loss function shown in Equation 2.12. λ is a weight for the depth loss, empirically set to 0.1 by Alhashim and Wonka [1].

$$L(y, \hat{y}) = \lambda L_{depth}(y, \hat{y}) + L_{grad}(y, \hat{y}) + L_{SSIM}(y, \hat{y}) \tag{2.12}$$

2.1.6 Data preparation

2.1.6.1 One-hot encoding

In many classification tasks, it is important that the model does not find unwanted correlation between classes. One-hot encoding is a popular encoding type for classifications for this reason. It assigns each class to an index in a list, instead of perhaps the intuitive way using a single number where each numeric value is a specific class (eg. 1, 2, 3), where the model could find incorrect correlations between class x and $x+1$. One-hot encoding works by having a list of length equal to the number of classes, where all values are 0 except the correct class which is 1. This also has the advantage that along with the softmax activation function, the result will be the probability distribution of classes. A comparative study of different encoding schemes and a further description is provided by Potdar et al. [38].

2.1.6.2 Augmentation

It has been shown that neural networks give better results proportional to the amount of data they have available for training. Perez and Wang [36] showed this when they released a dataset with a trillion-word corpus, which in turn drastically increased the performance of many text-models — even though the data was very unstructured and had lots of errors. Augmentation works similarly, where many datasets include a limited amount of data points, and most ways to increase the dataset will also improve the models.

Augmentation in our context means doing different type of affine transformations, cropping, hue/saturation and color changes to images as a data-preparation step to increase the available training data. Other, more advanced augmentation are also possible, such as shown in Perez and Wang [36], where a Generative adversarial network (GAN) is used to generate image variations.

2.1.7 Approaches for autonomous driving

Yurtsever et al. [47] divides the algorithmic design of autonomous driving systems into two approaches: modular approaches and end-to-end approaches. Modular approaches, sometimes referred to as mediated approaches, splits the complex task of driving into smaller tasks that are easier to implement separately. The end-to-end approaches, also referred to as direct perception approaches, directly map sensor inputs and known environment information to vehicle control actions. Figure 2.12 shows a comparison of the two approaches.

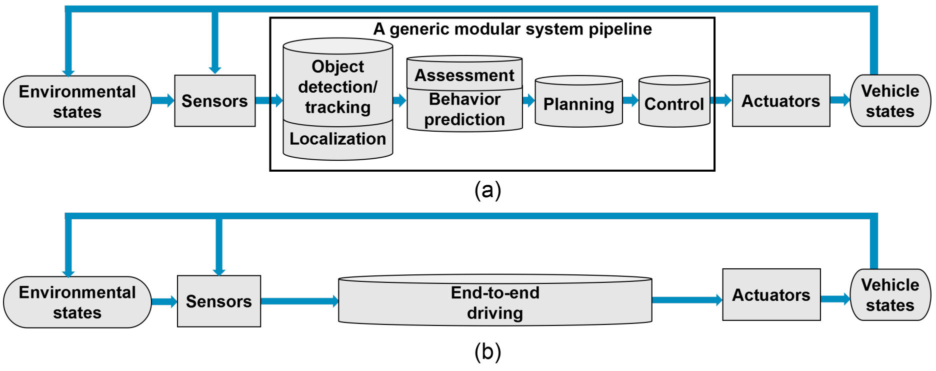


Figure 2.12: An illustration of the differences between a modular and an end-to-end system. Adapted from Yurtsever et al. [47].

2.1.7.1 Modular approach

Within modular systems, the implementation is divided into modules based on simpler tasks that together can form a driving system. The reasoning behind this approach is that it is much simpler to solve each problem individually, and then combine them into a bigger pipeline. Figure 2.12 (a) shows a typical modular system, with separate modules for object detection, localization, behavior prediction, planning, and control.

A run-through of such a pipeline could work along these lines: An autonomous car drives down a road, with a car in front of it. A camera image frame is sent into the object detection module, which detects a bounding-box for the car in front. The behavior prediction module predicts the movement of the detected car based on both this bounding box and previous bounding boxes it received. The planning module then uses this information to plan a safe path for the vehicle to move in, which the control module converts into actuator commands.

2.1.7.2 End-to-end approach

End-to-end systems, as shown in Figure 2.12 (b), are quite different from the modular systems. While both consume sensor data and output actuator commands, the end-to-end system treats everything in-between as one big decision-maker. There is no distinction between the roles of the neural network, as early layers that typically would do object detection could additionally be responsible for planning or even output controls. There are several approaches within end-to-end driving, and Yurtsever et al. [47] mention deep reinforcement learning and imitation learning as common approaches. We will not dive into deep reinforcement learning, but rather focus on imitation learning; a class of machine learning tasks focused on learning a policy by imitating the actions of an expert. In this thesis, we will work with behavioral cloning — a subclass of imitation learning where we in our case try to learn a driving policy by looking at examples from an expert driver. Note that imitation learning is often used interchangeably with behavioral cloning, but is actually a broader class of tasks that include reinforcement learning-based approaches as well.

Behavioral cloning. Imitation learning by behavioral cloning is a specific case of supervised learning. To gather training data, an expert performs the desired task, and we record the state of the environment and the expert's action in the given state. When doing this over time, we get a set of (state, action)-pairs, which can be used as inputs and targets for a supervised learner. In the context of autonomous driving, a state will typically be the outputs of the vehicle's sensors, e.g. a camera frame and speedometer reading. The action would then be a command for controlling the vehicle, which could be a new steering angle, throttle, and brake.

A big advantage behavioral cloning has over normal supervised learning, is that the data does not need labeling. The reason for this is that the target values, which is what you would normally need to label, is the outputs of the expert itself — gathered while doing data collection. As with any other type of learning, behavioral cloning has its own disadvantages. In driving, the set of (state, action) pairs are sequential; where a given state is dependent of the last action. This violates the independent and identically distributed (i.i.d.) principle of supervised learning. Another disadvantage is that the learner will only be able to learn what the expert driver did. Consequently, if the expert driver breaks traffic rules, or even crashes, the learner will try to learn this. This emphasizes the importance of having good training data from a wide variety of scenarios and from a reliable expert driver.

2.2 Technology

This section introduces the specific technology used in the thesis, from machine learning and simulation software, to physical equipment.

2.2.1 Machine learning software

To design and train our neural networks, we use Keras 2.2.4 with Tensorflow 1.13.1. Keras is a Python library which exposes an easy-to-use API for building neural networks. It does not train models itself, but rather abstracts it away to "backends" — libraries that implement the machine learning algorithms. In our case, the backend is Tensorflow. Tensorflow implements all the necessary building blocks for constructing neural networks, and provides GPU-accelerated (CUDA) implementations of these. This allows us to greatly improve both training and inference time relative to CPU-based implementations.

Training hardware. We use an Ubuntu desktop with a NVIDIA GeForce GTX 1080 Ti for training. With 11 GB of GPU memory, we can efficiently train large models with large data sets.

2.2.2 Simulator

Much of the early work within the field of autonomous driving were conducted using real-life vehicles (Pomerleau [37], LeCun et al. [31], Bojarski et al. [3]), and researches without access to such vehicles were at a great disadvantage. The Car Learning to Act (CARLA) simulator, introduced by Dosovitskiy et al. [10] in 2017, aims to make autonomous driving research available for everyone, by providing a leading open-source simulator for autonomous vehicle research. It is built upon Unreal Engine 4, which provides state-of-the-art real-time graphics and physics.

The simulator allows for simulating multiple vehicles, traffic flow, pedestrians, weather conditions, and more. It comes with several built-in environments (called towns), varying from small villages to highways — all of which can be used out-of-the-box. It has powerful APIs for both C++ and Python, which makes it possible to control most aspects of the simulation with custom scripts. Figure 2.13 shows an example of an environment in the simulator.

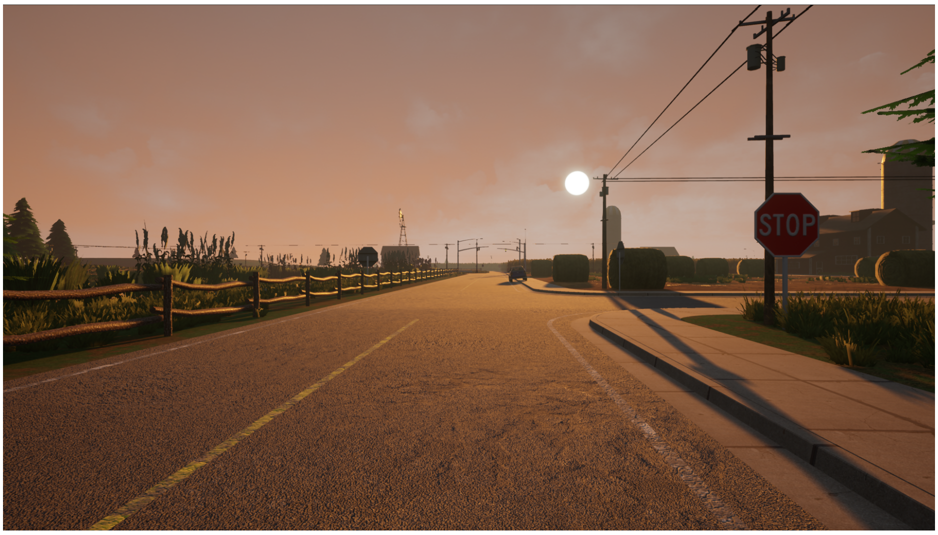


Figure 2.13: An example image from CARLA. The image was captured from a vehicle in Town07 at sunset.

As of CARLA 0.9.9, there are eight built-in towns, of which we mainly use Town01, Town02, and Town07. The weather API allows for extensive control of both the weather and time of day, which are both configured by a weather control parameters object. Some examples of weather control parameters are wind intensity, sun angle (controls day/night-cycle), cloudiness, and rain intensity. CARLA additionally provides an expanding variety of simulated sensors and outputs: RGB cameras, segmentation maps, depth maps, LIDARs, radars, GNSSes, IMUs, among others.

A problem with outside real-world experiments is that the environment is in constant change. It is impossible to reproduce the same conditions multiple times; as time progresses, weather changes, and external objects might move. In a simulator, we can control every aspect of the world, and if the simulator is deterministic, we can reproduce the exact same environment multiple times. CARLA has a mode where the simulation runs step-by-step as controlled by a user, where the whole simulation is independent of real-time frames per second. There are still some issues that prevent 100% determinism, but for most cases the simulator can be regarded as deterministic.

A driving simulator was essential for our early work, as it provided a reproducible environment for both data gathering and testing of our proposed models. CARLA had already been used by several authors (Codevilla et al. [6], Müller et al. [34], Xiao et al. [46], Haavaldsen et al. [17]), and seemed to be the best tool for our work. The simulator greatly helped saving time in the early phases of development of our models, as it eliminated the need to conduct time-consuming real-world experiments. The reproducibility and the deterministic behavior of CARLA further helped ruling out external factors from the experiments. This allows us to be confident that we will get the same results each time we perform an experiment, and that changes in performance is only dependent on our own models and evaluation code.

2.2.3 The SPURV Research vehicle

To conduct our real-world experiments, we used the SPURV Research — a small four-wheeled vehicle developed by KVS Technologies[1] for research in robotics and autonomous vehicles. The SPURV is battery-driven and contains a NVIDIA Jetson TX2 board with an integrated Tegra X2 GPU. This allows it to run CUDA-accelerated machine learning models, which Tensorflow supports creating, at a relatively high inference rate. All models tested in our real-world experiments were able to run an inference rate of at least 20 images per second. The robot is fitted with two RGB cameras, one facing forward, and one facing backwards, both at a height of approximately 20 cm above the ground. See Figure 2.14 for a picture of the SPURV.

[1]`https://www.kvstech.no/`

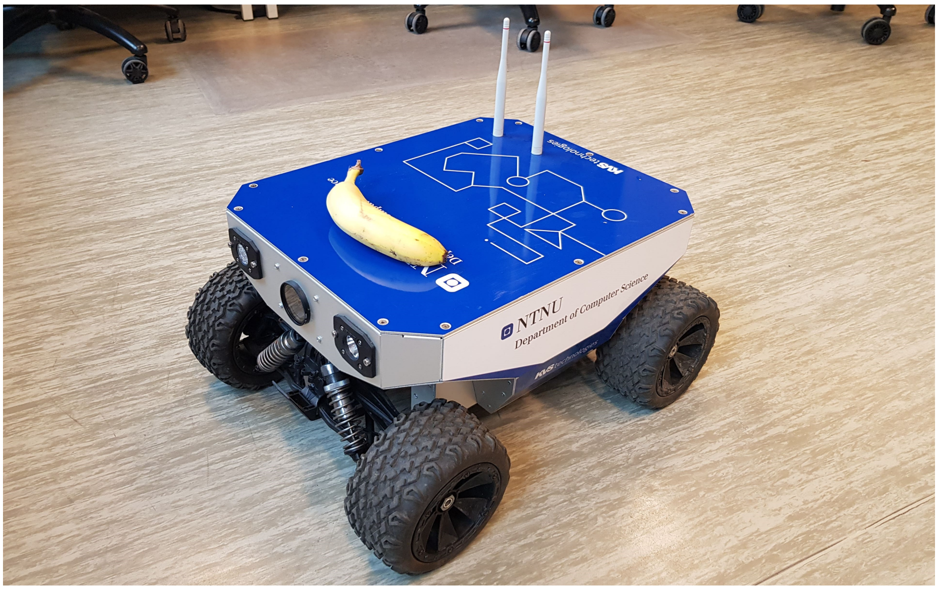


Figure 2.14: A picture of the SPURV Research vehicle, taken from the front.

The NTNU Autonomous Perception Laboratory (NAPLab) acquired a SPURV in 2018, and Kastet and Neset [23] used it to conduct autonomous driving experiments. Kastet and Neset did comprehensive work to enable the vehicle to be used for research, and wrote manuals to set up the necessary hardware and software for deep learning. Because of their work, the SPURV already had CUDA and Tensorflow available, and with easy-to-use manuals for commonly needed tasks; including data gathering, executing pre-trained machine learning models, and manually controlling the vehicle via Wi-Fi. We expanded on Kastet and Neset [23]'s work where we updated the existing software and created a new set of tools to complement theirs. These changes are further explained in Section 3.3.

2.3 Related Work

This section contains selected papers on end-to-end autonomous driving relevant to our thesis. We present the papers chronologically, giving a brief image of the field's history and a build-up to the state-of-the-art work.

2.3.1 ALVINN: An Autonomous Land Vehicle in a Neural Network (1989)

The work of Pomerleau [37] is one of the most well-known within autonomous driving, and was the first to use neural networks to create an end-to-end driving system. The ALVINN network, shown in Figure 2.15, consists of one input layer, one hidden layer, and one output layer; all of which are fully connected. The input layer contains input data from a camera and a range finder, which is then fed through the 29-unit hidden layer, and finally to the output layer. The output layer serves to output a steering angle for the vehicle, and consists of 46 neurons. Each neuron corresponds to a distinct steering angle, except one labeled "Road Intensity Feedback Unit", which states if the road is lighter or darker than the rest of the image. The road intensity feedback unit is fed into the next prediction as an input. Pomerleau showed that ALVINN was able to follow simple roads without intersections and forks, and they were optimistic to future applications of neural networks within autonomous driving.

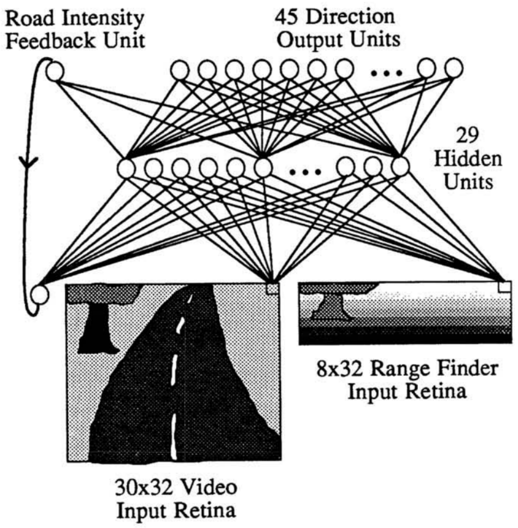

Road Intensity Feedback Unit
45 Direction Output Units
29 Hidden Units
8x32 Range Finder Input Retina
30x32 Video Input Retina

Figure 2.15: The ALVINN architecture as presented in the original paper. Adapted from Pomerleau [37]

2.3.2 Off-Road Obstacle Avoidance through End-to-End Learning. (2005)

LeCun et al. [31] built upon the work of Pomerleau [37], creating a novel network named DAVE. They present three major differences between ALVINN and DAVE: Firstly, they trained the network to do off-road obstacle avoidance instead of road following. Secondly, they used two cameras side-by-side to give the network stereo information. Instead of using known techniques to gather depth information from the stereo camera setup — and then use this as input for the network — they input the raw images and let the network learn its own representation. Lastly, they made use of convolutional layers to process the input images. The network itself consisted of six layers: four convolutional layers, where the first layer took a 147x56x6 input (two 147x56 images with three color channels each), a fully connected layer of size 100, and a fully connected output layer with two neurons; steer left or right.

To gather training data and test the network, LeCun et al. constructed a small four-wheeled vehicle with a length of 50 cm. The network itself was run on a remote computer, where a radio transmitter transferred the inputs and outputs of the network between the vehicle and the computer. They showed that an end-to-end system can perform some tasks as good or better than classical algorithms, with the network robustly detecting and avoiding obstacles in real-time.

2.3.3 End to End Learning for Self-Driving Cars (2016)

Inspired by the promising results of DAVE, Bojarski et al. [3] developed a new network, DAVE-2, by building upon the ideas of LeCun et al. [31]. DAVE-2 was designed to control full-scale cars on real roads, and was shown to drive on both highways and in residential areas, in diverse weather conditions. To improve upon both ALVINN and DAVE, they increased the number of trainable parameters from $\sim$72 000 in DAVE to $\sim$250 000 in DAVE-2. While the preceding networks had been limited by the computing capabilities of their time, Bojarski et al. could take advantage of GPU-accelerated neural networks that allowed for deeper networks and faster inference. Their final network consisted of one image normalization layer, five convolutional layers, and three fully-connected layers, as shown in Figure 2.16. The final layer outputs the inverse of the desired steering angle. 72 hours of varied driving data were used as input to the network. To improve error correction when driving, the data was additionally augmented by shifting and rotating the input images. When using the augmented images for training, the steering angles were adjusted to correct for the changed perspectives.

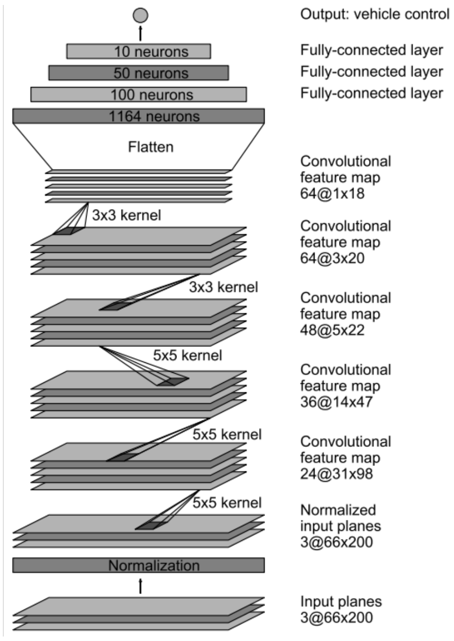
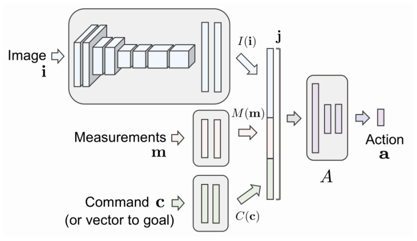


Figure 2.16: The DAVE-2 architecture as presented in the original paper. The output is a single node containing the raw steering angle. The image was adapted from Bojarski et al. [3].

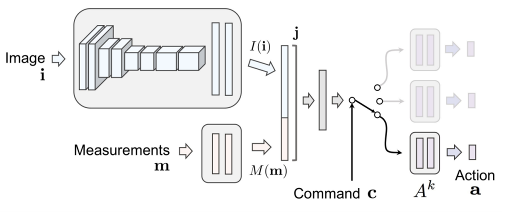
2.3.4 End-to-end Driving via Conditional Imitation Learning (2017)

In 2017, end-to-end imitation learning approaches like Eraqi et al. [11]'s had already shown great promise for lane- and road following. However, adapting these models to roads with intersections and forks was still a largely unexamined problem. Previous models were trained to learn one correct steering direction, and a road forking into two was therefore not something they could possible reason about. Codevilla et al. [6] addressed this problem by introducing a new input to behavioral cloning models — a High-Level Command (HLC) — thus naming the new type of imitation learning Conditional Imitation Learning (CIL). The paper includes a baseline implementation of CIL, where they define four HLCs to dictate the behavior in road forks: continue (follow the road), turn left, turn right, or drive straight. By providing this additional navigational information during data collection, the model was able to resolve any ambiguities about where to drive, and learn to choose a fork based on the given command. This allows for a person or even a route planner to choose which turn to take in a intersection, without explicitly telling it how it should control the vehicle.

Codevilla et al. [6] experimented with two alternatives for incorporating the HLC into the driving model. They both share a common component, a CNN which extracts features from images, which are then concatenated with the speed of the vehicle. The first architecture, shown in Figure 2.17a, concatenates the current HLC (as a one-hot encoded vector) with the image features and speed. This is then used as input for a small fully-connected network. The second approach, shown in Figure 2.17b, does not use the HLC as an input itself, but rather as a switch for selecting one of k (in our case 4) smaller fully-connected networks. Both networks outputted the same predictions: a steering angle, throttle amount, and brake amount. In their experiments, the switch-based network performed best, and they demonstrated that it could drive and follow commands in both a driving simulator and with a real RC car. To be able to run in both simulation and in the real world, they trained two models: one on only simulation data and the other only on real-world data.



(a)



(b)

Figure 2.17: An illustration of different approaches for incorporating HLC into a end-to-end driving network. The approach in Figure 2.17a is titled *command input network*, and uses HLC as an input to the model. Figure 2.17b shows the *branched network* architecture, where HLC is used as switch for different sets of learnable output layers. The figure is adapted from Codevilla et al. [6].

