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

Jostein Lilleløkken
Martin Hermansen

Improving Performance of Autonomous Driving in Simulated Environments Using End-to-End Approaches

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2021

Jostein Lilleløkken
Martin Hermansen

Improving Performance of Autonomous Driving in Simulated Environments Using End-to-End Approaches

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2021

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

Abstract

In recent years, autonomous vehicles have been subject to increased research and development. Due to recent advances in deep learning, the end-to-end approach has become a viable and cost-effective solution to creating autonomous driving systems. In the end-to-end approach, the complete task of autonomous driving is learned by a single comprehensive neural network. These networks can be trained by imitation learning or reinforcement learning methods.

Learning by Cheating (LBC) is an imitation learning approach which has proved to be effective for training neural networks for autonomous driving. This approach trains a network that uses RGB images as input and outputs a trajectory for the vehicle to follow. It uses the CARLA simulator to train and evaluate networks. This simulator provides a flexible and safe environment for quickly developing autonomous driving systems.

The models in end-to-end approaches will usually learn to perceive the scenery by processing RGB images. It is also possible to equip the models with explicit intermediate representations. Research shows that by using methods from computer vision, such as semantic segmentation and depth estimation, models can generalize better and increase task performance.

This thesis examines if using explicit intermediate representations can improve the performance of networks trained with the LBC approach. In the first experiment, the results show that LBC is reproducible in the latest version of CARLA (0.9.11). In the second experiment, it is shown that the performance and generalization of the networks increase significantly when using perfect explicit intermediate representations, which are supplied by the simulator. The results from the third experiment show that the networks also generalize better when using intermediate representations generated by trained perception models.

It is also investigated if the performance of the networks can be improved by using an additional reinforcement learning stage. An attempt was made to improve the networks further with proximal policy optimization, but this was found to be difficult.

Sammendrag

I de siste årene har forskningen og utviklingen av autonome kjøretøy økt. Takket være gjennombrudd innen dyp læring, så har ende-til-ende-tilnærmingen blitt en realistisk og kostnadseffektiv løsning for å skape selvkjørende kjøretøy. I ende-til-ende-tilnærmingen blir hele kjøreoppgaven lært av ett enkelt nevralt nettverk. Disse nevrale nettverkene kan bli trent ved enten imitasjonslæring eller forsterknings-læring.

Learning by Cheating (LBC) er en imitasjonslærings-metode som har vist seg å være effektiv for å trene nevrale nettverk for selvkjørende kjøretøy. Denne metoden trener et nettverk som tar inn RGB-bilder og gir ut en sti som kjøretøyet skal følge. CARLA-simulatoren blir brukt til å trene og evaluere nettverk. Denne simulatoren gir et fleksibelt og trygt miljø for å kunne utvikle selvkjørende-kjøretøy-modeller raskt.

Modellene i en ende-til-ende-tilnærming vil vanligvis lære å oppfatte omgivelsene ved å prosessere RGB-bilder. Det er også mulig å gi modellene eksplisitte mellomrepresentasjoner. Forskning viser at ved å bruke metoder fra datasyn, som for eksempel semantisk segmentering og dybdeestimering, så kan modellene generalisere bedre og øke ytelsen.

Denne masteroppgaven undersøker om man kan øke ytelsen av nettverk ved å bruke eksplisitte mellomrepresentasjoner i LBC-metoden. Det første eksperimentet viser at LBC kan gjenskapes i den nyeste versjonen av CARLA (0.9.11). Det andre eksperimentet viser at ytelsen og generaliseringen av nettverk økes betraktelig ved å bruke perfekte eksplisitte mellomrepresentasjoner. Disse representasjonene er gitt direkte av simulatoren. Det tredje eksperimentet viser at nettverk også generaliserer bedre når de får mellomrepresentasjoner som er generert av trente oppfatnings-modeller.

Det blir også undersøkt om man kan øke nettverks-ytelsen ved å legge til en fase med forsterknings-læring. Et forsøk ble gjort på å forbedre nettverkene videre med Proximal Policy Optimalization, men dette viste seg å være utfordrende.

Preface

This Master's thesis in Computer Science is a part of the research conducted within the NTNU Autonomous Perception Laboratory (NAPLab).

We would like to thank our supervisor Frank Lindseth, who provided us with valuable feedback and guidance throughout the course of this work.

Contents

List of Figures

List of Tables

Abbrevations

AI	Artifical Intelligence
ANN	Artifical Neural Network
API	Application Programming Interface
ASPP	Atrous Spatial Pyramid Pooling
AV	Autonomous Vehicle
BEV	Bird's-Eye View
CIL	Conditional Imitation Learning
CNN	Convolutional Neural Network
CIRL	Controllable Imitative Reinforcement Learning
DDPG	Deep Deterministic Policy Gradient
DRL	Deep Reinforcement Learning
FCN	Fully Convolutional Network
FPS	Frames Per Second
GPU	Graphical Processing Unit
GAE	Generalized Advantage Estimation
HLC	High Level Command
IA	Implicit Affordances
IL	Imitation Learning
LBC	Learning by Cheating
LSD	Learning Situational Driving
mIoU	Mean Intersection over Union
MLP	Multilayer Perceptron
MSE	Mean Squared Error
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNG	Random Number Generator
ReLU	Rectified Linear Unit
SOTA	State of the Art

Chapter 1

Introduction

This chapter is divided into four sections. Section 1.1 gives a brief summary of the background and motivation behind this thesis. Section 1.2 describes the research goal and the research questions of our thesis. Section 1.3 presents the contributions of our work, and Section 1.4 outlines the structure of the thesis.

1.1 Background and Motivation

A recent study on road traffic accidents by the U.N. shows that traffic-related fatalities rank as the eighth leading cause of death in the world [1]. The World Health Organization states that the total number of such fatalities reach approximately 1.35 million people every year [2]. Furthermore, approximately 94% of traffic accidents in the U.S. are caused by human errors [3].

With these statistics in mind, fully *autonomous vehicles* (AVs) could lead to substantial benefits for human life. AVs could also lead to more efficient traffic flow, productivity gains due to less time spent driving, and less emissions of green house gases [4]. In the U.S., widespread deployment of AVs is estimated to save hundreds of billions of dollars by 2050 [3].

In recent years, AVs have been subject to increased research and development [4]. This is reflected by the vast number of research articles that is published every year. It is also reflected by the efforts of corporations like Tesla, Uber and Google. However, autonomous driving in complex and unpredictable environments remains a difficult challenge.

There are predominately two main approaches for developing AVs; the *modular approach* and the *end-to-end approach*. The modular approach splits the task of driving into a pipeline of modules. Each module performs a separate task, e.g., localization, prediction or planning. Dividing the autonomous driving problem into separate modules makes the system more interpretable. However, developing such a pipeline is costly, and often results in over-engineered solutions. Meanwhile, the end-to-end approach treats the complete task of driving as a single machine learning task, learnable by *artificial neural networks* (ANNs).

The rise of the end-to-end approach has been enabled by the the recent breakthroughs of deep learning. This have allowed for the use of deep neural networks to solve complex tasks, which include object detection, speech recognition and natural language processing. ANNs have also achieved super-human performance in game environments such as Atari

[5] and Dota 2 [6]. In board games such as chess, shugi and go, the AlphaZero system has been capable of defeating world champions [7]. These successes have led to optimism regarding the development and deployment of neural networks in other domains, such as autonomous driving.

Tampuu et al. [8] believe that the end-to-end approach will be of higher interest to the automotive industry than the modular approach. An important reason for this is that the end-to-end approach is more affordable. A single RGB camera is the primary requirement for deploying a developed end-to-end system in the real world. Many other sensors, such as LiDAR [9], raise the costs of the system significantly. This motivates the pursuit of a camera only, end-to-end based solution for autonomous driving.

Simulators such as CARLA [10] have proven to be incredibly beneficial for developing end-to-end systems. These simulators let researchers quickly organize and conduct experiments, without any safety concerns. Research conducted in CARLA often evaluate systems on the challenging NoCrash benchmark [11]. This benchmark consists of tasks in urban environments with varying difficulty.

Neural networks in end-to-end approaches are usually trained to output actuator commands based on RGB images captured by a monocular camera. These neural networks can be trained with *imitation learning* (IL). This is a supervised learning method, where the networks learn on demonstrations performed by an expert. Another method that has garnered more attention recently is *reinforcement learning* (RL). This approach trains the network to maximize a reward signal by experimenting with different actions in the environment. RL allows the network to learn how to recover from mistakes, which can make the network more robust to diverse traffic scenarios.

Both IL and RL approaches have shown promising results. However, networks trained with IL often fail to generalize to new environments. This is often due to lack of variety and inherent bias in the demonstrations [11]. RL methods are usually not data efficient, and will often require more computation than IL methods. They are also challenging to apply in real environments. In general, end-to-end solutions suffer from a lack of *interpretability*. Understanding erroneous behavior is therefore difficult.

End-to-end systems can also use *intermediate representations*, such as semantic segmentation and depth images. Zhou et al. [12] investigated how using explicit intermediate representations from computer vision affects the performance of end-to-end systems. Their results showed that these representations help networks learn faster, generalize better, and achieve higher task performance. They deem semantic segmentation and depth images the most useful representations. These representations can also make the system more interpretable [13]. In this thesis, such representations are also referred to as *computer vision images*. Neural networks that generate such representations are referred to as *perception models*.

In this thesis, we investigate and build on the work by Chen et al. [14], who introduced the *Learning By Cheating* (LBC) approach for training AVs. Their research was conducted with CARLA, and was evaluated with the NoCrash benchmark. LBC separates the problem of learning to drive into two distinct tasks; learning to *act*, and learning to *see*. First, a *privileged* network is trained to drive on semantically segmented bird's-eye view (BEV) images. These BEV images contain high-level information about the state of the world, such as pedestrians, vehicles and traffic lights. This information allows the privileged network to focus on the task of learning to act. Next, a *sensorimotor* network that receives RGB images as input is trained by IL, where the privileged network acts as the expert.

The sensorimotor network can query the privileged network on what the optimal action is in any situation. This allows the sensorimotor network to focus on the task of learning to see. This two-staged approach proved to be very effective, with the sensorimotor network achieving *state-of-the-art* (SOTA) results on the NoCrash benchmark.

We were interested in LBC because it had proven to be an incredibly effective, camera-only-based approach. Additionally, it was an interesting and unique solution to the autonomous driving problem. The code for LBC was also available as an open-source repository, which included an implementation of the NoCrash benchmark. This code gave us a good starting point from where we could continue with our research.

CARLA has undergone significant updates after the release of the LBC paper. At the time of writing this thesis, the most recent version of CARLA is version 0.9.11. The authors of LBC used CARLA 0.9.6. Therefore, we attempt to reproduce the results of LBC in the newest version. This allows us to inspect how the LBC approach performs in an updated simulated environment, which has more realistic physics and graphics. Inspired by recent research, we then try to improve the performance of the sensorimotor network in two different ways; with methods from computer vision and RL.

We investigate what happens when the sensorimotor network is given access to explicit representations from computer vision, particularly semantic segmentation and depth images. This is motivated by the results of Zhou et al. [12]. We hypothesize that by providing additional representations from computer vision, the sensorimotor network will generalize better and achieve higher task performance. We investigate this by using ground truth computer vision images provided directly from the simulator.

Furthermore, we train and evaluate different neural networks that perform semantic segmentation and depth estimation. The output of these networks, together with the corresponding RGB image, is then used as input to train a new sensorimotor network.

In the RL experiment, we attempt to train the sensorimotor network further without supervision of the privileged network. Additionally, combining LBC with RL was something that the LBC authors stated could be a potential direction for future work. We use the proximal policy optimization algorithm and a pretrained sensorimotor network. The RL stage will hopefully allow the sensorimotor network to learn from the new situations it might put itself in. By starting with a good initial policy, the agent will have skipped the computationally expensive early stages of training [15]. Using RL is also inline with several other recent works, which have opted for methods that do not utilize experts with access to high-level information [16, 17].

1.2 Research Goal and Research Questions

In this thesis, we investigate LBC [14], a state-of-the-art IL-based approach for training neural networks for autonomous driving in CARLA. We want to expand on LBC, and improve the performance of networks trained with this approach. In this context, *performance* refers to the ability to drive efficiently and safely in urban environments. We formulate our research goal as follows:

Research Goal: Improve the performance of neural networks trained with the LBC approach.

To achieve the research goal, we pose the following research questions (RQs):

- **RQ1**: Can the results of LBC be reproduced in the newest version of CARLA?
- **RQ2**: Can the performance of the sensorimotor network be improved by providing it with perfect semantic segmentation and depth images directly from the simulator?
- **RQ3**: Can the performance of the sensorimotor network be improved by providing it with intermediate representations produced by networks trained for semantic segmentation and monocular depth estimation?
- **RQ4**: Can the sensorimotor network be improved with the use of an additional RL stage?

Four experiments have been conducted to answer each of these RQs.

1.3 Contributions

To the best of our knowledge, reproducing LBC in CARLA 0.9.11 has not been conducted in any published works. However, the concurrent work by Chen et al. [18], reproduced LBC in CARLA version 0.9.10. We are not aware of any other works which have extended LBC with explicit intermediate representations. Furthermore, we have not discovered any other works that have trained the sensorimotor network from LBC with an additional RL stage.

The results of the reproduction experiment indicate that the LBC approach can be reproduced in the newest version of CARLA. However, this requires extensive tuning of the PID controller parameters.

When the sensorimotor network is provided with additional ground truth semantic segmentation and depth images, the performance increases and generalizes better. The results show that the sensorimotor network drives more efficiently and safely, without requiring any tuning of controller parameters.

Furthermore, when the sensorimotor networks is provided with intermediate representations predicted by neural networks, the performance increased in test conditions. The performance was weaker in training conditions compared to the reproduced network. No tuning of PID controller parameters was required, and it had fewer traffic light violations compared to the reproduced network. The utilization of these intermediate representations also helped the system become more interpretable, which is beneficial for real world deployment.

In the RL experiment, the sensorimotor network was never able to improve. The performance degraded in all cases. This might be due to inefficient exploration, an unbalanced reward signal, or too many tunable network parameters.

1.4 Thesis Structure

This thesis is structured into six chapters:

- **Chapter 1 - Introduction**: Describes the background and motivation for the thesis, and explains where this work is situated in the field of end-to-end AV research.

The research goal, research questions, and contributions of this thesis are also presented in this chapter.

- **Chapter 2 - Background and Related Work**: Gives a theoretical foundation for the thesis. This includes machine learning basics, deep learning, computer vision, approaches to developing AVs, relevant technology, and related work.
- **Chapter 3 - Methodology**: This chapter explains the methodology of our work. It presents the technology we have used for our research. It also explains how we updated the LBC code for CARLA 0.9.11 and fixed various issues. Furthermore, this chapter describes the experimental setup and plan for the experiments. We reproduce the LBC approach in the first experiment. In the second experiment, we give the sensorimotor network ground truth depth and semantic segmentation images directly from the simulator. In the third experiment, we train and evaluate different networks for monocular depth estimation and semantic segmentation. The output of these networks is then used as input to train a sensorimotor network. In the fourth experiment, we expand LBC with an additional RL stage.
- **Chapter 4 - Results**: Presents the results from each of the four experiments.
- **Chapter 5 - Discussion**: Evaluates and discusses the results from the experiments. The RQs are also addressed. The chapter ends with a reflection over the shortcomings of the research.
- **Chapter 6 - Conclusion and Further Work**: Concludes the thesis, describes the significant findings, and presents ideas for further work.

Chapter 2

Background and Related Work

This chapter covers the theoretical foundations for this thesis. Section 2.1 and Section 2.2 introduces the machine learning foundations and deep learning techniques which are the basis for end-to-end autonomous vehicles. In Section 2.3 the tasks of semantic segmentation and monocular depth estimation are introduced. The approaches of organizing and creating autonomous vehicle systems are discussed in section 2.4. Various tools and technology used in the thesis are introduced in Section 2.5. Section 2.6 discusses research papers that are relevant to this thesis.

2.1 Machine Learning

In the field of AI, an *agent* is a general term meant to symbolize anything that perceives and exists in an environment [19]. *Machine Learning* is a sub-field of AI, where the main goal is to make a machine, i.e. an agent, learn by the use of data and experience. Machine learning systems can be divided into different categories based on how they perform the learning procedure. This section will briefly explore three categories of learning, namely *supervised*, *imitation* and *reinforcement* learning.

Before delving into these topics, some preliminary terminology must be explained:

- **Training data**: Training data is the information and experience that any machine learning system requires for learning. This data consists of a number of different *instances* or *examples*, and is denoted by D. The shape, form and origin of the data will vary depending on the type of learning being performed.
- **Offline and online learning**: In offline learning, the machine learning system is learning on a static dataset D which is collected prior to training. In online learning, the system can collect new data as training progresses, and instances can also be discarded.
- **Policies**: For some learning systems, the goal is to learn a *strategy* that specify the most optimal *action* available in any given situation. This strategy is also known as a *policy*, and is denoted by π. A policy is essentially a function which maps states of the environment to actions. States and actions are denoted by s and a, respectively. The policy can be *deterministic*, which means that π will always output the same action a for any arbitrary state s, i.e., $a = \pi(s)$. It can also be *stochastic*, which

means that the action is sampled from the policy, i.e., $a \sim \pi(\cdot|s)$. Furthermore, the sampled action will not always be the optimal action. On the other hand, an *expert policy* always chooses the optimal action, and is denoted by π^*.

Depending on whether the action and environment spaces are discrete or continuous, π can be implemented with numerous representations. For instance, π can be implemented as a tabular representation, a decision tree, or an artificial neural network.

2.1.1 Supervised Learning

In supervised learning, the training data consists of a number of different examples:

$$(x_1, y_1), (x_2, y_2), ...(x_n, y_n),$$

where each x_i is an input, and y_i is the corresponding desired output. That is, for all x_i, there exists an unknown function f^*, such that $f^*(x_i) = y_i$. This function is often referred to as the target function. The goal of supervised learning is to learn an approximation f of the target function f^* [19].

2.1.2 Imitation Learning

The goal of *imitation learning* (IL) is to learn a mapping between observations and actions, by learning from the demonstrations performed by an *expert* [20]. More specifically, the learner wants to learn a policy π that maps observations o_t to actions a_t.

Behavior Cloning
When this problem is formulated as a supervised learning problem, it is called *behavior cloning* [4]. In behavior cloning, the training data D consists of demonstrations collected by an expert, and comes in the form of several observation-action pairs:

$$(o_1, a_1), (o_2, a_2), ...(o_n, a_n),$$

where o_t is the observation at time t, and a_t is the corresponding desired action performed at time t.

DAgger
Dataset aggregation (DAgger) [21] is an online IL algorithm that uses an expert policy π^* to train a new deterministic policy $\hat{\pi}$. First, it uses the expert policy to collect a dataset of trajectories D_1. A trajectory is a sequence of observations and actions, i.e $o_0, a_0, o_1, a_1, ..., o_T, a_T$, and is denoted by τ.

Next, the trajectories in D_1 is used to learn a policy $\hat{\pi}_1$. For the next iteration, $\hat{\pi}_1$ is used to collect a new set of trajectories D_2. The new trajectories are aggregated with the previously collected dataset, i.e., $D_2 \leftarrow D_1 \cup D_2$. Then, a new policy $\hat{\pi}_2$ is trained on those trajectories. The algorithm repeats this step for n iterations. For every iteration, the current policy $\hat{\pi}_i$ is trained on an aggregate of all datasets (i.e. $D_i \leftarrow D_1 \cup D_2 \cup ... \cup D_i$), and is the policy that best imitates π^* on the aggregated dataset.

It also allowed to continue using the expert policy to collect data beyond the initial iteration. This is done by using the expert only a fraction of the time. This is desirable because the trained policies for the first few iterations might perform much worse than the expert, which in turn might result in a dataset containing irrelevant instances. The policy for iteration i is now defined as $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$. By setting $\beta_1 = 1$, there is also no need to provide an initial policy $\hat{\pi}_1$. The pseudocode for DAgger can be seen in Algorithm 1.

A major disadvantage of DAgger is the computational resources required for maintaining the growing size of the dataset D. Another downside is that the algorithm requires an expert policy, which might not be available.

Algorithm 1 DAgger - With slight modifications from the original paper [21]

```
1: Initialize $D \leftarrow \emptyset$
2: Initialize $\hat{\pi}_1$
3: for $i = 1$ to $N$ do
4:     Set $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
5:     Sample trajectories using $\pi_i$.
6:     Collect dataset $D_i = (o, \pi^*(o))$ by visiting states using $\pi_i$.
7:     Aggregate datasets: $D \leftarrow D \cup D_i$
8:     Train the next policy $\pi_{i+1}$ on $D$.
9: end for
10: return policy $\pi_i$ with the highest validation score.
```

2.1.3 Reinforcement Learning

Generally speaking, the goal of *reinforcement learning* (RL) is to learn a policy that maximizes a *reward* signal over time. The rewards are scalar values which are received when an agent applies actions to an environment. In Section 2.2.7, *deep reinforcement learning* (DRL) will be explored. DRL implements RL methods using deep neural networks.

Sutton and Barto [22], and Achiam [23] give comprehensive descriptions of RL.

Markov Decision Process
RL can be formulated as a sequential decision making problem known as a *markov decision process* (MDP). A MDP is an idealized, flexible, and abstract mathematical formulation of the RL-problem, with several useful theoretical properties. MDPs are defined by a set of states $\mathbf{S}$, a set of actions $\mathbf{A}$, a reward function R and a transition probability function p. The reward function outputs a scalar value r_t, based on state s_t, action a_t, and the next state s_{t+1} for time step t. That is $r_t = R(s_t, a_t, s_{t+1})$.

The *transition* probability function p outputs the probability of transitioning to the next state $s_{t+1} = s'$ conditioned on applying action $a_t = a$ in the current state $s_t = s$. It is defined as follows:

$$p(s'|s, a) \doteq Pr\{s_{t+1} = s'|s_t = s, a_t = a\}$$

This function exhibits the *markov property*, which means that the probability of transitioning to the next state is only dependent on the directly preceding state and action.

A MDP functions as follows: at time step t, the agent applies action a_t based on the current state s_t. With probability $p(s_{t+1}|s_t, a_t)$, the environment transitions from state s_t to s_{t+1} when applying action a_t. When the agent explores the world, it generates a trajectory τ, which is a sequence of states and actions. The first state s_0, is sampled from the *start state-distribution* ρ_0 [23].

Reward Functions

The agent tries to maximize the reward signal defined by the reward function R. This function was previously defined as $r_t = R(s_{t+1}, s_t, a_t)$. The reward function can also be defined over a trajectory τ. Following the notation of Achiam [23], we denote the rewards received from a trajectory as $R(\tau)$:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

This sum is calculated over a *finite horizon*, as it adds a fixed number of terms together. The sum can also be discounted with a parameter γ over an *infinite horizon*. This definition values rewards closer in time more than distant rewards. The γ parameter also guarantees that the sum of rewards will converge. The discounted cumulative sum of rewards is defined as follows:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

Sometimes we want the cumulative reward from a specific time step t'. Achiam [23] calls this a *reward-to-go*. The discounted reward-to-go for time-step t is defined as:

$$\hat{R}_t = \sum_{t'=t}^{T} \gamma^t R(s_{t'}) \tag{2.1}$$

Designing a good reward function is a crucial part of implementing a RL training procedure. As Sutton and Barto [22] explains, the reward function should be designed to focus on *what* we want the agent to achieve, and not on *how* we want the agent to achieve it. The reward function should also take into account the *density* of the rewards. If the rewards are too sparse, it can lead the agent to wander the environment aimlessly without learning anything useful. This problem occur when the reward signal produces approximately zero-valued rewards too often. This can lead the agent to wander the environment aimlessly without learning anything useful. Another challenge with reward-function design is the *credit assignment problem.* This is the problem of assigning credit or blame to actions for rewards that are received much later in time.

The Central Optimization Problem of RL

When the policy is stochastic, each action is sampled from the policy which is conditioned on the current state s_t. That is, $a_t \sim \pi(\cdot|s_t)$. The probability of a trajectory τ with T-steps, conditioned on the policy π then becomes:

$$p(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

The RL agent seeks to maximize the rewards over the trajectory τ, i.e. $R(\tau)$. The expected return of τ then becomes:

$$J(\pi) = \int_{\tau} p(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] \tag{2.2}$$

This means that the central optimization problem of RL is finding an optimal policy π^* that satisfies $\pi^* = \underset{\pi}{argmax}\ J(\pi)$.

Value Functions

Two essential functions in regards to RL is the *value function* $V^\pi(s)$, and the *action-value* *function* $Q^{\pi}(s,a)$. In simple terms, these functions give a measure of how good it is to be in a state. The value function $V^{\pi}(s)$ gives the expected discounted return when starting in state s, and following policy π forever after:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s] = \mathop{\mathbb{E}}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$$

The action-value function $Q^{\pi}(s,a)$ gives the expected discounted return after applying action a in state s, and following policy π forever after:

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] = \mathop{\mathbb{E}}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]$$

$Q^{\pi}(s_t,a_t)$ can also be defined in terms of $V^{\pi}(s_{t+1})$:

$$Q^{\pi}(s_t,a_t) = \mathbb{E}\left[r_t + \gamma V^{\pi}(s_{t+1})\right]$$

Value functions mitigate the effect of the credit assignment problem, as they give a measure of the value of an action before the rewards arrive [24]. They are utilized to some capacity in almost all RL algorithms. Another closely related function to $V^{\pi}(s_t)$ and $Q^{\pi}(s_t,a_t)$, is the *advantage function* $A^{\pi}(s_t,a_t)$. This function measures the relative advantage of applying action a_t in state s_t, in comparison to the default action chosen by the policy π [24]. It is defined as follows:

$$A^{\pi}(s_t,a_t) = Q^{\pi}(s_t,a_t) - V^{\pi}(s_t) \tag{2.3}$$

2.2 Deep Learning

Like other machine learning approaches, the main purpose of *deep learning* is to learn an approximation f of an unknown function f^* by fitting a set of training data D. In deep learning, f is a type of function known as an *artificial neural network* (ANN) [1], that is defined by set of tunable parameters θ.

This section will describe the preliminary theory required for understanding the procedure of creating, training and validating ANNs. It will begin by introducing perceptrons and feedforward neural networks. It will also describe activation functions, loss functions, gradient descent, in addition to regularization and optimization techniques. Some topics from deep reinforcement learning will also be presented, particularly the main principles of policy gradient optimization, and the TRPO and PPO algorithms.

Goodfellow et al. [25] and Nielsen [26] give comprehensive descriptions of deep learning.

2.2.1 Perceptron and Feedforward Neural Networks

The *perceptron* is a simple mathematical model that takes a weighted input $\sum_{i=1}^{n} x_i w_i$ plus a bias term b, and produces a single binary output. Here, x_i is an input, and w_i is the corresponding weight. The output of the perceptron is defined with the following equation:

[1]In this thesis, ANNs are also simply referred to as *neural networks*.

$$y = \begin{cases} 0 & \sum_{i=1}^{n} x_i w_i + b \leq 0 \\ 1 & otherwise \end{cases}$$

The perceptron is the first example of an *artificial neuron*, as its design is inspired by biological neurons found in the brain. Quite famously, the perceptron has a major limitation, as they are unable to learn non-linearly separable functions. By creating a composite function consisting of multiple layers of perceptrons, the result is a *multilayer perceptron* (MLP). However, this function is still only a linear function of its input [25].

To introduce non-linear approximation capabilities into MLPs, perceptrons must use non-linear activation functions. These types of perceptrons, with any arbitrary non-linear activation function, are often called *neurons* or *units*. *Feedforward neural networks* are composed of several layers of neurons. Figure 2.1 shows a visual example of this type of network.

When an input $\mathbf{x_i}$ is propagated through the network, all neurons in each layer are *activated* or *fired* in parallel, propagating their activations to the next layer. Using the same notation from the work of Nielsen [26], the activation of an arbitrary neuron j in layer l is defined with the following equation:

$$a_j^l = g(\sum_k a_k^{l-1} w_{jk}^l + b_j^l).$$

Here, g is an activation function, b_j^l is the bias, and w_{jk}^l denotes the weight from neuron k in layer $l-1$, to neuron j in layer l.

Feedforward neural networks are mathematically defined as composite functions: $f(\mathbf{x}) = f^n(f^{n-1}(...f^2(f^1(\mathbf{x}))))$, where $\mathbf{x}$ is an input vector, and f^i is the ith layer in the network. The first layer is called the input layer, the last layer is called the output layer, and any layer in-between are called *hidden* layers. Each layer perform matrix multiplication between a matrix of weights $\mathbf{W}$ and a vector of inputs $\mathbf{x}$. Next, a bias term $\mathbf{b}$ is added and the activation function g^i is applied. The output of layer i is then $f^i(\mathbf{x}) = g^i(\mathbf{xW} + \mathbf{b})$. Information only flows in the direction from the input layer to the output layer in these networks, hence the name *feedforward*.

When the term *architecture* is used in the context of neural networks, it is generally referring to the number of layers in the network, the amount of units in each layer, and how these layers are connected to each other. The term *deep learning* is derived from the fact that the network typically consists of many layers. When there are connections between every single neuron in all layers in the entire network, it is known as a *fully connected network*.

2.2.2 Activation Functions

Selecting the appropriate activation functions is a condition for the *universal approximation theorem* to apply. This theorem states that a neural network can approximate any continuous target function f^* on a closed and bounded subset of $\mathbb{R}^n$, if two criteria are met. Firstly, at least one hidden layer must have a sufficient amount of neurons. And secondly, that same layer must use an activation function that saturates for very positive or negative values, i.e. a "squashing" function" [25]. The activation functions also affect the training speed and convergence rate of the network.

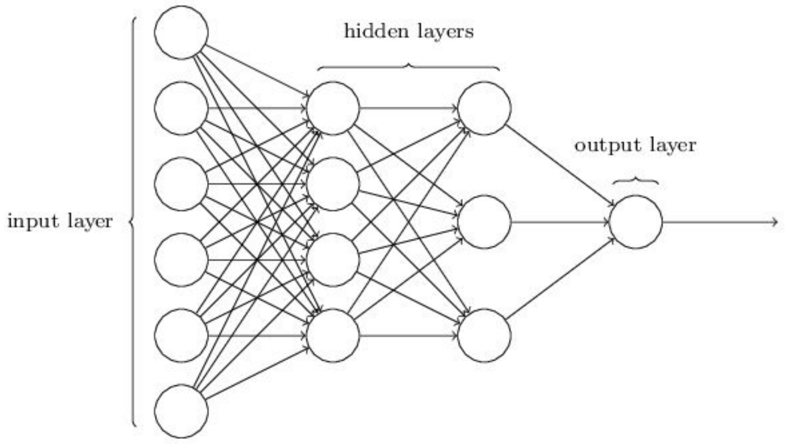


Figure 2.1: An example of a fully connected feedforward neural network with two hidden layers, and a single neuron in the output layer. The bias connections are not visualized. Image taken from the work of Nielsen [26].

Following the notation of Nielsen [26], we denote the weighted input plus bias as z, that is $z = \sum_{i=1}^{n} x_i w_i + b$. Here is a list of common activation functions:

Sigmoid
The sigmoid function outputs values in the range [0, 1]. It is denoted by σ:

$$\sigma(z) = \frac{1}{1 + e^{(-z)}}$$

Hyperbolic Tangent
The hyperbolic tangent function outputs values in the range [-1, 1]. It is denoted by *tanh*:

$$tanh(z) = 2\sigma(2z) - 1,$$

where σ is the sigmoid function.

Softmax
Softmax is a commonly used activation function for classification tasks. For each element z_i in a vector $\mathbf{z}$, it performs the following activation denoted by S:

$$S(z_i) = \frac{e^{z_i}}{\sum_j e^(z_j)}$$

This will map every element z_i to the range $[0, 1]$. $S(z_i)$ then represents the probability that instance $\mathbf{z}$ belongs to class i, and $\sum_i S(z_i) = 1$.

Rectified Linear Unit
The rectified linear unit (ReLU) function outputs z if z is higher than zero. Otherwise, its activation is equal to zero.

$$ReLU(z) = max(0, z)$$

ReLU is not a "squashing" function, but the universal approximation theorem have been proven to apply for ReLU as well [25].

2.2.3 Cost Functions

Cost functions give an estimate for the degree of error the network makes for every training instance that propagates through the network. Cost functions are often called loss functions, and we use both terms interchangeably in this thesis. The choice of cost function is an important design decision, as the gradient of the cost function is used to optimize the network. In the following list of cost functions, the predicted output is denoted by $\hat{\mathbf{y}}$, the target as $\mathbf{y}$, and the number of instances in the training data as n:

Mean Squared Error (MSE)

$$MSE(\hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2,$$

Root Mean Squared Error (RMSE)
This function outputs the root of the MSE.

$$RMSE(\hat{\mathbf{y}}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2},$$

Cross Entropy
This cost function is suitable if the network is performing classification or logistic regression. Cross entropy is denoted by H:

$$H(\hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i$$

2.2.4 Backpropagation and Gradient Descent

In order to update the weights of the network, the gradient of the cost function with regards to every single adjustable parameter in the network must be calculated. The gradient is the partial derivative of the cost function $\frac{\partial L}{\partial w_i}$, and indicates how much a slight adjustment of w_i will change the output of the cost function. Optimization algorithms use the gradients to update the parameters of the network [25].

To calculate the gradients, an efficient technique known as the *backpropagation* algorithm is applied. It consists of two stages. First, a subset of the training data is propagated through the network, producing predictions at the output-layer. These predictions are then applied to the cost function. This stage is known as the *forward pass.* The second stage is the *backward pass*, and involves applying the chain rule from calculus to compute the gradient of the cost function. It starts by computing the error gradient at the output layer, and works its way backwards to the input-layer by propagating the error gradient backwards. That is, it computes $\forall_j \frac{\partial L}{\partial w_j}$, and $\forall_j \frac{\partial L}{\partial b_j}$, where L is the cost function, and w_j and b_j is a weight and bias in the network, respectively [26].

The subset of training examples that passes through the network is called a *mini-batch.* However, in this thesis it is referred to as a *batch.* After the batch has passed through the network, and all gradients have been calculated, the weights and biases are updated with a *gradient descent* step as follows:

$$\forall_j w_j \longrightarrow w_j - \frac{\eta}{m}\sum^{m} \frac{\partial L}{\partial w_j},$$

$$\forall_j b_j \longrightarrow b_j - \frac{\eta}{m}\sum^{m} \frac{\partial L}{\partial b_j}$$

Here η is a hyperparameter called the *learning rate*, and m is the number of instances in the batch. The overall goal of gradient descent is to iteratively minimize the error of the cost function, by adjusting the parameters gradually. The learning rate η decides the step size the algorithm should take in the direction of the gradient. However, if the learning rate is too large, the optimization algorithm may oscillate instead of descending into a minimum.

Training a network involves sampling a single batch from the training data, passing the batch through the network, applying the loss function, calculating the gradient, then updating all parameters. Next, a new batch is sampled, and the same procedure is performed over again. When the entire training set has passed through the network, a single *epoch* of training has finished. Training continues until the average loss of each epoch has converged.

2.2.5 Training Neural Networks: Problems and Solutions

This section describes some of the common problems that might occur during training of neural networks, and how they can be mitigated. *Batch normalization* layers, the *Adam* optimizer, *data augmentation*, and *dropout* is also described in this section.

Common Problems
Overfitting is a typical problem where the network is able to make precise predictions for instances in the training data, but makes weak predictions for instances beyond the training data. That is, it has poor *generalization* capabilites. The effect of overfitting can be diminished by introducing more diverse training data, or by designing networks with less parameters [19]. We can create more diverse training data with *data augmentation*, which means that instances in the training data are augmented by random transformations. Another solution to reduce the effect of overfitting is to remove errors and outliers in the data, or to utilize regularization techniques such as *dropout*, or l_1 and l_2 regularization [27].

Covariate shift (or *distribution shift*) [28, 8] is a problem which can occur when the distribution of features is different between seen and unseen data. Covariate shift is especially relevant in regards to behavior cloning (explained in Section 2.1.2). Networks trained by behavior cloning tend to perform well for states that are present in the training data, but generalizes poorly for new states. A potential solution for covariate shift is using an online algorithm such as DAgger (Algorithm 1).

The *vanishing/exploding gradients* problem occurs when the error gradient values are either very small or very large. The problem is often caused by using saturating activation functions, where the derivative gets close to zero for extreme values. This means that the gradient update results in an insignificant change. This makes training extremely slow, as networks get stuck in local minima.

Training, Validation and Test Sets

In order to detect poor generalization capabilities, it is common practice to split the training data into a *training*, *validation* and *test* sets. The training set is used for training the network, while the test set is used for evaluation after training has completed. The predictions on the test set give an estimate of the overall performance of the network. The validation set is used after every epoch to detect if the network is overfitting the training set. If the loss on the validation set increases while the training loss decreases, it might be an indication of overfitting.

Batch Normalization

A batch normalization layer [29] normalizes inputs for the succeeding layer. They do this by calculating the mean μ and standard deviation σ. Let $\mathbf{X}$ be a batch of inputs propagating through a neural network. Then the normalized batch $\tilde{\mathbf{X}}$ is computed as follows:

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \mu}{\sigma}.$$

Batch normalization allows for using higher learning rates, saturating activation functions, and leads the network to be less sensitive to weight initialization. It also acts as a form of regularization, reduces the effect of vanishing/exploding gradients, and makes training faster. The exact reason why batch normalization is so effective is poorly understood, but research indicates it smoothes the optimization landscape [30]. The major negative aspect is the extra computational burden, which increases runtime when making predictions. However, this is balanced by the tendency of batch normalization layers to make models converge faster with less epochs needed [27].

Adam Optimizer

Adaptive moment estimation (Adam) [31] is an adaptive learning rate optimization algorithm. In contrast to regular gradient descent algorithms (e.g. SGD), where the network can be stuck in local minima for several epochs due to small gradient updates, Adam updates the parameters by using *momentum* optimization. Momentum optimization takes the history of previous gradients into consideration before applying the update. Adam calculates two different moment estimates to either accelerate or decelerate learning [27].

For every single batch containing m instances at time step t, Adam calculates $\mathbf{s}$ with decay rate ρ_1 and $\mathbf{r}$ with decay rate ρ_2. $\mathbf{s}$ and $\mathbf{r}$ are the first and second-order moment estimates, respectively. The decay rates are hyper-parameters set by the user[2]. These moment estimates are then rescaled for bias, denoted by $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$. Adam performs the following operations:

$$\mathbf{g} \longleftarrow \frac{1}{m}\sum_j \frac{\partial L}{\partial w_j}$$

$$\mathbf{s} \longleftarrow \rho_1\mathbf{s} + (1-\rho_1)\mathbf{g}$$

$$\mathbf{r} \longleftarrow \rho_2\mathbf{r} + (1-\rho_2)\mathbf{g}\odot\mathbf{g}$$

$$\hat{\mathbf{s}} \longleftarrow \frac{\mathbf{s}}{1-\rho_1^t}$$

$$\hat{\mathbf{r}} \longleftarrow \frac{\mathbf{r}}{1-\rho_2^t}$$

$$\boldsymbol{\Delta}\theta = -\eta\frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}.$$

[2]The PyTorch implementation of Adam sets ρ_1 to 0.9 and ρ_2 to 0.999 by default.

Here, $\mathbf{g}$ denotes the gradients computed for the current batch, η is the learning rate, δ is a small value used to avoid numerical instabilities, and θ is a set of parameters in a neural network. Finally, the weights are updated with: $\theta \longleftarrow \theta + \mathbf{\Delta}\theta$.

Dropout
Dropout is a regularization technique where the output of neurons are set to zero with probability p during training. This technique is motivated by the main principle of *ensemble learning*. Generally, ensemble learning is about training an ensemble of different models, then averaging the predictions from all models when making a prediction. A random set of neurons are dropped for every batch, and the remaining connections make up a new model. This means that after training has finished, the final model will be the average, i.e. the ensemble, of all models that were generated by dropout during training [27].

2.2.6 Convolutional Neural Network

A *convolutional neural network* (CNN) employ a special type of layer known as a *convolutional layer*. Convolutional layers use an operation designed for processing data with a grid-like topology, e.g. image data. It can be defined for any n-dimensional array of inputs. The definition for the two-dimensional case is presented here, where I is the input and K is a two-dimensional kernel (or *filter*) of weights:

$$S(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i-m, j-n)K(m,n)$$

This equation defines an operation where a kernel K of size m x n slides over the input, extracting features along the way. The end result is a two-dimensional output S, which is called a *feature map*. $S(i,j)$ is the neuron positioned at row i and column j in feature map S. After this weighted sum is calculated, an activation function is applied for all values in S before being sent to the next layer.

The size of S depends on the sizes of K and I, and whether or not *strided* convolutions are used, or if the input has been *padded* with zeros. A strided convolution will skip some specified number of rows and columns as the filter slides over the input. For instance, one *columns* will be skipped when the horizontal stride equals 2. *Padding* is an operation where rows and columns filled with zeros are inserted around the input. Padding is used for getting the desirable output resolution, and to ensure that all values in the input are used during convolution.

A convolutional layer can use several kernels, each with its own set of weights. Every kernel produces a separate feature map, which means that the next layer will receive a stack of n feature maps. To perform convolution over data with n-channels, the kernel must have n-channels as well. A n-dimensional kernel is trying to capture both spatial and cross-channel features simultaneously.

Pooling
CNNs can employ a *pooling* layer after the activation function. Pooling operations work by applying an operation over a rectangular subset of a feature map. *Max pooling* returns the maximum value of a rectangular subset. *Average pooling* returns the average value of a rectangular subset. *Global average pooling* returns the average value of the entire feature map. Pooling operations are used to reduce the amount of parameters, storage requirements and computational costs, while retaining most of the information. They also

help the network to be invariant to small translations. This means that small changes in the input will not always effect the output of the pooling operations.

Motivation
According to Goodfellow et al. [25] there are primarily three ideas which motivate the design of CNNs:

- **Sparse weights**: Neurons in convolutional layers does not have connections to every singly unit in the preceding layer, but only to a n-dimensional rectangular subset. This is in sharp contrast to fully connected feedforward networks, where every neuron is connected to every unit in the preceding layer. This means that fewer parameters need to be stored, and optimization takes less time.
- **Parameter sharing**: In fully connected networks, each layer's weight matrix $\mathbf{W}$ is only used once during the forward pass. For a convolutional layer, the weights of each kernel are used several times to compute the value for each neuron in a feature map. More precisely, neurons located in the same feature map share the same set of parameters in order to compute their own value. Sharing parameters significantly reduces the storage requirements.
- **Equivariance**: The convolutional operation is equivariant to translation. This concept can be explained with an example; let us say a CNN is used for detecting traffic signs in images. No matter where the signs are located in the image, the convolutional operation will be able to find useful features that are relevant for this specific task. Since neurons in a feature map share the same kernel, changing the location of a traffic sign in the image will also change the location of the activation in the feature map.

CNNs have been used to great success for image-processing tasks[3], and have become a fundamental building block in several network architectures related to computer vision. There also exists other variants of the convolutional operation, namely *transposed convolution*, *depthwise separable convolution*, and *atrous convolution.*

Transposed Convolution
CNNs typically decrease the height and width of feature maps the further the input propagates through the network. This is in contrast to *transposed convolutional* layers, which perform an upsampling operation. This operation is equivalent to adding rows and columns filled with zeros to the input, then applying a regular convolutional operation [27]. The result of this operation is a feature map with a larger height and width than the input.

Depthwise Separable Convolution
When the input to a convolutional layer is an n-dimensional array, the kernel will perform spatial and cross-channel feature extraction simultaneously. Meanwhile, a *depthwise separable convolutional* layer perform spatial and cross channel feature extraction *separately.* For instance, if the input is n-dimensional, the depthwise separable operation applies n separate spatial filters to each channel. This results in a stack of n-feature maps. Then, the depthwise operation is applied, which is a regular convolutional operation using a 1x1-kernel with n-channels. Separating the convolutional operation decrease memory and computational requirements, as well as reducing the number of parameters in the network. Depthwise separable convolutional layers have shown to perform at least as well as standard convolutional layers. However, they should generally not be used if the input has a

[3]CNNs have also proven to be effective for other tasks, such as voice recognition and natural language processing [27]

small of number of channels, e.g. the first layer in a network processing images with RGB channels [27].

Atrous Convolution

Atrous convolution (or *dilated convolution*) uses a special filter defined by the *atrous rate* parameter (denoted by r). Consider the example of a two-dimensional input I and a 3x3 kernel. The atrous operation with rate r, will add $r-1$ zero valued weights between any pairs of consecutive weight values in K. If the rate equals 2, K will *dilate* to the size of 5 x 5. As can be seen in Figure 2.2, the atrous rate decides the field of view of the kernel. When r equals 1, the atrous operation is equivalent to the standard convolutional operation.

Atrous convolution allows for using a kernel with a larger field of view without losing resolution of the input, while not increasing computational costs and storage requirements [32]. These type of layers play an important role in some network architectures that perform semantic segmentation [33, 34], which will be explained later in Section 2.3.2.

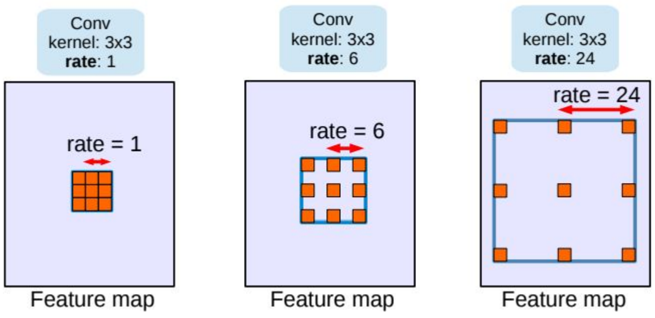


Figure 2.2: Examples of dilated kernels used in atrous convolutional layers. The blue background is a two-dimensional feature map. Orange squares represent non-zero weight values. The "holes" between orange squares represent zero-valued weights. When $r = 1$ the kernel is a standard convolutional kernel. Image taken from the DeepLabv3 paper [34].

2.2.7 Deep Reinforcement Learning

This section builds on Section 2.1.3, and will explore some selected topics from the field of *deep reinforcement learning* (DRL). This section will mainly explore policy optimization methods, such as TRPO [35] and PPO [36], and introduce GAE [24].

In DRL, the policies and value functions are represented mathematically as differentiable functions. These functions are implemented as neural networks with parameters θ. For instance, π_θ denotes the policy defined by θ. Furthermore, $\hat{Q}^{\pi}_{\phi}(s_t, a_t)$ and $\hat{V}^{\pi}_{\phi}(s_t)$ denotes approximations of the action-value function and value function with parameters ϕ, respectively. *Actor-critic* algorithms either use $\hat{Q}^{\pi}_{\phi}(s, a)$ or $\hat{V}^{\pi}_{\phi}(s)$ for learning π_θ. The *actor* is π_θ, which explores and applies actions to an environment, while *critic* is either $\hat{Q}^{\pi}_{\phi}(s, a)$ or $\hat{V}^{\pi}_{\phi}(s)$. The critic evaluate the behavior of the actor by giving estimates of the expected return.

DRL algorithms can roughly be split into two categories; *Q-learning* and *policy optimization* methods. Q-learning methods learn an approximation of the action-value function $\hat{Q}^{\pi}_{\phi}(s_t, a_t)$. Meanwhile, policy optimization methods directly optimize the policy π_θ by *gradient ascent* with the *policy gradient* $\nabla_\theta J(\pi_\theta)$. Some examples of policy optimization algorithms are *trust region policy optimization* (TRPO) [35] and *proximal policy optimization* (PPO) [36]. Additionally, the *deep deterministic policy gradient* (DDPG) [37] algorithm can be regarded as a mixture of Q-learning and policy optimization. DDPG is an actor-critic algorithm, where π_θ is optimized with gradient ascent by finding $\nabla_\theta \hat{Q}_\phi^{\pi_\theta}(s_t, \pi_\theta(a_t|s_t))$.

In general, DRL algorithms are known to be difficult to apply successfully. They are sensitive to hyperparameters, and training is often unstable [27]. Additionally, training deep networks with DRL often requires more data to be trained sufficiently [16].

A stochastic policy with a continuous action space is usually represented as a multivariate Gaussian distribution with a diagonal covariance matrix. If the action a is k-dimensional, the log-likelihood of is calculated as follows [23]:

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left[\sum_{i=1}^{k}\left(\frac{(a_i-\mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right], \tag{2.4}$$

where π_θ is a multivariate Gaussian distribution with a diagonal covariance matrix.