2.3.5 Driving Policy Transfer via Modularity and Abstraction (2018)

Müller et al. [34] proposed a CIL model that was able to be trained solely in a driving simulator and then drive in both the simulator and in the real world. The motivation for their work was that driving simulators allow for data collection and testing of models in a safe, cheap, and diverse environment. The problem with simulators is that the perception data differs considerably from real-world data, which makes directly transferring the models impossible. Müller et al. proposed to abstract the input data into a domain-independent format; a format where the same scene would be represented identically, independent of the environment (e.g. simulator or real-world) of the scene. In their implementation, they first trained a separate model, the "perception module", to predict a binary segmentation map from images. The perception module segmented images into the categories "road" or "not road", which was then used as input for the CIL network, the "driving policy". When training the driving policy, the weights of the perception module were held constant. With a perception model trained on only real-world images, they got good enough road segmentation of images from both simulation and real world. Figure 2.18 shows an overview of Müller et al.'s architecture.

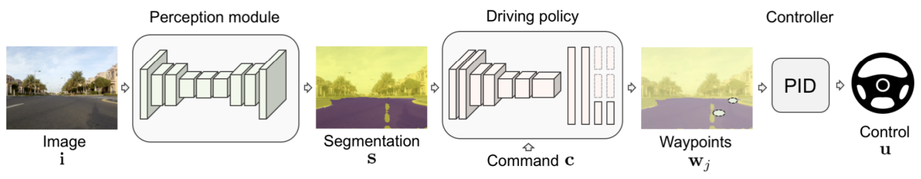


Figure 2.18: Architecture of Müller et al.'s system for transfer of driving policy between different domains. The image gets processed by a perception module, and is then used for a CIL driving network. The outputs of the driving network are waypoints that a PID controller uses to control the vehicle. The figure is adapted from Müller et al. [34].

To test their implementation, they first evaluated it in a driving simulator, and then on a 1/5-scale robotic truck. They achieved best results when adding two important changes to their architecture: Firstly, to not overfit their model to the specific camera configuration of the simulated car, they varied the camera's mount height, vertical field-of-view, and pitch. Secondly, the vehicle dynamics (how the vehicles react to steering, physics, etc.) differed from the simulated car and the robotic truck. Therefore, they abstracted away the raw steering angle and throttle outputs, instead outputting waypoints relative to the vehicle. With these adjustments, their model was able to drive in highly varied real-world environments, without any fine-tuning despite only learning to drive in a simulator.

2.3.6 Autonomous Vehicle Control: End-to-end Learning in Simulated Urban Environments (2019)

Haavaldsen et al. [17] proposed a CIL-based driving model with the addition of a LSTM network. Most of the earlier solutions for end-to-end driving only process spatial information, i.e. information from a single image, run through convolutions. Haavaldsen et al. built upon the idea that humans gather information from the scene over time, to get a good enough understanding to drive in urban environments. Their network, shown in Figure 2.19, combined a CNN with LSTM cells to learn from both spatial and temporal dependencies. Additionally, they separated steering and throttle/brake into two separate models, only sharing input images, HLCs and environmental information.

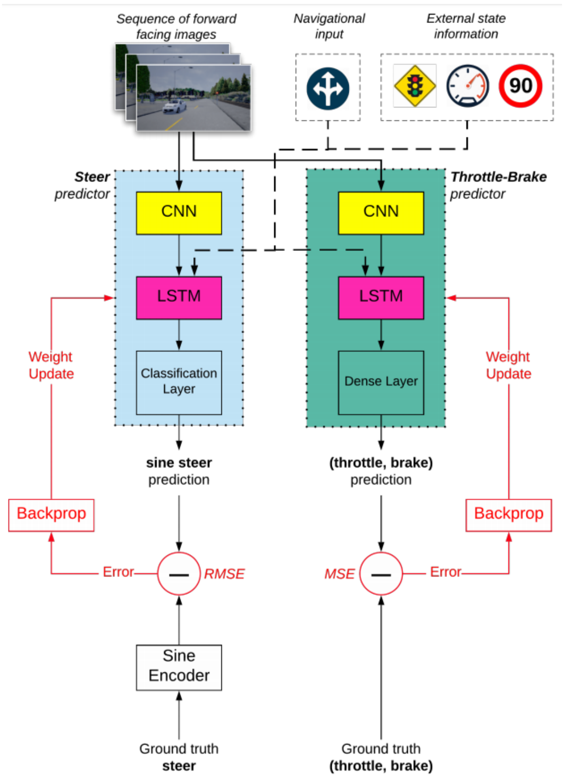


Figure 2.19: A high-level overview of Haavaldsen et al.'s architecture. Adapted from Haavaldsen et al. [17].

2.3.7 Multimodal End-to-End Autonomous Driving (2019)

The majority of papers on end-to-end autonomous driving focus mainly on RGB images as input sensor data, but using depth information has shown promising results in other computer vision tasks (see Viereck et al. [45], Gualtieri et al. [15]). Xiao et al. [46] pointed out that many autonomous vehicles are not only equipped with cameras, but with depth measuring sensors such as LIDARs as well. They proposed to use depth information received from such a sensor to train a driving model that performs better than with images only.

Xiao et al. experimented with three approaches for including the depth information into their CIL network. Figure 2.20 shows the approaches: early fusion, mid fusion, and late fusion. Their results showed that early fusion resulted in the best driving performance. Early fusion incorporates the depth data from the start of the network, by channel-wise concatenation of the RGB image and depth map, thus creating an RGBD (RGB and depth) representation. To test the driving performance of their network, they ran it through a driving simulator which generated camera images and accurate depth maps.

In addition to the models using both RGB and depth information, they also trained two separate control networks using exclusively RGB images or depth maps. As an extra experiment, they trained a separate network that created depth estimates from the RGB images, and used this as the input to their early fusion network, instead of the ground truth depth from the simulator. They showed that this variant could in some cases outperform both networks trained on ground truth depth only and images only, and noted that it was an approach worth pursuing in the future.

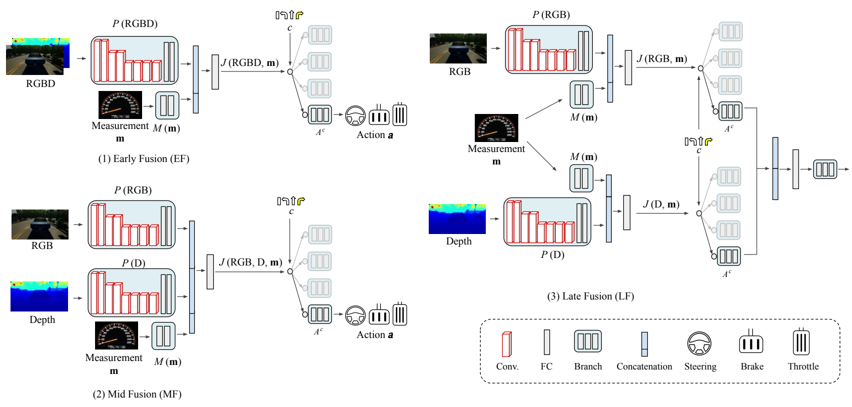


Figure 2.20: An overview of the various approaches Xiao et al. [46] built for including depth information in a CIL network. Early Fusion (1) fuses the image and depth data immediately. Mid Fusion (2) extracts features from both inputs separately, then fuses the features. Late Fusion (3) builds a separate CIL network for each input, and fuses their predictions together as input for a final fully-connected network. Adapted from Xiao et al. [46].

2.3.8 Urban Driving with Conditional Imitation Learning (2019)

Hawke et al. [19] released a paper in late 2019 where they attempted to improve upon earlier end-to-end autonomous driving approaches. As opposed to most recent papers on end-to-end driving, Hawke et al. trained and validated their model solely on real-world data. Using CIL and a perception model pre-trained on public datasets, they were able to drive a car in urban environments with simple traffic.

Their driving policy was an end-to-end neural network, but it was conceptually divided into three components: *perception*, *sensor fusion*, and *control*.

The *perception* component had a deep encoder-decoder architecture and used multi-task learning inspired by Kendall et al. [24] to create multiple predictions from the scene: an RGB image reproduction, segmentation, and monocular depth estimation. The perception component was pre-trained on several large, heterogeneous research datasets for vision, such as Mapillary [35] and CityScapes [8], to create a data efficient and robust perception component. When training the control component, only the encoded features were used, containing the learned compressed representation of the scene, which can be seen in Figure 2.21.

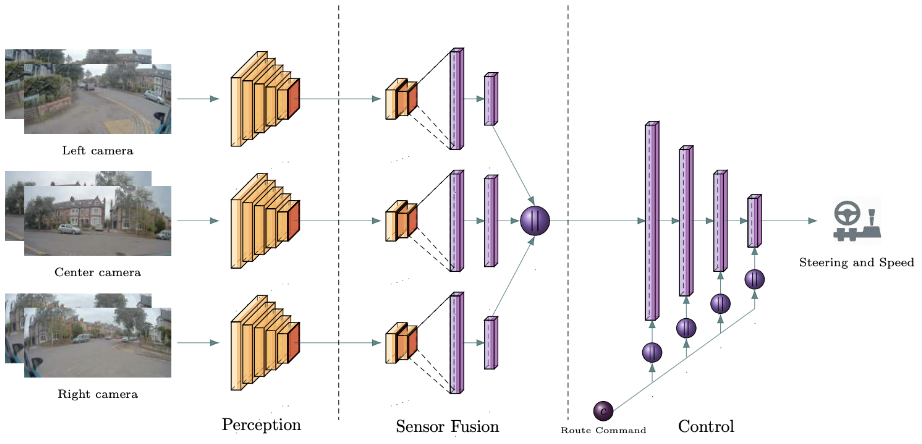


Figure 2.21: The end-to-end neural network architecture of Hawke et al. [19]. The architecture is divided into three components: *perception*, *sensor fusion* and *control*. Adapted from Hawke et al. [19].

In addition to the multi-task perception model, intermediate features from the separate optical flow model PWC-Net [44] was concatenated with the features from the separate multi-task perception model, resulting in a set of features complete with semantics, geometry, and motion information.

Hawke et al. also used sensor fusion to give the model full observability of relevant information, using two additional cameras; one to each side. The three cameras were joined into a single representation using sensor fusion. However, this increased the model's issue with basing its driving on non-causal elements, such as deciding driving speed based only on the input speed limit (called *causal confusion*, and described further by de Haan et al. [9]). Therefore techniques such as augmentation, noise (e.g. Gaussian) and dropout were used on the input to reduce the model's dependency on spurious information.

Finally, the control commands were generated from a network with fully connected layers, where the command input (HLC) was concatenated with each layer, as showed improved robustness. The output was a motion plan represented by a parameterized line for both steering and speed. During training, the loss was calculated by finding the mean squared error between the output motion plan and the actual expert driving N timesteps in the future, where the error was discounted by a factor for each timestep.

Chapter 3

Methodology

This chapter describes the methodology of our work. The first section contains a thorough description of the data we used, our neural network architectures, and metrics used to evaluate them.

Our approach to end-to-end driving is inspired mainly by two papers. Müller et al. [34] attempted to use modularity and abstraction to transfer their driving model directly from simulation to real-world. They pre-trained a perception model to learn semantic segmentation. The segmentation maps were then used when training a driving model in simulation. The driving model could then be transferred directly to real-world, using the more abstract representation for perception. Hawke et al. [19] expand their pre-trained perception model to use multi-task learning, generating depth estimations and RGB reproduction. The model has an encoder-decoder architecture, where the decoder is removed and only the encoded representation of the driving scene is used when training the driving model. Hawke et al. used only real-world data, and the paper did not focus on the simulation to real-world problem.

The approach used in this thesis attempts to incorporate elements from the paper by Hawke et al. into the approach by Müller et al. Specifically, we use multi-task learning where the perception model generates depth maps in addition to segmentation maps. We also expand to use five segmentation classes, as opposed to Müller et al. who used binary classification of road/no-road. Several combinations of training datasets are used in our experiments, where the goal was to get the best model using as little real-world data as possible. Public perception datasets were utilized for training the perception model, as it did not require any further data-collection. Figure 3.1 shows an overview of our network, where a pre-trained perception model predicts segmentation and depth maps that are further used as input to the driving model.

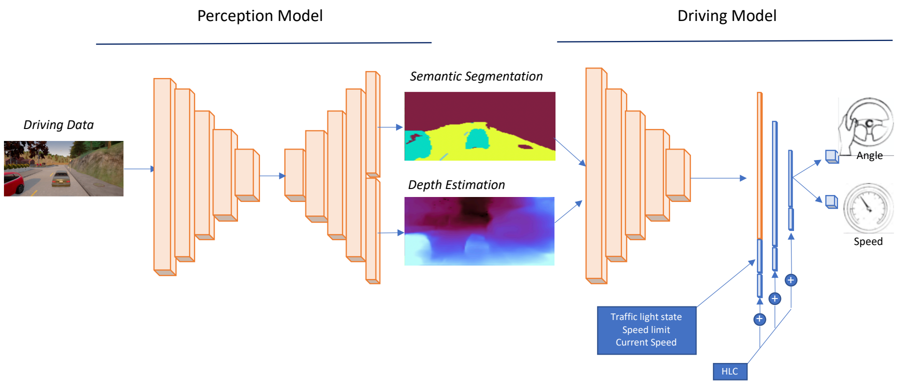


Figure 3.1: An overview of the architectures used in this thesis. The perception model takes driving data as input, which it converts to segmentation and depth maps. The output is then used as input to the driving model, which also process HLC and other high level information and finally outputs the angle of steering and speed.

3.1 Perception

The perception model is trained separately to the driving model, and therefore we first focused on finding the optimal perception model. The model was chosen by testing a different set of neural network architectures for semantic segmentation. The architectures were then trained on combinations of real-world and simulation datasets. The real-world dataset was the public Mapillary Vistas dataset [35] (henceforth Mapillary), while the simulation dataset was generated in CARLA.

3.1.1 Data collection

Simulated data. In order to improve the performance during driving in simulation, we included a synthetic dataset from CARLA. This was generated using a Python script that spawns a large variety of moving vehicles and pedestrians. Images were taken by cameras with a random distribution of yaw angle and field of view. The camera height was fixed to three meters above ground, as we wanted to avoid having the camera blocked by any of the vehicles it was mounted on.

During data-collection, environmental conditions were also randomized. The weather conditions were changed periodically between varying cloudiness, amount of rain, time of day and more. The numbers and locations of pedestrians and vehicles were also randomized between 40-80 and 0-40 respectively. The total size of the synthesized dataset is about 20 000 images.

Real-world data. The Mapillary Vistas dataset is a high quality, street-level dataset from driving situations from around the world. The dataset includes fine-grained annotations of both semantic segmentation and instance-specific image segmentation, with the semantic segmentation dataset including 66 different classes. During this thesis, we only used the semantic segmentation images.

3.1.2 Data preparation

The Mapillary and CARLA segmentations and images were first cropped and resized to 384x384 pixels. Then we reduced the number of classes in the segmentation annotations to five: road, lane markings, humans, vehicles and unlabeled (all other classes). The images were normalized using mean normalization, where the mean values are computed from the whole ImageNet dataset, similar to most implementations of VGG (Simonyan and Zisserman [42]).

As we wanted to train monocular depth estimation in addition to segmentation, we needed to find a dataset for depth images. To reuse a lot of work done with the Mapillary dataset, we decided to generate depth labels for the dataset using the pre-trained Monodepth2 model [13]. The Monodepth2 model was pre-trained on the KITTI dataset [12], and the data format was a depth value for each pixel. The depth value was represented as a normalized value between [0, 1], where 0 was 0.001m and 1 was 80m. For the CARLA dataset, these depth maps were built into the software, and we could therefore choose to generate depth maps as well as segmentation maps and RGB images. The depth value of a pixel in CARLA represented a distance in space 0m to 1000m forward (CARLA's max render distance), and was normalized to the range [0, 1], closer and farthest away respectively. Figure 3.2 shows an example of Mapillary data with segmentation and our generated depth maps.

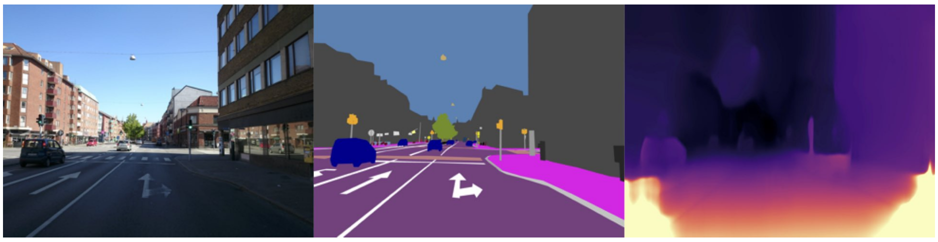


Figure 3.2: Sample from the Mapillary Vistas dataset. A selected image from the Mapillary training data. It shows the original image to the left and the labeled segmentation in the center. The right-most image is a depth map generated by Monodepth2.

The depth information had some differences between the datasets, which may make it difficult to train models on combinations of the datasets. A possibly improved approach could have been to use Monodepth2 to generate depth images from CARLA data as well. A visual comparison between the different formats can be seen in Figure 3.3. Note that the CARLA data has a higher maximum depth value, and the depth visualization is therefore stretched over a longer interval, as the visualization uses the min and max value to determine the color value for a given depth.

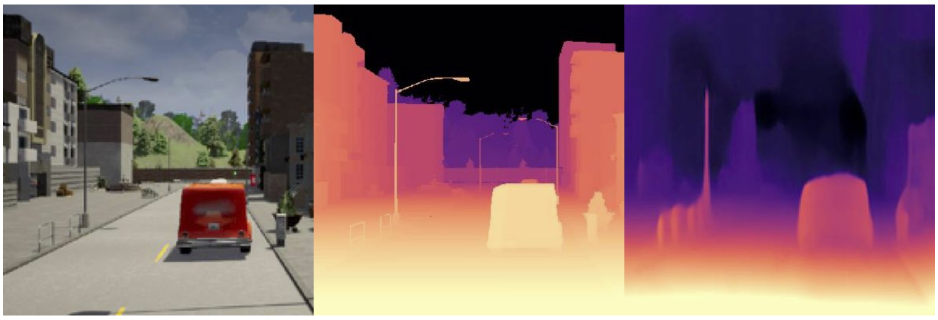


Figure 3.3: Compared depth maps between datasets. The leftmost image is the original image. The center image shows the source of truth image from CARLA, while the final rightmost image shows the generated depth map from Monodepth2.

Before training, the images were augmented in order to increase the dataset size. The augmentation used was the built-in augmentation to the prediction library we used [16]. These augmentations, some of which are shown in Figure 3.4, include cropping, affine transformations (e.g. rotation, translation), blur, contrast change, and hue/saturation changes. For the depth models, these augmentations were not used as we did not implement new augmentations for depth.

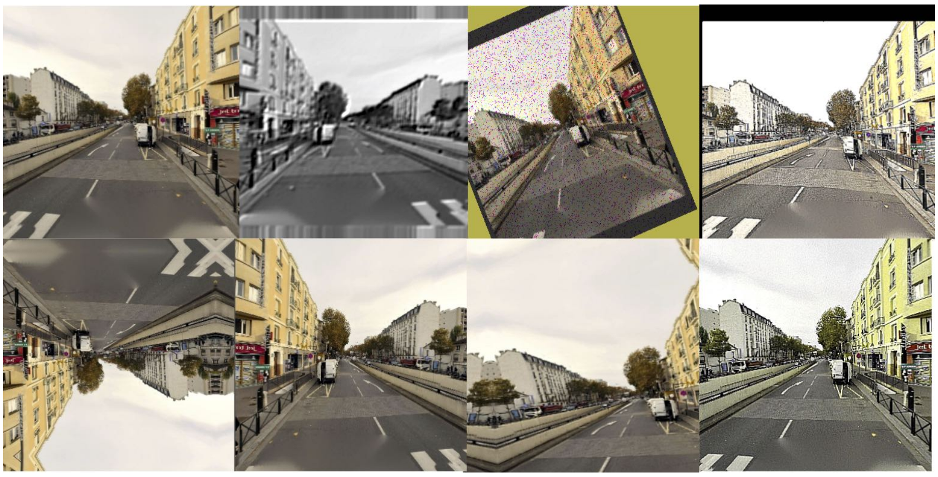


Figure 3.4: Perception data augmentations. A set of randomly generated augmentations from a single image from the Mapillary dataset. Each image includes several different augmentation of different magnitudes. Some of the obvious augmentations ordered from top left are: greyscale (second image), rotation (third image), mirroring (sixth image) and hue/saturation changes (eight image). The top left image is the original.

3.1.3 Architecture

The architectures used in this thesis are well known existing segmentation models or variations of these. A library with implementations of many popular segmentation networks called *image-segmentation-keras* [16] allowed us to explore several different architectures, as well as compare these using the relevant driving data we had collected. Several encoders and decoders were tested for our model architecture, with encoders being MobileNet [22], ResNet-50 [20], and a vanilla CNN. The decoders we experimented with were SegNet [2], U-Net [39], and PSPNet [49].

Multi-task learning was also explored as an attempt to improve the overall prediction performance. Standley et al. [43] showed good improvements when training segmentation with depth estimation. Depth estimation has also been used successfully by several papers [45, 15] as a means to generalize between simulated and real-world data. One of our approaches was therefore to train the segmentation models on an additional depth estimation task. A visualization of one of our main architectures using both depth estimation and semantic segmentation can be seen in Figure 3.5.

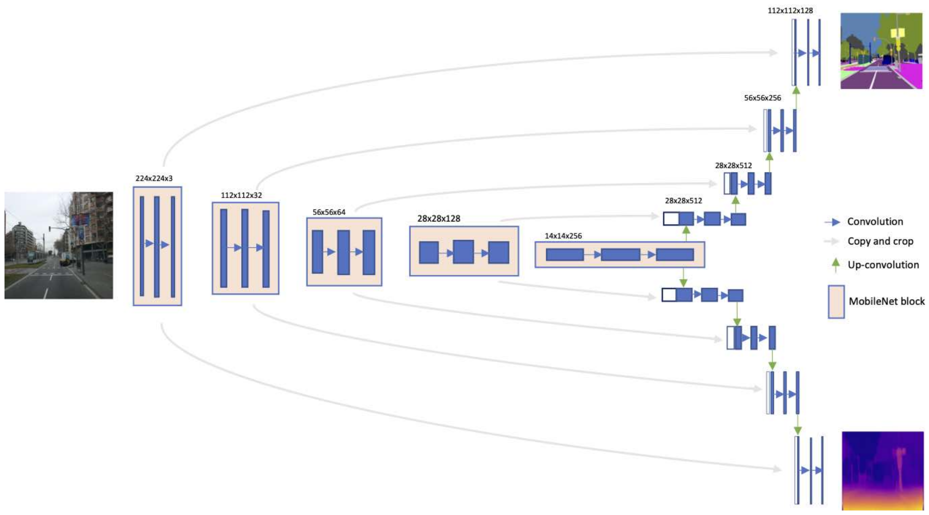


Figure 3.5: A visualization of the MobileNet+U-Net architecture with an additional depth branch. MobileNet encodes the image, while two similar U-Net decoders up-sample the encoding to a segmentation map and a depth map. The visualization is not an exact representation of the final architecture, and the detailed description can be seen in Appendix C.

The depth estimation branch was implemented similar to the existing U-Net segmentation architecture. The two branches differ in the activation function used on the final layer, loss function, and number of channels in the final layer. As depth estimation only predicted a single value per pixel, we reduced the last layer to only include one channel, as opposed to five channels for semantic segmentation (one for each of the predicted classes). The final layer activation function was changed from softmax to sigmoid to represent depth estimation as a regression task instead of classification. Finally, a loss function for depth estimation, adapted from Alhashim and Wonka [1], was implemented as an additional loss function to the categorical crossentropy used in segmentation. It is described further in Section 2.1.1.4. All models were trained with the AdaDelta optimizer, see Section 2.1.1.5.

3.1.4 Evaluation and metrics

To evaluate the perception models, two different metrics were used: one for semantic segmentation and one for depth estimation. Semantic segmentation was evaluated using Intersection over Union (IoU), also referred to as the Jaccard index. IoU is calculated by taking the area of intersection (true positives) for each segmentation class, and dividing it by the area of union (true positives + true negatives + false negatives) for each class. Calculating the mean of the class-wise IoU, we end up with Mean IoU. IoU is further described in Section 2.1.4.5.

For depth evaluation, we used the *accuracy within threshold* metric. We set the threshold th to 1.25, 1.25^2, and 1.25^3, adapted from Cao et al. [4]. The actual accuracy δ was calculated by $\delta = max(\frac{d_{gt}}{d_p}, \frac{d_p}{d_{gt}}), \delta > th$, where d_p is the predicted depth value and d_{gt} is the ground truth value calculated for each pixel. Accuracy within threshold is further described in Section 2.1.5.

3.2 Driving

The driving model passes images through the perception model, and then uses the estimated segmentation and depth maps as inputs to a CNN. These training images are collected during recordings made from an expert driver's driving. This data was generated in CARLA. The CNN processes the images, and uses a HLC and known information about the vehicle and its environment to output control signals for the vehicle to apply. To evaluate the models, we use a metric measuring the mean completion rate for each validation route.

3.2.1 Data collection

For the driving model, we gathered data both from an autopilot in CARLA and manual driving with SPURV. A summary of the information we collected per data point is shown in Table 3.1.

Data	Description	Simulated	Real
Forward center image	The image from the camera later used for inference	490x224 PNG	432x324 JPEG
Forward left image	Image from left-aligned front-facing camera	490x224 PNG	N/A
Forward right image	Image from right-aligned front-facing camera	490x224 PNG	N/A
Steering angle	The steering angle [left, right] of the vehicle	float [-1, 1]	float [1, -1]
HLC	The given high-level command: left (1), right (2), straight (3), or follow lane (4)	int	int
Target speed	The desired speed of the vehicle	float (m/s)	float (m/s)
Speed	The speed of the vehicle	float (m/s)	float (m/s)
Speed limit	The speed limit at the vehicle's location	float (m/s)	N/A
Traffic light	1 if not affected by a traffic light or it is green, 0 otherwise	int	N/A

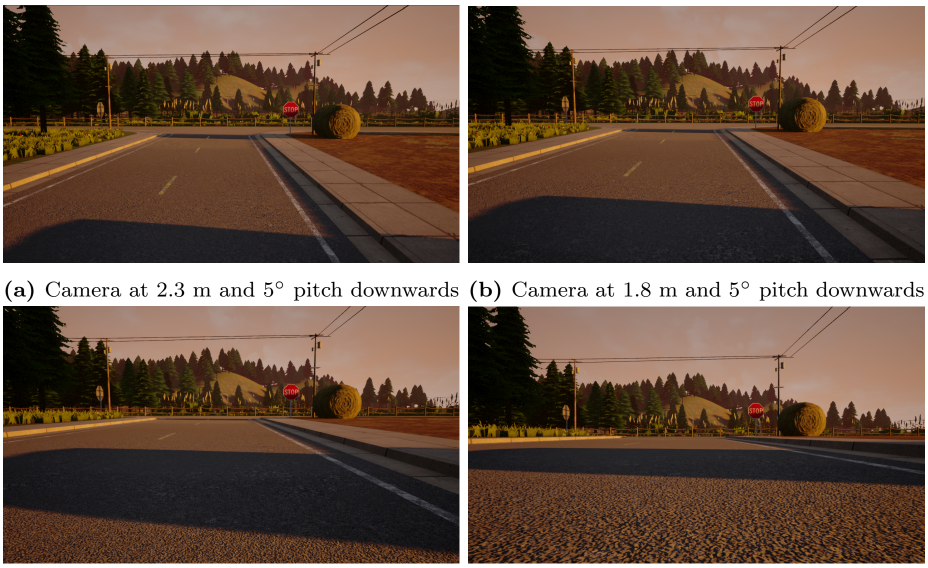
Table 3.1: The contents of each driving data point gathered. N/A specifies that the data were not collected in the given domain.

Simulated data. We used CARLA 0.9.9 to generate the simulated training data. To get the information we need, we utilized CARLA's Python API to make a script that automatically drove a vehicle through a set of routes in various CARLA maps. The script is based on Haavaldsen et al. [17]'s work, and uses the API's built-in autopilot which has access to the full state of the virtual world. This includes the location and velocity of all vehicles and pedestrians and a HD map of the whole environment. The information is used to generate waypoints for a driving path, which is fed into a PID controller which outputs steering angle and throttle values. Note that this dataset is different from the one described in Section 3.1.1, as it consists of continuous sequences of images from the same vehicle, paired with its controls and environment state.

Our script supports capturing data from different camera configurations, inspired by Müller et al. [34]. This allows us to make a dataset with images from a diverse set of different camera heights, fields of view, and camera pitches. The reason for doing this is to make the driving network learn to drive independently of the camera specifications and where the camera is put on the vehicle. Figure 3.6 shows several camera configurations in CARLA, some which were used in the training set. We varied the following camera parameters when generating training data:

- The horizontal field of view, alternating between 65°, 90°, and 110°.
- The camera height, alternating between 2.8m, 2.3m, 1.8m, and 0.8m (as opposed to the fixed 3m used in the perception dataset).
- The camera's pitch, alternating between −5°, 0°, and 5°.

The lower the camera is positioned, and the lower field of view, the less information is gained about the scene.



(a) Camera at 2.3 m and 5° pitch downwards **(b)** Camera at 1.8 m and 5° pitch downwards

(c) Camera at 0.8 m and 5° pitch downwards **(d)** Camera at 0.3 m and 5° pitch. Not used in training, but approximates SPURV's camera configuration.

Figure 3.6: An overview of different camera configurations in CARLA.

Real-life data. We collected real-life data by driving the SPURV on a set of small-scale roads. We collected 22530 data points; mostly while following the right-side of the road, occasionally taking turns in intersections. In an attempt to learn steering correction, we applied noise to our steering angles, which we manually had to correct while driving.

Automatic noise generation has been important for adequate performance, described by Codevilla et al. [6]. We adapted their technique for including triangular noise in the steering signal during data-collection. The algorithm worked by first randomly selecting a noise magnitude and direction. Then over a fixed period the noise was gradually increased until the max value, following a similar reduction until it reached no noise. This process was then repeated for the entire data-collection period.

The generated noise was excluded from the training data to ensure the model only learned from the expert drivers driving signal and not the noise signal. A visualization of the entire noise process can be seen in Figure 3.7

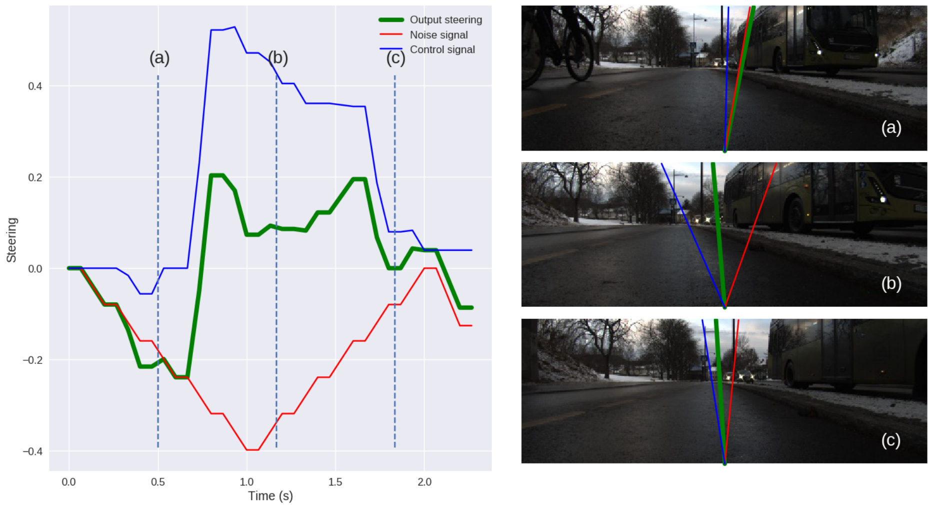


Figure 3.7: Triangular noise in real-world training data. A visualization of a period with triangular noise generated during data-collection. The blue line is the actual control signal from the expert driver, the red line is the noise and the green line is the final output steering. In stage (a), the vehicle starts drifting from the noise signal. During stage (b), the expert driver has started correcting to the noise. Finally, in (c), the expert driver slowly reduces the correction as the vehicle straightens and the noise signal is reduced.

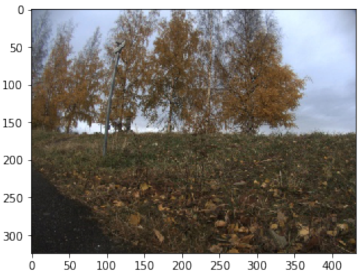
3.2.2 Data preparation

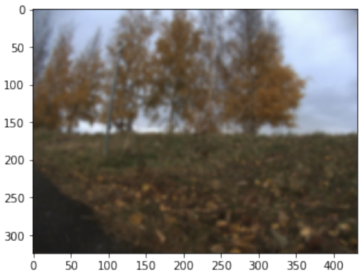
To utilize the collected data better, we employed various data augmentation and balancing techniques.

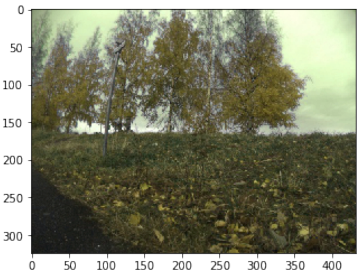
Randomly moving HLCs. To prevent the network from learning an identity mapping between a HLC and steering, e.g. always steering left when getting a ”turn left in next intersection” HLC, we randomly move some HLCs back a few frames, such that it is activated before the intended turn. This method is based on Haavaldsen et al. [17]’s work.

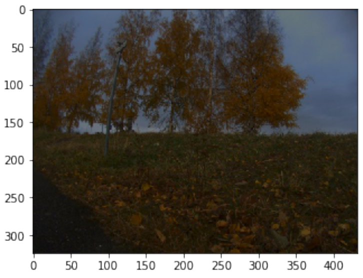
Balancing. The data we gather from CARLA initially has a skewed data distribution. Many of the data points are quite repetitive, as they capture the vehicle driving straight while following a lane, or the vehicle waiting for a green light. If the data contains considerably more examples of a certain action, the driving model will become biased to that action. To counteract the bias effect, we balance the data. To learn all HLCs equally good, we down-sample with respect to HLCs by removing data points until we get a equal amount per HLC.

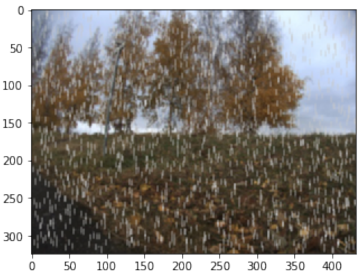
Image augmentation. To help our models generalize, we apply augmentation to our training images. Each mini-batch has a 25% probability of being augmented with one of the augmentations shown in Figure 3.8. These are the same augmentations as were used in Haavaldsen et al. [17]: Gaussian blur, adjusted hue, adjusted brightness, simulated rain, and simulated shadows.


(a) Original picture


(b) Gaussian blur


(c) Adjusted hue


(d) Adjusted brightness


(e) With lines representing rain

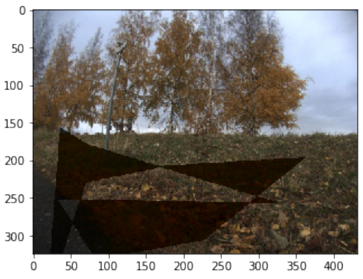

(f) With polygons representing shadows

Figure 3.8: A training image with augmentations applied to it.

3.2.3 Architecture

The driving model architecture is built to use the outputs of the perception model (segmentation and depth maps), a HLC, and known information about the environment and the vehicle state. These inputs are processed within the network to produce two outputs: a steering angle and a target speed. Figure 3.9 shows a simplified version of the model. Note that we concatenate the HLC value with the output of several layers. This is an approach taken from Hawke et al. [19], where they suggest that it improves the model's robustness.

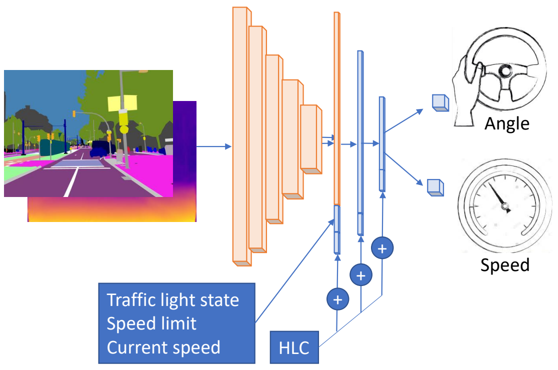


Figure 3.9: Driving network architecture. A simplified illustration of the driving network. The segmentation and depth maps inputs are concatenated directly from the outputs of the perception model, shown in Figure 3.5. See Appendix C for the implementation.

The driving model first processes the outputs from the perception model. We concatenate its predicted segmentation and depth map channel-wise such that we get an input of size 112x112x6 (112x112x5 segmentation map concatenated with 112x112x1 depth map), inspired by Xiao et al. [46]'s best performing approach, early fusion. The combined segmentation and depth representation is then processed through five convolutional blocks consisting of zero padding of 1, a 2D convolution, batch normalization, a ReLu activation, and finally max pooling with pool size of 2. The convolutions have the following parameters, in order:

1. 64 filters and kernel size of 3
2. 128 filters and kernel size of 3
3. 256 filters and kernel size of 3
4. 256 filters and kernel size of 3

5. 256 filters and kernel size of 3

The resulting feature map then gets flattened into a vector of length 2 304. This is concatenated with the following inputs:

- *info_input*: a vector of size 3. It contains the traffic light state (1 for green or if the vehicle is not near any traffic lights, 0 otherwise), the speed limit in km/h, and the current speed of the vehicle, in km/h. Both the speed limit and current speed values are normalized by division of 30 and subtraction of 1. The number 30 was chosen as the original intended maximum speed was 30 km/h, but was not adjusted subsequently.
- *hlc_input*: the current HLC, one-hot encoded ([left, right, straight, follow lane]).

The resulting vector of length 2 311 is followed by a fully-connected layer of size 100. The outputs are concatenated with the HLC vector, and followed by a LSTM layer with 10 cells. The output of the LSTM is once again concatenated with the HLC vector, before branching off to the two outputs of the model:

- *steer_pred*: The predicted steering angle, with ReLu activation. To get the correct angle in CARLA, we multiply the value by 2 and subtract 1, mapping the value to the [-1, 1] range. ReLu activation was chosen over sigmoid because we saw slight improvement in convergence during training.
- *target_speed_pred*: The predicted target speed, with sigmoid activation. To get the target speed in km/h, multiply the value by 100.

The sequence length used for the LSTM layer is always 1, so no temporal dependencies are considered because of this. The LSTM was put into the architecture early in the design process, and we decided not to remove it — even though it served no real purpose — as many models had already been trained with it, and we did not want to introduce changes which could make the comparison of models incorrect. Moreover, the inclusion of the LSTM layer allows for inclusion of temporal dependencies in any future extensions of the network. The Adam optimizer (see Section 2.1.1.5) was used for training, just as in Haavaldsen et al. [17].

3.2.4 Evaluation and metrics

Many machine learning tasks include standardized metrics validating the performance of models. These metrics can usually be calculated directly when validating the models on a separate test dataset. However, for the task of imitation learning for autonomous driving, there are no such standardized metrics, partly because there could be several correct driving actions for a given point in time. While there has been some work on creating methods to better compare different solutions (see Codevilla et al. [7]), there is no standard methods for evaluation. We follow the work of Haavaldsen et al. [17], and use the mean completion rate (henceforth abbreviated as MCR) to measure our driving models' performance. MCR is defined in Equation 3.1, where d_c is the completed distance of the model, d_t the total distance of the route, and R the set of all routes.

$$MCR = \frac{\sum_{r \epsilon R} \frac{d_c}{d_t}}{|R|} \tag{3.1}$$

We additionally watch the validation loss of each model, calculated from the 20% of the driving dataset not used in training. We discovered that many of our trained models had a significantly higher loss than the best-performing ones, and we used that as a rough method to weed out failed training sessions. Our most successful models all had a validation loss of less than 0.20, and we therefore discarded any model with higher values.

3.3 Real-world validation

The work of Kastet and Neset [23] got us started using the SPURV, and we ended up building on this when we made our own tools. After applying some initial software updates, we made new scripts for data collection and model evaluation. Our data collection scripts supports recording of steering angles and HLCs, where the HLC was collected by manually pressing buttons on the Xbox controller used for driving during data collection. The buttons used were: X (turn left), Y (keep straight) and B (turn right). Additionally, we used a button to optionally enable the triangular noise explained in Section 3.2.1. Finally, the model evaluation script was upgraded such that it could switch between models in quick succession.

Figure 3.10 gives a brief overview of our work related to the SPURV, which we call the SPURV pipeline. With these tools, it should be relatively easy to collect training data, train models, and then finally test them on the SPURV. For more information about the pipeline, see Appendix B, which provides documentation and a user manual.

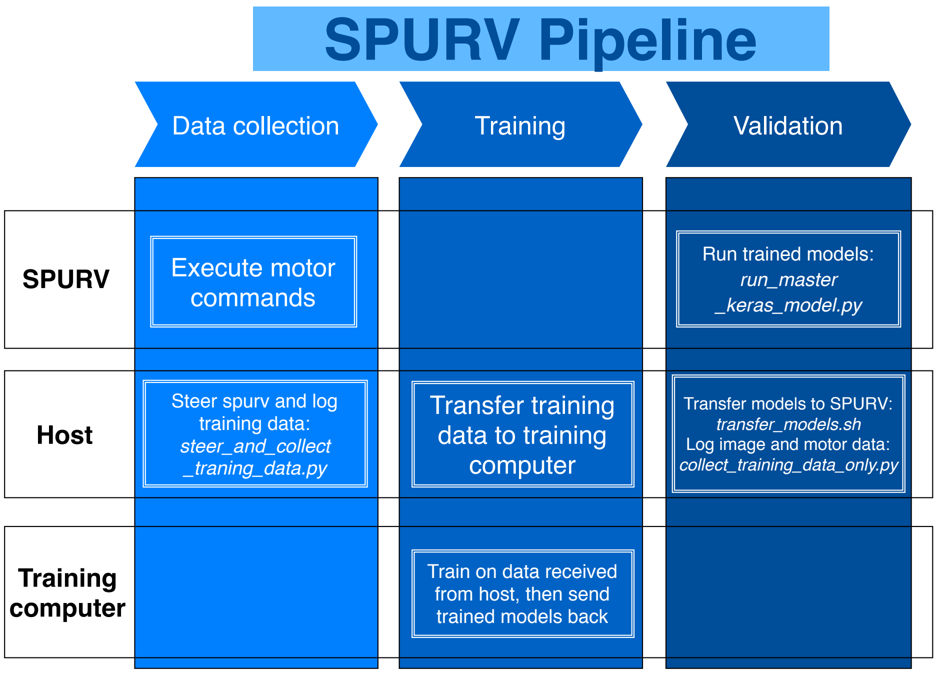

SPURV Pipeline
Data collection
Training
Validation
SPURV
Host
Training computer
Execute motor commands
Run trained models: *run_master _keras_model.py*
Steer spurv and log training data: *steer_and_collect _traning_data.py*
Transfer training data to training computer
Transfer models to SPURV: *transfer_models.sh*
Log image and motor data: *collect_training_data_only.py*
Train on data received from host, then send trained models back

Figure 3.10: The SPURV pipeline, consisting of three main steps. Our host machine was a laptop which connected to the SPURV via Wi-Fi, while the training computer is any computer with suitable hardware for neural network training.

Chapter 4

Experiments and Results

We conducted three main experiments as a part of our work. The first experiment was created to determine the best perception model for further experiments. The second experiment uses the perception model and a second driving model to train and evaluate driving performance in the simulated driving environment CARLA. Each model's generalizability is also tested by using different previously unseen environments. The third and final experiment attempts to validate the final driving model in a real-world environment. This experiment is conducted on a downscaled urban driving environment, using a small four-wheeled vehicle by the name of SPURV Research.

4.1 Experiment 1: Perception model

This experiment consists of three sub-experiments that were created in order to determine which perception model has the best performance. The first sub-experiment trains three encoders and two decoders on the Mapillary dataset in order to optimize segmentation performance. The second sub-experiment uses different combinations of the Mapillary and CARLA datasets, with and without augmentation. The third sub-experiment uses a second decoder branch that generates depth maps in addition to the segmentation maps. The results are then compared to a subset of the data configurations from sub-experiment 2. All the perception experiments use the same dataset for evaluation, and the result can therefore be compared across experiments.