Policy Optimization
Policy optimization methods use $\hat{V}_\phi^\pi(s_t)$ in some capacity to find suitable gradient updates. This section will follow the notation of Achiam [23]. As previously described in Section 2.1.3, the RL agent seeks to maximize the expected return $J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[R(\tau)]$. Policy optimization methods use the policy gradient $\nabla_\theta J(\pi_\theta)$ to update the parameters by gradient ascent. That is:

$$\theta_{k+1} = \theta_k + \eta \nabla_{\theta_k} J(\pi_{\theta_k}) \tag{2.5}$$

In this equation θ_k refers to the k-th iteration of the parameters θ, and η is the learning rate. It can be shown that:

$$\nabla_{\theta_k} J(\pi_{\theta_k}) = \nabla_{\theta_k} \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)] = \underset{\tau\sim\pi}{\mathbb{E}}\left[\sum_{t=0}^{T} \nabla_{\theta_k}\log\pi_{\theta_k}(a_t, s_t)R(\tau)\right] \tag{2.6}$$

Furthermore, it can also be shown that $\underset{a_t\sim\pi_\theta(\cdot|s_t)}{\mathbb{E}}[\nabla_\theta \log\pi_\theta(a_t|s_t)] = 0$. This means we can write Equation 2.6 in a more general form as:

$$\nabla_{\theta_k} J(\pi_{\theta_k}) = \underset{\tau\sim\pi}{\mathbb{E}}\left[\sum_{t=0}^{T} \nabla_{\theta_k}\log\pi_{\theta_k}(a_t|s_t)\Phi_t\right] \tag{2.7}$$

where Φ_t is any function that is only dependent on the current state s_t[4].

A common choice for Φ_t is an estimate of the advantage function A_t, which is denoted by $\hat{A}_t$. To compute $\hat{A}_t$, the approximation $\hat{V}_{\phi_k}(s_t)$ is used. $\hat{V}_{\phi_k}$ is usually updated with MSE using discounted rewards-to-go $\hat{R}$ as target values. That is, the parameters ϕ is optimized by:

$$\phi_{k+1} = \underset{\phi_k}{\operatorname{argmin}} \underset{s_t,\hat{R}_t\sim\pi_{\theta_k}}{\mathbb{E}}\left[(\hat{V}_{\phi_k}(s_t) - \hat{R}_t)^2\right] \tag{2.8}$$

Policy optimization algorithms calculate an estimate $\hat{g} \approx \nabla_\theta J(\pi)$ of Equation 2.7 by collecting a set of trajectories D. If $\Phi_t = \hat{A}_t$, the estimate of the gradient $\hat{g}$ is computed

[4]This is allowed because Φ_t can be treated as a constant, and $\mathbb{E}[c\mathbf{X}] = c\mathbb{E}[\mathbf{X}]$, where c is a constant and $\mathbf{X}$ is a random variable.

as follows:

$$\hat{g} = \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t, s_t) \hat{A}_t$$

The parameters are then updated with

$$\theta_{k+1} = \theta_k + \eta \hat{g} \tag{2.9}$$

Equation 2.9 is regarded as the standard policy gradient update. This update can be very unstable, as it can lead to huge updates to the policy [36]. The TRPO [35] algorithm counteracts this by constraining a "surrogate" objective function by the *Kullbäck-Leibler divergence* (KL-divergence), between the new policy $\pi_{\theta_{k+1}}$ and old policy π_{θ_k}. A simplified explanation of the KL-divergence is that it measures the distance between two probability distributions. If the difference is to large, it can lead to unstable policy updates. Therefore, TRPO only updates θ if the KL-divergence is below some given threshold δ, which ensures stable policy updates:

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \underset{a_t, s_t \sim \pi}{\mathbb{E}} \left[r_t(\theta) A_t^{\pi_{\theta_{\text{old}}}}(a, s) \right]$$
$$\text{subject to } KL(\pi_\theta | \pi_{\theta_{\text{old}}}) < \delta,$$

$$\text{where } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

The motivation behind the expression $r_t(\theta) A_t^{\pi_{\theta_{\text{old}}}}(a, s)$ is that actions with positive advantages will be chosen with a higher probability (and vice versa for negative advantages) during the next iteration of the policy. TRPO is known to be a stable algorithm, able to learn policies with good performance for several tasks [23]. However, TRPO is complicated and hard to implement, as it requires a complex second-order estimate of the KL-divergence in order to apply the update [36].

Proximal Policy Optimization
Proximal policy optimization (PPO) [36] is an actor-critic policy optimization algorithm that is known to be stable, easy to implement, and allows for parameter sharing between policy and value-function. PPO has been shown to achieve similar performance as TRPO. It is also as data efficient [23, 36]. PPO uses a clipped objective function L_{CLIP}:

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \underset{s_t, a_t \sim \pi_{\theta_k}}{\mathbb{E}} [L_{\text{CLIP}}(s_t, a_t, \theta)], \tag{2.10}$$

where L_{CLIP} is defined as:

$$L_{\text{CLIP}}(s_t, a_t, \theta) = \min(r_t(\theta) A_t^{\pi_{\theta_k}}(a_t, s_t),\ g(\epsilon, A_t^{\pi_{\theta_k}}(a_t, s_t))),$$

and g is defined as

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0 \\ (1-\epsilon)A & \text{if } A < 0 \end{cases}$$

Here, ϵ is a small value called the *clip ratio.* To understand this objective, recall that when advantages are positive, $r_t(\theta)$ must be increased by raising the probability of $\pi_\theta(a_t|s_t)$. Similarly, when advantages are negative, $r_t(\theta)$ must be decreased by lowering the probability of $\pi_\theta(a_t|s_t)$. The L_{CLIP} objective ensures that the ratio $r_t(\theta)$ is kept within the interval $[1-\epsilon, 1+\epsilon]$ when the objective improves. If the objective does not improve, the

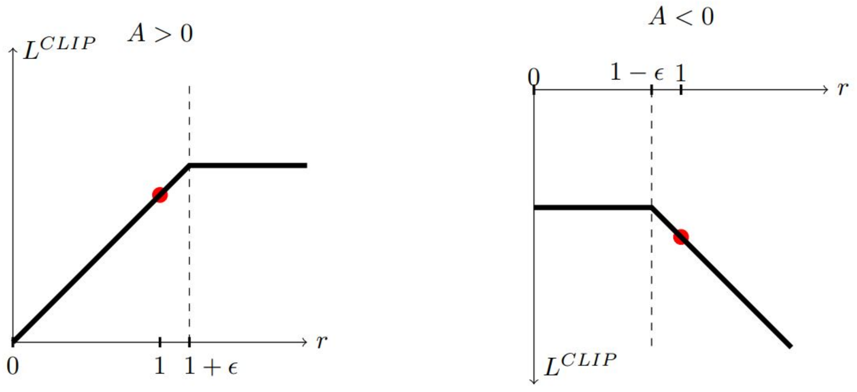


Figure 2.3: L_{CLIP} function for a single timestep t. The horizontal axis is the ratio r_t between new and old polices, while the vertical axis is L_{CLIP}. Left graph displays L_{CLIP} when A_t is positive, right graph displays L_{CLIP} when A_t is negative. Image taken from original PPO paper [36].

min function creates a pessimistic lower bound, which ensures that these values are included in the range of L_{CLIP}. Figure 2.3 visualizes how the function behaves for a single timestep.

PPO is ran for a number of *episodes*. Each episode starts by collecting a set of trajectories D using N parallel actors. We will also refer to trajectories as *rollouts*. Then, rewards-to-go $\hat{R}$ and advantage estimates $\hat{A}$ are computed. $\hat{A}$ is found by utilizing the value-function estimate $\hat{V}_\phi(s_t)$. The parameters of the policy and value function are optimized by finding the gradients of L_{PPO}, which is defined as the following weighted sum:

$$L_{\text{PPO}} = -\sum_{\tau \in D} \sum_{t=0}^{T} L_{\text{CLIP}}(s_t, a_t, \theta_k) + c_1 \sum_{\tau \in D} \sum_{t=0}^{T} (\hat{V}_{\phi_k}(s_t) - \hat{R}_t)^2 + c_2 S \tag{2.11}$$

The first term is an estimate of Equation 2.10, and the second term is an estimate of Equation 2.8. If the actor and critic do not share parameters, the c_1 coefficient should be set to 1. Also notice the negative sign $(-)$ in front of the first term. This sign will result in parameters being updated with *gradient ascent* when applying this objective function in a *gradient descent* optimizer such as Adam [31].

During the course of training, the policy will hopefully learn what causes positive rewards by experimenting with applying different actions. As training progresses, the randomness of the stochastic policy decreases. The pseudocode for PPO is shown in Algorithm 2. This pseudocode is based on the work of Achiam [23] and Schulman et al. [36].

Generalized Advantage Estimation

Previously we mentioned that a common choice for Φ_t in Equation 2.2.7 is the advantage function $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$. The motivation behind this choice is that the true function A_t^π leads to the lowest possible variance. When $\Phi_t = A_t$ the policy gradient estimate will point in the direction of increased probability for actions that result in higher rewards on average when $A_t^\pi > 0$. However, A_t^π is unknown and must be estimated [24].

Following the notation from the work of Schulman et al. [24], we define $\delta_t^{\hat{V}}$ as an estimate of the true and unknown advantage of action a_t:

$$\delta_t^{\hat{V}} = r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)$$

Algorithm 2 PPO with clipped objective function.

```
 1: Initialize $D \leftarrow \emptyset$
 2: Initialize policy parameters $\theta_0$
 3: Initialize policy parameters $\theta_{\text{old}} \leftarrow \theta_0$
 4: Initialize value function parameters $\phi_0$
 5: for $k = 1$ to $n$ do
 6:     Collect a set of trajectories $D$ with $\pi_{\theta_{\text{old}}}$ using $N$ parallel actors
 7:     Compute advantage estimates $\hat{A}$ using the current value function $\hat{V}_{\phi_k}$
 8:     Compute rewards-to-go $\hat{R}$
 9:     Compute gradients $\nabla_{\theta_k} L_{\text{PPO}}$ and $\nabla_{\phi_k} L_{\text{PPO}}$
10:     Update $\theta_{k+1}$ and $\phi_{k+1}$ with a gradient descent algorithm
11:     Set $\theta_{\text{old}} \leftarrow \theta_{k+1}$
12: end for
```

If $\hat{V}^{\pi}$ equals the true value-function V^{π}, then $\delta_t^{V^{\pi}}$ is an unbiased estimator[5] for the true advantage function A_t^{π}. This is because:

$$\mathbb{E}\left[\delta_t^{V^{\pi}}\right] = \mathbb{E}\left[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right] = \mathbb{E}\left[Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)\right] = A^{\pi}(s_t, a_t)$$

In practice, the approximation $\hat{V}^{\pi}$ does not equal V^{π}, which means that $\delta_t^{\hat{V}}$ is a biased estimator. Schulman et al. propose $\hat{A}_t^{(k)}$, which is defined as the following discounted sum:

$$\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^{\hat{V}^{\pi}}$$

Finally, the *generalized advantage estimate* (GAE) for action a_t is defined as the following exponentially weighted sum:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = (1-\lambda)(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + ...) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^{\hat{V}^{\pi}}$$

GAE has two parameters, γ and λ, that contribute to a bias-variance trade-off. γ is the discounting term, which introduces bias to the estimate no matter how accurate $\hat{V}^{\pi}$ is. When $\hat{V}^{\pi}$ is inaccurate, the λ parameter will contribute with some bias. Schulman et al. report that they find empirically that λ should be set to a lower value than γ for good results.

2.3 Computer Vision

This section will describe some selected topics related to the field of computer vision. It will briefly explain transfer learning, and provide general insight into the fields of semantic segmentation and monocular depth estimation. Several network architectures are also presented, including U-Net [38], MobileNet [39], ResNet [40], among others.

2.3.1 Transfer Learning

Consider the task of training a network to detect traffic signs in images. If trained successfully, the parameters of the first layers will be adjusted to detect low-level features, such

[5]An estimator $\hat{\theta}$ of the true and unknown parameter θ is unbiased when $\mathbb{E}[\hat{\theta}] = \theta$.

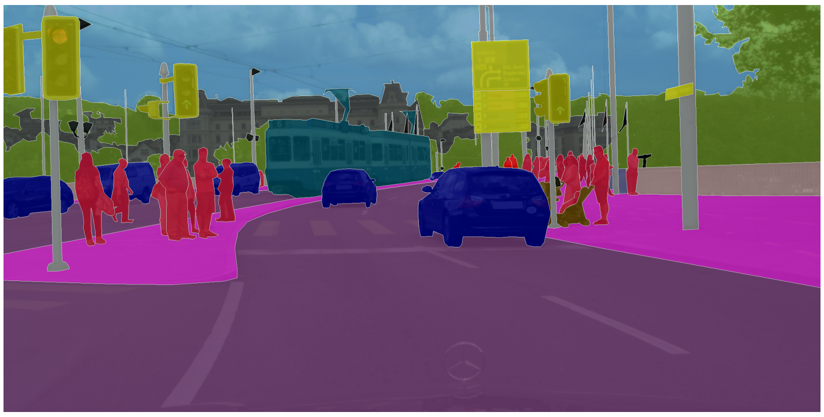


Figure 2.4: Example of a semantically segmented image. This image is from the Cityscapes [43] dataset for semantic segmentation. The colors used in this image are known as the Cityscapes color palette.

as lines or edges. The next layers will be adjusted to detect mid-level features from the low-level features, such as rectangles or circles. For each subsequent layer, the parameters will be adjusted to detect features on a higher-level than the preceding layer. Eventually, the output layer will be adjusted to detect traffic-signs.

Instead of randomly initializing the weighs of the network, it is possible to reuse the parameters of an already trained network. For instance, if the reused weights of the first few layers are already trained to detect low-level features, this can speed up training. Reusing weights is an example of *transfer learning* [27]. Transfer learning is not only restricted to the field of computer vision.

The *backbone* is an important component of several architectures that process images. It is a neural network which extracts features from images. It is common practice to reuse backbones that have been trained on large datasets, e.g. the ImageNet [41] or COCO [42] datasets. In this case, the backbone is said to be *pretrained* on the dataset.

2.3.2 Semantic Segmentation

Semantic segmentation is the task of classifying every pixel in an image to one of n classes. A semantically segmented image can be visualized by assigning each class a specific color, which is showcased in Figure 2.4. As can be seen in the image, each color corresponds to a different class, such as *vehicle, sidewalk, vegetation* etc. All pixels not belonging to any particular class gets classified as *unlabeled*, and is assigned the black color when visualized.

When performing semantic segmentation, the last layer of the network must output as many feature maps as the number of semantic classes. For instance, consider an image I that can be segmented into four semantic classes. The last layer of the network must then output four feature maps, i.e. $[S_1, S_2, S_3, S_4]$. The value positioned at column i and row j in S_k, i.e. $S(i,j)_k$, is the probability of pixel (i,j) belonging to class k. This means that the following two requirements hold:

$$\forall_{i,j,k}\, S(i,j)_k \in [0,1],$$

$$\sum_{k=1} S(i,j)_k = 1.$$

In order for the output to satisfy the latter requirement, a suitable activation function for the final layer would be *Softmax2D*. This function takes an n-dimensional array as input, and applies the Softmax function over the channel dimension. Since semantic segmentation is a classification task, cross-entropy would be a good choice for computing costs. Finally, the classification of an arbitrary pixel (i, j) is decided by

$$argmax_k \in [S(i,j)_1, S(i,j)_2, S(i,j)_3, S(i,j)_4]$$

where k is the index referring to feature map S_k.

Two commonly used datasets for semantic segmentation are *Cityscapes* [43] and *Mapillary* [44]. Both Cityscapes and Mapillary contain images of urban driving environments.

Evaluation Metrics
Multiple evaluation metrics for semantic segmentation are based on the *intersection over union* (IoU) values. These are calculated as follows: for class k, let $\hat{S_k}$ be a prediction from a neural network and S_k be the corresponding ground truth. Both $\hat{S_k}$ and S_k are two-dimensional boolean masks with values indicating the presence of class k. The IoU for class k is measured as the intersection of correctly classified pixels between $\hat{S_k}$ and S_k, divided by the area of union between them. IoU for class k is computed as follows:

$$IoU_k = \frac{\hat{S_k} \cap S_k}{\hat{S_k} \cup S_k} = \frac{TP}{TP + FP + FN},$$

where TP, FP, and FN are the number of true positives, false positives, and false negatives, respectively. More precisely, TP is the number of pixels correctly classified as class k. FP is the number of pixels incorrectly classified as class k. FN is the number of pixels belonging to class k, but classified as something else. After IoU_k for each class k has been found, the *mean intersection over union* (mIoU) is calculated as $\frac{1}{k}\sum_k IoU_k$. The mIoU is a common evaluation metric for semantic segmentation.

When calculating mIoU, every class IoU is of equal importance when calculating the mean. A network can be good at predicting the most frequently occurring classes, but bad at the rarely occurring classes. The network will be heavily punished for this on the mIoU score. There exists another variant of this metric, the *frequency weighted IoU*, that mitigates this effect. Frequency weighted IoU will scale all class IoUs according to how prevalent the class is in the image. Let P be the total number of pixels in an image. Let p_k be the number of pixels of class k in the ground truth image. For an image with n classes, frequency weighted IoU is then calculated as follows:

$$Frequency\ weighted\ IoU = \frac{1}{n}\sum_{k=1}^{n} \frac{p_k}{P} IoU_k$$

2.3.3 Monocular Depth Estimation

Monocular depth estimation is the task of estimating distances to objects in an RGB image captured by a monocular camera. Neural networks performing depth estimation produce two-dimensional outputs known as *depth maps*. A depth map contains distance estimates to objects for every single pixel in the image.

There is a wide variety of approaches for training depth estimation networks [45]. Some networks are trained in a self-supervised fashion using image pairs captured by a stereo camera. On the other hand, supervised methods generally rely on *absolute depth* images. These are single image depth estimates. Absolute depth is easily provided in a simulated environment.

Loss functions
There are several loss functions used for depth estimation. Some loss functions are designed for stereo images. Other loss functions are based on absolute depth targets.

Alhashim and Wonka [46] introduced a novel loss function for absolute depth. Let $\hat{y}$ be the depth prediction from a neural network, and y be the corresponding ground truth. Then, the loss function is defined as sum of the following three terms:

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

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

$$L_{SSIM}(y, \hat{y}) = \frac{1 - SSIM(y, \hat{y})}{2}$$

L_{depth} is the pixel point-wise difference between $\hat{y}$ and y. L_{grad} is the L_1 difference between the *image gradients*[6] of $\hat{y}$ and y. L_{SSIM} calculates the *structural similiary* (SSIM) between $\hat{y}$ and y. SSIM is a metric used to compute overall similarity of two images. SSIM has the ability to catch similarity in groups of pixels that are spatially close, not just in a pixel-by-pixel fashion. The final loss function L is then defined as a sum of the three preceding terms, where L_{depth} is weighted by a value λ which is set to 0.1:

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

Evaluation Metrics
Accuracy within threshold[7] is an evaluation metric that is commonly used in research articles for monocular depth estimation [47, 45, 46]. This metric measures the percentage of pixels with a depth estimate below some specific threshold σ. That is, it calculates the percentage of pixels that satisfy the following requirement:

$$\delta_p = max(\frac{y_p}{\hat{y_p}}, \frac{\hat{y_p}}{y_p}) < \sigma \tag{2.13}$$

Here, $\hat{y_p}$ is a depth estimate and y_p is the ground truth for a particular pixel p. The accuracy within threshold metric is usually computed for several values of σ. Common values for σ are 1.25, 1.25^2 and 1.25^3.

RMSE can also be used for comparison of depth images. The formula can be found in Section 2.2.3.

2.3.4 Architectures

This section will describe a selection of network architectures that is used in the field of computer vision.

[6]Image gradients are a measure of change of intensity in an image, and are mathematically defined as 2D vectors.

[7]This metric is also known simply as *threshold* [45], or *threshold accuracy* [46].

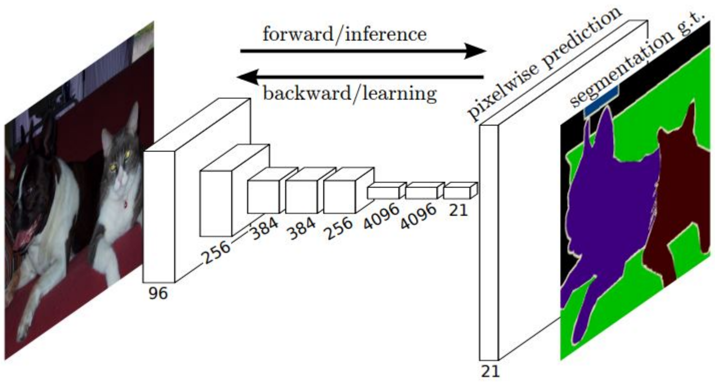


Figure 2.5: Visualization of a FCN. Each number denotes the number of feature maps produced by that layer. Notice how the network does not have a fully connected output layer. Instead, it uses a convolutional output layer with a 1x1 kernel mapping to the number of semantic classes. Image taken from the original FCN paper [48].

2.3.4.1 Fully Convolutional Networks

In 2014 Long et al. [48] introduced *fully convolutional networks* (FCN), which are designed for semantic segmentation. As the name entails, all layers in the network are convolutional layers. This also includes the output layer, which uses a 1x1 kernel to map to the desired number of semantic classes. This was in contrast to the standard design at the time, where the output layer usually was implemented as a fully connected layer. Using a fully connected output layer will only work for a fixed input size, as these layers expect a certain number of neurons in the preceding layer. Meanwhile, a convolutional layer can process images of any size. This means that FCNs can perform semantic segmentation for any arbitrary input size.

2.3.4.2 Residual Networks

Kaiming He et al. [40] introduced a network architecture known as a *resiudal network* (ResNet), which have since become ubiquitous within the field of computer vision. These networks consists of several *residual blocks* (RBs). Each RB perform *residual learning* by using a shortcut connection between layers. An example of a RB is shown in Figure 2.6. In the figure, $\mathbf{x}$ is an input tensor which is propagated through two convolutional layers. Batch normalization and ReLU is applied after every convolutional operation. The RBs approximate a residual function defined as $F(x) = H(x) - x$, where x is the input and $H(x)$ is the output from the convolutional layers.

The motivation behind this design is that the task of approximating the function $H(\mathbf{x}) - \mathbf{x}$ is easier than approximating $H(\mathbf{x})$. The reason for this is that the original target function $H(\mathbf{x})$ is often close to the the identity function, i.e. $H(\mathbf{x}) \approx \mathbf{x}$.

There are several variants of ResNet, each consisting of a different amount of RBs. For instance, ResNet-18 consists of 18 RBs, and ResNet-34 consists of 34 RBs, etc. The ResNet architecture allowed for efficient training of very deep networks, which marked an important milestone in deep learning. The networks reached state of the art results on multiple computer vision tasks.

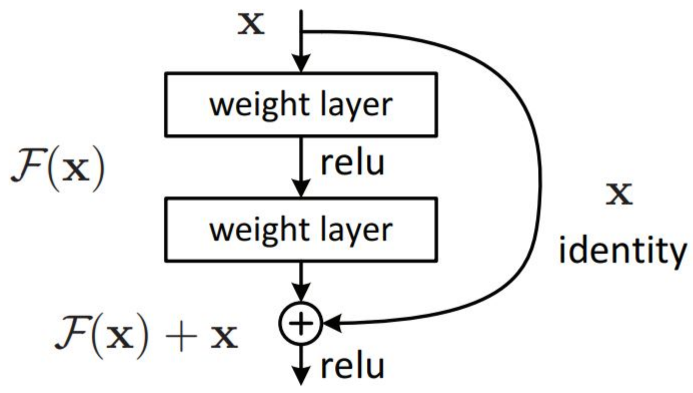


Figure 2.6: Example of a skip connection utilized in ResNets, as presented in the original paper [40].

2.3.4.3 U-Net

Ronneberger et al. [38] introduced a neural network architecture called U-Net, designed for performing biomedical segmentation tasks. U-Net is a fully convolutional network, with an encoder-decoder structure. The encoder is called the *contracting path*, and consists of four repeated blocks. Each block is a convolutional layer followed by ReLU and max pooling. For each subsequent block in the encoder, the resolution is reduced by a factor of two, while the number of feature maps is doubled.