4.1.1 Setup

To train and test several architectures on image segmenation, we adapted an existing library by Gupta [16] that had implemented a set of the most common encoders and decoders for image segmentation. As described in Section 3.1.2, we used an existing dataset from Mapillary, as well as a synthesized dataset from CARLA to train the different architectures. The models were evaluated using a common CARLA dataset with 4 400 images with corresponding segmentation maps and depth maps. Some of the models were also evaluated on Mapilliary data, where that test dataset was 2 000 images taken directly from the Mapillary test set. Evaluation of segmentation results were done with Mean IoU and Frequency Weighted IoU as described in Section 2.8, while depth estimations were evaluated using *accuracy within threshold* as described in Section 2.9.

4.1.2 Experiment 1-1: Encoder-decoder models

This experiment attempts to find the best encoder and decoder to use for the perception network. We evaluated six combinations consisting of three different encoders and two decoders.

4.1.2.1 Setup

In order to evaluate the different encoder-decoder combinations, we used a set of pre-defined models from the *image-segmentation-keras* library [16]. All variations of encoders and decoders were tested in combination. The encoders tested were VanillaCNN, MobileNet and ResNet50, while the decoders tested were SegNet and U-Net. The dataset was Mapillary with 18 000 images for training and 2 000 for validation without any augmentation. All the models were trained for 90 epochs, and the version with lowest validation loss was chosen. Time per epoch was set as the total training time divided by number of epochs. The evaluation data was a synthesized dataset from CARLA, and the models are therefore tested in an unseen domain.

4.1.2.2 Results

Model	Mean IoU	Weighted IoU	Time per Epoch
VanillaCNN-SegNet	0.324	0.712	37s
VanillaCNN-U-Net	0.351	0.705	55s
MobileNet-SegNet	0.368	**0.775**	**19s**
MobileNet-U-Net	0.403	0.774	24s
ResNet50-SegNet	**0.405**	0.767	71s
ResNet50-U-Net	0.383	0.733	98s

Table 4.1: Experiment 1-1 - Evaluation of encoder-decoder architectures. Evaluation of three different encoders and two decoders on a CARLA evaluation dataset. Weighted IoU is shortened from Frequency Weighted IoU. Each model was trained on the Mapillary dataset. The best scores are marked in bold and the table is sorted by encoders of increasing complexity. Higher values are better in Mean IoU and Weighted IoU, and lower values are better in Time per Epoch. A description of the IoU metrics can be found in Section 2.1.4.5.

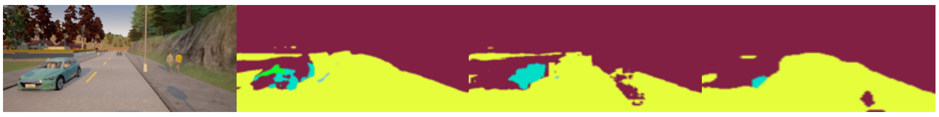


Figure 4.1: Images segmentation samples. This image shows an original image from CARLA, with segmentation maps generated by ResNet50-SegNet, MobileNet-SegNet and VanillaCNN-SegNet in order from left to right. Note that the models were trained on real-world data and therefore have never seen a simulated environment.

4.1.2.3 Discussion

Figure 4.1 shows the outputs of some of the trained models. Table 4.1 show a correlation between the complexity and size of a model to the resulting segmentation result, perhaps unsurprisingly. U-Net shows a better performance paired with the VanillaCNN and MobileNet, however, it performs worse when paired with ResNet. While SegNet showed overall shorter training times than U-Net, the use of MobileNet as the encoder showed the most drastic improvement in training time overall.

Looking at the resulting Mean IoU, Weighted IoU and time per epoch, MobileNet-U-Net showed overall best results. It had comparable performance to ResNet50-

SegNet with almost three times less training and prediction time. MobileNet-U-Net was therefore chosen for the next experiments. Note that as the models were trained for slightly different number of epochs, time per epoch is just an estimation for training time duration.

4.1.3 Experiment 1-2: Training data

This experiment uses the MobileNet-U-Net architecture chosen in Experiment 1-1, and introduces different variations of training data and augmentation in order to improve the overall results in both simulated and real-world environments.

4.1.3.1 Setup

The dataset variations introduced in this experiment was a CARLA dataset and a set of augmentation techniques. We created a dataset from a combination of Mapillary and CARLA data, naming it Mapillary+CARLA. The Mapillary+CARLA dataset consisted of 20 000 datapoints from the Mapillary dataset and 3 250 samples from *Town01* and *Town02* in CARLA. In addition to a combined dataset, we created a dataset of only CARLA data, with 15 000 samples from Town 1-4 for training, and 4 000 samples from Town 5 for validation. We evaluated the dataset on a separate dataset from Town 1-2.

We trained most datasets with and without augmentations, where augmentations included rotation, translation, cropping, hue and saturation changes, blur, noise, among others. The different augmentations are described more in depth in Section 3.1.2.

The resulting models were evaluated on both the original CARLA test data, as well as a second Mapillary test dataset. We included evaluation with Mapillary as we wanted to find the model that generalized best between the two domains.

4.1.3.2 Results

	CARLA Eval		*Mapillary Eval*		*Mean*	
Training dataset	mIoU	wIoU	mIoU	wIoU	mIoU	wIoU
Mapillary	0.425	0.771	0.632	0.887	0.529	0.829
Mapillary+Aug	0.436	0.809	0.574	0.873	0.505	0.841
Mapillary+CARLA	0.469	0.846	**0.633**	**0.889**	**0.551**	**0.868**
Mapillary+CARLA+Aug	0.478	0.850	0.568	0.874	0.523	0.862
CARLA+Aug	**0.572**	**0.909**	0.384	0.785	0.478	0.847

Table 4.2: Experiment 1-2 - Evaluation of perception datasets. mIoU is shortened from Mean IoU, while wIoU is shortened from Frequency Weighted IoU. A table of the MobileNet-U-Net model trained on different datasets. All models were evaluated on a CARLA and Mapillary dataset. The best results are in bold and higher values are better in all the metrics.

4.1.3.3 Discussion

Table 4.2 shows the resulting metrics for each model. Introducing CARLA data into the Mapillary dataset seem to be very beneficial. The Mapillary+CARLA model had a slight increase in performance on the Mapillary dataset, and a large increase on the CARLA dataset, compared to the original model trained on Mapillary data only. We also see that the CARLA data gives, perhaps as expected, the best result when evaluating on CARLA data; however, the performance decreases drastically when evaluated on Mapillary data. It seems a model trained on real-world data is able to generalize better to simulation than the other way around.

Augmentation as part of the training gave varied results. Mapillary improved its results on CARLA data when introducing augmentation, however, it decreased the results on real-world data compared to the model trained only on the Mapillary dataset. The same results are shown on the augmented Mapillary+CARLA dataset. An interpretation could be that the model generalizes better, however penalizing perfomance on data similar to the training dataset. In continued experiments, we do not include augmentations even though it seemed to increase the generalizability, which could have further improved the perception model on real-world driving data.

On average, Mapillary+CARLA data performed best overall in both mIoU and wIoU, with a mean value on both datasets of 0.551 and 0.868 respectively. As there is only a small difference in results, and no model with best performance on both datasets, we continue to use most of the datasets in further experiments, with the main training dataset being Mapillary+CARLA.

4.1.4 Experiment 1-3: Multi-task perception

Training a model on multiple perception tasks has shown to improve the performance on each task in general. For instance, Standley et al. [43] shows that training a segmentation model with depth estimation improves the segmentation model's performance by 4.17%. Similarly, Hawke et al. [19] and Xiao et al. [46] introduced depth as part of their models with good results. In this experiment, we added a second decoder branch and trained the segmentation model on both segmentation and depth estimation simultaneously using multi-task learning.

4.1.4.1 Setup

The MobileNet-U-Net model was further expanded to include a second decoder branch for depth estimation, as described in Section 3.1.3. Depth maps were generated by using a pre-trained monocular depth estimation model for the Mapillary data, as it did not include depth information originally. CARLA data included depth maps as part of our data-generation scripts, and we thereby had access to ground-truth depth maps for both datasets. The models were trained on the same dataset combinations as in Experiment 1-2, however we removed the augmentation as we had not implemented depth augmentations.

The depth estimation performance was evaluated using *accuracy withing threshold*, as described in Section 2.9. Note also that all models were evaluated using only CARLA data.

4.1.4.2 Results

	Segmentation		*Depth*		
Training dataset	Mean IoU	Weighted IoU	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Mapillary	0.458 (+7.8%)	0.817	0.320	0.572	0.684
Mapillary+CARLA	0.520 (+10.9%)	0.854	0.295	0.542	0.679
CARLA	**0.717 (+25.3%)**	**0.960**	**0.775**	**0.806**	**0.816**

Table 4.3: Experiment 1-3 - Evaluation of multi-task perception model. The table shows metrics for both the segmentation and depth branches of the model, evaluated on CARLA data. Mean IoU is also compared to the Mean IoU from Experiment 1-2 on the corresponding dataset. Depth is estimated using *accuracy within threshold*, where the set threshold is presented in the column title. Higher values are better in all the metrics. The numbers in parenthesis shows change in Mean IoU from non-depth equivalents, non-augmented if available.

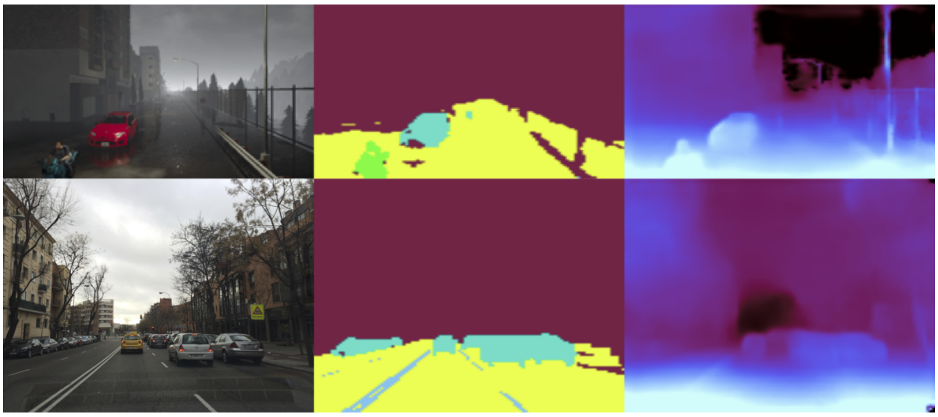


Figure 4.2: Final perception model samples. This image illustrates the performance of the final perception architecture trained on Mapillary+CARLA data. The top image is from CARLA during rainy weather, while the bottom image is from the real-world Mapillary test set.

4.1.4.3 Discussion

The results show that the segmentation model improves its performance on every dataset variation. On the CARLA dataset, the increase is most drastic, with

25.3% increase in accuracy. As the CARLA data is the only dataset with perfect ground truth depth annotations, this may have impacted the results as well.

The depth performance is just an indication of how well the models performed. As the model trained on only CARLA data was evaluated on a similar dataset, it has an advantage when compared to the other two models. Therefore, the improvements in segmentation compared to models without depth estimation is the most interesting part of these results.

Compared to what was reported by Standley et al. [43] with a 4.17% increase in performance when training segmentation with depth, we saw significantly better results with an average of 14.67% increase in performance. We believe that the perfect CARLA depth data was an important factor for our much better performance, as the increase for Mapillary is similar to Standley et al.'s.

4.1.5 Discussion

The final model was trained on a combination of Mapillary and CARLA data, as we wanted a model that performed good in both real-world and simulated environments. A depth estimation branch was also included as it improved the overall segmentation performance.

Choosing a perception model by training on different architectures and dataset variations was difficult, however we believe we found a model with overall good performance. We had some issues with dataset differences, which again impacted the resulting metrics for each model, for instance depth estimation, where the CARLA and Mapillary data formats had some differences. Depth performance was therefore difficult to compare between models and datasets.

Furthermore, we did not continue with augmentations after seeing good generalization results on Experiment 1-2. In retrospect, we would have included augmentation in the final depth model as well.

4.2 Experiment 2: Driving model

In this experiment, we tested the performance of various driving models based on the best-performing perception models from Experiment 1. Instead of doing static testing, i.e., predicting frame-by-frame vehicle controls from an unseen dataset, we ran the models in CARLA. The models are given control of a vehicle by applying the model outputs to the vehicle controls. The same vehicle is fitted with a camera and can provide all the inputs the model requires. A consequence of this real-time evaluation is that each prediction will affect the state of the next input, meaning that one incorrect prediction could possibly lead to unrecoverable errors.

4.2.1 Setup

4.2.1.1 Driving Models and Training Data

In Experiment 1, we explored several different architectures for building a perception model to be used for our system. This experiment seeks to verify that such perception models are able to provide good enough scene understanding information such that a driving model can drive based on it. We compare the best-performing perception models to a baseline driving model that only uses raw RGB images as input. A brief summary of the models is shown in Table 4.4.

Model	Perception model	Perception data
RGB	*N/A*	*N/A*
S-CARLA	Segmentation	CARLA
S-Mapillary	Segmentation	Mapillary
S-Combined	Segmentation	Mapillary+CARLA
SD-CARLA	Segmentation + depth	CARLA
SD-Mapillary	Segmentation + depth	Mapillary
SD-Combined	Segmentation + depth	Mapillary+CARLA

Table 4.4: Experiment 2 - Overview of evaluated driving models. Perception models and data are explained in Experiment 1. RGB does not use any perception model, indicated by *N/A* in the table.

The driving dataset used for this experiment comes from CARLA, and was collected as described in Section 3.2.1. We did, however, not collect the driving data using multiple camera angles; only from a height of 2.3 meters, camera pitch of $5°$ downwards, and field of view of $90°$. This experiment was conducted before any real-world testing, and the camera configuration bias was irrelevant as we evaluated the models using the same camera configuration as in the training data. Additionally, we did not include augmentation of the driving data as we wanted to keep the training fast, simple, and reproducible.

4.2.1.2 Evaluation and Metrics

Haavaldsen et al. [17] conducted evaluation of models in a similar way to what we wanted; therefore, we used their scenario runner source code as a base for our evaluation. The scenario runner works as follows: It loads the model to be evaluated, and creates an accompanying vehicle in an empty CARLA town. The runner inputs the information that the model needs from the vehicle into the model, and sends the outputs of the model to the vehicle's controller.

The scenario runner is designed to create scenarios which each model should try to complete. Each scenario consists of spawning the vehicle at a start location, then giving it HLCs to navigate through a predefined route. The route consists of a set of waypoints, which each specifies a HLC to activate when the vehicle passes it. The performance of a model is measured by its mean completion rate (MCR), as mentioned in Section 3.2.4. The scenario runner calculates the completion rate of a route by summing the distance between all the waypoints it has passed. A scenario ends either when the vehicle completes the route, or if any of the following happens: the vehicle gets stuck, it drives in the oncoming lane without returning after five seconds, or it ignores a given HLC.

The scenario runner runs each model through the same routes in different weather, and repeats all route and weather combinations three times. The reason for repeating each configuration multiple times is that CARLA 0.9.9 is not perfectly deterministic, which in some cases resulted in slightly different results with the same configurations.

4.2.1.3 Environments and Routes

The models were tested in two environments, Town02 and Town07. There are three routes in each environment, which the models will try to complete in six different weather conditions. Three of the weather conditions have already been observed in the training data, while the three remaining are unknown to the model. The training data only contain samples from day-time weathers, but two of the unknown weathers are at midnight.

Town02, seen in Figure 4.3, has many similar features to Town01, the town used for collection of the training data. It is situated in an urban setting, and its roads only meet in perpendicular intersections — just as in Town01. It includes both a fenced parking lot with a few parked cars, and a larger paved area with a stationary police car. The three routes in Town02 are shown in Figure 4.4. Route

1 and 2 partly overlap, and feature several HLC following tasks in intersections. Route 3 starts with a sharp turn, and includes a long stretch of high speed limit.

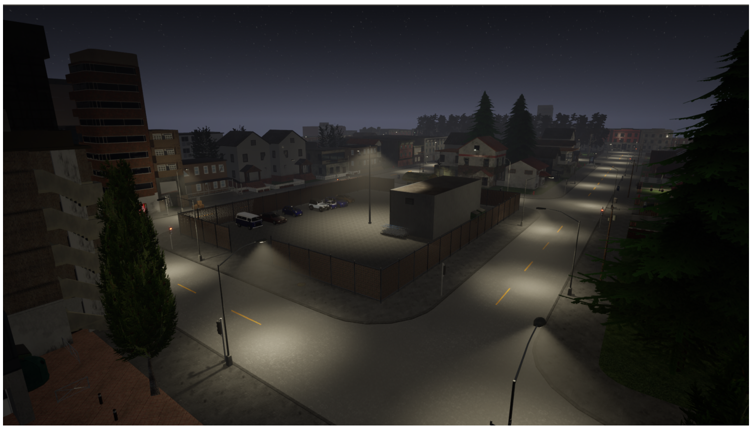


Figure 4.3: A picture from CARLA's Town02.

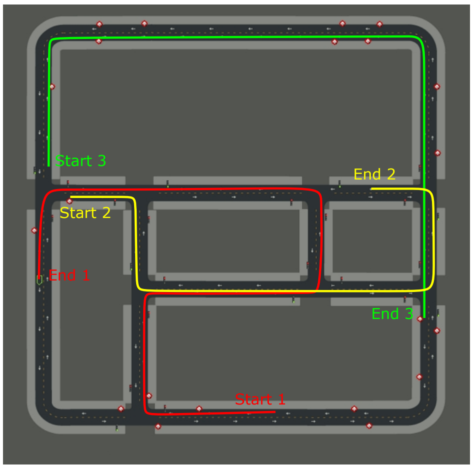


Figure 4.4: Evaluation routes in CARLA's Town02.

Town07, shown in Figure 4.5, is quite different from the two others. It is located in a rural area with narrow roads (some without any centre marking), fields, and barns. Some of the intersections, which can be seen along with the routes in

Figure 4.6, are perpendicular, but many are not. Route 1 covers a long section of road in a curved and hilled forest road. Route 2 starts with a sharp turn, before turning and continuing over a bridge. Route 3 starts in the middle of a field, with limited sight in its intersections.

Figure 4.5: A picture from CARLA's Town07.

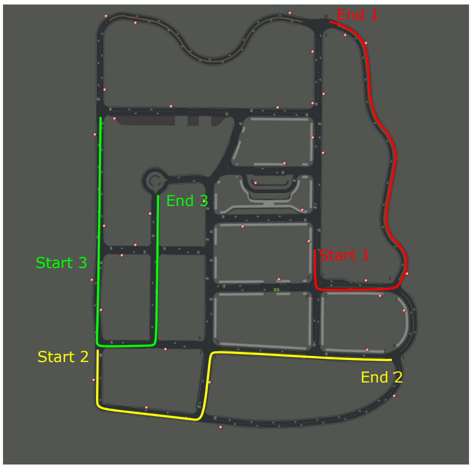


Figure 4.6: Evaluation routes in CARLA's Town07.

4.2.2 Results

4.2.2.1 Quantitative results

The results in Table 4.5 presents the MCR of the driving models in all the routes and weather conditions from the evaluation. Table 4.6 shows the MCR calculated from all runs, revealing the final score of each model.

	Seen weather			*Unseen weather*			
Model	**Clear (D)**	**Rain (D)**	**Wet (S)**	**Clear (N)**	**Rain (N)**	**Fog (S)**	**Mean**
RGB	100.00 %	28.72 %	36.05 %	100.00 %	11.22 %	100.00 %	62.67 %
SD-CARLA	100.00 %	43.30 %	55.36 %	100.00 %	23.46 %	44.96 %	61.18 %
S-CARLA	93.83 %	7.23 %	43.51 %	100.00 %	24.61 %	66.66 %	55.97 %
SD-Mapillary	88.51 %	42.16 %	67.22 %	100.00 %	11.59 %	24.91 %	55.73 %
SD-Combined	93.53 %	9.92 %	47.42 %	100.00 %	9.92 %	46.53 %	51.22 %
S-Combined	90.71 %	44.02 %	23.12 %	72.12 %	2.72 %	7.23 %	39.98 %
S-Mapillary	72.12 %	43.30 %	40.85 %	39.34 %	2.72 %	2.72 %	33.51 %

(a) Results for *Town02*.

	Seen weather			*Unseen weather*			
Model	**Clear (D)**	**Rain (D)**	**Wet (S)**	**Clear (N)**	**Rain (N)**	**Fog (S)**	**Mean**
SD-CARLA	84.95 %	61.14 %	84.95 %	60.81 %	88.88 %	17.19 %	66.32 %
SD-Mapillary	77.28 %	77.28 %	55.54 %	51.61 %	33.33 %	14.84 %	51.65 %
SD-Combined	50.52 %	61.14 %	57.39 %	38.60 %	66.67 %	33.33 %	51.27 %
S-Combined	77.83 %	55.10 %	55.10 %	17.32 %	50.65 %	33.33 %	48.22 %
RGB	44.49 %	44.49 %	44.49 %	44.49 %	49.75 %	44.49 %	45.37 %
S-CARLA	55.10 %	55.10 %	57.39 %	17.32 %	17.32 %	61.87 %	44.02 %
S-Mapillary	51.61 %	50.52 %	44.49 %	55.10 %	44.49 %	0.00 %	41.04 %

(b) Results for *Town07*.

Table 4.5: Experiment 2 - Results of driving evaluation in CARLA. Mean completion rate (MCR) in (a) *Town02* and (b) *Town07*, in six weather conditions. See Table 4.4 for explanation of the models. Day, Sunset and Night is shortened to *D*, *S*, *N* respectively. The individual cells are colored on a scale where green is the best, and red is the worst.

Model	Combined MCR
SD-CARLA	63.75%
RGB	54.02%
SD-Mapillary	53.69%
SD-Combined	51.25%
S-CARLA	50.00%
S-Combined	44.10%
S-Mapillary	37.27%

Table 4.6: Experiment 2 - Overview of MCR for all tested models. The MCR of a model is the mean completion rate from all its runs (all weather configurations and all routes).