The decoder also consists of four repeated blocks, and is known as the *expanding path.* For every block in the decoder, feature maps are sampled to a higher resolution by bilinear interpolation. The feature maps are then concatenated along the channel dimension with feature maps from the corresponding level in the encoder. This is followed by two convolutional layers with ReLU activation. Every block in the decoder increases the resolution, but reduces the number of feature maps. The final layer uses a 1x1 kernel that maps to the desired number of classes. A figure of the network is showcased in Figure 2.7. The name is derived from the shape of the architecture that looks like a "U" when visualized.

2.3.4.4 MobileNet

Howard et al. [39] introduced a family of efficient convolutional networks, known as MobileNet. The name is derived from the intended use case of the network, which is for mobile and embedded vision applications. The paper introduced several variants of the architecture. Each variant is scaled uniformly with a different width multiplier (i.e. changing the number of feature maps for each layer), and a different resolution multiplier. The standard MobileNet-architecture is computationally inexpensive and has few parameters. This is because it uses depthwise separable convolutional layers. However, the first and penultimate layers are standard convolutional layers, while the final layer is a fully connected layer.

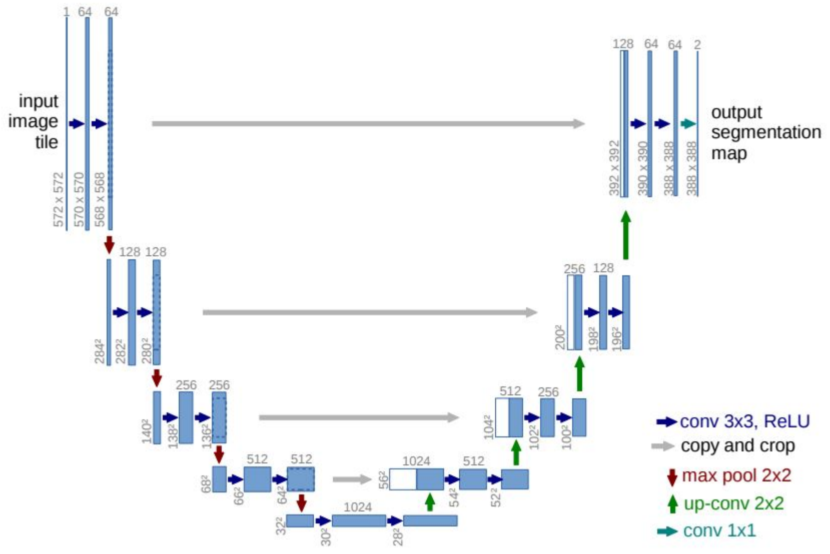


Figure 2.7: The architecture of U-Net. Notice the contracting path on the left side. The gray lines represents feature maps that are copied and cropped from the encoder to the corresponding level in the decoder. Image taken from the original U-Net paper [38].

2.3.4.5 DeepLab

Chen et al. [33] introduced *DeepLab*, which is an architecture designed for semantic segmentation. The authors of DeepLab addressed several issues with semantic segmentation. One issue is that the reduction of resolution caused by striding and pooling in deep CNNs severely impacts the performance of the networks. This is because semantic segmentation is regarded as a *dense* prediction task. This means that it requires a high level of detail for good performance. Another difficulty is classifying objects belonging to the same class, but at different scales. For instance, an image of an urban traffic environment might contain several vehicles. Depending on the relative distance to the camera, the size of the vehicles will appear different in the image. Different scales can make it challenging for the network to realize that these objects all belong to the same class.

DeepLab uses a ResNet-like backbone. To solve the problem of feature maps losing their resolution, they switch some of the downsampling operations with atrous convolution in the backbone. As previously described in Section 2.2.6, atrous convolution allows for denser feature extraction and a larger field of view, without adding extra parameters to the network.

To solve the problem of classifying objects at different scales that belong to the same class, they introduced *atrous spatial pyramid pooling* (ASPP). An ASPP operation uses four atrous convolutions with different atrous rates in parallel on the same feature map. Feature maps from each of the four atrous convolutions are then concatenated together along the channel dimension. A visualization of this operation can be seen in Figure 2.8. The authors of DeepLab would later go on to design two other versions of the DeepLab architecture. The newest version is known as *DeepLabv3* [34].

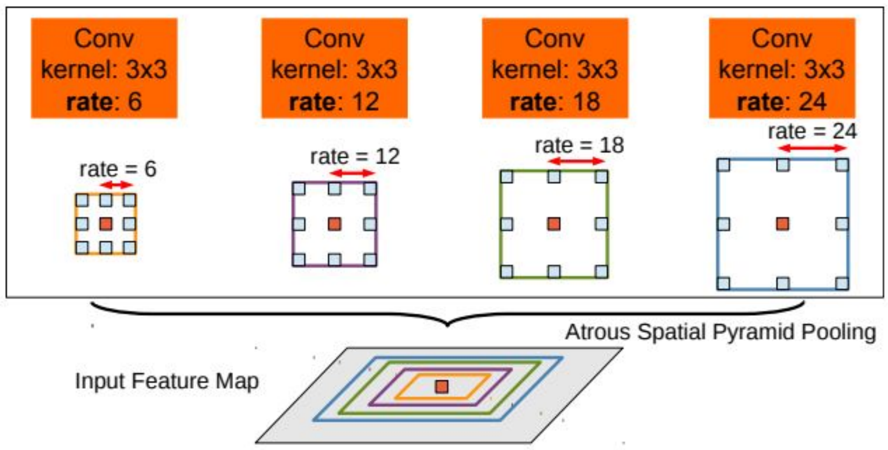


Figure 2.8: Example of an ASPP operation. This ASPP uses four atrous convolutions with rates of 6, 12, 18, and 24. All atrous operations are computed in parallel on the same feature map to capture information at multiple scales. The higher the atrous rate, the higher the scale at which features are extracted from the feature map. Image taken from original DeepLab paper [33].

2.3.4.6 MiDaS

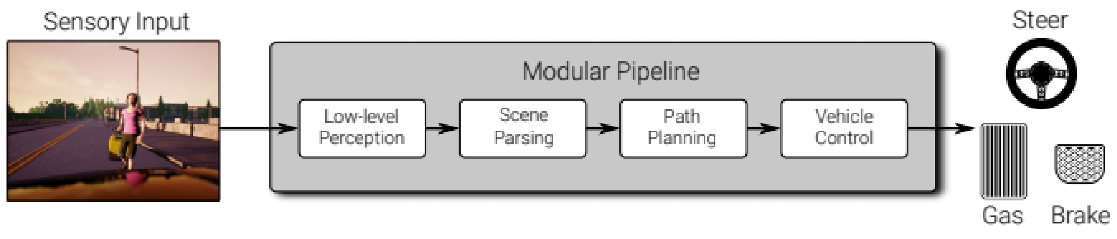
Ranftl et al. [47] introduced a ResNet-based network called *MiDaS*. This network was trained on a diverse set of monocular depth estimation datasets covering several months of GPU time. These datasets were captured by stereo cameras, light sensors and laser scanners. Additionally, they also extracted a dataset from 3D Hollywood movies. The motivation behind training on this mixture of datasets is to make the network robust to variety of different scenarios. They proposed a novel loss function that is invariant to the conflicting representations of depth between the datasets. They evaluated the performance of MiDaS with an approach called *zero-shot cross dataset-transfer*. This method evaluates the network on an entirely new dataset that was not present during training. They report SOTA performance on several datasets.

2.4 Approaches to autonomous driving

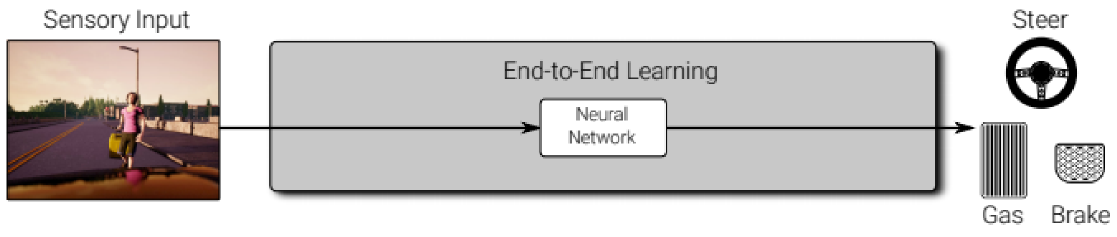
The designs of autonomous driving systems can be divided into two classes: modular approaches and end-to-end approaches. In modular approaches the task of autonomous driving is broken down into sub-tasks, which are then solved independently by dedicated modules. In end-to-end approaches the whole task is solved by a single monolithic neural network. This neural network will take raw sensor data as input and output vehicle controls. Figure 2.9 shows the conceptual difference between the two approaches. Descriptive overviews of both these approaches are given by Yurtsever et al. [3] and Tampuu et al. [8].

2.4.1 Modular approach

According to the recently published work of Janai et al. [28], the modular approach is the most commonly used autonomous driving approach in the industry. In a modular



(a)



(b)

Figure 2.9: A visual comparison of autonomous driving approaches as presented by Janai et al. [28]. (a) Modular approach. (b) End-to-end approach.

approach the autonomous driving system is organized as a pipeline of modules. Each of these modules have the responsibility to solve some part of the complex autonomous driving problem. This has the advantage of allowing each module to be hand-crafted and to exploit high-level knowledge in its domain. There is no standardization for which modules a pipeline should include, and the complexity of each module can vary greatly. As in the example from Janai et al. [28], which can be seen in Figure 2.9a, examples of such modules can be low-level perception, scene parsing, path planning and vehicle control. In this example, the low-level perception and scene parsing modules might be implemented by neural networks, while the path planning and vehicle control can be implemented as classical search algorithms and control models.

This approach has a major advantage in terms of explainability. It is possible to trace down which module is responsible for a certain decision in the case of an error. The modules can also be rigorously tested individually before being incorporated into the larger system.

There are also some disadvantages to the modular approach. As the pipeline and modules are hand-crafted and specialized, they may fail to generalize to unusual conditions and unforeseen situations. While it is possible to make individual rules for all edge cases, doing so requires a high amount of information, and is often hard in practice.

2.4.2 End-to-end approach

While the modular approaches will divide autonomous driving into tasks like perception, planning and control, an end-to-end approach will take on the combined task and solve it with a single neural network. The end-to-end approach has historically seen little use in comparison to its counterpart [28]. But with the recent breakthroughs and advances in the field of deep learning, this approach has recently been subject to much research. Neural networks have shown to have superhuman performance in many domains, and there is a belief that this eventually might be the case for autonomous driving as well.

Neural networks are capable of leveraging large amount of data. In the context of driving such large amounts of data is easily available, as cameras and other sensors can be mounted on a vehicle. By recording the actions of the drivers, this data can be used to perform imitation learning, which is described in Section 2.1.2. However, these agents can only learn from situations that the experts have found themselves in, not from situations that arises from the agents actions. This is problematic, because the agents will not always be able to do expert-like decisions, and as a result will sometimes find itself in unforeseen situations.

The end-to-end systems can also be trained by RL, this process is described in Section 2.1.3 and Section 2.2.7. With this approach, the agents will be able to learn from the unusual situations that it might put itself in. The drawback is that the training process is substantially slower and a stable training algorithm can also be hard to implement. RL is easier in a simulated setting, because there are no consequences of exploration. Simulation time can also be sped up compared to real time, which allows for shorter training times. It is also possible to use RL as a fine-tuning stage after IL. This is done by reusing the weights trained in the IL stage in a RL algorithm. This can significantly reduce the long training times that is associated with RL [8].

RL reward functions for AVs are often designed to make the agent behave efficiently and safely [4]. This can be done by giving the agent negative rewards for collisions, driving on the sidewalk, running red lights, and not being near the center of the lane, and giving positive rewards for efficient driving. Efficient driving can be rewarded by giving the agent rewards that are proportional to the current speed of the vehicle [8]. It can be difficult to balance the reward function in terms of how big the magnitude of the rewards should be in different scenarios [4].

End-to-end learning is often defined as directly mapping raw sensor input e.g. RGB images into vehicle controls. But in some works, the architectures are split into modules in an end-to-end setting, for example *perception* and *driving* modules [49, 50, 11]. There might be some confusion of where this crosses the line of the modular approaches. We find that different works use different definitions. In our thesis we regard systems that are exclusively composed of neural networks as end-to-end.

Input Modalities
The type of input modality used in an end-to-end network can vary. RGB images are naturally often used as input. Additionally, the network may also get information about state of the vehicle, such as the current speed. The network may also receive a navigational command that indicates the direction to drive in an intersection [51].

Intermediate representations from computer vision tasks such as semantic segmentation and depth images can also be used. If training is conducted in an simulator such as CARLA or Grand Theft Auto V (GTAV), these representations can be retrieved with perfect quality [10, 12]. Another way of acquiring these representations is to use models which are specifically created to perform these tasks. After the representations have been acquired, they must be combined in some way. This can be accomplished by simply concatenating the computer vision representations together with the corresponding RGB image. The resulting stack of images can then be used as input to a neural network that outputs the action that control the vehicle. This approach, where multiple types of inputs are fused together before propagating the data through the neural network, is known as *early fusion* [8].

Output Modalities
Most end-to-end networks output steering angle and speed [8]. It is also possible to

output a trajectory of waypoints [4] which the vehicle should follow. These waypoints can be transformed into low-level control commands by a PID controller, which is explained in Section 2.5.3. Tampuu et al. [8] explain several advantages by using waypoints as an output modality. Outputting waypoints forces the network to plan ahead. They are also easy to interpret and analyze. Figure 2.10 shows an example of what the waypoints might look like.

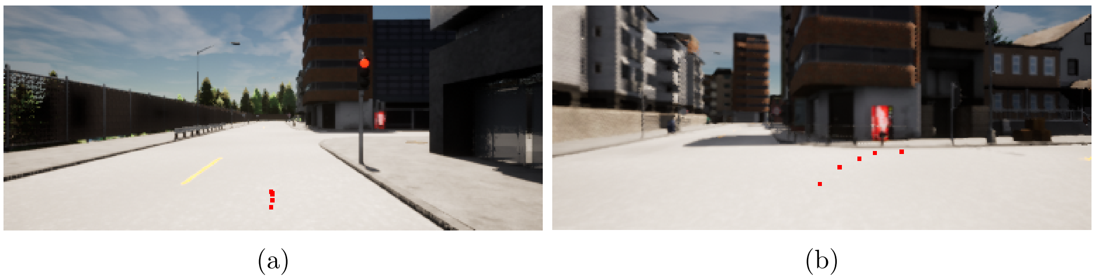


Figure 2.10: Visualization of how waypoints can be used as an output modality. The waypoints are shown as red dots. (a) Contracted waypoints, indicating full stop. (b) Arcing waypoints, indicating the path through a turn.

2.5 Technology

This section introduces some of the technology used in this thesis. It presents CARLA [10], which is a simulator used for autonomous driving research. Machine learning frameworks and the functioning of a PID controller are also discussed.

2.5.1 CARLA

Dosovitskiy et al. [10] introduced CARLA (Car Learning to Act), which is a freely available open-source simulator for autonomous driving research. This simulator have become a prevalent tool used by researchers developing end-to-end AVs since its introduction in 2016. This is clearly reflected by the amount of research papers that have utilized CARLA[8].

In the early days of the end-to-end approach for AVs, researchers such as Pomerlau [54] and Lecun et al. [55] had to use real-life vehicles in order to evaluate the performance of their networks. Now, with simulators such as CARLA available, developing end-to-end AVs has become much more accessible. Simulators remove many difficulties related to using real-life vehicles. In a simulator, training and testing AVs in a controlled and customizable environment can safely be accomplished.

CARLA is built as a layer on top of Unreal Engine 4, and uses a client-server architecture. The client sends commands to the server through an API. The commands give a high level of control over the simulated environment. It is possible to spawn and control vehicles, pedestrians, and sensors. Environmental properties like the weather and illumination are also fully controllable. There are several sensors available in CARLA, including RGB cameras, LiDAR, and semantic segmentation sensors. The job of the server is to run the simulation, render graphics, and process commands from the client. The server responds to the client by sending back the simulation state.

[8]Some examples include [51, 15, 11, 14, 17, 16, 52, 53], among others.

CARLA has numerous assets available for use. This includes vehicles, buildings, pedestrians, vegetation, urban layouts, highways, and other miscellaneous assets. The simulator has several pre-built *towns* available for use. Two commonly used towns for research are *Town01* and *Town02*, which consists of 2.9 and 1.4 kilometers of drivable road, respectively. The towns use different assets but have the same visual style. This naturally sets up *Town02* as a test environment for models trained in *Town01*. Figure 2.11 displays two images showing the visual style of *Town01* and *Town02*.

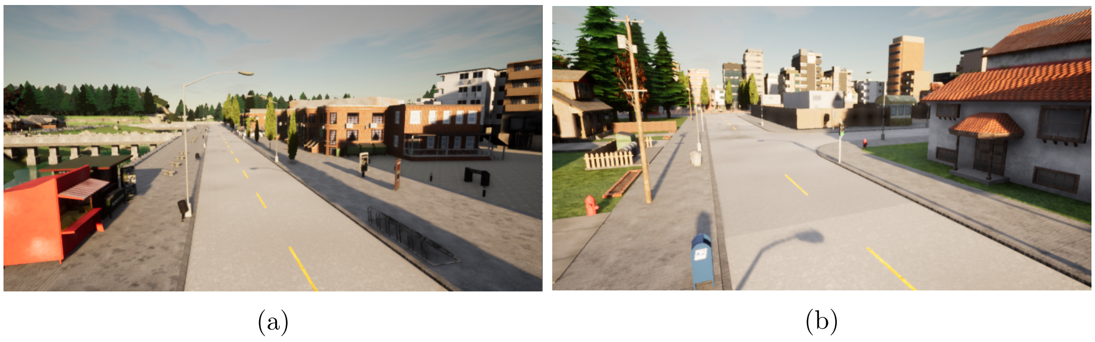


Figure 2.11: Images showing the similarity in style between *Town01* and *Town02* in CARLA. (a) *Town01*. (b) *Town02*.

The CARLA Benchmark

Dosovitskiy et al. [10] also proposed a benchmark for evaluating AV-systems in CARLA. This benchmark is known as the *original CARLA benchmark* (OCB). In this benchmark, *Town01* is used for training. *Town01* and *Town02* is used for evaluation. Four weather conditions are seen during training, and two additional weathers are used during evaluation. The benchmark consists of several goal-directed episodes, where the agent has to navigate through a path. Some episodes are set in a dynamic environment, where pedestrians and other vehicles are spawned. An episode is considered successful if the agent reaches the end-position within a given time-limit. Episodes are not terminated if collisions or traffic-light infractions occur.

The NoCrash Benchmark

Three years after the release of CARLA, Codevilla et al. [11] proposed a new benchmark called *NoCrash*. The NoCrash benchmark was designed for evaluating AVs in complex and dynamic urban driving scenarios. NoCrash is significantly more demanding than OCB, as it requires the agent to adapt to dynamic traffic situations involving a higher number of pedestrians and vehicles. The evaluation procedure of NoCrash consists of three increasingly difficult tasks. These tasks are:

- *Empty*: No pedestrians or actors are spawned.
- *Regular*: A moderate amount of pedestrians and vehicles are spawned.
- *Dense*: A high number of pedestrians and vehicles are spawned.

Town01 is used for training, while both *Town01* and *Town02* are used for evaluation. Four different weather conditions are seen during training, which are the same weathers as for OCB. During evaluation, two additional weather conditions are used. For every combination of task, town and weather, the agent has to complete 25 goal-directed episodes, which are defined by a start and end position. Similarly to OCB, the agent is given high-level navigational commands from a navigational planner that guides it through the path. These navigational commands are called HLCs. The episode is terminated when a collision occurs or if the time limit is reached.

We refer to the different weather conditions by their name in the CARLA API. The four training weathers are *ClearNoon, WetNoon, HardRainNoon*, and *ClearSunset*. The test weathers are *WetSunset* and *SoftRainSunset*. All six weathers are visualized in Figure 2.12.

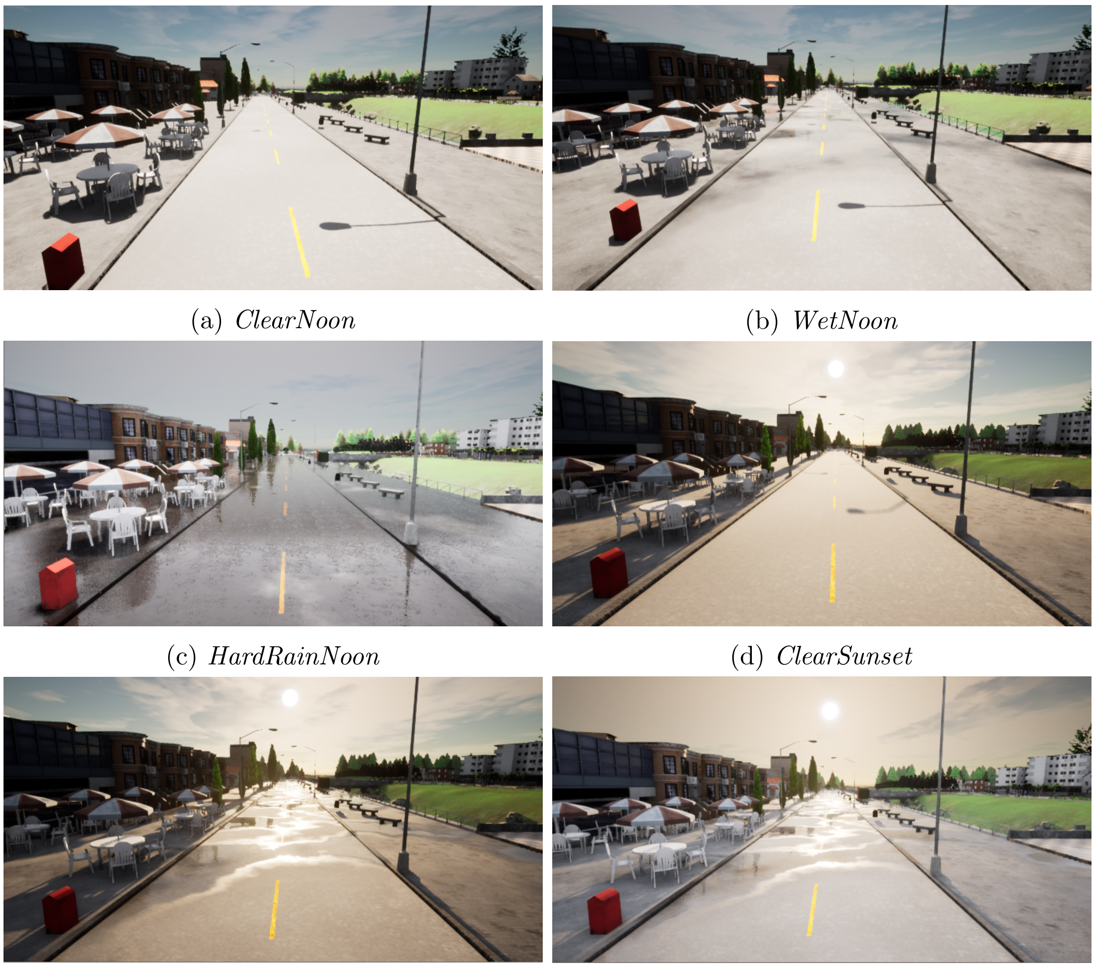

(a) *ClearNoon* (b) *WetNoon*
(c) *HardRainNoon* (d) *ClearSunset*
(e) *WetSunset* (f) *SoftRainSunset*

Figure 2.12: The weathers used in the NoCrash benchmark. (a) through (d) are training weathers. (e) and (f) are test weathers.

2.5.2 Machine Learning Frameworks

The two dominant machine learning frameworks used in the industry and by researchers are PyTorch [56] and TensorFlow [57]. They are used for designing, training and validating deep neural networks. Both frameworks have become integral parts of the ecosystem of open-source Python libraries used by data scientists. PyTorch is developed by the AI Research Lab at Facebook, while TensorFlow is developed by Google Brain. Both libraries are primarily used with Python. TensorFlow comes bundled with its own implementation of a high-level API known as Keras [58].