With a combined MCR of 63.75%, SD-CARLA performed significantly better than all other models. In Town01, SD-CARLA performed considerably better than RGB in most weathers, but worse overall because it only had a 44.96% completion rate in Fog (S). In Town02, however, it had the highest score; much better than RGB. RGB got a combined MCR of 54.02%. It was the best performing model in Town01, as it drove perfectly in Fog (S). In Town07, which was completely unknown, RGB's performance degraded considerably; it was only able to complete around half of the routes in each weather condition.

SD-Mapillary had the third best combined MCR, 53.59%, very close to RGB. The model worked well in most situation, and tackled previously seen weather in Town07 well. While not as good as SD-CARLA, it showed impressive results despite not having been trained on CARLA perception data at all. SD-Combined was not able to compete with SD-CARLA and SD-Mapillary, and got a slightly lower combined MCR of 51.25%. In Town01, it struggled with rainy weather, but performed better than RGB in Town07.

S-CARLA got a combined MCR of 50.00%, not too far away from SD-Combined. As with the rest of the segmentation-only models, it was not able to perform as good as the segmentation + depth models. S-Combined ended up with a combined MCR of 44.10%, which indicates that combined segmentation data might benefit driving performance. The worst overall model was S-Mapillary, with a combined MCR of only 37.27%. It worked decently in known weather but struggled with a lot of the unknown weather types.

4.2.2.2 Qualitative results

In addition to providing quantitative results for each model, we recorded a video of SD-CARLA, the best performing model overall. The video shows the model in some of the evaluation routes and weather conditions where it performed the best. The video itself can be accessed at `https://youtu.be/gf0qNRXXwX0`.

4.2.3 Discussion

The results clearly show that the usage of depth data (in addition to segmentation data) improves driving performance. It might not come as a big surprise that SD-CARLA beats SD-Mapillary, as it is fine-tuned to CARLA. However, we predicted that SD-Combined would be better than SD-Mapillary as it had seen some CARLA data. We are not exactly certain why it performed worse, but the following results of the segmentation-only models may give us a hint of what has happened.

S-CARLA is performing best as expected; but here, S-Combined works better than S-Mapillary. Our intuition is therefore that the CARLA segmentation and Mapillary segmentation go well together, but that the difference between the CARLA depth data and Monodepth2-generated depth data from Mapillary is too large; thus, hindering the prediction quality of the combined model.

The RGB model performed surprisingly well in Town07, considering that it is trained only on driving images from Town01. From Table 4.5 we can see that its good score is mainly from its 100% completion rate of Clear (D), Clear (N), and Fog (S) in Town01. In the other weathers in Town01, the model's performance degraded. Overall, we can conclude that the RGB model is able to generalize to some unseen weathers, but struggles with unseen environments.

4.3 Experiment 3: Real-world validation

This experiment attempts to validate that our results from simulated environments are representative in the real-world. We use model architectures created from the two previous experiments, and deploy them to a real-world vehicle in two different routes in a downscaled urban driving environment.

4.3.1 Setup

4.3.1.1 Driving Models and Training Data

The models used for this experiment were derived from the results of the last two experiments. All models use the Mapillary+CARLA perception data, as well as the multi-task architecture with depth and segmentation.

The data used in the experiment is a combination of CARLA driving data, like Experiment 2 but with multiple camera configurations, as described in Section 3.2.1. An additional dataset was included, which was created from the same driving environment as the real-world evaluation. The first model tested was trained only on CARLA data, while the second model was trained on a combination of CARLA data and real-world SPURV data.

The CARLA dataset included about 66 000 datapoints before HLC balancing and 45 000 datapoints after balancing, where 20% of the data was used for validation. Each datapoint included three images, so the dataset was therefore practically three times the original size. The real-world dataset has 22 530 datapoints, where also here 20% was used for validation. Both the CARLA dataset and real-word SPURV data is further described in Section 3.2.1. As an extra measure for making the driving models robust, we apply the augmentation described in Section 3.2.2 when training.

4.3.1.2 Evaluation and Metrics

The real-world experiments were conducted using the SPURV Research vehicle (shown in Figure 4.7, which is further described in Section 2.2.3. The vehicle is a small four-wheeled vehicle with a single forward-facing camera, and a GPU for running machine learning models.

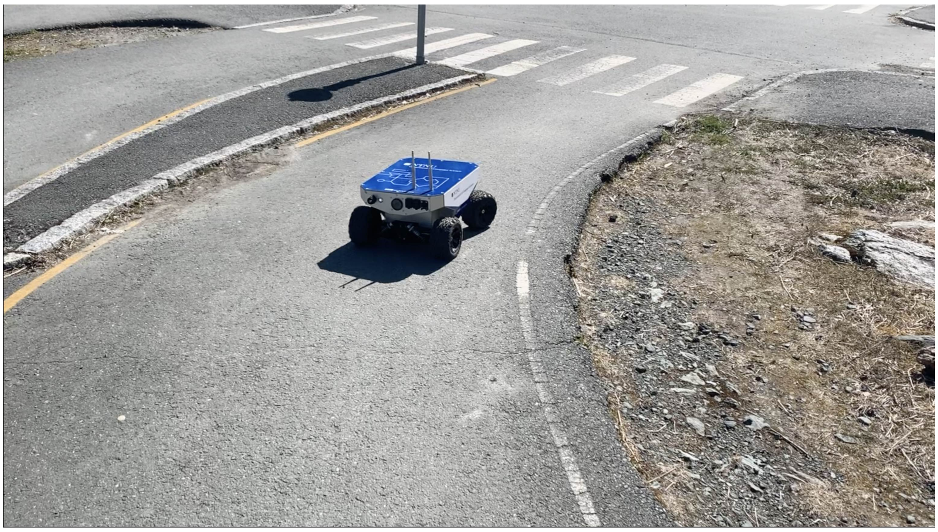


Figure 4.7: SPURV Research vehicle during evaluation. An image of the SPURV Research vehicle while driving on our test track. The driving environment is a downscaled urban environment with intersections and traffic lights.

The testing track is a bike path in a downscaled urban driving environment, created to teach bike riding in an urban environment. A bird's-eye view of the routes used is shown in Figure 4.8.

Measuring performance was done by counting the number of interactions we had with the vehicle during driving. The performance measure was inspired by Codevilla et al. [6], where an interaction was counted if the vehicle drove outside its lane for more than four seconds, or if it was about to crash with an obstacle. Codevilla et. al also included more measures to test HLC performance, however we excluded the same metrics because good HLC performance was not a focus of this thesis.

4.3.1.3 Environments and Routes

Two different routes were used for evaluation, one for simple lane following and the other for more complex behavior and variable light conditions. Both routes are displayed in Figure 4.8.

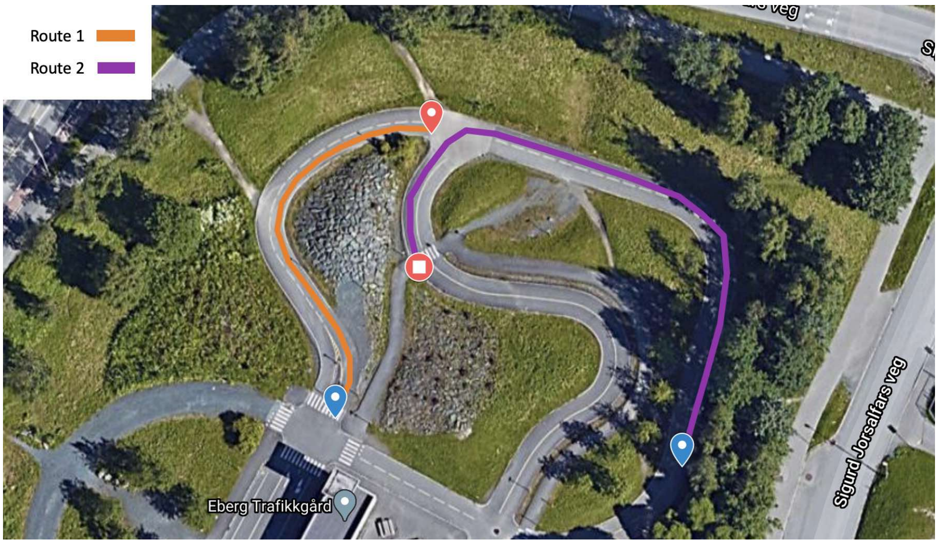


Figure 4.8: Real-world evaluation routes. An aerial view of the training track as well as mapping of the different routes. Blue waypoints mark the beginning of the route, while the red and white squares mark the end. Each route was driven in both directions every other lap.

Route 1 is a simple lane-following route, where we verify if the vehicle is able to follow the correct side of the road, as well as stay on the road even when the lane markings are worn off. Figure 4.9 shows a sample of where the lane line has worn off, and the vehicle still must stay inside the road. Route 2 is a longer and more complicated route that includes shadows from trees and an intersection for testing HLC.

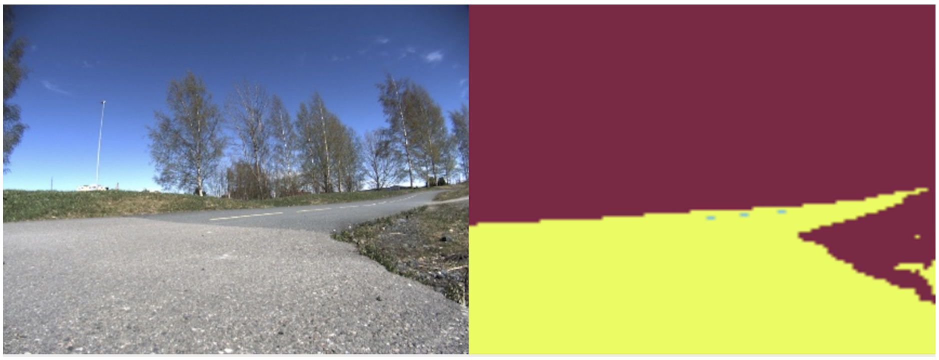


Figure 4.9: Real-world driving data. A visualization of a road without lane lines. The left image is the original image from the vehicle's camera, while the right image is the resulting segmentation after processing by the perception model. Yellow is categorized as road, blue is lane lines and red is unlabeled.

4.3.2 Results

4.3.2.1 Quantitative results

Each training dataset was tested for four laps on both routes. The metric measures number of interactions required to complete the lap.

Training dataset	**Lap 1**	**Lap 2**	**Lap 3**	**Lap 4**	**Mean**
CARLA	1	1	1	7	2.5
Combined	2	0	2	1	**1.25**

(a) Results for route 1.

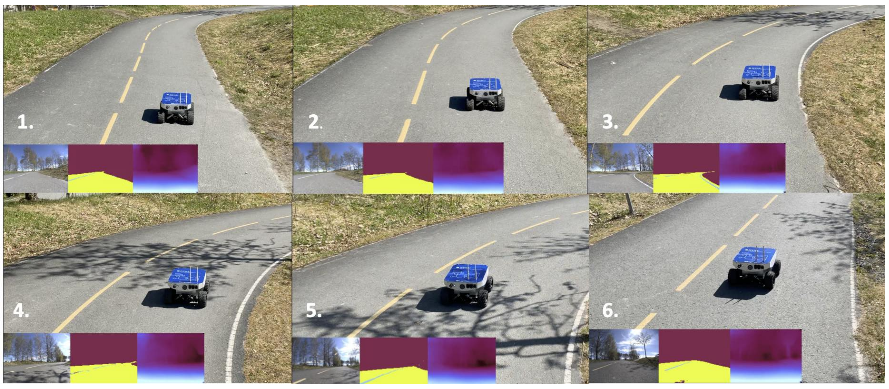
Training dataset	**Lap 1**	**Lap 2**	**Lap 3**	**Lap 4**	**Mean**
CARLA	2	0	1	0	**0.75**
Combined	6	6	N/A	N/A	6

(b) Results for route 2.

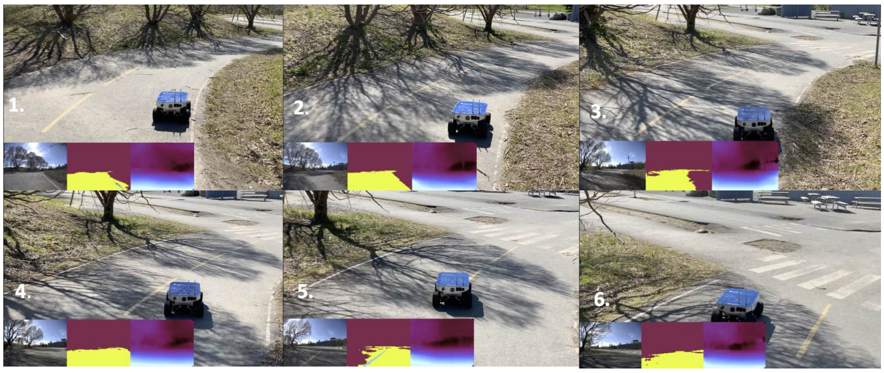
Table 4.7: Experiment 3 - Results from real-world evaluation. The tables show the number of interactions from real-world driving with one model trained only in simulation (CARLA) and one trained with a combination of simulation and real-world data (Combined). N/A in this case means that the lap was canceled because of poor performance. A final model trained only on real-world data was not included in the results as it failed to complete most laps.

4.3.2.2 Qualitative results

The following figures show two samples from SPURV evaluation laps. Figure 4.10a shows the vehicle successfully following the correct lane line. The second figure, Figure 4.10b shows a failed attempt, where the SPURV drives outside its lane.



(a) The SPURV successfully following the road.



(b) The SPURV had issues with segmentation of the road because of shadows, and ended up driving into the wrong lane.

Figure 4.10: Experiment 3 - Driving samples. These figures show two different driving events. Both images are read from top left and the sequence is numbered. The driving model was trained only on CARLA driving data.

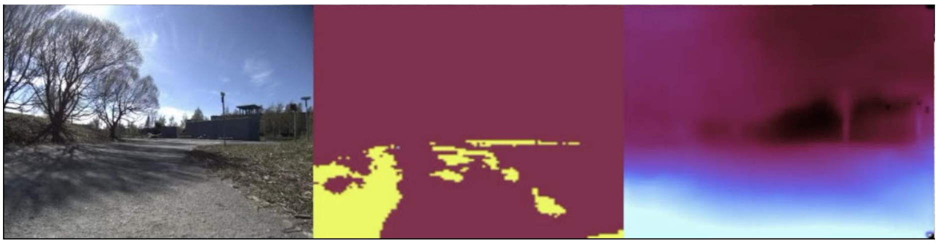


Figure 4.11: Experiment 3 - Real-world segmentation issues. An image from the SPURV's perspective driving a part of Route 2 with a lot of shadow. The segmentation in the center image is very poor, which again caused the SPURV to drive out of the road. Note that the segmentation in similar situations often were much better, where this is an outlier.

A video that displays a summary of the real-world driving is available at: `https://youtu.be/gf0qNRXXwX0?t=46`. Note that the models driving in the video is trained exclusively on CARLA driving data, unless stated otherwise, and that we let the model continue driving after completing its route.

4.3.3 Discussion

Overall, the results show that the model trained in simulation handles real-world adequately, especially Route 2, where it manages two laps without any interaction, even including a correct HLC turn. The model trained only on real-world data performed worst, and was therefore removed from the resulting table, as it did not complete any laps. Finally, the model trained on combined data performed similarly to only CARLA data.

We found testing in the real-world to be much more unpredictable than in simulation. Models we expected to perform good (trained on similar data to the real-world testing) performed very poorly, while the model trained in simulation performed very well. The exact results, with drastically different results between the models should therefore be taken with a grain of salt. The experiment's main contribution is therefore the validity that the model's trained in simulation can drive in the real-world as well to some extent.

As seen in Figure 4.11, the perception model had some issues with shadows, which further made the driving model perform poor in situations with a lot of shadows. This impacted especially the *Combined* model, as it was trained on very noisy steering angles (see triangular noise in Section 3.2.1), which in turn made the driving model quickly drive out of the road when the perception was inconclusive. The model trained only in simulation on the other hand had less sharp turns, and was therefore often able to correct itself before driving out of the road.

Chapter 5

Discussion

5.1 Simulation to real-world domain transfer

5.1.1 Perception model

The choice of perception model was important as it was the main tool for increasing the generalizability of our finished architecture. We therefore did extensive testing on everything from encoder-decoder variations and datasets, to additional perception tasks. Luckily, we found a library with implementations of the most popular encoder and decoder architectures, which allowed us to test many variations. Finally, we found the MobileNet encoder with a U-Net decoder to be the best combination of computational cost and accuracy.

Finding 1: MobileNet with U-Net achieves the best Mean IoU relative to computational cost of the tested architectures.

A disadvantage of selecting U-Net was that the architecture did not allow for using the immediate representation as seen in Hawke et al. [19]. The architecture heavily depends on information from the encoder layers, which in turn means that no single point will include all the compressed information by the encoder.

We further expanded the perception model to use a technique described by Hawke et al. [19] where we include another perception task in order to improve the model's overall understanding of the image. This improved the overall results significantly, with the multi-task model improving both existing segmentation accuracy and overall driving performance, even outperforming the driving model trained on raw images.

Finding 2: Perception models trained to predict both segmentation and depth maps achieves up to 25.3% better Mean IoU score for segmentation compared to models predicting only segmentation.

The improved results when including depth information was, however, only the case for models trained exclusively on either CARLA or Mapillary data, not combined. As described in Section 3.1.2, there were differences in the depth formats from Mapillary and CARLA, which in turn seems to decrease the overall performance of the segmentation model. A possible improvement could therefore be to better normalize the depth maps between CARLA and Monodepth2. Using Monodepth2 to generate depth maps from CARLA data could also be a solution, as we can then guarantee that the data formats are equal between datasets.

Finding 3: The data format when combining simulated with real-world depth data is critical for the overall perception performance.

5.1.2 Driving model

We tested a total of seven driving models, where the choice of perception model made big impacts. We found that using only segmentation maps, the driving models were stripped of too much scene information to drive properly; the original RGB images worked much better.

Finding 4: A driving model only trained on segmentation maps will not perform as good as just using the original RGB images.

The driving performance was tightly connected with the perception training data we used: CARLA data was enough to drive decently, but pure Mapillary data made it difficult for the model to drive in a simulated environment.

Finding 5: Using only Mapillary segmentation data for the perception model results in a driving model struggling with anything but the easiest routes.

When we introduced depth estimation to our perception model, we saw clear driving improvements. It seems that the addition of a predicted geometry of the scene helps the driving model learn. In all but one case, the models using depth estimations performed better than all others, regardless of whether they had a perception model trained on CARLA, Mapillary, or combined data.

Finding 6: Driving models utilizing depth predictions drives better than all but one of those with segmentation only.

Finally, we found that using the resulting image representation generated by the perception model makes it easier to learn driving, in comparison to the original RGB images.

Finding 7: Using both estimated segmentation and depth maps yields mostly better driving performance than the original RGB images, in addition to being more general.

5.1.3 Real-world driving

In Experiment 4.3 we show that a driving policy trained solely in CARLA can drive on a real road as well. As the driving model only requires semantic segmentation and depth maps, it should be able to drive in any environment where a perception model can perform sufficiently well.

Finding 8: Our perception model provides segmentation and depth maps which are domain independent, and makes domain transfer from simulation to real world feasible.

Our initial attempts at transferring our models to the real world did not work as well, as we had not yet started using multiple camera configurations in our training data. This resulted in a driving policy that was overfitted to the specific camera configuration we had set up in CARLA, which differed significantly from the one of the SPURV. The static camera configuration we used from the start is the one shown in Figure 3.6a, while the SPURV's camera is approximated in Figure 3.6d.

Finding 9: A driving policy can easily overfit if the camera configuration in the training data is static.

After training with the additional camera configurations shown in Figure 3.6, we went from seemingly random behavior to a model that understands the environment around itself and use this to drive. The model was able to follow roads without any forks or intersections without problem, and were able to follow high-level command with moderate success, even if it only had been trained on driving from a CARLA autopilot.

Finding 10: A driving model with perception data from both CARLA and Mapillary, and driving data only from CARLA, can follow roads and will try to respect HLCs.

5.2 Comparison to related work

Our model combines ideas and techniques from several existing papers, thus making it natural to compare our results to theirs. Xiao et al. [46] evaluates several methods for incorporating semantic segmentation and depth maps into a driving model. The paper's focus was mainly on the usage of the maps, and not their origin, i.e. if they are predictions or ground truth data. They showed that using channel-wise concatenation of ground-truth segmentation and depth maps as input to a CIL network performs better than both raw RGB images and segmentation alone. Our results showed similar relative performance, where our channel-wise concatenated predicted segmentation and depth maps performed best overall.

Müller et al. [34] did not research the use of multimodal input data; they focused on predicting a binary segmentation of images — either road or not road — to use as input for their CIL network. We extend their idea of a perception module to segmentation with five classes and depth prediction within the same network. Müller et al.'s system was able to learn to drive using a perception module trained on a real-world segmentation dataset, and with a driving policy trained only on driving in CARLA. They showed that their control abstraction using waypoints is essential to their approach. We only abstracted away our vehicle control to steering angle and target speed, and we might have experienced performance gains if we were to implement an abstraction like theirs. However, we showed that our approach, too, can learn to drive in both simulation and in real-world environments from similar training data.

The driving model presented in Hawke et al. [19] extended the idea of a perception module, which included prediction branches for reconstructing the input image, semantic segmentation maps, and depth maps. The perception module did not output the predicted values, but rather an intermediate representation of the scene which is common for all the prediction branches. The output of their driving policy is a motion plan, and thus more abstract than our outputs. They accomplished very impressive results in real-world driving, but with no focus on domain transfer. It is plausible that our approach would have benefited from their approaches of using intermediate layers and vehicle control abstractions.

Our work is based on previous work originating from the NAPLab research group, specifically Haavaldsen et al. [17] and the same authors' master thesis. Their work focused mostly on how to create a driving model that could tackle complex simulated environments, and suggested future work on how to improve upon this. We decided to go in a slightly different direction, instead focusing on domain transfer from simulation to the real world, but with simpler simulated environments. Future research within the NAPLab could utilize both our and Haavaldsen et al.'s findings, and we believe that a combination of these will provide directions for further research in the field.

5.3 Fulfillment of research questions

RQ 1 In the first research question, we wanted to see if a end-to-end model can learn to drive solely on segmentation maps, or segmentation + depth maps, generated by a pre-trained perception model. Our findings show that our end-to-end driving model learned to drive, both in simulation and in real-world scenarios. In most cases, the driving model using the pre-trained perception models performs best, better than the baseline RGB models. Depth estimations is also shown to improve the baseline results by a significant margin.