PyTorch and TensorFlow are easy to use and have well-written documentation. They support eager execution, which results in easier debugging. Additionally, both frameworks have support for accelerating training with the use of graphical processing units (GPUs). On Nvidia GPUs, this is done with the *Compute Unified Device Architecure library* (CUDA), and with the *CUDA Deep Neural Network library* (cuDNN). CUDA enables developers to perform parallel computations on GPUs, while cuDNN includes several optimized neural network operations for use on GPUs.

PyTorch includes a computer vision library called *Torchvision.* This library includes datasets, image transformations, and implementations of network architectures. PyTorch has a large ecosystem of open-source libraries. One such library is the *Segmentation-Models-PyTorch* [59], which include several network architectures for semantic segmentation and other computer vision tasks.

2.5.3 PID Controller

A PID controller is a control loop feedback mechanism which is used to control and regulate variables in various control systems. It can be used to keep a control system variable constant over time. PID stands for Proportional, Integral, and Derivative, which are the names of the three terms that the PID controller uses to calculate a correction in the control loop.

The full mathematical motivation behind a PID controller is beyond the scope of this work. In principle, the system has a desired target value called the setpoint (SP), and the current system value called the process variable (PV). The latter is measured by sensors in the environment. The task of the controller is to keep PV as close to SP as possible at every time step. For each time step, the controller calculates the error $e(t)$ between SP and PV. This error is the basis for the proportional, integral and derivative terms.

The proportional term is equal to the error $P(t) = e(t)$. This term will have its value directly proportional to how large the error is in that time step. The integral term is $I(t) = \int_0^t e(\tau)d\tau$. This term will have a larger value if the error is present over time, and it is also proportional to the magnitude of the errors. The derivative term is $D(t) = \frac{de(t)}{dt}$. This term is proportional to the slope of the error over time, $e(t)$.

These three terms are then multiplied by some coefficients K_p, K_i, K_d and summed together to make up the control output, $u(t) = K_pP(t) + K_iI(t) + K_dD(t)$. The extended mathematical formula for control output is then:

$$u(t) = K_pe(t) + K_i\int_0^t e(\tau)d\tau + K_d\frac{de(t)}{dt}$$

The control output directly controls system action, i.e., how much throttle to apply or how much a valve should be opened.

The coefficients K_p, K_i and K_d that scale the error terms need to be set suitably to configure the PID controller to work as intended. Finding a set of coefficients that give desired control is a task known as tuning the controller. PID controller tuning is a difficult problem [60], and many sophisticated strategies have been developed to do this efficiently. Manual tuning may also be an option, depending on the task requirements.

An example that illustrates the functioning of a PID controller is the cruise control system of a car. Here the driver will reach his desired speed before clicking a button to enable cruise control, which sets the target speed SP. The car has sensors that measure the current speed PV. If the car goes up a slight hill, the measured speed PV will go down. Based on the calculated error $e(t)$ between current speed and target speed, the PID controller will calculate a higher $u(t)$. The system will then apply a little more throttle. If the throttle increase was not enough, the integral error term will increase for the next time step. The controller will then apply more throttle. This process continues with each time step until the car reaches its target speed, which it will if the controller is tuned appropriately.

2.6 Related Work

This section details the history of relevant research papers that have made essential contributions to the topics of this thesis.

Many of the papers build and follow up on each other naturally, and we try to describe the main aspects and the novel contributions of each paper.

2.6.1 ALVINN: Autonomous Land Vehicle In a Neural Network (1989)

The work of Pomerlau [54] is the first example of a neural network trained for controlling a vehicle. Pomerlau used a network called ALVINN, which is a fully connected feed-forward network, consisting of three layers. The final layer consists of 46 neurons, where all but one neuron predict the steering angle. The final neuron, called the *road intensity feedback unit*, is recurrently connected to the first layer. A figure of ALVINN can be seen in Figure 2.13. The network was trained for 40 epochs on a dataset consisting of 1200 examples. Each example consisted of an image from a camera mounted at the front of the vehicle, depth information from a laser rangefinder, along with the desired steering angle of the vehicle.

After training, Pomerlau state that the vehicle was able to output desirable steering angles for approximately 90% of all test examples. These test examples were generated by using a simulated road generator. ALVINN was also deployed and evaluated on a real life vehicle. The network was able to drive sufficiently well along a 400 meter long road with a speed of 0.5 meters per second.

2.6.2 Off-Road Obstacle Avoidance through End-to-End Learning - DAVE (2005)

Sixteen years after ALVINN [54], LeCun et al. [55] made huge advances in the field of end-to-end learning for AVs. They used a six-layer CNN that was trained on dataset with 80 thousand training examples. This is in sharp contrast to ALVINN, which used a three-layer fully connected network that was trained on a much smaller dataset. This network controlled a small vehicle nicknamed DAVE (DARPA Autonomous Vehicle). This network was specifically trained for off-road obstacle avoidance. Each example consisted of an image captured by a camera mounted at the front of the car, along with the desired steering wheel angle. To make the network robust to various environmental conditions, the training data was collected manually in a variety of different lightning and weather conditions. LeCun et al. state that the vehicle was able to output suitable steering wheel angles.

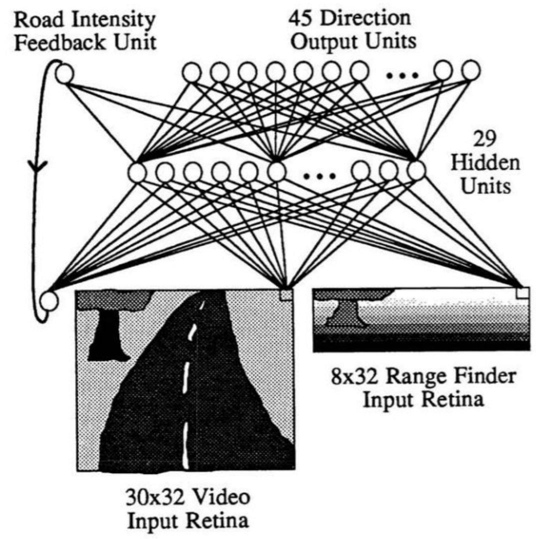

Road Intensity Feedback Unit
45 Direction Output Units
29 Hidden Units
8x32 Range Finder Input Retina
30x32 Video Input Retina

Figure 2.13: The architecture of ALVINN, as presented in the original paper [54]. It consists of three fully connected layers, with a single recurrently connected neuron from the output layer to the input layer.

2.6.3 End-to-End Learning for Self-Driving Cars - DAVE-2 (2016)

Building on the DAVE architecture [55], Bojarski et al. [61] proposed DAVE-2. This system uses a neural network which starts with an image normalization layer, then five convolutional layers, and then a three-layer feedforward network which outputs predictions of the steering wheel angle. Figure 2.14 shows the architecture of the network. The network was trained on 72 hours of video, which was collected by several data-collecting cars. Each car had three cameras mounted at the front of the vehicle, with each camera pointing in different directions. They collected data in various weather and lighting conditions. They also wanted the vehicle to learn how to navigate if the car drifted off the road. In order to learn this, the training data included examples where the vehicle had drifted from the center of the road, and subsequently corrected itself. Data augmentation was also applied by randomly shifting and rotating the images.

The network was evaluated in simulated and real environments, where it performed lane and road following. The network was able to detect important patterns in the images, such as the outline of the road, and disregard irrelevant information in the image as noise. Bojarski et al. conclude by stating that they have empirically demonstrated that it is possible for neural networks to learn the tasks of lane and road following.

2.6.4 End-to-End Driving via Conditional Imitation Learning (2017)

Codevilla et al. [51] made huge contributions in the field of end-to-end learning for AVs with the introduction of a novel IL procedure called *conditional imitation learning* (CIL). Previous AV approaches [54, 55, 61] had the goal of learning to follow roads and lanes. In addition to this, CIL tries to disambiguate the correct course of action in situations where there are multiple actions to choose from. For example, consider a vehicle entering an intersection. Deciding whether to go left, right or straight forward is not something that can be extracted from the environment, but relies on the internal state, i.e., intended destination of the agent.

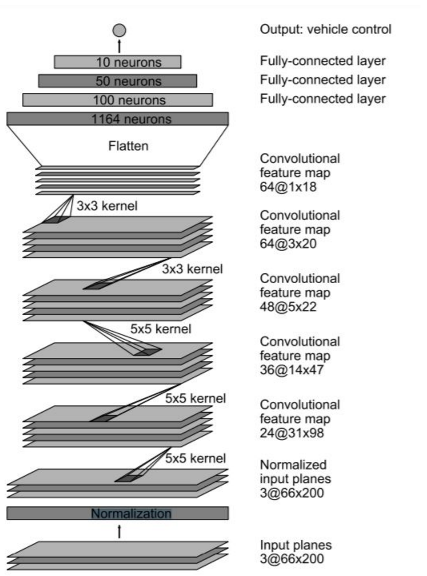


Figure 2.14: The architecture of DAVE-2. It starts with an image normalization layer, then five convolutional layers, and ends with a three-layer feedforward network.

In order to accomplish this, each training instance includes an additional piece of information called a *high level command* (HLC). A HLC indicate the navigational direction, and is represented as an one-hot encoded vector. The training set is then represented as $D = \{\mathbf{o_t}, \mathbf{c_t}, \mathbf{a_t}\}_{i=1}^{N}$, where $\mathbf{o_t}$ is the observation of the environment, $\mathbf{c_t}$ is the HLC, and $\mathbf{a_t}$ is the desired action.

Codevilla et al. investigate two different network architectures which are visualized in Figure 2.15. Both networks include an image module and a measurements module, implemented as a convolutional and feedforward network, respectively. The image module processes images, while the measurements module processes longitudinal and lateral information. The difference between the two types of networks is how they process the HLC.

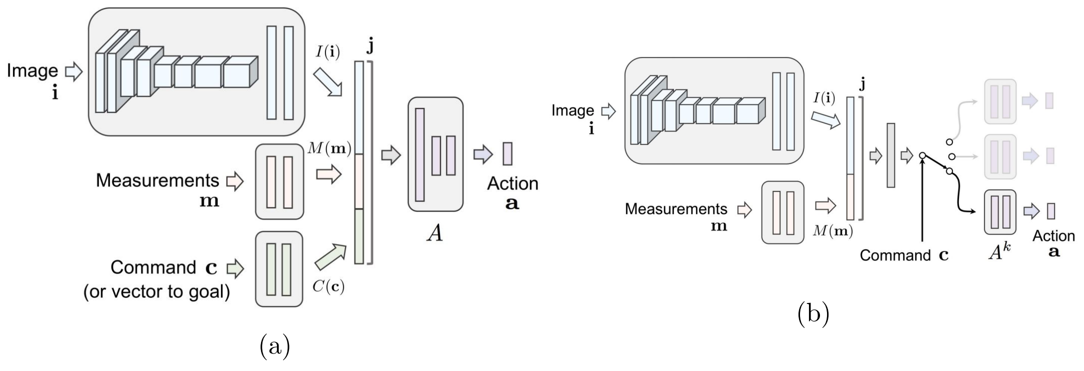


Figure 2.15: The two CIL networks as presented in the original CIL paper [51]. (a): The *command input* network. (b): The *branched* network.

Figure 2.15a shows the *command input* network. This network processes HLCs in a separate feed forward network called the command module. The output of the command module is then concatenated with the outputs of the image and measurements modules, before being propagated through a three-layer network that outputs the action $\mathbf{a_t}$.

Figure 2.15b shows the *branched* network. This network uses the HLC as a switch between different branches, where each branch is a neural network. The motivation behind the branched network is that each branch has to learn its own sub-policy. That is, one branch specializes in turning left, another in turning right and so forth.

They trained and evaluated both networks in simulation using CARLA, and in real life using a small toy truck. In both environments, an expert manually collected two separate datasets containing two hours of driving video. After training, the networks were evaluated by having the vehicle navigate through a set of predefined paths. The performance was measured by the average time between infractions, and travel time from start to destination.

Additionally, they applied data augmentation during training to improve generalization. This augmentation consists of a random subset of transformations, where the magnitude of each transformation is randomly sampled from a normal distribution. These transformations include Gaussian blur, Gaussian noise, randomly dropping pixels in the image, along with changing different visual qualities, such as hue, contrast and brightness.

The results shows that the branched network perform significantly better than the command input variant. Codevilla et al. found that that data augmentation was crucial for generalization. The networks trained with data augmentation performed dramatically better than networks trained without data augmentation.

2.6.5 CIRL: Controllable Imitative Reinforcement Learning for Vision-based Self-driving (2018)

Liang et al. [15] wanted to solve some of the problems that had been encountered in past end-to-end AV research. They describe several limitations with these approaches [54, 55, 61, 51]. This includes the large amount of data required to sufficiently train the networks, and their weak generalization capabilities.

Motivated by these problems, they introduce a two-staged procedure called *controllable imitative reinforcement learning* (CIRL). First, the network is trained with an IL stage. The second stage involves a RL phase, where the trained weights from the IL stage is loaded into the actor in the DDPG algorithm. The motivation behind this two-stage process, is that the actor can reuse the pretrained weights from the IL stage. With the pretrained weights, the network is already capable of outputting reasonable actions. This removes much of the time-consuming exploration of the state space that usually happens at the beginning of RL training. Due to the exploration during the RL stage, the network can improve beyond the policy learned from IL.

The actor and critic networks use HLCs to switch between different branches, similarly to the branched network from the work of Codevilla et al. [51]. A visualization of the overall process is depicted in Figure 2.16. The actor and critic networks are almost identical. Both include a convolutional backbone that process images, and a three-layer feedforward network which processes speed information. The difference is that the critic includes an additional three-layer feed-forward network that processes actions chosen by the actor.

They evaluated the actor network on the original CARLA benchmark, and achieved state of the art results. With pretrained weights, the RL stage trained for 300000 simulations. This is in sharp contrast with the RL approach from Dosovitskiy [10], which required over 10 million simulation steps.

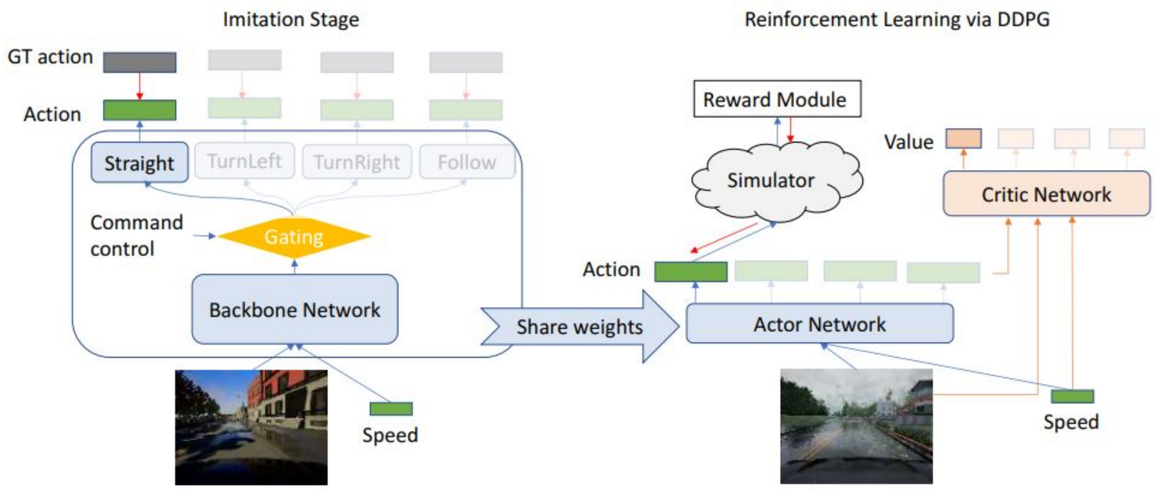


Figure 2.16: The two stages of training with CIRL [15]. The image on the left depicts the initial IL stage. The image on the right depicts the RL stage, where the trained weights from the IL stage is loaded into the actor-network.

2.6.6 Exploring the Limitations of Behavior Cloning for Autonomous Driving (2019)

Codevilla et al. [11] investigated the key limitations of behavior cloning for AVs, and introduced a new benchmark called *NoCrash* for the CARLA simulator. The NoCrash benchmark is described in more detail in Section 2.5.1. They also introduce a new architecture called CILRS. This model was trained and evaluated on the original CARLA and NoCrash benchmarks.

Limitations
Bias, *high variance* and *causal confusion* are identified as as three important problems for behavior cloning. The causal confusion problem happens when the model fails to understand correlations between environment state and certain actions. An example is when a vehicle reducing its speed to stop at a red light. Instead of learning that red lights should result in stopping, the model learns the false correlation that driving slowly should result in stopping. This particular example is a type of causal confusion problem known as the *the inertia problem.*

CILRS
Building on the architecture of CIL [51], Codevilla et al. introduce a new architecture that uses a ResNet-34 backbone to process images. In addition to the branch that outputs actions, they add a speed prediction branch. The speed branch outputs predictions of the current speed. The output of the ResNet34 backbone is propagated through the speed and action branches. This forces the perception module to extract speed-related features in the images. The results show that using a deeper network and a speed prediction branch improved performance. The resulting architecture is called CILRS (CIL with ResNet and Speed prediction), and is visualized in Figure 2.17.

The model was trained on a dataset containing more than 400 hours of driving video. It achieved state of the art results on both the original CARLA and NoCrash benchmarks. They also evaluate the performance for models trained on different amounts of data. The results show that more training data give worse results, i.e., the model trained with the

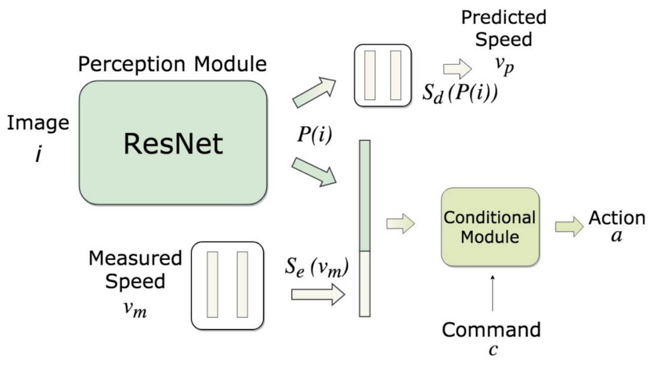


Figure 2.17: The CILRS architecture [11]. It uses ResNet-34 backbone and a speed prediction regularizer.

least amount of data achieves the highest task performance. They suspect that this is caused by the inherent bias and lack of diversity in the data.

2.6.7 Does Computer Vision Matter for Action? (2019)

Zhou et al. [12] investigated the effect of computer vision on agents acting in complex 3D environments. They conducted their research using the video games Doom and GTAV.

In GTAV, they trained and evaluated different agents for the task of urban driving. All agents used three-layer convolutional networks. The type of input modality was different for each agent. One agent was only trained on RGB images, while other agents were trained on a stack of RGB images and ground truth computer vision representations. Some were trained on a stack of RGB images and representations produced by an U-Net with multiple decoders. Each decoder performed a different computer vision task. The type of computer vision representations they investigated were semantic segmentation, instance segmentation, optical flow, depth, and albedo[9].

The agents were evaluated by driving through multiple paths, defined by a start and target position. HLCs were not used for navigation. When the agent reached an intersection, it could always reach the target destination by turning to the right. An episode was considered successful if the agent could reach the target position without colliding with any object.

Their results clearly showed that computer vision matter for action. Agents trained with explicit representations from computer vision generalized better to unseen environments, learned faster, and achieved higher task performance. This performance boost applies when the representations are ground truth computer vision, or when they are predicted by neural networks. Semantic segmentation and depth estimation was deemed the most essential representations. Additionally, their results showed that networks trained with explicit intermediate representations require less training data to achieve the same performance as an RGB-only agent.

[9]Albedo is the intrinsic surface color of different materials in the scene [12].

2.6.8 Learning by Cheating (2019)

Chen et al. [14] split the task of learning to drive into two separate tasks; learning to see, and learning to act. They call their approach *Learning by Cheating* (LBC). Using CARLA, they collect a dataset D in *Town01*. The dataset consists of RGB images, velocities, HLCs, and semantically segmented bird's-eye view (BEV) images. The BEV images include information about traffic light state, lanes, and other actors surrounding the vehicle.

A *privileged* network is trained using the BEV images. Since the privileged agent has direct access to most of the relevant environment information (i.e. it "cheats"), it can focus on the task of learning to *act*. The BEV images are augmented by random perturbations and rotations during training. An example of a BEV image and an augmented BEV image can be seen in Figure 2.18. The privileged network is trained to output five waypoints in world coordinates (i.e. (x, y, z)), which are denoted by $\hat{\mathbf{w}}$. These waypoints indicate the future trajectory of the vehicle.

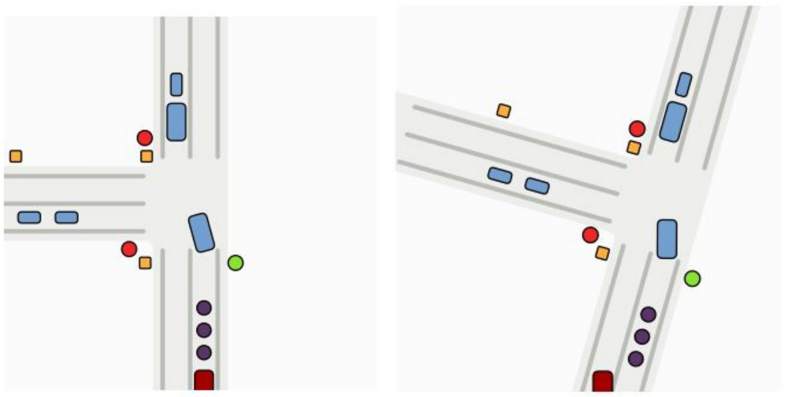

(a) (b)

Figure 2.18: Examples of semantically segmented BEV images, as presented in the original paper [14]. These images are used to train the privileged agent. Each BEV image has seven classes: road, lane markings, vehicles, pedestrians, and each traffic light state. The red square is the agent, and the purple dots are the predicted waypoints. The agent, waypoints, and the *unlabeled class* are not included in the BEV images that the privileged agent uses, but they are visualized in these images. (a) The original BEV image. (b) An augmented BEV image.

Given a set of waypoints $\{\hat{w}_1, ...\hat{w}_5\}$, two different PID controllers are used to output throttle and steering. A longitudinal PID controller produces a throttle value which minimizes the error between the target velocity v, and the average velocity $v*$ required for passing through all waypoints. That is, it minimizes $v - v*$, where $v*$ is defined as:

$$v* = \frac{1}{N}\sum_{i=1}^{N}\frac{||\hat{w}_i - \hat{w}_{i-1}||}{\delta_t},$$

and δ_t is the temporal spacing between waypoints. The velocity value v is set to zero if $v < \epsilon$, where ϵ is set to $2.0\frac{km}{h}$ by default. A lateral PID controller outputs a steering angle s^*. Using least squares fitting, a circular arc is fitted on the waypoints. A single point $p = \{p_x, p_y\}$ is then chosen on the arc, and is used to calculate the steering angle by $s^* = \arctan(\frac{p_y}{p_x})$.

The next stage involves training a *sensorimotor* network. This network is trained with RGB images to output five waypoints in camera coordinates (i.e. (x, y)), which are denoted by $\tilde{\mathbf{w}}$. To train the sensorimotor network, each waypoint in camera coordinates $\tilde{\mathbf{w}} = \{\tilde{w_x}, \tilde{w_y}\}$ is transformed to world coordinates $\hat{\mathbf{w}} = \{\hat{w}_x, \hat{w}_y, \hat{w}_z\}$ with a linear transformation T_p. All waypoints are projected to the ground plane, i.e., $\hat{w_z} = 0$. Given the height of the camera above ground p_y, the field of view (fov) of the camera, and the coordinates of the center of the RGB image $\{c_x, c_y\}$, the linear transformation T_p is given by:

$$\hat{w_x} = \frac{\tilde{w_x} - c_x}{c_y - \tilde{w}_y} p_y$$

$$\hat{w_y} = \frac{c_x}{\tan(\frac{fov}{2})(c_y - \tilde{w}_y)} p_y$$

Training the sensorimotor network involves three different phases. Here, f^* denotes the privileged network, and f_θ denotes the sensorimotor network parameterised by θ.