RQ 2 The second research question focused on the generalizability of our perception model, where we asked if a perception model trained only on real-world data could generalize to simulated environments. We found that a model trained only on the Mapillary dataset was able to generalize to CARLA images to some extent. However, we saw that a combination of Mapillary and CARLA data performed better on both real and simulated images.

RQ 3 The last research question sought to determine if an end-to-end model trained exclusively on driving data from simulation, could successfully drive in the real-world. We showed that by first using a perception model trained on public datasets, we were able to transfer the entire driving model trained exclusively in CARLA to a real-world environment. Using the SPURV Research vehicle, the model was able to follow a path in clear weather, and handle simple visual obstacles, specifically shadows. It was also able to adhere to the provided High Level Commands in most situations.

5.4 Potential shortcomings and reflection

While we are satisfied with our research and results, we wanted to address any potential shortcomings of this thesis. To design our architecture, we had to obtain knowledge on many subjects: autonomous vehicles, conditional imitation learning, segmentation tasks, monocular depth estimation, vehicle control, simulation, among others. Many possible directions were left unexplored, simply because there were time constraints and that we wanted to demonstrate a system that could drive both in simulation and in the real world. In the process of achieving this goal, we had to skip investigating many possible directions, where some could have helped us considerably.

Müller et al. [34] and Hawke et al. [19] both had a greater focus on the driving outputs, and it is shown to improve their driving performance significantly. Had we implemented a similar abstraction from the start, maybe we could have ended up with even better performance. However, we did not realize the full extent of their benefits before we were well into our own evaluation; and we deemed it too late to make any major design changes.

We made several design choices during this project, where many were inspired by work of others. However, we did not verify every choice before applying it to our work, and we cannot prove that they improved our system; only that they seemed to be good based on other author's results. Again, a possible solution could have been to properly investigate the effects of these choices early on, for example by conducting preliminary experiments which tested them.

Here we present a few specific improvements that we believe might have impacted the overall driving results:

- Representing the output steering value of the driving model using a more abstract representation, perhaps using motion plans such as Hawke et al. [19].
- Include motion/temporal information in the perception model or driving model, through optical flow, or recurrent networks respectively.
- Equal data formats for depth data in Mapillary and CARLA data.
- Using a decoder network optimal for compressing the perception semantics into a single network layer, which in turn would allow us to use only the encoded intermediate representation for the driving model.

Chapter 6

Conclusion and Future Work

In this chapter, we conclude based on the results and findings of this thesis, and suggest directions for future work.

6.1 Conclusion

We conclude our work by looking back at our initial goal: to make an end-to-end autonomous driving system that can utilize both simulated and real-world data to drive. This was a bold goal as domain transfer from simulation to the real world has been extensively researched and has shown to be a very difficult problem. During this thesis, we have mainly explored how existing computer vision datasets and the simulated CARLA environment can be used to reduce the amount of real-world driving data required for training autonomous vehicles. We used a separate perception model that was pre-trained on these computer vision datasets, and thereby was able to interpret difficult driving scenes in both simulation and the real world. This allowed us to train a separate driving model in simulation, while simplifying the specifics of each domain into segmentation and depth maps. Finally, we showed how this driving model trained solely in simulation was able to successfully drive on a bike path with lane lines and intersections.

A lot of research has been conducted using end-to-end models for autonomous driving, both in the real world and in simulated environments. However, few of these papers have focused on a combination of the two. We used inspiration from recent papers that had state-of-the-art results from both the simulated and real-world domain, and attempted to combine the successful approaches from each domain into a domain-independent solution. There are still many approaches left to explore, such as using an immediate driving scene representation for the driving model input, as shown by Hawke et al. [19], or focusing on better and more abstract steering and velocity output representations.

As with any work, this thesis also has its set of shortcomings. Looking back, we have found several situations where our approach could have been improved. Inconsistent depth map representations between simulated and real-world environments, not fine-tuning the perception model to our real-world environment, as well as selecting a decoder architecture that did not easily allow for directly using the compressed scene representation are a few of these. Often, these were early design decision that were difficult to revert because of the time constraints on the thesis. In situations where we encountered such shortcomings, we attempted to highlight them so they could be improved upon in future work.

During the work with this master's thesis, we wrote a paper covering the parts of this thesis related to driving in the CARLA simulator. The paper is currently being reviewed for the Colour and Visual Computing Symposium 2020 (CVCS 2020)[1], and is available in full in Appendix A.

6.2 Future work

We believe our work has shown promising results, and is an interesting approach for future work in end-to-end driving for autonomous vehicles. There are still many directions left unexplored, as well as plenty of possible improvements to the directions we did explore. Throughout this thesis, we have highlighted situations were the results could have been improved. In this section, we will attempt to extract a few of the directions we believe would have the largest impact on the results, and therefore are good candidates for future work.

A lot of this master's thesis was inspired by Hawke et al. [19], and we implemented a few of their approaches with good results. However, there are still several approaches that we did not try. For our perception model, we found four

[1] `https://www.cvcs.no/`

important improvements in Hawke et al.'s work:

- Only include the encoded features of the perception model as input to the driving model, which we believe would significantly increase the end-to-end model's efficiency and maybe even driving performance.
- Use a perception model trained on an additional RGB reproduction task, in addition to the existing segmentation- and depth maps, to improve the overall scene understanding of the driving model.
- Train the perception model on several datasets (e.g. Cityscapes and Mapillary) to improve the generalizability of the perception model.
- Include motion information using a separate optical flow model to incorporate temporal information.

Another direction we did not focus on, but that we later understood the importance of, was the output representation of our driving model. Our approach used the target speed and steering angle as output representations, while Hawke et al. and Müller et al. [34] attempted to improve upon this representation with more abstract outputs. Two approaches worth exploring based on these two papers are:

- Output the speed and steering as a parameterized line for the motion plan on the current and future timesteps, as opposed to only a single value on the current timestep for more stable and robust vehicle control.
- Use estimated waypoints on the road instead of steering angle, which could improve transferability of steering between domains, where the model does not have to take into account differences in speed and steering feedback.

During real-world driving, our driving model had issues with instabilities when encountering inconclusive perception results. When the perception model had temporary issues with interpreting the driving scene, the driving model had issues handling the situation, as it lacked context. An interesting approach could be to include temporal information to the model — perhaps similar to how Haavaldsen et al. [17] used a recurrent neural network to give their driving model temporal information.

Finally, we would like to conclude with the hope that future work can use what we have learned during this thesis, and continue exploring domain-transfer for end-to-end autonomous driving. Our source code is available on GitHub.[2][3]

[2]`https://github.com/AudunWA/master-thesis-code`
[3]`https://github.com/AudunWA/autonomous-vehicle`

Bibliography

[1] I. Alhashim and P. Wonka. High quality monocular depth estimation via transfer learning. *arXiv:1812.11941 [cs]*, Mar 2019. URL `http://arxiv.org/abs/1812.11941`. arXiv: 1812.11941.

[2] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv:1511.00561 [cs]*, Oct 2016. URL `http://arxiv.org/abs/1511.00561`. arXiv: 1511.00561.

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars. 2016.

[4] Y. Cao, T. Zhao, K. Xian, C. Shen, Z. Cao, and S. Xu. Monocular depth estimation with augmented ordinal depth relationships. *arXiv:1806.00585 [cs]*, Jul 2019. URL `http://arxiv.org/abs/1806.00585`. arXiv: 1806.00585.

[5] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv:1412.3555 [cs]*, Dec 2014. URL `http://arxiv.org/abs/1412.3555`. arXiv: 1412.3555.

[6] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy. End-to-end Driving via Conditional Imitation Learning. *arXiv:1710.02410 [cs]*, Oct. 2017. URL `http://arxiv.org/abs/1710.02410`. arXiv: 1710.02410.

[7] F. Codevilla, E. Santana, A. M. López, and A. Gaidon. Exploring the Limitations of Behavior Cloning for Autonomous Driving. *arXiv:1904.08980 [cs]*, Apr. 2019. URL `http://arxiv.org/abs/1904.08980`. arXiv: 1904.08980.

[8] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. *arXiv:1604.01685 [cs]*, Apr 2016. URL `http://arxiv.org/abs/1604.01685`. arXiv: 1604.01685.

[9] P. de Haan, D. Jayaraman, and S. Levine. Causal confusion in imitation learning. *arXiv:1905.11979 [cs, stat]*, Nov 2019. URL `http://arxiv.org/abs/1905.11979`. arXiv: 1905.11979.

[10] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[11] H. M. Eraqi, M. N. Moustafa, and J. Honer. End-to-End Deep Learning for Steering Autonomous Vehicles Considering Temporal Dependencies. 2017.

[12] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics: the kitti dataset. *The International Journal of Robotics Research*, 32:1231–1237, 09 2013. doi: 10.1177/0278364913491297.

[13] C. Godard, O. Mac Aodha, M. Firman, and G. Brostow. Digging into self-supervised monocular depth estimation. *arXiv:1806.01260 [cs, stat]*, Aug 2019. URL `http://arxiv.org/abs/1806.01260`. arXiv: 1806.01260.

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[15] M. Gualtieri, A. t. Pas, K. Saenko, and R. Platt. High precision grasp pose detection in dense clutter. *arXiv:1603.01564 [cs]*, Jun 2017. URL `http://arxiv.org/abs/1603.01564`. arXiv: 1603.01564.

[16] D. Gupta. Image segmentation keras : Implementation of segnet, fcn, unet, pspnet and other models in keras., 2020. URL `https://github.com/divamgupta/image-segmentation-keras/blob/31d1ba660ec16d6032d8719841c4f00c6bf934b0/keras_segmentation/data_utils/augmentation.py`.

[17] H. Haavaldsen, M. Aasboe, and F. Lindseth. Autonomous Vehicle Control: End-to-end Learning in Simulated Urban Environments. *arXiv:1905.06712 [cs]*, May 2019. URL `http://arxiv.org/abs/1905.06712`. arXiv: 1905.06712.

[18] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, June 2000. ISSN 1476-4687. doi: 10.1038/35016072. URL `https://doi.org/10.1038/35016072`.

[19] J. Hawke, R. Shen, C. Gurau, S. Sharma, D. Reda, N. Nikolov, P. Mazur, S. Micklethwaite, N. Griffiths, A. Shah, and A. Kendall. Urban Driving with Conditional Imitation Learning. *arXiv:1912.00177 [cs]*, Dec. 2019. URL `http://arxiv.org/abs/1912.00177`. arXiv: 1912.00177.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv:1512.03385 [cs]*, Dec 2015. URL `http://arxiv.org/abs/1512.03385`. arXiv: 1512.03385.

[21] S. Hochreiter and J. Schmidhuber. Long Short-term Memory. *Neural computation*, 9:1735–80, 1997. doi: 10.1162/neco.1997.9.8.1735.

[22] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861 [cs]*, Apr 2017. URL `http://arxiv.org/abs/1704.04861`. arXiv: 1704.04861.

[23] A. Kastet and R. C. Neset. *End-to-End Steering Angle Prediction for Autonomous Vehicles*. NTNU, 2018. URL `http://hdl.handle.net/11250/2566504`.

[24] A. Kendall, Y. Gal, and R. Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. *arXiv:1705.07115 [cs]*, Apr 2018. URL `http://arxiv.org/abs/1705.07115`. arXiv: 1705.07115.

[25] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, Jan. 2017. URL `http://arxiv.org/abs/1412.6980`. arXiv: 1412.6980.

[26] A. Kirillov, K. He, R. B. Girshick, C. Rother, and P. Dollár. Panoptic segmentation. *CoRR*, abs/1801.00868, 2018. URL `http://arxiv.org/abs/1801.00868`.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*, page 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[28] L. Ladicky, J. Shi, and M. Pollefeys. Pulling things out of perspective. page 89–96, Jun 2014. doi: 10.1109/CVPR.2014.19.

[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov. 1998. doi: 10.1109/5.726791.

[30] Y. Lecun, L. Bottou, G. Orr, and K.-R. Müller. Efficient backprop. 2000.

[31] Y. LeCun, U. Muller, J. Ben, E. Cosatto, and B. Flepp. Off-Road Obstacle Avoidance through End-to-End Learning. 2005.

[32] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. 6(6):861–867, 1993.

[33] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[34] M. Müller, A. Dosovitskiy, B. Ghanem, and V. Koltun. Driving Policy Transfer via Modularity and Abstraction. *arXiv:1804.09364 [cs]*, Dec. 2018. URL `http://arxiv.org/abs/1804.09364`. arXiv: 1804.09364.

[35] G. Neuhold, T. Ollmann, S. Rota Bulò, and P. Kontschieder. The mapillary vistas dataset for semantic understanding of street scenes. In *International Conference on Computer Vision (ICCV)*, 2017. URL `https://www.mapillary.com/dataset/vistas`.

[36] L. Perez and J. Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv:1712.04621 [cs]*, Dec 2017. URL `http://arxiv.org/abs/1712.04621`. arXiv: 1712.04621.

[37] D. A. Pomerleau. *Advances in Neural Information Processing Systems 1*, page 305–313. Morgan Kaufmann Publishers Inc., 1989. ISBN 978-1-55860-015-7. URL `http://dl.acm.org/citation.cfm?id=89851.89891`.

[38] K. Potdar, T. Pardawala, and C. Pai. A comparative study of categorical variable encoding techniques for neural network classifiers. *International Journal of Computer Applications*, 175:7–9, 2017. doi: 10.5120/ijca2017915495.

[39] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. *arXiv:1505.04597 [cs]*, May 2015. URL `http://arxiv.org/abs/1505.04597`. arXiv: 1505.04597.

[40] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, and et al. Imagenet large scale visual recognition challenge. *arXiv:1409.0575 [cs]*, Jan 2015. URL `http://arxiv.org/abs/1409.0575`. arXiv: 1409.0575.

[41] L. Sifre and S. Mallat. Rigid-motion scattering for texture classification. *arXiv:1403.1687 [cs]*, Mar 2014. URL `http://arxiv.org/abs/1403.1687`. arXiv: 1403.1687.

[42] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556 [cs]*, Apr 2015. URL `http://arxiv.org/abs/1409.1556`. arXiv: 1409.1556.

[43] T. Standley, A. R. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese. Which tasks should be learned together in multi-task learning? *arXiv:1905.07553 [cs]*, May 2019. URL `http://arxiv.org/abs/1905.07553`. arXiv: 1905.07553.

[44] D. Sun, X. Yang, M.-Y. Liu, and J. Kautz. Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume. *arXiv:1709.02371 [cs]*, Jun 2018. URL `http://arxiv.org/abs/1709.02371`. arXiv: 1709.02371.

[45] U. Viereck, A. t. Pas, K. Saenko, and R. Platt. Learning a visuomotor controller for real world robotic grasping using simulated depth images. *arXiv:1706.04652 [cs]*, Nov 2017. URL `http://arxiv.org/abs/1706.04652`. arXiv: 1706.04652.

[46] Y. Xiao, F. Codevilla, A. Gurram, O. Urfalioglu, and A. M. López. Multimodal End-to-End Autonomous Driving. *arXiv:1906.03199 [cs]*, June 2019. URL `http://arxiv.org/abs/1906.03199`. arXiv: 1906.03199.

[47] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *arXiv:1906.05113 [cs, eess]*, June 2019. URL `http://arxiv.org/abs/1906.05113`. arXiv: 1906.05113.

[48] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012. URL `http://arxiv.org/abs/1212.5701`.

[49] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia. Pyramid scene parsing network. *arXiv:1612.01105 [cs]*, Apr 2017. URL `http://arxiv.org/abs/1612.01105`. arXiv: 1612.01105.

Appendix A

CVCS Paper: Autonomous Driving in Simulation using Domain-Independent Perception

Autonomous Driving in Simulation using Domain-Independent Perception

Audun Wigum Arbo[*][0000-0002-8572-2402], Even Dalen[*][0000-0001-6032-3678], and Frank Lindseth[**][0000-0002-4979-9218]

Norwegian University of Science and Technology, Trondheim, Norway

Abstract. There have been great advancements within the fields of computer vision and autonomous driving in recent years. Autonomous vehicle systems have typically consisted of mostly handmade algorithms and rules, where neural networks have been assisting in perception tasks. However, increased usage of neural network have shown promising results, and many state-of-the-art systems now utilize these for increasingly complex tasks.
This paper builds upon recent work in building end-to-end networks for autonomous driving. Conditional imitation learning is combined with a perception network that generate high level abstractions from RGB images. We examine the driving performance implications of learning to drive from raw images, semantic segmentation, and depth estimates. Secondly, we propose a driving network which uses raw images to predict semantic segmentation and depth maps of the scene, which furthermore is used to predict output steering angle and target speed for the vehicle. The models are evaluated in CARLA, an open-source simulator for autonomous driving research, in various environments: an urban town, a rural road with surrounding farms and fields, and in greatly varied weather conditions.
Our experiments show that the driving network trained on higher lever abstractions generalize better than a model trained directly on RGB images in simulation, even when the perception model is trained on real-world data. We also show that the perception model trained on several tasks using multi-task learning, leads to better-performing driving policies than learning only semantic segmentation.

Keywords: End-to-end Autonomous Driving · AV Domain Transfer · Multi-task Learning · Conditional Imitation Learning · Semantic Segmentation · Depth Estimation

1 Introduction

Autonomous vehicles have been a popular research domain for many years, and there has recently been large investments from both technology and car companies to be the first to solve the problem. The most prominent approach in recent

[*] These authors contributed equally to this work.
[**] Corresponding author.

years has been the modular approach, where the driving is divided into several sub-tasks such as perception, localization, and planning. The modular approach often results in a very complex solution, where each module has to be fine tuned individually. The scalability of this approach, when the vehicles has to tackle more complex environments, has been questioned.

Another rising approach is the end-to-end approach, where the entire driving policy is generated within a single system. The system then takes sensor-input and converts it directly to driving commands, similar to how humans drive vehicles. End-to-end systems for autonomous vehicles require large amounts of data, and the ability to train on many different scenarios. Therefore, simulated environments have been approached for the ability to train for different scenarios. These environments, however, differ significantly from the real world, and the learned driving policy does not transfer adequately between environments.

This paper attempts to improve the ability for driving policies to be transferred between domains by abstracting away both the perception task, raw throttle and brake control of the vehicle, with focus on perception. Existing public datasets is used for perception in a real-world driving environment, as well as the autonomous vehicle simulator CARLA [8] for both driving and perception data. The ultimate goal of this paper is to reduce the amount of real-world data required to train an autonomous vehicle, by utilizing simulated environments for training.

The paper is structured as follows: Section 2 investigates related work, while additionally providing a brief history of the field itself. Section 3 presents our method, including the data, neural network architectures, and evaluation metrics. Section 4 describes our experiments, their results, and discussion related to these. Section 5 discusses the overall implications of the experimental results, and compare our results with conclusions from related work. Section 6 draws a final conclusion of the work conducted, and addresses the paper's merits, weaknesses, and potential future work.

2 Related Work

Autonomous vehicle control research is mainly divided into two categories: modular approaches and end-to-end approaches. Modular approaches divides the responsibility of driving into several sub-tasks, such as perception, localization, planning, and control. Conversely, end-to-end approaches can be defined as a function $f(x) = a$ where x is any input needed to make decisions — typically sensor data and environmental information — and a, the controls which are sent to the vehicle's actuators.

The end-to-end approach was first demonstrated in the ALVINN project, described by Pomerleau [21]. While being able to follow simple tracks, ALVINN had no means to handle more complex environments. Since then, large advancements have been done within neural networks, resulting in new research within end-to-end vehicle control. Bojarski et al. [3] approaches the problem using modern techniques, and showcase a driving policy capable of driving on both highways and residential roads; in varied weather conditions. More recent approaches [5, 19, 14, 13, 25] are based on Conditional Imitation Learning (CIL), introduced by Codevilla et al. [5] in 2017, where the driving policy is given instructions — high-level commands (HLCs) — on which actions to take (e.g. turn left in next intersection). Codevilla et al. [5] shows that an architecture can be re-used for both simulated and physical environments, but they make no attempt to use the same weights across the two domains. Codevilla et. al outputs a steering angle, and either throttle or brake, which are sent to the vehicle's control systems. Hawke et al. [14] outputs the target speed of the vehicle, leaving the raw throttle and brake adjustments to a lower-level system. Müller et al. [19] proposes to abstract the commands even further; into several waypoints in space. Their model outputs two waypoints, 5 and 20 meters away from the vehicle, which a PID controller uses to control the vehicle's steering and velocity.

Transfer from simulation to real world. A lot of studies have been done on transferring learning from simulation to the real world. Johnson-Roberson et al. [17] used images from the driving game Grand Theft Auto, to train their object detection model, and achieved state of the art performance on the KITTI [9] and Cityscapes [7] datasets. **?**] successfully used simulation to train a model for robotic grasping of physical unseen objects. Among the techniques used was applying randomization in the form of random textures, lighting, and camera position, to enable their model to generalize from the simulated source domain to their physical target domain.

Transferring driving policy between domains also require an abstraction of the perception data. Müller et al. [19] uses a perception model to generate segmentation maps which are forwarded to the driving model, in order to generate similar perception environments for both simulation and real-world. Xiao et al. [25] combines ground-truth segmentation and depth data from CARLA to increase driving performance. Hawke et al. [14] uses an encoder-decoder network with three decoder-heads — segmentation, depth estimation and original RGB reproduction — to maximize the model's scene understanding. Hawke et al. also removes the decoding-process when training their driving policy, making their driving policy model take only the compressed encoding of scene understanding as input. Kendall et al. [18] finds that the performance of such multi-task prediction models depend highly on the relative weighting between each task's loss. Tuning these weights manually is an error prone and expensive process, and they therefore suggest a solution for tuning weights based on the homoscedastic uncertainty of each task. They show that the multi-task approach outperformes separate models trained individually. The uncertainty based weighing was later used by Hawke et al. and produced good results for generating optimal encoding of a driving scene. Depth images has also been proven as a useful approach in other simulation-to-real world knowledge transfers, such as robotic grasping [24, 11].