- **Phase 0**: Initially, the sensorimotor network tends to output waypoints that are close to the center of the image, i.e. $\{\tilde{w_x} \approx c_x, \tilde{w_y} \approx c_y\}$. This causes the denominators in the transformations to get very small, which in turn leads to non-sensical waypoint values. This can cause exploding gradients. To counteract this, the output of the privileged agent is transformed to camera coordinates. The sensorimotor network is optimized using the L_1 distance between its own output and the transformed privileged waypoints. This phase is regarded as a warmup phase, and only lasts for two epochs.
- **Phase 1**: The sensorimotor network is trained by behavior cloning using D. Given an RGB image I, a BEV image M, a velocity v, and a HLC c for any arbitrary state, the parameters of the sensorimotor network θ is optimized with the following cost function:
$$\underset{\theta}{\text{minimize}}\, \mathbb{E}[||T_p(f_\theta(I, v, c)) - f^*(M, v, c)||_1] \tag{2.14}$$
- **Phase 2**: The sensorimotor is trained with a DAgger-like algorithm [21], where the privileged network acts as the expert policy π^*. For each timestep of the rollout, the sensorimotor and privileged networks take turns controlling the vehicle. The probability of the sensorimotor network controlling the vehicle is decided with probability $p = \frac{1}{2} + \frac{1}{2}(1 - 0.95^i)$, where i is the number of the current episode. After each rollout, the sensorimotor network is trained using Equation 2.14 on the aggregated dataset. At any point during a rollout, the sensorimotor network can query the expert on what the current optimal action is. This means that the sensorimotor agent can focus on the task of learning to *see.*

The privileged and sensorimotor networks use a ResNet-18 and a ImageNet pretrained-ResNet-34 backbone, respectively. The output of the backbone is propagated through a series of transposed convolutional layers. The current velocity v is used to produce 128 feature maps filled with the value of v, which are later concatenated with the output of the transposed convolutional layers along the channel dimension. The HLC acts a switch between different branches, similarly to CIL [51]. Each branch consists of batch normalization, convolutional, and spatial softmax layers. All branches are always optimized for each data point, even if the branch is not corresponding to the current HLC. They call this *white-box supervision.* They report white-box supervision as being extremely effective.

An important point is that the sensorimotor agent only has access to RGB images, and not to any privileged information. It achieved state-of-the-art results on the original CARLA and NoCrash benchmarks, achieving 100% success rate for the first time on the CARLA benchmark. It also outperformed CILRS [11] on the NoCrash benchmark.

Chen et al. conclude by stating that a potential direction for future work would be to combine the LBC approach with reinforcement learning.

2.6.9 Other Recent Work (2020 - 2021)

This section includes other works that have come out either concurrently or after LBC [14]. These works also evaluate their approaches on the CARLA and NoCrash benchmarks. This will give an overview of some of the most recent results in this field. It also includes a brief section about the response to the LBC approach.

End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances (2020)

Toromanoff et al. [16] introduced a novel RL procedure. They trained a convolutional network with an encoder-decoder architecture. This network was trained to predict semantic segmentation images and the current traffic light state. The encoder is a ResNet-18 backbone. During the RL training procedure, they freeze the ResNet-18 encoder. The output of the encoder is then flattened, and propagated through the rest of the network. They call the information from the encoder *implicit affordances*. This gives the name of the model, *IA*. IA gets similar results as LBC on the CARLA benchmark. On NoCrash they get slightly lower success rates on training weather conditions, and approximately half the success rates with test weather conditions, compared to LBC. However, IA was trained on multiple CARLA towns and weathers. The test environments of the benchmarks have therefore been seen during training.

Learning Situational Driving (2020)

Ohn-Bar et al. [52] proposed a novel approach for learning urban driving policies called *learning situational driving* (LSD). The goal of the learner is to optimize a policy which is decomposed into a mixture model of probabilistic expert policies (MoE), in addition to an image-based context embedding. Each expert policy in the MoE are implemented as neural networks, and are trained by behavior cloning. The context embedding is updated through supervised learning, and is implemented as a variational autoencoder. The final stage of learning involves RL, where the parameters of the MoE are kept frozen. LSD+ and LSD refers to the models trained with and without RL refinement, respectively. LSD+ achieved state of the art results on the CARLA 0.8.4 NoCrash benchmark.

Affordance-based Reinforcement Learning for Urban Driving (2021)

Agarwal et al. [17] built on the work of IA [16] by training three different models that use *affordances*. Affordances are small pieces of information about the environment [53]. Both explicit and implicit affordances are used. Explicit affordances are pieces of information that comes directly from the simulator, such as traffic light information, distance and speed to the vehicle ahead, and distance to the goal destination. Implicit affordances come from a convolutional autoencoder, which is trained on semantically segmented BEV images. This approach is denoted as ABRL (*affordance-based reinforcement learning)*[10]. They proposed three models:

- **ABRL-A**: only uses explicit affordances.
- **ABRL-A+I**: uses a combination of explicit and implicit affordances.
- **ABRL-I**: only uses implicit affordances.

All models are trained with PPO. They propose a novel reward function defined as follows:

$$R = R_s + R_d + \mathbf{I}(i)R_i \tag{2.15}$$

$$\text{where } R_s = \alpha v, R_d = -\beta d, R_i = -\phi v - \delta, \text{ and}$$

$$\mathbf{I(i)} = \begin{cases} 1 & \text{If a traffic light violation or collision occur} \\ 0 & \text{Otherwise} \end{cases}$$

Here, v is the current velocity, and d is the lateral deviation from the optimal trajectory to the center of the vehicle. α, β, ϕ, and δ are hyperparameters set by the user. R_s rewards the agent for driving efficiently. R_d penalizes the agent for drifting too far away from the center of the lane. R_i penalizes collisions and traffic light violations. This penalty is scaled with the magnitude of the velocity v. This reward function is *dense*, which means that it outputs rewards constantly.

The three proposed models are evaluated on the CARLA and NoCrash benchmarks in CARLA 0.9.6. ABRL-A achieves highest task performance, although all three variants get significantly better results compared to other approaches, i.e., CIL [51], CILRS [11] and LBC [14] on both benchmarks. However, the comparison is not fair. The affordances used give a significant advantage, as they utilize high-level information from the simulator.

Response to LBC

The authors of LSD [52] state that LBC [14] should be seen as an "upper limit on performance", since it uses highly privileged information during training that would not be available in real life. This is in sharp contrast to LSD, which has no access to such information. Carton et al. [62] share a similar sentiment about the use of privileged information, and instead favors algorithms that only use visual input during training. Furthermore, Zhao et al. [63] also criticizes the semantically segmented BEV images for being expensive and requiring access to ground truth information that would only be available in simulated environments. Agarwal et al. [17] questions the generalization capability of the LBC approach to new environments. Tampuu et al. [8] describes how the LBC approach, with its online training phase, mitigates the negative effect of the *distribution shift* problem. However, they also state that performing a similar training procedure in real life could potentially be very difficult. However, the authors of LBC state that the approach could be deployable in real life and refers to methods from sim-to-real-transfer.

Learning to Drive From a World on Rails (2021)

On May 3rd 2021, three of the four LBC authors, Chen, Koltun and Krähenbühl released their next work on autonomous vehicles [18]. In this work they first train a world model which can simulate the actions of an agent without executing them. This forward world model is learned from a dataset of driving trajectories. The trajectories also include

[10]As far as we know, there is no established acronym for this work in the literature.

information about the environment. They then estimate an action-value function using the forward model, dynamic programming and Bellman equations. The action-value function is then used to supervise training of a visuomotor driving policy. By using the action-value function, the agent is able to predict the outcomes of all possible actions before taking them. The agent only uses RGB images and current speed as input when driving.

The *world-on-rails assumption* is an assumption that the actions of the agent does not influence the environment, only its own state. This significantly simplifies the state space for the model, as the world is now only moving passively based on the pre-recorded trajectories. This makes using the Bellman equations to learn a forward model feasible.

This approach achieves the highest score on the *CARLA leaderboard*[11], while using 40 times less training data than the second ranking model. The method also achieves new state of the art results on the NoCrash benchmark. The approach also generalizes to other tasks. This is shown by achieving good results on the ProcGen benchmark, while being more sample-efficient than competing methods.

We will refer to this model as *Rails*. The authors also publish LBC results on the NoCrash benchmark in CARLA version 0.9.10. These results differ from the original results, which are from CARLA version 0.9.6.

[11] `https://leaderboard.carla.org/leaderboard/`

Chapter 3

Methodology

This chapter describes the methodology of our work.

In Section 3.1 we describe the choice of simulator, machine learning framework, and hardware. In Section 3.2 we describe the task of converting the LBC code to the most recent version of the CARLA simulator. This section also explains how some other important issues were fixed. Section 3.3 through 3.6 describe the four experiments that have been conducted for this thesis.

In Experiment 1, we tried to reproduce the LBC approach in the CARLA 0.9.11 simulator. In Experiment 2, the sensorimotor network was trained with ground truth computer vision images. In Experiment 3, we trained and evaluated perception models. We then trained a sensorimotor network with explicit intermediate representations given by the trained perception models. In Experiment 4, we trained the sensorimotor network with an additional RL stage.

3.1 Environment and Technology

Choice of simulator
We wanted to conduct research in a simulator since it allowed us to freely and safely experiment with different approaches. As described in Section 2.5.1, the CARLA simulator [10] is widely used by researchers. We believe that CARLA will continue to see widespread use in the future. The existing LBC code already supported CARLA. Implementing similar functionalities in a different simulator would have required a lot of time and effort. Therefore, the natural choice was to continue using CARLA. This thesis then belongs to the growing field of research that use the CARLA simulator for training and evaluating end-to-end AVs.

Choice of Machine Learning Framework
Since the implementation of LBC used PyTorch, we chose to continue using this framework. The use of PyTorch is also in line with the research community who often use PyTorch over TensorFlow [64]. Another benefit with PyTorch is the inclusion of the Torchvision library, as well as the available open-source Segmentation Models [59] library. These libraries have implementations of many different architectures, which is helpful as it makes implementing, training and evaluating neural networks a quick and efficient process.

Hardware

We had several computational resources available. Due to COVID-19, we could not physically access school areas at NTNU. Instead we received access to a virtual desktop with a Linux operating system, that had 8GB or 12GB VRAM available. We also had access to the IDUN cluster [65], which is a compute platform organized by the High Performance Computing Group at NTNU. Due to limited rendering capabilities on this platform, we did not use it. Additionally, we had two Windows PCs available. One PC was equipped with a RTX3070 with 8GB VRAM, and the other with a GTX1060 with 6GB VRAM.

Most of the computation was done on the RTX3070 and GTX1060 machines, as they were the fastest and were always available. The virtual desktop was used for data collection, because the data saving library created unnecessarily large files on Windows. The privileged network was also trained on the virtual desktop. Every phase of all sensorimotor networks were done on the RTX3070 machine. Perception data collection was done on the RTX3070. The training and evaluation of the perception models was spread across the RTX3070, the GTX1060 and the virtual desktop. All benchmarking was done on the RTX3070 and GTX1060.

3.2 Converting and updating to CARLA 0.9.11

We built on the work by Chen et al. [14] who proposed LBC. They utilized a modified version of CARLA 0.9.6, which was the newest CARLA version at the time of their research. Pedestrian logic was severely lacking in CARLA 0.9.6. Because of this, Chen et al. included multiple modifications to the pedestrian system, including crosswalk logic, teleportation of stuck pedestrians and car avoidance. The code behind their research, along with their custom version of CARLA 0.9.6, was published and is available for use under a MIT license[1].

Instead of using the customized implementation of CARLA by Chen et al., we instead opted for CARLA 0.9.11. This is the most recent version at the time of writing this thesis. The reason we decided to use version 0.9.11 over 0.9.6 is because we wanted to use the latest technology available. As new research tend to use the newest available technology as well, we hope that our results will be easier to compare with concurrent and later works. Using the newest version also let us investigate how the LBC approach fared in a more realistic simulated environment.

CARLA version 0.9.11 has many significant changes from the version used by Chen et al. The 0.9.11 version is more realistic in terms of graphics and physics, and includes a more diverse set of vehicle and pedestrian models. Although the complete list of changes is vast and the effects are complex, some changes have more clear effects on the simulated environment. Some of the changes with direct effects are listed in Table 3.1. All version changelogs can be found at CARLA's official blog[2].

Aside from changes of the simulator itself, some other problems arose in the conversion process. Our code base was forked from the official LBC repository.

Updating the NoCrash benchmark

We found that we needed to update the NoCrash benchmark routes. These routes are defined in terms of pairs of spawn points. Each town has a fixed number of spawn points

[1] `https://github.com/dotchen/LearningByCheating/`
[2] `https://carla.org/posts/`

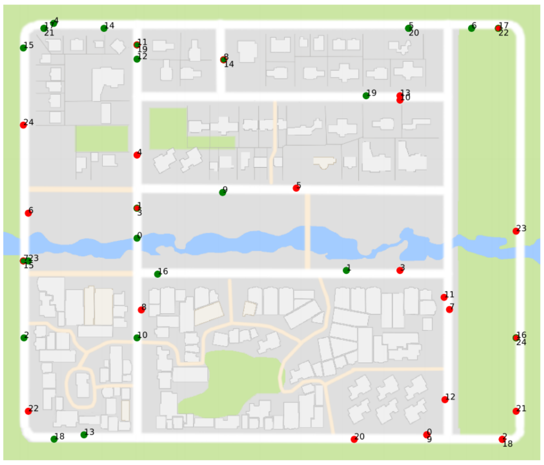
Version	Change	Description
0.9.7	New pedestrian navigation module	Pedestrians are now functioning properly. This change covers the features that Chen et al. implemented in their custom 0.9.6 version.
0.9.7	New illumination system (SSAO, HDRi)	The overall lighting and shadows are now more realistic.
0.9.7	Camera lens distortion	Camera-based sensors feature realistic effects, such as lens distortion.
0.9.7 to 0.9.11	Traffic manager	Multiple improvements to the traffic manager, which affects behavior of vehicles. There are now configurable parameters for the probability of running red lights or ignoring pedestrians.
0.9.8	Weather update	Weather effects and particles are now more realistic. They are also fully configurable. This also affects color temperature and intensity of the scene lighting.
0.9.10	Eye adoption for RGB cameras	Image exposure values are now adjusted according to scene luminance.
0.9.10	New sky atmosphere	Lighting is more realistic.
0.9.10	Enhanced vehicle physics	Changes to vehicle physics, such as suspension and center of mass. This specifically affects how the vehicles turn.
0.9.8, 0.9.10	Pedestrian gallery extension	Updated existing models and added new pedestrian models. Features new animations.
0.9.11	New vehicle models	New vehicle models, which gives more diverse visuals in the scene.

Table 3.1: Overview over a selection of changes following CARLA version 0.9.6 up to version 0.9.11. This table does not include every change which may affect model performance between versions. These changes all have immediate effects on the simulator environment, and the effects are given in the descriptions. Most changes are found in the CARLA update blog posts.

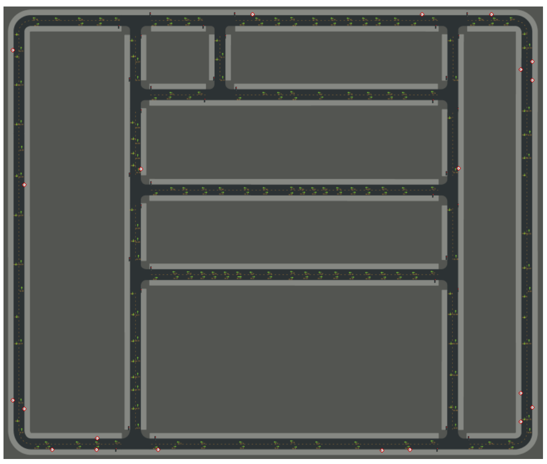
which all have their own ID. These IDs correspond to their index in the list of all spawn points for the given town. We first attempted to run the original CARLA benchmark for testing purposes. This caused the program to crash when running Town01 routes with spawn points 255 and 256. This was because we ran routes intended for CARLA 0.9.6 in CARLA 0.9.11.

By inspection of the list of spawn points, we realized that two spawn points have been removed from Town01, although we did not know which. This meant that every route in the benchmark setup for Town01 could be wrong. If left uncorrected, the benchmark would run completely different routes than intended. We were not able to find any information regarding this change, so we had to resort to manually updating the routes.

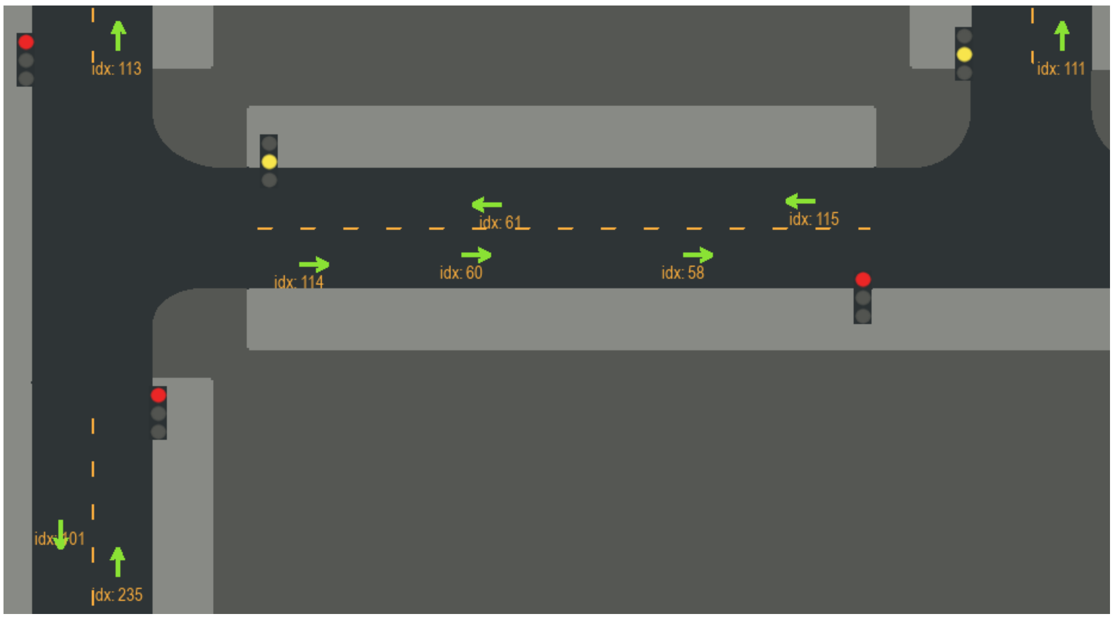
We figured out a strategy to correct the benchmark routes for CARLA 0.9.11. The paper that introduced the NoCrash benchmark [11] includes an image with an overview over start and finish points of all routes in the benchmark. However, these images are made for CARLA version 0.8.4. We call this the 0.8.4 map. The LBC framework includes routes for both CARLA version 0.8.4 and 0.9.6. These indexes are different, but they represent the same routes in the simulator. This means that we had a mapping from spawn points in 0.8.4 to those in 0.9.6. We also used an example script that comes with CARLA, *no_rendering_mode.py*. This script has the functionality to draw a complete map of the town, and we extended it to draw each spawn point together with its index. We call this the 0.9.11 map. The two maps are shown in Figure 3.1.



(a)



(b)



(c)

Figure 3.1: Images showing the setup we used for updating the NoCrash routes for CARLA version 0.9.11. (a) The overview over start and stop points of each route in the NoCrash benchmark, created for CARLA 0.8.4. Taken from the NoCrash paper by Codevilla et al. [11]. (b) The dynamic map created from the modified *no_rendering_mode.py* script, showing spawn point indexes for CARLA 0.9.11. (c) A zoomed in image from the dynamic map.

Using these resources we could now correct the routes. For each route, we looked at the spawn points in the 0.8.4 map and found the corresponding spawn points in the 0.9.11 map. To speed up the conversion, we utilized the fact that the 0.9.11 indexes are always 0, 1 or 2 numbers below their 0.9.6 counterparts. As we evaluated more routes, we could logically induce in which range the two removed spawn points were lying in. For example, we found a rule that $i_{0.9.11} = i_{0.9.6} - 2$ for $i_{0.9.6} \geq 104$. We also found that no spawn points were removed under index 61, which means that $i_{0.9.11} = i_{0.9.6}$ for $i_{0.9.6} \leq 61$. This sped up the conversion considerably. This was a tedious process, and great focus was required to not make any mistakes.

Fixing an issue with Random Number Generation and multiprocessing in PyTorch

In mid-April, Tanel Pärnamaa released a blog post[3] which addresses an issue when using PyTorch with NumPy. In a PyTorch training loop, there is a *DataLoader* object which fetches data from a *Dataset* subclass object. It then feeds data batches to the GPU for computation. A Dataset subclass implements a method which returns a single sample, the *__getitem__()* method. This method also typically performs pre-processing tasks like data augmentation. The DataLoader object has a *num_workers* parameter, which specifies how many subprocesses should be used to load data. These subprocesses are referred to as *workers* in PyTorch. Each of these workers can work in parallel on the CPU. This parallelization speeds up training if the data preprocessing is the bottleneck in the training pipeline, which is often the case.

The data augmentation in the Dataset fetch method usually has some randomness, which is commonly implemented with the NumPy *random number generator* (RNG). When a worker is created, it will copy the parent process state. This means that by default, each worker will have identical copies of the RNG. And this is the cause of the issue, as the RNG of each worker will have the same state. They will therefore produce the exact same sequence of numbers. If the data augmentation relies on the RNG, each batch from each worker will have the same augmentations. This is a problem, because the point of data augmentation is to create variance in the data.

This issue can be fixed by specifically seeding the RNG of each worker. This is done by supplying a seeding function as the *worker_init_fn* argument when creating the DataLoader object. Although this issue is commonly related to the NumPy RNG, it also exists for other RNGs. This issue is not considered a real bug. But it is definitely a lesser known feature, and is easily overseen. Pärnamaa made an analysis of this problem, and the blog post states that 95% of PyTorch repositories on GitHub which use custom Dataset objects, NumPy RNG and $num\_workers > 1$ have this problem. We were not aware of this issue before we read the blog post.

When setting $num\_workers > 1$ in LBC, the ramifications of this issue are possibly even worse. LBC implements three subclasses of the Dataset class, one for the privileged network, one for phase 0 and phase 1 of the sensorimotor network, and one for phase 2. These are named *BirdviewDataset*, *ImageDataset* and *ReplayBuffer* respectively. The BirdviewDataset and the ImageDataset are covered in a wrapper class before being given to their respective DataLoader. This wrapper class is also a Dataset subclass, and implements the *__getitem__()* method. In this method, the wrapper class uses the NumPy RNG to pick samples. This leads to each worker picking the exact same samples for every round of batches. The first batches of each worker will be identical, the second batches will be identical and so forth. This issue is demonstrated in Figure 3.2.

The unmodified LBC repository does not have this problem for phase 0 and phase 1, as *num_workers* is set to 0. For these phases, we initially set the *num_workers* to a higher number, as we found that it sped up the training considerably. However, phase 2 of the

[3]https://tanelp.github.io/posts/a-bug-that-plagues-thousands-of-open-source-ml-projects/

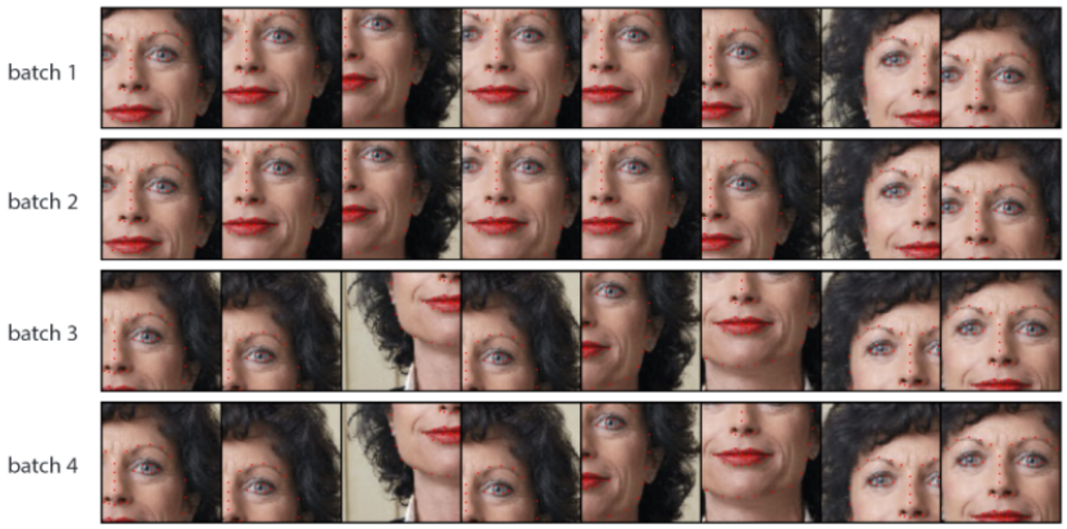


Figure 3.2: A visualization of the batch augmentation issue which commonly arises when using multiprocessing with PyTorch and NumPy. In this case, there are two workers. The first worker produces batch 1 and 3, while the second worker produces batch 2 and 4. Because the RNG states of the workers are identical, they both produce the same augmentations. The image is taken from the blog post of Tanel Pärnamaa and showcases the issue in an official PyTorch example.

official LBC implementation also uses $num\_workers = 4$, which means that the official results probably were affected by this issue as well.