3 Data and Methods

Our approach consists of two separately trained models: a perception model and a driving model. The reason for this separation is to decouple the task of scene understanding from the task of driving. This opens up the possibility of improving the tasks independently, and we can train the models separately and with different data sets. An important goal was then to make the output of the perception model domain-independent, which in turn makes the driving policy model domain-independent. Domain independence is in this context defined as the ability to generalize between multiple domains (e.g. simulated and real), as well as completely unseen domains.

The perception model is trained on datasets containing RGB images paired with semantic segmentation and depth information. These data sets can contain images not directly related to driving, as they are used to train a general scene understanding.

The driving model is trained on data sets recorded from an expert driver. The data set contains RGB images and driving data such as steering angle, current speed and target speed. The data sets for both models can either be collected from the real world, generated from the CARLA simulator, or a combination of both real-world and simulated data.

3.1 Perception Model

The perception model takes raw RGB images as input, and tries to predict one or more outputs related to scene understanding; always semantic segmentation, and in some experiments an additional depth map. The model has an encoder-decoder structure, compressing the input into a layer with few neurons (encoder) before expanding towards one or more prediction outputs (decoders). Figure 2 shows a high-level illustration of the model. The model was trained on data from driving situations in different environments and geographics. Some experiment also use data generated from CARLA as a means to improve the model's performance in simulated environments.

Data. The *Mapillary Vistas* dataset [20] (henceforth Mapillary) was used for RGB and ground-truth semantic segmentation data. The dataset consists of 25 000 high-resolution images from different driving situations, with a large variety of weather and geographical locations. To simplify the environment for the perception network, the number of classes for segmentation was reduced from the original 66 object classes, to five classes; unlabeled, road, lane markings, humans and vehicle. To train the model's depth decoder, ground truth depth maps were generated using the Monodepth2 network from Godard et al. [10], as Mapillary lacks this information. Figure 1 shows a sample from this dataset.

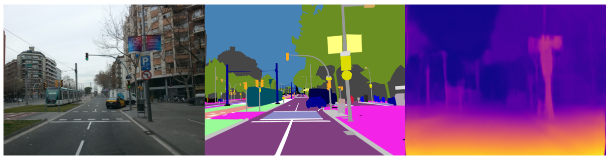


Fig. 1: Sample of the data used when training the perception model. The left image is the original RGB. The center image is segmentation ground truth from Mapillary. The right depth map was generated from RGB images with the Monodepth2 network.

In addition to using real-world perception data, we generated synthesized data in CARLA. A Python script spawns a large variety of vehicles and pedestrians, and captures RGB, semantic segmentation, and depth data from the vehicles as they navigate the simulated world. The field of view (FOV) and camera yaw angle were randomly distributed to generalize between different camera setups. The simulated weather was additionally changed periodically; varying cloudiness, amount of rain, time of day, and other modifiable weather parameters in CARLA. The final size of this synthesized dataset is about 20 000 images.

Architecture. Several encoders and decoders were explored when deciding the model's architecture. Encoders tested were: MobileNet [16], ResNet-50 [15] and a vanilla CNN, while decoders tested were: U-Net[22] and SegNet[2]. To generate a network which could predict both depth and segmentation estimations, we modified the existing MobileNet-U-Net architecture to include a second U-Net decoder. The decoder was modified to predict only one value per pixel, use the sigmoid activation function, and train with a regression loss function for depth estimation, adapted from Alhashim and Wonka [1].

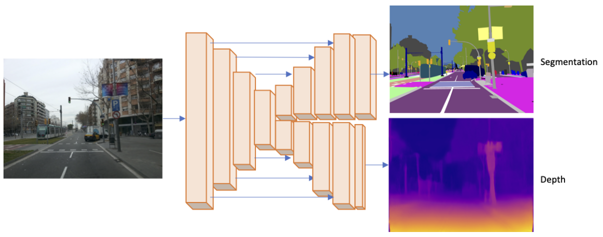


Fig. 2: A simplified illustration of the perception model. The different architectures all used a variant of the encoder-decoder architecture. The figure represents the MobileNet-U-Net model with a second depth estimation decoder, where each layer in the encoder is connected to the corresponding layer in the decoder.

Evaluation and Metrics. The segmentation prediction was evaluated using Intersection over Union (IoU), calculated with the following equation: $\frac{gt \cap p}{gt \cup p}$, where gt is the ground truth segmentation and p is the predicted segmentation. Mean IoU was used as the main indicator for performance, calculated by taking the mean of the class-wise IoU. Frequency weighted IoU was also calculated, measured as the mean IoU weighted by the number of pixels for each class.

The metric used for depth estimation was "accuracy within threshold *th*", where *th* was set to 1.25, 1.25^2 and 1.25^3, equal to Cao et al. [4]. The actual accuracy δ was calculated by $\delta = max(\frac{d_{gt}}{d_p}, \frac{d_p}{d_{gt}}), \delta > th$, where d_p is the predicted depth value and d_{gt} is the ground truth value calculated for each pixel.

3.2 Driving Model

The driving model runs raw RGB images through the perception model, and uses its output segmentation and depth predictions as input. These images are coupled with driving data recorded from an expert driver, which is further described in *Data* below. The driving model processes these inputs through its own layers, before outputting a steering angle and target speed.

Data. The driving data was generated in CARLA version 0.9.9. This was done by making an autopilot control a car in various environments, and recording video from three forward-facing cameras, its steering angles, speed, target speed, and HLCs (*left*, *right*, *straight*, or *follow lane*). The autopilot has access to the full state of the world, which includes a HD map, its own location and velocity, and the states of other vehicles and pedestrians. It uses this information to generate waypoints, which are finally fed into a PID-based controller to apply throttle, brake, and steering angle. The collected training data was unevenly distributed in regards to HLCs and steering angles, and we therefore down-sampled over-represented values for an improved data distribution.

Various data sets were gathered for training the driving policy, all of which were collected in *Town01*. These have different amount of complexity; steering noise magnitude the autopilot has to account for, different weather conditions and different light conditions. 30 641 samples were collected in total, where the weather varied according to CARLA's 15 default weather presets. The training data was effectively multiplied by three, as we made two copies of each data point, where we used the recorded image from each side camera instead of the main camera. To adjust for a slightly modified camera perspective, we added an offset of 0.05 and -0.05 in steering angle respectively for the left and right camera variants. This technique was first introduced by Bojarski et al. [3], and has later proved successfully in other papers [5, 13].

Architecture. The input of the driving model is a concatenation of the output from the perception model and an information vector containing the HLC (one-hot encoded), current speed, current speed limit, and the upcoming traffic light's state. The driving model is trained on simulation data with all the layers of the perception model frozen (that is, non-trainable) to preserve generalizability. Figure 3 shows an overview of the model.

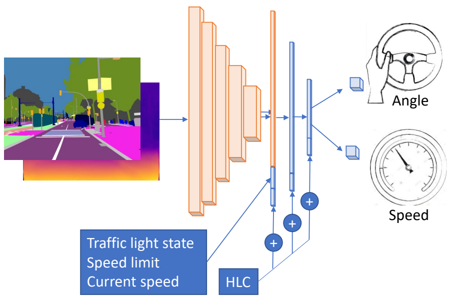


Fig. 3: A simplified illustration of the driving network. The segmentation and depth maps inputs are concatenated directly from the outputs of the perception model, shown in Figure 2.

The segmentation and depth output of the perception model are concatenated channel-wise, and resemble a RGBD (RGB + depth) image. This representation is then run through 5 convolutional blocks, each consisting of zero padding of 1, 2D-convolution with kernel 3, batch normalization, ReLu activation, and finally max-pooling with pool size 2. The filter sizes are 64, 128, 256, 256, 256, respectively. The current HLC, whether the traffic light was red or not, speed, and speed limit are concatenated with feature vectors generated from the perception data. The last layers are a combination of fully-connected layers, where we concatenate the HLC vector at each step, similar to Hawke et al. [14]. The first output of the model is the steering prediction; one neuron outputting the optimal steering (between 0 and 1, 0 being max leftward, 1 being max rightward), later mapped to CARLA's [-1, 1] range. The second output is the optimal vehicle speed, outputted as a percentage of 100 km/h (between 0 and 1).

Evaluation and Metrics. The main metric used for measuring driving model performance was Mean Completion Rate (MCR) during real-time evaluation. This is calculated by dividing the completed distance d_c by the total route distance d_t of each run-through of a route, averaged over all run-throughs R: $\frac{\sum_{r\epsilon R} \frac{d_c}{d_t}}{|R|}$. Traffic violations were not included as metrics, as the scope of this paper is mainly within completing routes without major incidents. The model's validation loss was also used as a rough metric for performance. By empirical observations we only picked models with validation loss $\lessapprox 0.03$ for further evaluation. The validation loss metric was used as an initial performance estimation because the MCR evaluation was very time consuming.

4 Experiments and results

There are two main experiments conducted in this paper. The first experiment and its sub-experiments focuses on generating the best perception model to be used when training the driving network. Model architecture, dataset variants, augmentation, and multi-task learning are parameters experimented with to increase performance. The second experiment is conducted in CARLA. This experiment assess the driving policy performance given the different models derived in the first set of experiments. The generalizability of each model is tested using different unseen environments. Each perception model is then compared to a baseline model trained only on the CARLA dataset using Mean Completion Rate as the metric.

4.1 Experiment 1: Perception Model

The perception experiments use semantic segmentation and occasionally depth estimation to generalize the driving environment when training and testing the driving models. All of the perception experiments use the same dataset for evaluation, and the results can therefore be compared across experiments. The CARLA data generated and used for evaluation consists of 4 400 images with corresponding ground truth segmentation and depth maps from Town 3-4. The Mapillary evaluation dataset is a set of 2 000 images from the original Mapillary test set.

Experiment 1-1: Encoder-decoder models. This experiment attempts to find the best encoder and decoder to use for the perception network. All the encoders tested were picked because they have previously shown good results in other papers, and were implemented in a common library by Gupta [12].

The three encoders showed increased performance as the complexity and size of the encoder increased. The Vanilla CNN encoder performed worst with the lowest Mean IoU score, however, it was also the fastest model during training and testing. MobileNet gave better results while keeping a lot of the speed advantage from the Vanilla CNN network. MobileNet also showed very good results, with MobileNet-U-Net displaying the best overall performance when combining scores. ResNet50 performed good as expected with a higher Mean IoU than MobileNet-U-Net, however the difference from MobileNet-U-Net was less than expected. MobileNet was used for futher experiments as it was significantly faster than ResNet50.

Model	Mean IoU	Weighted IoU
VanillaCNN-SegNet	0.324	0.712
VanillaCNN-U-Net	0.351	0.705
MobileNet-SegNet	0.368	**0.775**
MobileNet-U-Net	0.403	0.774
ResNet50-SegNet	**0.405**	0.767
ResNet50-U-Net	0.383	0.733

Table 1: Evaluation of three different encoders (Vanilla CNN, Mobilenet and ResNet50), and two decoders (SegNet and U-Net). Each model was trained on the Mapillary dataset (18 000 samples for training and 2 000 for validation) without any augmentation. The best scores are marked in bold.

Experiment 1-2: Training data. To improve the model further some CARLA data was introduced to the Mapillary dataset. Augmentation was also introduced for further improvements and better generalization. The Mapillary+CARLA dataset consisted of 20 000 datapoints from the Mapillary dataset and 3 250 samples from *Town01* and *Town02* in CARLA. The dataset with only augmented CARLA data (CARLA+Aug) used a different dataset of 15 000 samples from Town 1-4, and 4 000 samples from Town 5 as validation. The results were evaluated on Town 3-4 as Town 1-2 was used when training Mapillary+CARLA. The augmentation included consists of among others gaussian noise, translation, rotation, hue and saturation augmentations, and was adapted from Gupta [12].

	CARLA Eval		*Mapillary Eval*	
Training dataset	Mean IoU	Weighted IoU	Mean IoU	Weighted IoU
Mapillary	0.425	0.771	0.632	0.887
Mapillary+Aug	0.436	0.809	0.574	0.873
Mapillary+CARLA	0.469	0.846	**0.633**	**0.889**
Mapillary+CARLA+Aug	0.478	0.850	0.568	0.874
CARLA+Aug	**0.572**	**0.909**	0.384	0.785

Table 2: Evaluation of different datasets on the MobileNet-U-Net model. Mapillary is the original dataset while the CARLA dataset was generated directly from CARLA. Each dataset consist of about 20 000 samples, and the Mapillary+CARLA dataset consist of about 80/20 Mapillary and CARLA data respectively. The two sets of columns show evaluation on CARLA data and Mapillary data respectively. The best scores are marked in bold.

The dataset experiment shows that including CARLA data as a component when training the perception models increases the total performance. As the model's goal is to make good predictions in both real and simulated environments, combining data from both seems to be a reasonable approach. CARLA+Aug achieves great results when evaluating on CARLA data, however the performance decreased drastically when predicting in real-world environments. Models trained on real-world data tends to generalize better to unseen simulated environments than the other way around. Incorporating some CARLA data into the real-world data in addition to augmenting the images yields the best results overall.

Experiment 1-3: Multi-task perception. Inspired by Hawke et al. [14] we introduced a depth estimation decoder to the MobileNet-U-Net model. The model was trained with ground truth data generated by the Monodepth2 network using images from the Mapillary dataset, while depth maps for the CARLA data was included in the generated CARLA dataset.

	Segmentation		*Depth*		
Training dataset	Mean IoU	Weighted IoU	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Mapillary	0.458 (+0.03)	0.817	0.320	0.572	0.684
Mapillary+CARLA	0.520 (+0.05)	0.854	0.295	0.542	0.679
CARLA	0.717 (+0.15)	0.960	0.775	0.806	0.816

Table 3: Results after adding a depth estimation decoder to the Mobilenet-U-Net model. Each model was trained on the same dataset as in Experiment 2. Mean IoU additionally presents a difference in parantheses: the difference between these models' mean IoU and their counterparts' from Experiment 1. Depth is estimated using *accuracy within threshold*, where the set threshold is presented in the column title. A high value is best for all metrics in the table.

Including a depth estimation decoder increases the segmentation performance for each model. The mean increase in IoU on the CARLA test set is 8%, which conforms with the results reported by Standley et al. [23], who reported a 4.17% increase in performance when training semantic segmentation with depth estimation. An increase in overall scene understanding can also be expected as depth is introduced to the model, however this has to be verified as part of the overall driving policy experiments.

4.2 Experiment 2: Driving Model

This experiment aims to assess the overall performance of the two-part (perception and driving policy) architecture. We run real-time evaluations on variants of our proposed architecture, including a baseline network where the complete network is trained at once. The evaluation is conducted with a custom scenario runner for CARLA, originally introduced by Haavaldsen et al. [13], and extended for our experiments. The real-time nature of this experiment makes it different from the previous experiments: The models' steering and speed outputs affect the camera input in subsequent simulation steps, and each prediction is therefore dependent on the ones before.

The scenario runner. The scenario runner makes each model drive through a predefined set of routes, each of which is defined by a set of waypoints. The model navigates each route using HLCs provided automatically when passing each waypoint. Each attempt at a route ends either when the vehicle completes the route, or when the vehicle enters any of the following erroneous states: stuck on an obstacle, leaving its correct lane and not returning within five seconds, or ignoring a HLC. Quantitative measurements are made by logging the distance completed and traffic violations (e.g. lane touches and collisions) of each route attempt. The models are then compared mainly on their mean route completion rate and mean traffic violations per route.

Environments and Routes. The models were tested in two environments, *Town02* and *Town07*. *Town02* is similar, but not identical to the one in the driving policy's training data, which is *Town01*. *Town07* is quite different, and is rural with narrow roads (some without any centre marking), fields, and barns. There are three routes in each environment, which the models will try to complete in six different weather conditions. Three of the weather conditions have already been observed in the training data, while the three remaining are unknown to the policy. The training data only contain samples from day-time weathers, but two of the unknown weathers are at midnight.

Results. Table 4 summarizes the driving performance of the different models. The model trained only on driving data and without a frozen perception model, *RGB*, was the best-performing model on *Town02*, but it struggles with *Town07*.

The model names starting with *SD* indicates that they use the segmentation and depth perception model (henceforth SD). *SD-CARLA*, which uses SD trained only on perception data from CARLA, outperforms all other models when ranked by Mean Completion Rate (MCR) over both towns. To demonstrate its performance, we made a video (`https://youtu.be/HL5LStDe7wY`) showing some of its good performing moments. *SD-Mapillary* uses SD as well, but only had perception training data from Mapillary. While not performing as good as SD-CARLA, it still has impressive results. Its perception model has not seen any CARLA data, but is still able to predict segmentation and depth good enough for the driving model to beat even the RGB model. *SD-Combined* used perception data from both Mapillary and CARLA, and performs a little bit worse than SD-Mapillary.

The model names starting with *S* indicates that they use the segmentation-only perception model (henceforth S). *S-CARLA* is the S-counterpart of SD-CARLA, and it performs very well in *Town02*. In *Town07* however, it struggles

	Seen weather			*Unseen weather*			
Model	**Clear (D)**	**Rain (D)**	**Wet (S)**	**Clear (N)**	**Rain (N)**	**Fog (S)**	**Mean**
RGB	100.00 %	28.72 %	36.05 %	100.00 %	11.22 %	100.00 %	62.67 %
SD-CARLA	100.00 %	43.30 %	55.36 %	100.00 %	23.46 %	44.96 %	61.18 %
S-CARLA	93.83 %	7.23 %	43.51 %	100.00 %	24.61 %	66.66 %	55.97 %
SD-Mapillary	88.51 %	42.16 %	67.22 %	100.00 %	11.59 %	24.91 %	55.73 %
SD-Combined	93.53 %	9.92 %	47.42 %	100.00 %	9.92 %	46.53 %	51.22 %
S-Combined	90.71 %	44.02 %	23.12 %	72.12 %	2.72 %	7.23 %	39.98 %
S-Mapillary	72.12 %	43.30 %	40.85 %	39.34 %	2.72 %	2.72 %	33.51 %

(a) Results for *Town02*.

	Seen weather			*Unseen weather*			
Model	**Clear (D)**	**Rain (D)**	**Wet (S)**	**Clear (N)**	**Rain (N)**	**Fog (S)**	**Mean**
SD-CARLA	84.95 %	61.14 %	84.95 %	60.81 %	88.88 %	17.19 %	66.32 %
SD-Mapillary	77.28 %	77.28 %	55.54 %	51.61 %	33.33 %	14.84 %	51.65 %
SD-Combined	50.52 %	61.14 %	57.39 %	38.60 %	66.67 %	33.33 %	51.27 %
S-Combined	77.83 %	55.10 %	55.10 %	17.32 %	50.65 %	33.33 %	48.22 %
RGB	44.49 %	44.49 %	44.49 %	44.49 %	49.75 %	44.49 %	45.37 %
S-CARLA	55.10 %	55.10 %	57.39 %	17.32 %	17.32 %	61.87 %	44.02 %
S-Mapillary	51.61 %	50.52 %	44.49 %	55.10 %	44.49 %	0.00 %	41.04 %

(b) Results for *Town07*.

Table 4: Mean completion rate in (a) *Town02* and (b) *Town07*, in six weather conditions. Day, Sunset and Night is shortened to *D*, *S*, *N* respectively. The individual cells are colored on a scale where green is the best, and red is the worst.

with night-time weather. S-Mapillary is the S-counterpart of SD-Mapillary, and it has the lowest MCR in both towns. In any run with *Fog (S)*, it fails almost immediately. *S-Combined* uses combined perception data, the same as *SD-Combined*. It is performing a bit better than S-Mapillary in *Town02*, and is the fourth best in *Town07*.

5 Discussions

5.1 Results in comparison to related work

Xiao et al. [25] uses ground-truth semantic segmentation data generated from CARLA, not predicted as we do, and combine segmentation with both ground-truth depth maps and depth estimated by a separate network. Their results aligns with our results; using semantic segmentation data beats just using raw images, and combining both segmentation and depth performs the best. With a combination of ground truth segmentation and estimated depth, their policy is still able to beat the raw image-based policy. Our models estimate both segmentation and depth, and is still able to perform good in comparison to our baseline RGB-model.

Müller et al. [19] use predicted binary segmentation (road/not road) as driving input, and our work extends this with predicted depth, giving additional performance benefits. Haavaldsen et al. [13] achieved higher completion rates even with traffic, but focused more on the impact of larger data sets and encoding temporal information in the model, while this paper focused mainly on generalizability.

The driving model by Hawke et al. [14] did not include the perception model's decoding layers in its architecture, which seems to be an overall more efficient approach. Because the U-Net architecture used in our paper had connections between each encoder-decoder layer, information could have been lost by not including the decoding layers. In future work, a model without connections between the encoder-decoder layers could be explored to take advantage of Hawke et al.'s approach.

5.2 Driving models

We find that models with a learned understanding of both the semantics and/or geometry of the scene are able to navigate never-before-seen environments and weather. Our real-time experiment shows that these driving models often perform better than learning from raw image inputs directly, with models utilizing both semantics and geometry performing best overall.

Variance. It is important to note that we observed a high variance when training and evaluating our models. Two models trained from the exact same setup could perform significantly different, despite having the exact same training data. We suspect that this is the same variance problem as Codevilla et al. [6] experienced. The variance was handled by training and testing the models several times to make sure the results were representative. Still, conclusions based on the results in Experiment 2 must be drawn carefully. A more robust approach could be to train multiple models with the same parameters, and averaging their results.

6 Conclusion

Splitting end-to-end models for autonomous vehicles into separate models for perception and driving policy is shown to give good results in simulated environments. Perception models trained from public datasets such as *Mapillary Vistas* can be used to reduce the amount of driving data needed when training an end-to-end driving policy network. This approach opens up for training the driving policy in a simulated environment, while still getting good performance in real-world environments.

Future work should explore how these results transfers into the real world. Evaluating the performance of a model trained solely in simulation directly in a real-world environment will be an important next step as a means of testing the validity of these results.