When we found out about this issue we had to retrain every model we had trained so far. This issue did not reveal itself easily. Seeding each worker also fixed image augmentations and other random elements in each of the three Dataset subclasses.

We also fixed another issue related to multiprocessing on Windows. In the *BirdviewDataset* and *ImageDataset* classes, the reading streams to the data files on disk were kept open for the entire lifetime of the object. When using multiple workers, the program was trying to open multiple reading streams for each data file. This was allowed on Linux, but not on Windows. The error was fixed by having each worker open and close the connection every time data was loaded. While this had a performance cost, it was still faster than using a single worker.

Modifying the Replay Buffer for DAgger

As described in Section 2.1.2, a major disadvantage with DAgger is the computational resources required for maintaining the growing size of the dataset D. In the LBC implementation of DAgger, D is implemented as a replay buffer that is stored entirely in memory. The replay buffer is extended with 4 000 new instances for every subsequent rollout, which means that it eventually grows to several GBs in size. Since the replay buffer is extended every episode, the time required for a single epoch increases as DAgger progresses.

Storing the entire replay buffer in memory makes DAgger extremely slow on our hardware. This is because every worker that fetches batches of data must create its own copy of the replay buffer for every single epoch. The system memory is exceeded, and disk caches must be used. When we added ground truth depth and semantic segmentation images to the replay buffer in Experiment 2, it grew to an even larger size.

Another problem we experienced was frequent server timeouts. When this happened, we had to restart the entire DAgger procedure, and subsequently lose all data in the replay buffer. The authors of LBC ran their DAgger script on more powerful hardware, and might not have experienced these issues to the same extent as us.

To make the DAgger implementation more compatible with our hardware, we modify the the replay buffer so that it writes and reads data from disk instead of keeping it in memory. The PC that ran the DAgger procedure is equipped with SSD storage, which made the modified replay buffer a viable option for training. The modified replay buffer significantly decreased the time required for DAgger, while also allowing for stopping and resuming training.

3.3 Experiment 1: LBC Reproduction

This section describes our approach for reproducing LBC. For all stages of training we tried to follow the LBC paper [14] and the README documentation as closely as possible. We refer to the original hyperparameter choices as the default settings.

3.3.1 Data Collection

To collect data we used the data collection script written by the authors of LBC. However, we modified the script so that it captures additional semantic segmentation and depth images. These images were not relevant for this experiment, but will be utilized and discussed further in Experiment 2, which is described in Section 3.4.

Data collection was performed in Town01, and was conducted over the course of several episodes. For each episode, a random path was sampled from the original CARLA benchmark. A Ford Mustang was used as the data collection vehicle, and 250 pedestrians and 100 vehicles were spawned into the environment. The weather condition was randomly selected out of the four training weathers from the NoCrash benchmark. As the vehicle moved through the path, it captured RGB images, HLCs, velocities, and BEV images. It also stored the necessary information needed for calculating the waypoints, which were used as target values for training the privileged network. The RGB images and BEV images were captured with 160x364 and 320x320 resolutions, respectively. The RGB camera was mounted at the front of the car, and positioned 1.4 meters above ground. The camera field-of-view was set to 90 degrees.

Over the course of 117 episodes, 177 428 training frames and 41 424 validation frames were collected. This matches the numbers in the original paper, where they use 174 000 training frames and 39 000 validation frames. We refer to the collected data as the *driving dataset*. This dataset was used for training the privileged network. It was also used for training the sensorimotor networks during the zeroth and first phases in Experiments 1, 2, and 3. Table 3.2 shows the distribution of HLCs for the training and validation sets.

3.3.2 Training the Privileged Network

The privileged network was trained by supervised learning with the default settings. The BEV images were used as input, and the calculated waypoints were used as targets. The network used a ResNet-18 backbone, and was trained with the Adam optimizer with

HLC	Training	Validation
Left	4 158 (2.34 %)	967 (2.33 %)
Right	2 637 (1.49 %)	408 (0.98%)
Straight	7 572 (4.27 %)	1 886 (4.54%)
Follow	163 061 (91.90 %)	38 263 (92.15%)
Total	177 428	41 524

Table 3.2: Distribution of HLCs in the driving dataset for training and validation sets.

a learning rate of 0.0001 and a batch size of 128. The BEV images were augmented as described in Section 2.6.8. The README documentation of LBC states that a well trained privileged network should expect a validation loss less than 0.005. We used this as a guiding principle, and trained with the goal of attaining this validation loss. We also attempted to train the network on a dataset with a balanced HLC distribution.

The best network was used as the privileged network in all experiments.

3.3.3 Data Augmentation

The LBC repository includes a data augmentation script which is described here. The script is written by Codevilla et al. for the *COiLTRAiNE* framework[4], and uses the *imgaug* [66] library to sequentially apply different augmentations to RGB images.

The possible augmentations include Gaussian blur, Gaussian noise, dropping pixels, changing brightness, and changing grayness. The augmentations are applied in random order. The probability of applying an augmentation depend on an initial number N. Multiple or none of the augmentations might be applied to a single image. The degree of how much the augmentation should alter the image is also decided by N. The variable N is increased by 1 for each image that is fetched. This ensures that images are augmented differently every time an image is fetched.

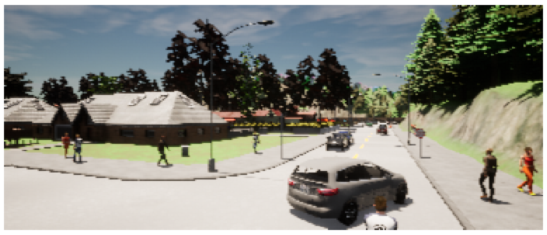
There are also several augmentation settings. These settings influence the probability and degree of each augmentation. Sorted from least to most extreme form of augmentation, the settings are: *soft, soft harder, medium, medium harder, high*, and *super hard.* Examples of *super hard* data augmentations are shown in Figure 3.3. Gaussian noise is not shown in this figure, because the effect is hard to spot with the eye.

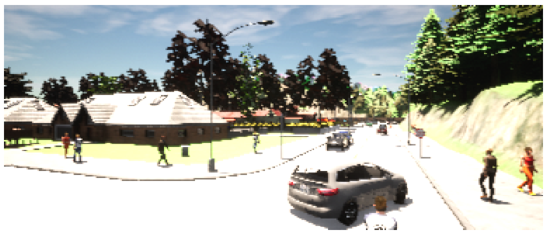
3.3.4 Training the Sensorimotor Network

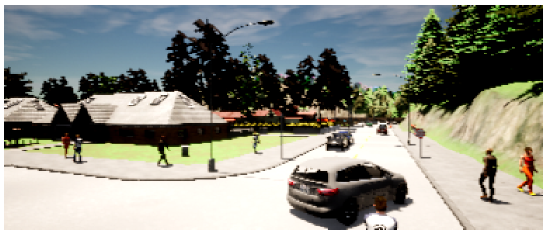
The training of the sensorimotor network involved three different phases of training, which is described more thoroughly in Section 2.6.8. For all phases of training we tried to adhere to the default settings as closely as possible. Adam with a learning rate of 0.0001 was used as the optimizer, and super hard data augmentation was used in phase 1 and 2. The only setting we changed was the batch size, which was done due to inferior hardware.

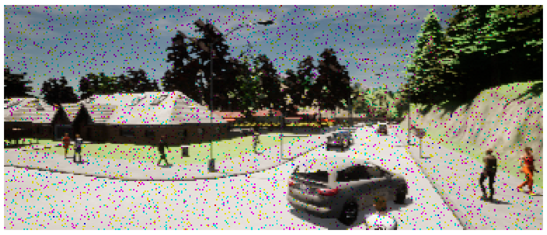
In phase 2, the LBC implementation of DAgger was used to train the sensorimotor net-

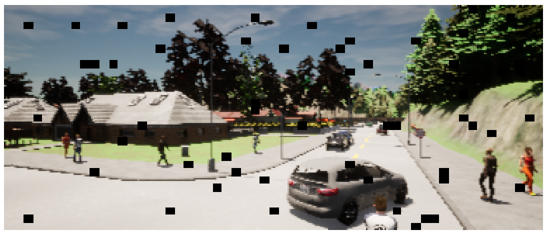
[4]The COiLTRAiNE framework can be found here `https://github.com/felipecode/coiltraine`. We assume that this augmentation script was used by Codevilla et al. for the CIL [51] paper. This is because the list of augmentations mentioned in the CIL paper correspond with augmentation script, and because Felipe Codevilla was involved with both CIL and COiLTRAiNE.


(a) Original RGB image

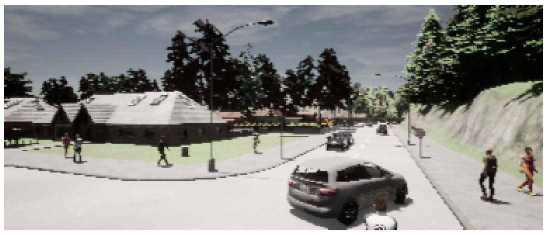

(b) Change of brightness


(c) Change of contrast


(d) Dropping pixels


(e) Dropping rectangles of pixels

(f) Gaussian blur


(g) Change in grayness

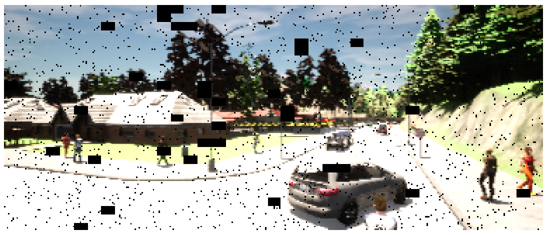

(h) Combined augmentations

Figure 3.3: Examples of image augmentations used during training with the *super hard* augmentation setting. From the top left: (a) original RGB image, (b) brightness change, (c) contrast change, (d) random percent of pixels dropped, (e) random subset of rectangles dropped, (f) Gaussian blur, (h) some randomly selected combined augmentations.

work. Whenever we refer to DAgger from this point on, we are referring to the LBC implementation. Their implementation differs slightly from the original DAgger pseudocode. For instance, $\hat{\pi}$ is trained for five epochs after every rollout. It also uses a special sampling mechanism, which performs a weighted choice of instances after the first epoch. The weights are based on the loss in the preceding epoch.

A summary of each phase follows:

- **Phase 0**: The sensorimotor network was trained to stabilize waypoint prediction for two epochs. A ResNet-34 backbone pretrained on ImageNet was used as the backbone. The batch size was set to 60, which differs from the default setting of 96.
- **Phase 1**: Using the weights from phase 0 as a starting point, behavior cloning was performed for 65 epochs using the privileged network as the expert.
- **Phase 2**: The phase 1 checkpoint was used as the inital policy π_1 in the DAgger algorithm. The privileged network was used as the expert policy π^*. The sensorimotor network was trained with DAgger for 11 episodes. Each episode consisted of a *rollout* and *training* stage. Each rollout took place in Town01, and added 4 000 new instances to a replay buffer. After each rollout, the network was trained for five epochs on all instances in the replay buffer, using super hard data augmentation. The sensorimotor and privileged agents took turns at controlling the vehicle.

  We ran CARLA and the phase 2 script on the RTX3070 GPU. Due to limited VRAM, we had to significantly reduce the batch size from the default settings. We set the batch size to 29, while the default setting is 128. We restrict phase 2 to a total of 11 DAgger iterations, which is in accordance with the configuration file of their published sensorimotor network. Due to long computation times, we followed this restriction for the rest of the experiments.

After phase 2 finished, some adjustments to the PID controller parameters were made. After training, the agent may make too hard turns, which leads to cutting corners. The agent can also make too soft turns, which can make it leave its lane or go off the road. By tuning the PID controller, we can give the agent smoother and more fitting turns. As each HLC has its own set of PID values, we could specifically adjust for the HLC it struggled with. For the *Lane follow* HLC, which features both left and right turns, we tried to find a balance so that it can handle the difficult left turns as well as the difficult right turns. Finding the right balance was difficult. An example that illustrates this is that tightening the left turns could lead to corner cutting for the right turns.

We call this sensorimotor network *LBC-R* (*LBC-Reproduction*). The network was evaluated on the NoCrash benchmark. The benchmark was ran three times with three different RNG seeds. The statistical mean and standard deviation of each task result was estimated. The original CARLA benchmark was dropped due to time constraints.

The privileged agent was also benchmarked for a single RNG seed. This benchmark gave an indication of the upper limit of the performance of the sensorimotor network.

3.4 Experiment 2: LBC with Ground Truth Computer Vision

This section describes our approach for extending the sensorimotor network to use ground truth depth and semantic segmentation images. This experiment is motivated by the results of Zhou et al. [12], who explored the effects of utilizing extra intermediate representations as input to agents acting in complex 3D environments.

3.4.1 Data Collection

As mentioned in Section 3.3, the data collection procedure was modified to include additional ground truth computer vision images, in the form of semantic segmentation and depth images. These images were captured by semantic segmentation and depth sensors. These sensors have the same viewpoint, field-of-view, height above ground, and image resolution as the RGB sensor.

The semantic segmentation sensor in CARLA 0.9.11 has 22 different classes. The semantic segmentation images are stored on disk as RGB images, where the red channel is encoded with an integer indicating which class the pixel belongs to. The depth sensor captures images in an RGB format, which are then converted by a logarithmic depth converter. This gives pixel depth values with millimeter precision. The depth images are stored as grayscale images, with values ranging from 0 to 255. Low values indicate objects closer to the camera, while higher values indicate objects farther away.

3.4.2 Training the Sensorimotor Network with Extra Input Modalities

The sensorimotor network is trained on a stack of inputs consisting of RGB images, semantic segmentation images, and depth images. Instead of using all the available semantic classes in CARLA, we choose eight classes that we deem important for autonomous navigation. Everything else is classified as *unlabeled*, which means there are nine semantic classes in total. The semantic classes which we used are presented in Table 3.3.

We had previously observed that the RGB agents could both ignore red lights and fail to continue driving when the lights turned green. To potentially reduce this problem, the semantic segmentation images included the traffic lights class. This allowed us to study the effect of computer vision representations on traffic light violations.

The depth images were normalized from the range $[0, 255]$ to $[0, 1]$, while the RGB images were normalized using the ImageNet mean and standard deviation values. RGB images were also augmented with the super hard data augmentation setting. RGB, semantic segmentation, and depth images were then concatenated together along the channel dimension. This resulted in a stack of images, consisting of 13 channels[5]. A visualization of this input stack is shown in Figure 3.4. We used the same network architecture as described in Section 2.6.8, but increased the number of input channels to 13 to make it compatible with the new input type. The ResNet-34 backbone is not pretrained on ImageNet. We argue that the task of learning to drive from RGB images is different than learning to drive from RGB, semantic segmentation and depth images.

[5] 3 RGB channels + 9 semantic channels + 1 depth channel = 13 channels

Class Name	**Color**	**Description**
Sidewalk	Pink	All parts of the ground that are meant for pedestrians and cyclists.
Road	Purple	All sections of the ground plane designated for vehicles.
Roadline	Green	Road markings on the surface of the road.
Pedestrian	Red	All humans. Mostly walking kids or adults. Can also be riding motorcycles or bicycles.
Vehicle	Blue	All motorized vehicles and bicycles.
Pole	Grey	Vertical poles. Includes poles for traffic signs and traffic lights, but not the actual signs or lights themselves.
Traffic Light	Orange	This class is for the traffic light *boxes*. Does not show the current light state.
Ground	Dark Purple	All horizontal ground areas not classified as sidewalk or road.
Unlabeled	Black	All pixels not belonging to any of the other classes.

Table 3.3: An overview over the classes that were used for semantic segmentation. The color and a short description of each class is provided. The colors follow the Cityscapes color palette.

We modified the live video view of the agent to show the complete input stack. This gave better visualizations, easier debugging, and better interpretability of the model.

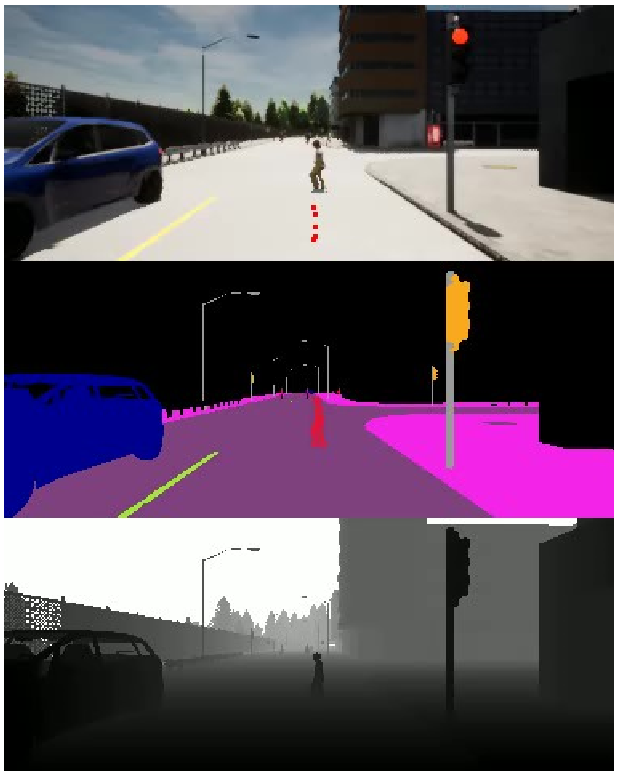


Figure 3.4: An example of the input stack which the sensorimotor network is trained on. From top to bottom: RGB, semantic segmentation, and depth.

To train the sensorimotor network, we used the privileged network as the expert policy. This network was trained in the previous experiment (Section 3.3.2). As for Experiment 1, we used the default settings when possible, but altered the batch size when necessary. See Section 2.6.8 and Section 3.3.4 for a more thorough description of the sensorimotor training phases. A summary of each phase follows:

- **Phase 0**: The sensorimotor network was trained to stabilize waypoint prediction for two epochs using batch size 60.
- **Phase 1**: The sensorimotor network was trained by behaviour cloning for 65 epochs.
- **Phase 2**: The DAgger algorithm was ran for 11 episodes, using the privileged network as the expert policy. The batch size was set to 29.

Before benchmarking we changed the braking threshold ϵ value, which was set to $1.5\frac{km}{h}$. We used the default PID parameters. We call this sensorimotor network *LBC-GTCV* (*LBC-Ground Truth Computer Vision*). The network was evaluated on the NoCrash benchmark with three different RNG seeds. We estimated the statistical mean and standard deviation of each task result.

3.5 Experiment 3: LBC with Trained Perception Models

This section describes our approach to extending the original LBC architecture with trained perception models. In the previous experiment the model was given ground truth computer vision images as input. These images are of perfect quality, and using them can be advantageous compared to only using RGB, and it would not be possible in the real world. To create a more realistic setting we trained and integrated perception models which take RGB images as input and produce semantic segmentation and depth estimation images. This output was then used as input for the sensorimotor network. The combined model then had the same input modality as the original LBC approach. We wanted to explore how much improvement, if any, could be made on the original LBC approach when using these intermediate representations.

This experiment was done in two parts. In the first part we trained, evaluated and compared different perception models. In the second part we integrated the selected models into the LBC architecture. The combined network was then trained and evaluated on the NoCrash benchmark.

The goal of this experiment was not to create perfect perception models. Because of time constraints, we wanted to create models that would give satisfactory results for our study on autonomous vehicles.

3.5.1 Perception Model Training, Evaluation and Selection

To create intermediate representations for the sensorimotor network we considered multiple types of architectures. We tried to fine-tune models, train models from scratch, and use unmodified pretrained state of the art models. An evaluation scheme was set up to compare the different architectures. We then chose the most fitting models.

Data Collection
To fine-tune and train models from scratch, we needed perception data. The perception models were only trained on images from the training settings. This was done to ensure fair comparison with the original LBC approach. This means that the perception models have not seen images from the test settings.

We collected a training dataset of 46 500 frames from *Town01* with the training weather types[6]. For model evaluation we collected two test sets. Both test sets were collected from the unseen environment of *Town02*. The first set, *Test1*, consisted of 500 frames from each training weather type, which totaled to 2 000 frames. The second set, *Test2* consisted of 500 frames from each test weather type[7], which totaled to 1 500 frames.

The perception datasets were collected with a dedicated data collection procedure. We set up an environment with a moderate amount of cars and pedestrians. A set of sensors was mounted on each vehicle. This set consisted of RGB, semantic segmentation and depth cameras. To avoid having the sensors getting placed inside the body of some of the cars, we placed the sensors at 3 meters height. This was different than the height used by the original LBC approach, which was 1.4 meters. All cars roamed around the environment

[6] *ClearNoon, ClearSunset, HardRainNoon* and *WetNoon* which are the training weathers for both the original CARLA benchmark and the NoCrash benchmark.

[7] *WetCloudyNoon* from the original CARLA benchmark, *WetSunset* from the NoCrash benchmark and *SoftRainSunset* which both benchmarks have in common.

with the standard CARLA autopilot. To ensure variety in images, each camera saved an image every 30 seconds in simulation time. The camera yaw degree was sampled from a normal distribution with a 0 degree mean and a 45 degree standard deviation. This gave some more variation in the datasets. Each training weather type had the same number of images.

Model Selection
Because of time constraints we could only train and evaluate a limited number of models. We therefore tried to select a varied set of architectures to find a good fit for our purpose. We looked for pretrained state-of-the-art architectures at *paperswithcode.com*[8]. For semantic segmentation, we looked at the best ranking models on the *Cityscapes val* benchmark. This is a benchmark with images from various urban traffic situations. Specifically, we looked at the highest ranking model *HRNet-OCR* [67] which is available on the authors GitHub page[9].

For monocular depth estimation we looked at three pretrained architectures: *MiDaS* [47], *AdaBins* [68] and *Monodepth2* [69]. We found *AdaBins* via *paperswithcode.com* where it had the highest score on the *KITTI Eigen split* benchmark. *Monodepth2* was used by Wigum et al. [49] who also tried to solve a similar task. We found *MiDaS* via internet searching and a machine learning internet forum. This architecture had good results on multiple benchmarks, but it was not listed on *paperswithcode.com.* We suspect that was because the full training code was not available, so there was no open-source implementation that could reproduce their results, which the website requires. The models are available on their respective GitHub repositories[10].

When selecting these pretrained models we also considered how they would fit into our pipeline. While some models were available for standalone prediction, it may have been hard to integrate them into a live driving pipeline. *MiDaS* is available as a *Torchvision* module, which would be easy to integrate. Meanwhile, *Monodepth2*, *AdaBins*, and *HRNet-OCR* were hard to integrate into the pipeline. They also gave poor results during initial testing, and were consequently dropped from further evaluation.

For semantic segmentation, we also selected a number of architectures from the *Torchvision* module which is shipped with PyTorch. This module provided a simple interface to load of models. The torchvision models can optionally use weights pretrained on a subset of *COCO train 2017.* The available models and their scores evaluated on *COCO val 2017* can be seen in Table 3.4. Based on these scores, we selected *FCN ResNet101*, *DeepLabv3 ResNet50*, *DeepLabv3 ResNet101* and *DeepLabv3 MobileNetV3-Large* for further fine-tuning and evaluation. We also evaluated a U-Net architecture with a ResNet-34 backbone pretrained on ImageNet[11].

For monocular depth estimation, we struggled to fine-tune the pretrained state of the art models. Some architectures required data on specific formats, some were trained with stereo images which were not easily available, and some gave unsatisfying results. Our main advantage was that we had large amounts of training data with perfect targets available from the simulator. We investigated if we could train a ResNet to learn this task. We realized that it would make sense to add a decoder network following the ResNet, to

[8]This is a website with comparisons and rankings of open-source implementations of machine learning models. The website has rankings on a on a wide variety of machine learning tasks and benchmarks.