Bibliography

[1] Alhashim, I., Wonka, P.: High quality monocular depth estimation via transfer learning. arXiv:1812.11941 [cs] (Mar 2019), URL `http://arxiv.org/abs/1812.11941`, arXiv: 1812.11941

[2] Badrinarayanan, V., Kendall, A., Cipolla, R.: Segnet: A deep convolutional encoder-decoder architecture for image segmentation. arXiv:1511.00561 [cs] (Oct 2016), URL `http://arxiv.org/abs/1511.00561`, arXiv: 1511.00561

[3] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., Zieba, K.: End to End Learning for Self-Driving Cars (2016)

[4] Cao, Y., Zhao, T., Xian, K., Shen, C., Cao, Z., Xu, S.: Monocular depth estimation with augmented ordinal depth relationships. arXiv:1806.00585 [cs] (Jul 2019), URL `http://arxiv.org/abs/1806.00585`, arXiv: 1806.00585

[5] Codevilla, F., Müller, M., López, A., Koltun, V., Dosovitskiy, A.: End-to-end Driving via Conditional Imitation Learning. arXiv:1710.02410 [cs] (Oct 2017), URL `http://arxiv.org/abs/1710.02410`, arXiv: 1710.02410

[6] Codevilla, F., Santana, E., López, A.M., Gaidon, A.: Exploring the Limitations of Behavior Cloning for Autonomous Driving. arXiv:1904.08980 [cs] (Apr 2019), URL `http://arxiv.org/abs/1904.08980`, arXiv: 1904.08980

[7] Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., Schiele, B.: The cityscapes dataset for semantic urban scene understanding. arXiv:1604.01685 [cs] (Apr 2016), URL `http://arxiv.org/abs/1604.01685`, arXiv: 1604.01685

[8] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, pp. 1–16 (2017)

[9] Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision meets robotics: The kitti dataset. International Journal of Robotics Research (IJRR) (2013)

[10] Godard, C., Mac Aodha, O., Firman, M., Brostow, G.: Digging into self-supervised monocular depth estimation. arXiv:1806.01260 [cs, stat] (Aug 2019), URL `http://arxiv.org/abs/1806.01260`, arXiv: 1806.01260

[11] Gualtieri, M., Pas, A.t., Saenko, K., Platt, R.: High precision grasp pose detection in dense clutter. arXiv:1603.01564 [cs] (Jun 2017), URL `http://arxiv.org/abs/1603.01564`, arXiv: 1603.01564

[12] Gupta, D.: Image segmentation keras : Implementation of segnet, fcn, unet, pspnet and other models in keras. (2020), URL `https://github.com/divamgupta/image-segmentation-keras/blob/31d1ba660ec16d6032d8719841c4f00c6bf934b0/keras_segmentation/data_utils/augmentation.py`

[13] Haavaldsen, H., Aasboe, M., Lindseth, F.: Autonomous Vehicle Control: End-to-end Learning in Simulated Urban Environments. arXiv:1905.06712 [cs] (May 2019), URL `http://arxiv.org/abs/1905.06712`, arXiv: 1905.06712

[14] Hawke, J., Shen, R., Gurau, C., Sharma, S., Reda, D., Nikolov, N., Mazur, P., Micklethwaite, S., Griffiths, N., Shah, A., Kendall, A.: Urban Driving with Conditional Imitation Learning. arXiv:1912.00177 [cs] (Dec 2019), URL http://arxiv.org/abs/1912.00177, arXiv: 1912.00177

[15] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. arXiv:1512.03385 [cs] (Dec 2015), URL http://arxiv.org/abs/1512.03385, arXiv: 1512.03385

[16] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861 [cs] (Apr 2017), URL http://arxiv.org/abs/1704.04861, arXiv: 1704.04861

[17] Johnson-Roberson, M., Barto, C., Mehta, R., Sridhar, S.N., Rosaen, K., Vasudevan, R.: Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? arXiv:1610.01983 [cs] (Feb 2017), URL http://arxiv.org/abs/1610.01983, arXiv: 1610.01983

[18] Kendall, A., Gal, Y., Cipolla, R.: Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. arXiv:1705.07115 [cs] (Apr 2018), URL http://arxiv.org/abs/1705.07115, arXiv: 1705.07115

[19] Müller, M., Dosovitskiy, A., Ghanem, B., Koltun, V.: Driving Policy Transfer via Modularity and Abstraction. arXiv:1804.09364 [cs] (Dec 2018), URL http://arxiv.org/abs/1804.09364, arXiv: 1804.09364

[20] Neuhold, G., Ollmann, T., Rota Bulò, S., Kontschieder, P.: The mapillary vistas dataset for semantic understanding of street scenes. In: International Conference on Computer Vision (ICCV) (2017), URL https://www.mapillary.com/dataset/vistas

[21] Pomerleau, D.A.: Advances in Neural Information Processing Systems 1, p. 305–313. Morgan Kaufmann Publishers Inc. (1989), ISBN 978-1-55860-015-7, URL http://dl.acm.org/citation.cfm?id=89851.89891

[22] Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. arXiv:1505.04597 [cs] (May 2015), URL http://arxiv.org/abs/1505.04597, arXiv: 1505.04597

[23] Standley, T., Zamir, A.R., Chen, D., Guibas, L., Malik, J., Savarese, S.: Which tasks should be learned together in multi-task learning? arXiv:1905.07553 [cs] (May 2019), URL http://arxiv.org/abs/1905.07553, arXiv: 1905.07553

[24] Viereck, U., Pas, A.t., Saenko, K., Platt, R.: Learning a visuomotor controller for real world robotic grasping using simulated depth images. arXiv:1706.04652 [cs] (Nov 2017), URL http://arxiv.org/abs/1706.04652, arXiv: 1706.04652

[25] Xiao, Y., Codevilla, F., Gurram, A., Urfalioglu, O., López, A.M.: Multimodal End-to-End Autonomous Driving. arXiv:1906.03199 [cs] (Jun 2019), URL http://arxiv.org/abs/1906.03199, arXiv: 1906.03199

Appendix B

SPURV Pipeline Manual

This manual assumes that you have already followed the steps in Appendix A of Kastet and Neset [23], and therefore have a client with the needed software for communicating with and controlling a SPURV with an Xbox controller. We use the same terminology of host machine and SPURV as in their manual. This is a copy of the manual provided in our own specialization project, updated with new changes.

B.1 Pipeline overview

The pipeline helps streamline the following recurring tasks within end-to-end model research: data collection, training, and validation of performance. Figure B.1 shows an overview of the pipeline, and where and what the scripts are used for.

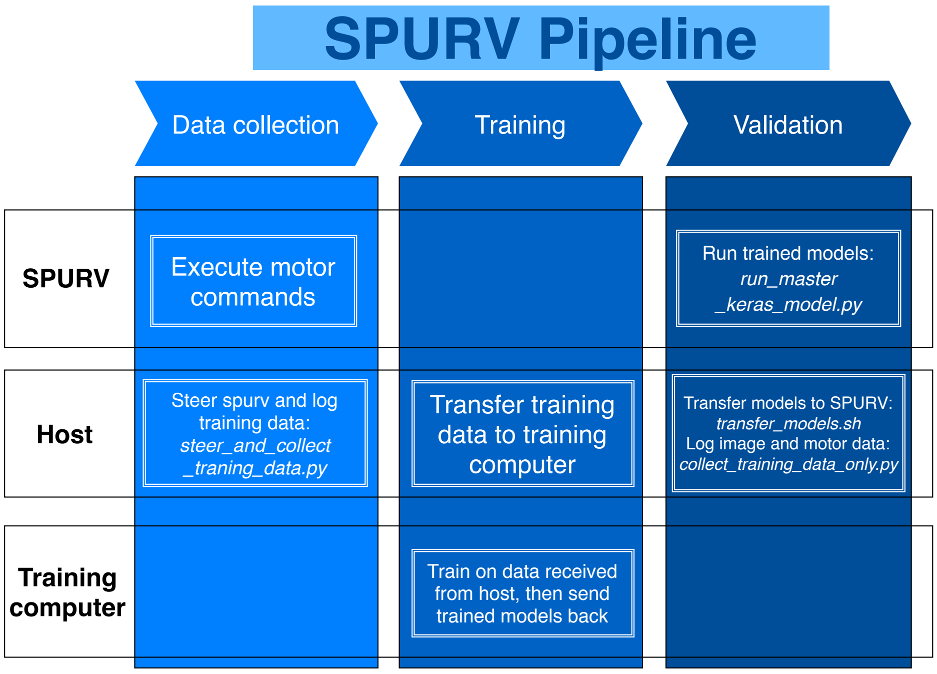

SPURV Pipeline
Data collection
Training
Validation
SPURV
Host
Training computer
Execute motor commands
Run trained models: run_master _keras_model.py
Steer spurv and log training data: steer_and_collect _traning_data.py
Transfer training data to training computer
Transfer models to SPURV: transfer_models.sh Log image and motor data: collect_training_data_only.py
Train on data received from host, then send trained models back

Figure B.1: The SPURV pipeline, consisting of three main steps.

B.2 Script development

We recommend using PyCharm Professional[1] to develop Python scripts for the SPURV. To be able to run scripts on the host machine, PyCharm needs to be run through bash, as ROS-based scripts requires the shell environment defined in ∼/.bashrc. PyCharm can easily be launched this way by running

[1]https://www.jetbrains.com/pycharm/

```
/snap/pycharm-professional/current/bin/pycharm.sh
```

(or whatever the install path of PyCharm is) from a bash instance.

B.2.1 Automatic deployment to SPURV

PyCharm Professional has a feature that automatically uploads scripts to a remote destination as they are saved or run. This enables us to develop scripts on the host machine and instantly run them on the SPURV. To set up this workflow for yourself, follow PyCharm's guide on setting up a remote interpreter: `https://www.jetbrains.com/help/pycharm/configuring-remote-interpreters-via-ssh.html`. Use

```
    nvidia@spurv:22
```

as the SSH connection host and port. Remember to choose Python 2 as the interpreter on the SPURV..0

PyCharm will attempt to start the script using Python directly, but this will fail as the ROS environment is not loaded. As a workaround, we have made a small wrapper which can be inserted into each script, which restarts the script in the correct environment. An example usage of this wrapper is found in the

```
    run_master_keras_models.py
```

script.

B.3 Transferring models - *transfer_models.sh*

To streamline the process of transferring models, we have made a script called *transfer_models.sh*, which transfers all the models (files ending with .h5) in the working directory and subdirectories to the

```
    /home/nvidia/ros/src/spurv_research/spurv_examples/src/models
```

directory on the SPURV. This is the directory used by the *run_master_keras_models.py* script. Alternatively, one can use the *SCP* command or mount the SPURV's file system over SSH.

B.4 Running models - *run_master_keras_models.py*

The *run_master_keras_models.py* script handles the selection and execution of models. It is designed to work with any model that fulfills the following criteria:

- It is in the HDF5 format (.h5 extension)
- It is compatible with Python 2, Tensorflow 1.10.0, and Keras 2.2.0.
- If it has any custom layers, its implementation has to be copied to the SPURV for loading. See the *model_structures* folder for examples.
- It has an input layer named *forward_image_input* for inputting images with shape (w,h,3) in the BGR color space. The image input will be normalized, based on VGG16's normalization, before being sent to the model.
- It has an input layer named *hlc_input* of dimension 4, containing an one-hot encoded high-level command (left, right, or straight, or follow lane).
- Its second output layer contains one node whose value is the desired speed, as a fraction of 100 km/h, of the SPURV.
- It has an output layer named *steer_pred* which consists of 1 node whose value is a steering angle in the range [0, 1]. 0 is 100% to the left, 1 is 100% to the right.

B.4.1 Xbox controls

The model selection, starting and stopping, and all other interactions with the script is done with an Xbox controller. Table B.1 contains an overview of all available actions.

Xbox buttom	Action
D-pad (left and right)	Iterate through available models (in the *models* directory).
RB	If no model is loaded: load the selected model. If a model is loaded: start autonomous mode.
LB	Stop autonomous mode.
X	Apply HLC=left(1).
B	Apply HLC=right(2).
Y	Apply HLC=straight(3).
A	Apply HLC=follow lane(4).

Table B.1: Overview of Xbox controls for *run_master_keras_model.py*.

B.5 Collecting data

This section covers the process of collecting data, both during training and while running a model. This is an alternative to using the *rosbag* utility of ROS.

B.5.1 Training data collection - *steer_and_collect_training_data.py*

This script allows you to collect training data while steering the SPURV. The data will be saved in the directory *~/data_dump/%iso_time%*. A file with the name *data.csv* will be created, and it contains data values for each time step. A description of the columns in *data.csv* is found in table B.2.

Field name	Description
speed	The speed of the SPURV, in m/s.
angle	The steering angle of the SPURV, in the range [-1, 1].
high_level _command	The current high-level command (HLC). It is either left (0), straight (1), or right (2).
image_path	The absolute path (on host machine) of the recorded image.
Field defined below are only available in *steer_and_collect_training_data.py*:	
ackerman _timestamp	The nanosecond timestamp of the latest Ackermann (steering command) message. Note: Based on the host machine clock.
img_timestamp	The nanosecond timestamp of the saved image. Note: Based on the SPURV OS clock.
angle_w_noise	The steering angle which is sent to the SPURV. This is the raw steering angle from the Xbox controller, plus a generated triangular noise.

Table B.2: Description of field in the data.csv file.

All recorded images will be saved in the *images* folder. For an overview of the available actions in the script, see table B.3.

Xbox buttom	Action
Left joystick (left and right)	Steer left or right
Right joystick (up and down)	Set desired speed.
Start	Toggle recording of data.
Back	Toggle triangular noise.
LB	Stop autonomous mode.
X	Apply HLC=left(1) for 10 seconds.
B	Apply HLC=right(2) for 10 seconds.
Y	Apply HLC=straight(3) for 10 seconds.
A	Apply HLC=follow lane(4) until changed.

Table B.3: Overview of Xbox controls for *steer_and_collect_training_data.py*.

B.5.2 Data collection for testing - *collect_training_data_only.py*

This script is meant to be run simultaneously as a model is running. It collects the camera images and the steering commands executed by the SPURV, and saves them in a similar format to *steer_and_collect_training_data.py.* This script uses only the *Start* button of the Xbox controller, which toggles recording of data. The data format is specified in Table B.2

Appendix C

Model Architectures

C.1 Perception model - Keras implementation

```
"""
    A lot of this code is modified code from the library
    utilized during model implementation.
    The original source can be found at:
    https://github.com/divamgupta/image-segmentation-keras

    See references chapter for more detailed citation.
"""

def relu6(x):
    return K.relu(x, max_value=6)


def _conv_block(inputs, filters, alpha, kernel=(3, 3), strides=(1, 1)):

    filters = int(filters * alpha)
    x = ZeroPadding2D(padding=(1, 1), name='conv1_pad')(inputs)
    x = Conv2D(filters, kernel,
               padding='valid',
               use_bias=False,
               strides=strides,
               name='conv1')(x)
    x = BatchNormalization(name='conv1_bn')(x)
    return Activation(relu6, name='conv1_relu')(x)


def _depthwise_conv_block(inputs, pointwise_conv_filters, alpha,
                          depth_multiplier=1, strides=(1, 1), block_id=1):

    pointwise_conv_filters = int(pointwise_conv_filters * alpha)
    x = ZeroPadding2D((1, 1),
                      name='conv_pad_%d' % block_id)(inputs)
    x = DepthwiseConv2D((3, 3),
                        padding='valid',
                        depth_multiplier=depth_multiplier,
                        strides=strides,
                        use_bias=False,
                        name='conv_dw_%d' % block_id)(x)
    x = BatchNormalization(name='conv_dw_%d_bn' % block_id)(x)
    x = Activation(relu6, name='conv_dw_%d_relu' % block_id)(x)

    x = Conv2D(pointwise_conv_filters, (1, 1),
               padding='same',
               use_bias=False,
               strides=(1, 1),
               name='conv_pw_%d' % block_id)(x)
    x = BatchNormalization(name='conv_pw_%d_bn' % block_id)(x)
    return Activation(relu6, name='conv_pw_%d_relu' % block_id)(x)


def get_mobilenet_encoder():

    input_height = 226
    input_width = 226
    alpha = 1.0
    depth_multiplier = 1
    dropout = 1e-3

    img_input = Input(shape=(input_height, input_width, 3))

    x = _conv_block(img_input, 32, alpha, strides=(2, 2))
    x = _depthwise_conv_block(x, 64, alpha, depth_multiplier, block_id=1)
    f1 = x

    x = _depthwise_conv_block(x, 128, alpha, depth_multiplier,
                              strides=(2, 2), block_id=2)
    x = _depthwise_conv_block(x, 128, alpha, depth_multiplier, block_id=3)
    f2 = x

    x = _depthwise_conv_block(x, 256, alpha, depth_multiplier,
                              strides=(2, 2), block_id=4)
    x = _depthwise_conv_block(x, 256, alpha, depth_multiplier, block_id=5)
    f3 = x

    x = _depthwise_conv_block(x, 512, alpha, depth_multiplier,
                              strides=(2, 2), block_id=6)
    x = _depthwise_conv_block(x, 512, alpha, depth_multiplier, block_id=7)
    x = _depthwise_conv_block(x, 512, alpha, depth_multiplier, block_id=8)
    x = _depthwise_conv_block(x, 512, alpha, depth_multiplier, block_id=9)
    x = _depthwise_conv_block(x, 512, alpha, depth_multiplier, block_id=10)
    x = _depthwise_conv_block(x, 512, alpha, depth_multiplier, block_id=11)
    f4 = x

    x = _depthwise_conv_block(x, 1024, alpha, depth_multiplier,
                              strides=(2, 2), block_id=12)
    x = _depthwise_conv_block(x, 1024, alpha, depth_multiplier, block_id=13)
    f5 = x

    return img_input, [f1, f2, f3, f4, f5]

def get_unet_decoder(levels, n_classes):
    [f1, f2, f3, f4, f5] = levels
    o = f4
    o = (ZeroPadding2D((1, 1)))(o)
    o = (Conv2D(512, (3, 3), padding='valid' , activation='relu'))(o)
    o = (BatchNormalization())(o)

    o = (UpSampling2D((2, 2)))(o)
    o = (concatenate([o, f3]))
    o = (ZeroPadding2D((1, 1)))(o)
    o = (Conv2D(256, (3, 3), padding='valid', activation='relu'))(o)
    o = (BatchNormalization())(o)

    o = (UpSampling2D((2, 2)))(o)
    o = (concatenate([o, f2]))
    o = (ZeroPadding2D((1, 1)))(o)
    o = (Conv2D(128, (3, 3), padding='valid' , activation='relu'))(o)
    o = (BatchNormalization())(o)

    o = (UpSampling2D((2, 2)))(o)

    o = (concatenate([o, f1], axis=MERGE_AXIS))

    o = (ZeroPadding2D((1, 1)))(o)
    o = (Conv2D(64, (3, 3), padding='valid', activation='relu'))(o)
    o = (BatchNormalization())(o)

    o = Conv2D(n_classes, (3, 3), padding='same')(o)

    return o


img_input, levels = get_mobilenet_encoder()

segm_output = get_unet_decoder(levels, n_classes)

depth_output = get_unet_decoder(levels, 1)

output_segm = (Reshape((output_height * output_width, -1)))(output_segm)
output_segm = (Activation('softmax', name="segm_pred"))(output_segm)

output_depth = (Activation('sigmoid', name="depth_pred"))(output_depth)

model = Model(inputs=img_input, outputs=[output_segm, output_depth])

loss_weights = {"depth_pred": 0.5, "segm_pred": 0.5}

model.compile(loss=['categorical_crossentropy', depth_loss_function],
                  loss_weights=loss_weights,
                  optimizer="adadelta",
                  metrics=['accuracy'])
```

C.2 Driving model - Keras implementation

```
"""
    The model is partly based on these sources:
```

```
    - "Autonomous Vehicle Control:
      End-to-end Learning in Simulated Urban Environments"
      by Hege Haavaldsen and Max Aasboe
    - https://github.com/divamgupta/image-segmentation-keras
"""
def get_lstm_model(freeze_segmentation: bool, segm_model: str):
    hlc_input = Input(shape=(1, 4), name="hlc_input")
    info_input = Input(shape=(1, 3), name="info_input")

    # Load the correct perception model,
    # either with trainable weights or frozen weights
    segmentation_model = get_segmentation_model(segm_model, freeze_segmentation)
    [_, height, width, _] = segmentation_model.input.shape.dims
    forward_image_input = Input(shape=(1, height.value, width.value, 3),
                                name="forward_image_input")
    x = TimeDistributed(segmentation_model)(forward_image_input)

    # Based on the vanilla encoder
    # from https://github.com/divamgupta/image-segmentation-keras
    kernel = 3
    filter_size = 64
    pad = 1
    pool_size = 2
    x = TimeDistributed(ZeroPadding2D((pad, pad)))(x)
    x = TimeDistributed(Conv2D(filter_size, (kernel, kernel),
                               padding='valid'))(x)
    x = TimeDistributed(BatchNormalization())(x)
    x = TimeDistributed(Activation('relu'))(x)
    x = TimeDistributed(MaxPooling2D((pool_size, pool_size)))(x)

    x = TimeDistributed(ZeroPadding2D((pad, pad)))(x)
    x = TimeDistributed(Conv2D(128, (kernel, kernel),
                               padding='valid'))(x)
    x = TimeDistributed(BatchNormalization())(x)
    x = TimeDistributed(Activation('relu'))(x)
    x = TimeDistributed(MaxPooling2D((pool_size, pool_size)))(x)

    for _ in range(3):
        x = TimeDistributed(ZeroPadding2D((pad, pad)))(x)
        x = TimeDistributed(Conv2D(256, (kernel, kernel),
                                   padding='valid'))(x)
        x = TimeDistributed(BatchNormalization())(x)
        x = TimeDistributed(Activation('relu'))(x)
        x = TimeDistributed(MaxPooling2D((pool_size, pool_size)))(x)

    x = TimeDistributed(Flatten())(x)
    x = concatenate([x, hlc_input, info_input])

    x = TimeDistributed(Dense(100, activation="relu"))(x)
    x = concatenate([x, hlc_input])

    # We have a LSTM layer, but the sequence length is always 1
    x = CuDNNLSTM(10, return_sequences=False)(x)
    hlc_latest = Lambda(lambda x: x[:, -1, :])(hlc_input)
    x = concatenate([x, hlc_latest])

    steer_pred = Dense(1, activation="relu", name="steer_pred")(x)
    target_speed_pred = Dense(1, name="target_speed_pred",
                              activation="sigmoid")(x)

    model = Model(inputs=[forward_image_input, hlc_input, info_input],
                    outputs=[steer_pred, target_speed_pred])

    return model
```

Audun Wigum Arbo, Even Dalen

Domain-Independent Perception for Autonomous Driving

NTNU
Norwegian University of
Science and Technology