[9]`https://github.com/NVIDIA/semantic-segmentation`

[10]*MiDaS*: `https://github.com/intel-isl/MiDaS`, *AdaBins*: `https://github.com/shariqfarooq123/AdaBins`, *Monodepth2*: `https://github.com/nianticlabs/monodepth2`

[11]`https://github.com/qubvel/segmentation_models.pytorch`

Network	Mean IoU	Global pixelwise accuracy
FCN ResNet50	60.5	91.4
FCN ResNet101	63.7	91.9
DeepLabv3 ResNet50	66.4	92.4
DeepLabv3 ResNet101	67.4	92.4
DeepLabv3 MobileNetV3-Large	60.3	91.2
LR-ASPP MobileNetV3-Large	57.9	91.2

Table 3.4: Scores on the *COCO val 2017* dataset by models available from the semantic segmentation subpackage of *torchvision*. This table is reproduced from `https://pytorch.org/vision/stable/models.html`.

create better looking outputs.

U-Net was also considered for depth estimation, which is an encoder-decoder architecture. Even though the U-Net architecture was created for the semantic segmentation task, we gave it a single output channel and used it for monocular depth estimation. Zhou et al. [12] used a U-Net to create depth estimation images as well. However, they used it in a multi-task learning situation where they had multiple decoders attached to a single encoder. Two U-Net implementations were evaluated, one with the original encoder[12] and one which used a ResNet-34 pretrained on ImageNet. The latter used the same implementation as the one we used for semantic segmentation, but with a single output channel.

Model Training
We wrote two PyTorch training scripts: one for semantic segmentation models and one for depth estimation models. As all models were available as PyTorch modules, the training scripts were easily configured to train the different models.

All models were trained with an Adam optimizer with learning rate $\eta = 0.001$, momentum $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The models were trained on different hardware with some difference in available VRAM. We therefore used different batch sizes for different models. All models were trained until convergence. Data augmentation was applied using the LBC data augmentation script, which is described in Section 3.3. Semantic segmentation models were trained with the *medium harder* setting, while depth estimation models were trained with the *medium* setting. We found that these were the hardest augmentations we could apply which did not lead to unacceptable decrease in output quality.

For semantic segmentation we used a cross entropy loss function. This loss function also allowed for using weighted loss to incentivize the network to better learn certain classes. For depth estimation we used the loss function designed by Alhashim and Wonka [46], described in Section 2.3.3.

During training we discovered that the U-Net models created very good-looking output, but that it completely failed to learn the traffic light class. As in Experiment 2, we wanted to study traffic light violations. We therefore trained two additional U-Net models, where we weighted the traffic light class losses 2.5 and 5 times higher than other classes, respectively.

Model Evaluation
To compare the models we set up an evaluation procedure with multiple metrics. We

[12]`https://github.com/milesial/Pytorch-UNet/tree/master/unet`

compared the semantic segmentation models using the *Mean IoU* and *Weighted IoU* metrics. For depth estimation we used the *accuracy within threshold* and *RMSE metrics.* For accuracy within threshold we used the commonly used threshold values of 1.25, 1.25^2 and 1.25^3. Evaluation metrics for semantic segmentation and depth estimation are further described in Section 2.3.2 and Section 2.3.3, respectively.

To get an idea if the models were fast enough to be deployed in an autonomous system, we also measured the prediction speeds of the models. Prediction speeds were measured by calculating the average time of 500 single predictions. A single prediction time was found by saving the system time before input was given to the model and after the output was returned, and then calculating the difference. Because the speed of the model can be unstable on the first predictions, we made 600 total predictions and cut the first 100 prediction times. This is not the most accurate way of measuring speed, as other tasks on the system might uncontrollably interrupt the CPU and use system resources. But in this case the procedure was sufficient as we only needed rough estimates of prediction speeds.

As we began benchmarking the sensorimotor network, we discovered that the semantic segmentation predictions were of lower quality than expected. This negatively affected the performance of the sensorimotor network. We suspected that the quality drop was caused by a difference in the images seen during training, and images seen during inference. As mentioned earlier, for the original RGB sensor height of 1.4 meters, there was a bug where the camera was placed inside the body of some of the data collecting cars. To solve this, we moved the camera to a height of 3 meters during data collection.

We attempted to improve the prediction quality by retraining the semantic segmentation model with a newly collected dataset. In this dataset, we kept the camera height at 1.4 meters, but only mounted sensors on cars that were not susceptible to the camera bug.

3.5.2 Training of the Sensorimotor Network with Trained Perception Models

A summary of each phase follows:

- **Phase 0**: The sensorimotor network was trained to stabilize waypoint predictions for two epochs using batch size 45.
- **Phase 1**: The sensorimotor network was trained by behaviour cloning for 33 epochs.
- **Phase 2:** The DAgger algorithm was ran for 11 episodes, using the privileged network as the expert policy. The batch size was set to 29.

As in Experiment 2, we modified the live video view of the agent to include the intermediate representations. This gave better visualizations, easier debugging and better interpretability of the models.

The default PID parameters were used during benchmarking. However, the brake threshold value ϵ was set to $1.5\frac{km}{h}$. We call this sensorimotor network *LBC-TCV* (*LBC-Trained Computer Vision*), and evaluate it on the NoCrash benchmark. The statistical mean and standard deviation of each task result was estimated.

The complete PyTorch architecture of LBC-TCV can be found in Appendix B.

3.6 Experiment 4: LBC with Reinforcement Learning

In this experiment, the sensorimotor network was trained with an additional RL stage. The main motivation for this experiment was to attempt to improve the performance of the sensorimotor network without supervision of the privileged network.

Training without the privileged network is desirable for numerous reasons. For instance, the privileged network may be biased on the dataset it was trained with. Training the privileged network also requires a dataset consisting of hundreds of thousands of instances that must be collected with a slow data collection procedure. As discussed in Section 2.6.9, several authors of other recent end-to-end approaches criticized LBC for its use of privileged information.

PPO was used for this experiment because it is known to be stable, easy to implement, and with desirable convergence properties [23]. Inspired by CIRL [15], a pretrained sensorimotor network was used. The hope by using a pretrained sensorimotor network would give the initial policy with a reasonable starting policy, that is capable of driving well. The critic used semantically segmented BEV images to predict the expected return of rewards. This information is easier to interpret than an RGB image.

This experiment built on the other experiments and the existing LBC framework. Therefore the actor used the same architecture as the sensorimotor network. It also meant that waypoints were used as the output modality. We could not find any other published work on RL which have used waypoints as the output modality.

3.6.1 PPO Implementation

The PPO implementation was written by modifying the phase 2 script. This implementation followed the pseudocode described in Algorithm 2. The phase 2 script already included a rollout phase where instances were collected using a policy π_{θ_k}, and stored in a replay buffer. This is similar to the trajectory collection in PPO. Furthermore, the next part of the phase 2 script trains the next iteration of the policy $\pi_{\theta_{k+1}}$ on the instances in the replay buffer. This is similar to the training phase of PPO. The PPO implementation was also inspired by the minimal PPO implementation of Barhate [70].

3.6.2 Actor and Critic Networks

The actor network had the same architecture as the sensorimotor network. It was initialized with the pretrained weights from phase 1 in Experiment 1. By using the phase 1 weights, the policy had more room for improvement compared to using phase 2 weights. This made it easier to see if the network had improved.

During the RL stage, the sensorimotor network would not be limited to only learning from expert demonstrations. It would be able to experiment with different actions and learn from the new situations it encountered during exploration.

The critic used a similar architecture to the privileged network. However, the branches were designed to evaluate states, instead of outputting waypoints. Each branch had the following layers: batch normalization, convolution, max pooling, ReLU, batch normalization, convolution, max pooling, and ReLu again. The output of these layers was then

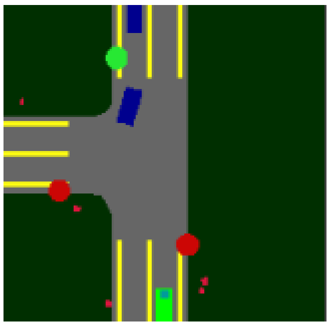


Figure 3.5: Example of a BEV image used as input to the critic in PPO. Image generated by the LBC code. The green rectangle is the vehicle of the sensorimotor network. Small red dots are pedestrians. Big red circles are red traffic lights. Green circles are green traffic lights. Blue squares are vehicles. The gray is the road. The yellow lines are the road lines.

flattened, before being propagated through two feedforward layers with batch normalization and ReLU activation. The final layer had linear activation so that it could evaluate states to any possible value.

The vehicle of the sensorimotor network was also added as a class to the semantically segmented BEV images. This meant that the BEV image consisted of eight classes in total. The extra class was added to ensure that the critic could perceive the current position of the the actor on the road. An example of this type of BEV image can be seen in Figure 3.5.

3.6.3 Reward Function

The reward function from ABRL [17] was used. This function was defined in Equation 2.15, and was used because it is *dense* and simple. It also followed the principle by Sutton and Barto [22] on how reward functions should be designed. The rewards should not tell the agent *how* it should achieve the task, rather, the rewards should tell the agent *what* it should achieve. The agent should drive efficiently, safely, and close to the center of the lane. This was also the motivation behind the design of the ABRL reward function.

The reward function of ABRL [17] has four hyperparameters α, β, ϕ, and δ that have to be set by the user. The hyperparameters were set to the values used in the ABRL paper, which were:

$$\alpha = 1, \beta = 1, \phi = 250, \text{ and } \delta = 250.$$

This meant that the final reward function was defined as:

$$R = v - d + I(i)(-250v - 250)$$

where v is the current velocity of the agent, and d is the lateral deviation from the optimal trajectory. $I(i)$ equals 1 if a collision or traffic light infraction occurs, and 0 otherwise.

3.6.4 Training

The actor network π_{θ_k} was initialized with the weights from phase 1 in Experiment 1, while the critic network $\hat{V}_{\phi_k}(s_t)$ was randomly initialized.

For every episode, a dataset of trajectories D was collected in Town01 with the current actor policy π_{θ_k}. Each episode consisted of 20 rollouts, where each rollout collected a maximum of 1000 instances. Each instance consisted of a RGB image, speed, HLC, and BEV image. Each rollout involved the agent driving through a random route from the NoCrash benchmark in Town01. The high-level navigational planner provided the agent with HLCs to guide the it from start to end position. The weather condition was uniformly sampled as one of the four NoCrash training weather conditions. A rollout was aborted if a collision happened or if the lateral deviation from the optimal trajectory was larger than 5 meters. Following Agarwal et al. [17], the amount of pedestrians was uniformly chosen as a number between 100 and 250. Likewise, the amount of vehicles was chosen uniformly between 60 and 120.

Following the Spinning Up implementation of PPO [23], GAE was used to compute advantage estimates $\hat{A}_t$. For GAE, γ was set to 0.97, and λ was set to 0.94. The c_1 coefficient was set to 1, and the entropy coefficient c_2 to 0.001. The parameters of the policy θ_k and ϕ_k was updated with L_{PPO} function, as defined in Equation 2.11. Two separate Adam optimizers for the parameters of the actor and critic were used. The Adam optimizer for the actor had learning rate set to 0.0001, while the learning rate for the critic's optimizer was set to 0.00001.

The actor policy was represented as a multivariate Gaussian policy, with a diagonal covariance matrix. Since there are five waypoints $[\tilde{w_1}, \tilde{w_2}, \tilde{w_3}, \tilde{w_4}, \tilde{w_5}]$, each defined by an x and y position, ten numbers had to be sampled for each timestep during rollouts. This meant that the multivariate Gaussian distribution was defined by a mean vector of ten numbers, and a $10x10$ diagonal covariance matrix. The output of the actor policy was regarded as the mean μ. Following Carton et al. [62] and the implementation of Barhate [70], the standard deviation σ of the policy was linearly decreased as training progressed.

As already mentioned, previous works have not used waypoints as the output modality in RL for AVs. Therefore, several methods of sampling waypoints have been proposed. For each method we ran PPO with different hyperparameters. These hyperparameters are explained below.

Method 1:

All waypoints were sampled with the same σ value.

The initial σ value was set to 0.008 and linearly decreased by 0.0001 every 5000 time steps. The actor and critic was updated for two epochs in each episode. The clip ratio ϵ was set to 0.1.

Method 2:

Each waypoint had a different σ value. Waypoint $\tilde{w_5}$ was sampled with the largest σ, waypoint $\tilde{w_4}$ was sampled with a lower σ, and so forth. For example if $\sigma = 1$, the five waypoints $[\tilde{w_1}, \tilde{w_2}, \tilde{w_3}, \tilde{w_4}, \tilde{w_5}]$ would have the following σ values $[0, 0.25, 0.5, 0.75, 1]$.

The initial σ value for $\tilde{w_5}$ was set to 0.15, and every 5000 time steps it was linearly decreased by 0.001. The actor and critic was updated for two epochs in each episode. The clip ratio ϵ was set to 0.1.

Chapter 4

Results

This chapter describes the experimental results of this thesis. In Section 4.1, an overview over the NoCrash benchmark results for all trained models is given. This overview includes comparisons to other state-of-the-art approaches. Section 4.2 presents the results from Experiment 1 where LBC was reproduced in the updated CARLA 0.9.11 environment. Section 4.3 presents the results from Experiment 2 where LBC architecture was extended to use semantic segmentation and depth estimation images given directly by the simulator. Section 4.4 presents the results from Experiment 3, where LBC was extended to use semantic segmentation and depth estimation images, but provided by trained perception models instead of the simulator. Section 4.5 presents the results from Experiment 4, where LBC was extended with a RL phase.

The models trained in this thesis are denoted by LBC-R, LBC-GTCV and LBC-TCV. These are the models from Experiment 1, 2, and 3, respectively. A video showing example demonstrations for these models is available at the following link: `https://youtu.be/OdWVNI-DxRQ`.

4.1 Comparison of Key Models on the NoCrash Benchmark

This section presents Table 4.1, which compares the results of many models on the NoCrash benchmark. The table includes the results of various state-of-the-art approaches, the LBC results from the original authors, and the results from the experiments of this thesis. The key takeaways from this table are discussed in the following sections.

		Training Town						Test Town					
		Training Weather			Test Weather			Training Weather			Test Weather		
Model	Version	Emp.	Reg.	Den.	Emp.	Reg.	Den.	Emp.	Reg.	Den.	Emp.	Reg.	Den.
LSD+ [52]	0.8.4	-	-	-	-	-	-	94	68	30	96	65	32
IA [16]	0.9.10	85	85	63	-	-	-	77	66	33	-	-	-
Rails [18]	0.9.10	98	100	96	90	90	84	94	89	74	78	82	66
LBC [14]	0.9.6	97	93	71	87	87	63	100	94	51	70	62	39
LBC†[18]	0.9.10	89	87	75	60	60	54	86	79	53	36	36	12
LBC-R	0.9.11	96	94	69	89	91	68	86	82	42	71	69	30
LBC-GTCV	0.9.11	98	99	85	97	96	89	100	96	60	100	96	59
LBC-TCV	0.9.11	85	92	76	70	81	79	99	96	52	75	71	38

Table 4.1: A comparison of various architectures on the NoCrash benchmark, presented with mean success rates on each task. The *version* column indicates which CARLA version was used, and is included because there can be significant differences between versions. *Emp.*, *Reg.*, and *Den.* refers to the empty, regular, and dense tasks, respectively.

4.2 Experiment 1: LBC Reproduction

The results of LBC-R on the NoCrash benchmark with comparisons to other approaches is given in Table 4.1. More detailed results of LBC-R can be seen in Table 4.2. Table 4.3 shows the benchmark results with respect to weather types.

Table 4.1 shows that LBC-R performed similarly to LBC in the training town, but slightly worse in the test town. However, it performed significantly better than LBC†, which was benchmarked in CARLA 0.9.10. Table 4.2 shows that the traffic light violation rates were significantly higher for Town02 than for Town01. Table 4.3 shows that it performed worse in the the *ClearSunset* and *WetSunset* weathers.

LBC-R failed in many different scenarios, and some examples are mentioned here. Sometimes it turned too hard, which led to collisions. In other situations it did not turn hard enough, which led to driving out of the road. It sometimes ran red lights, which led to collisions with other vehicles. It could also fail to continue driving after stopping at a red light, causing a timeout failure.

In the dense and regular environments, there were some failures which were not caused by the agent. In some cases, other vehicles ran red lights and crashed into the agent. In other routes, the traffic was so dense that the agent could not reach the goal within the time limit, even if it did not make any mistakes.

A common failure pattern of LBC-R was that the agent was reluctant to drive into shadows. This was especially evident for the *ClearSunset* weather, where there are many shadows with high contrast to the sunset lighting. This led to the failure of many routes. An example of this behavior is shown in Figure 4.1. In this situation, the agent was supposed to make a hard right turn to get into the right lane.

When training the privileged network, the final epoch loss was 0.008. This is of interest because the authors of LBC explicitly state that a well trained privileged agent should get a loss lower than 0.005. The benchmark results of the privileged agent are supplied in Appendix A.

The privileged network showed erroneous behavior in some situations involving intersections and yellow traffic lights. In these situations, the agent first legally passed the traffic light and entered the intersection. If the traffic light turned yellow after the agent had passed it, the network would output waypoints that were too stretched out. The speed of the agent would then continually increase. Eventually the agent would crash or timeout, and fail the benchmark route. For the empty environment in Town01, every route fail was caused by this issue.

Task	Town	Weather	Success	Collision	Timeout	Lights ran
Empty			95.7 ± 1.2	2.7 ± 1.5	1.7 ± 0.6	9.2 ± 0.3
Regular	train	train	93.7 ± 4.2	5.0 ± 3.6	1.3 ± 0.6	8.6 ± 1.0
Dense			69.0 ± 4.4	28.7 ± 5.5	2.7 ± 1.2	16.2 ± 1.6
Empty			89.3 ± 2.3	3.3 ± 1.2	8.0 ± 3.5	7.6 ± 1.5
Regular	train	test	90.7 ± 3.1	2.7 ± 1.2	6.7 ± 2.3	7.4 ± 1.1
Dense			68.0 ± 12.2	32.0 ± 13.9	0.7 ± 1.2	12.2 ± 2.0
Empty			86.0 ± 1.7	5.7 ± 0.6	8.3 ± 1.2	24.6 ± 0.6
Regular	test	train	81.7 ± 2.1	10.0 ± 3.0	8.3 ± 2.1	26.7 ± 0.8
Dense			41.7 ± 5.9	36.7 ± 3.1	21.7 ± 7.5	27.4 ± 2.0
Empty			71.3 ± 1.2	7.3 ± 1.2	21.3 ± 1.2	24.0 ± 1.8
Regular	test	test	68.7 ± 4.2	12.7 ± 1.2	18.7 ± 3.1	25.4 ± 3.2
Dense			30.0 ± 4.0	44.7 ± 5.0	25.3 ± 1.2	28.0 ± 2.3

Table 4.2: Results of LBC-R on the NoCrash benchmark. Shows the success rates, collision rates and timeout rates of each benchmark task. Collision and timeout rates constitute the total failure rate. The traffic light violation rate is also shown. All rates are presented as the estimated statistical means and standard deviations, calculated over three benchmark runs with different RNG seeds.

Weather	Success	Collision	Timeout	Lights ran
ClearNoon	86.7 ± 1.2	8.2 ± 1.4	5.3 ± 1.8	18.3 ± 1.9
WetNoon	84.0 ± 3.1	12.2 ± 4.4	3.8 ± 3.2	16.3 ± 0.8
HardRainNoon	79.3 ± 4.2	16.4 ± 3.4	4.2 ± 0.8	17.2 ± 1.5
ClearSunset	61.8 ± 2.5	22.2 ± 2.5	16.0 ± 4.4	20.7 ± 1.4
WetSunset	61.8 ± 2.1	19.1 ± 1.4	19.1 ± 2.7	16.5 ± 1.6
SoftRainSunset	77.6 ± 3.9	15.1 ± 5.0	7.8 ± 1.4	16.5 ± 1.1

Table 4.3: A modified view of the results of LBC-R on the NoCrash benchmark. Shows the success rates, collision rates and timeout rates for each weather type. Collision and timeout rates constitute the total failure rate. The traffic light violation rate is also shown. All rates are presented as the estimated statistical means and standard deviations, calculated over three benchmark runs with different RNG seeds. Weathers *ClearNoon*, *WetNoon*, *HardRainNoon* and *ClearSunset* were seen in training data, while weathers *WetSunset* and *SoftRainSunset* are unseen.

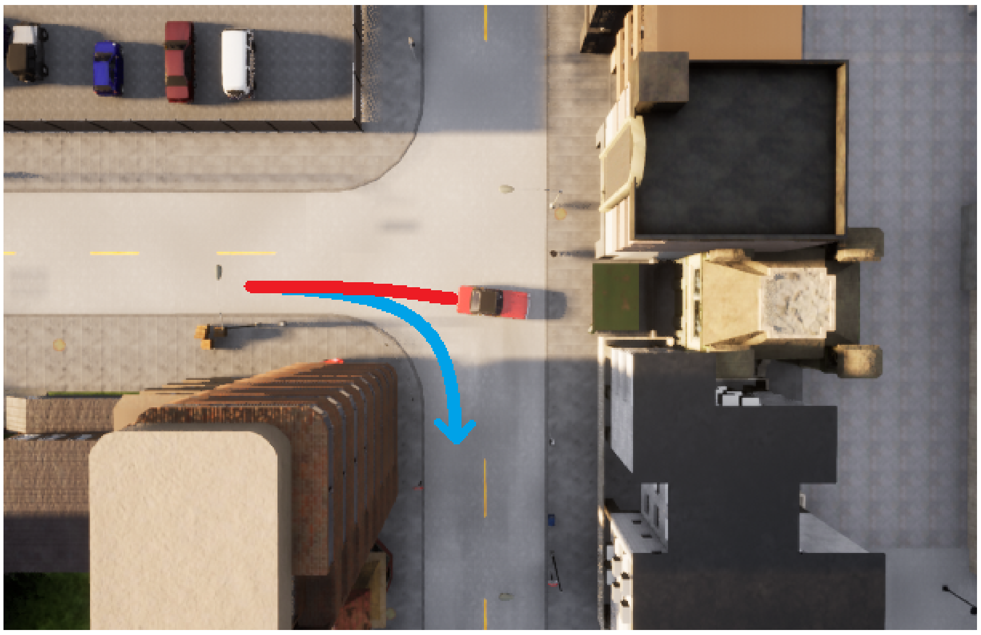


Figure 4.1: A bird's-eye view of a situation where the LBC-R agent is reluctant to drive into a shadow. The intended route is marked with blue and the actual route is marked with red.

4.3 Experiment 2: LBC with Ground Truth Computer Vision

The results of LBC-GTCV on the NoCrash benchmark with comparisons to other approaches is given in Table 4.1. More detailed results of LBC-GTCV can be seen in Table 4.4. Table 4.5 shows the benchmark results with respect to weather types.

Table 4.1 shows that LBC-GTCV outperformed the state-of-the-art approach, *Rails*, on multiple tasks on the NoCrash benchmark. The improvements were the highest for the test weathers. The table also shows that LBC-GTCV outperformed LBC-R on every task. Table 4.4 shows that LBC-GTCV had a significantly lower traffic light violation rate compared to LBC-R. This difference was higher in Town02 than in Town01. Table 4.5 shows that the agent performed similarly well in all weathers.

Several failures occurred when the agent tried to cross an intersection while the traffic light was green, but subsequently got hit by another vehicle that drove when the traffic light was red. There were also several instances when the agent did not behave according to the received HLC. For example, the agent was given a left HLC, but did not turn left. Instead it continued to drive straight, which caused the left HLC to be active until the time limit was reached or a collision occurred.

On the empty task it achieved a perfect score in Town02, for both training and test weather conditions. It did not fail on routes that were partially covered by shadows. For instance, it succeeded in situations like the one presented in Figure 4.1. In Town01 on the empty task, it nearly achieved a perfect score. For the *ClearSunset* and *WetSunset* weather conditions, there was a particular intersection in Town01 where the agent regularly failed. When the agent received a right HLC from the navigational planner, the agent abruptly stopped. The agent stayed there indefinitely until the time limit was reached. This scenario is shown in Figure 4.2.

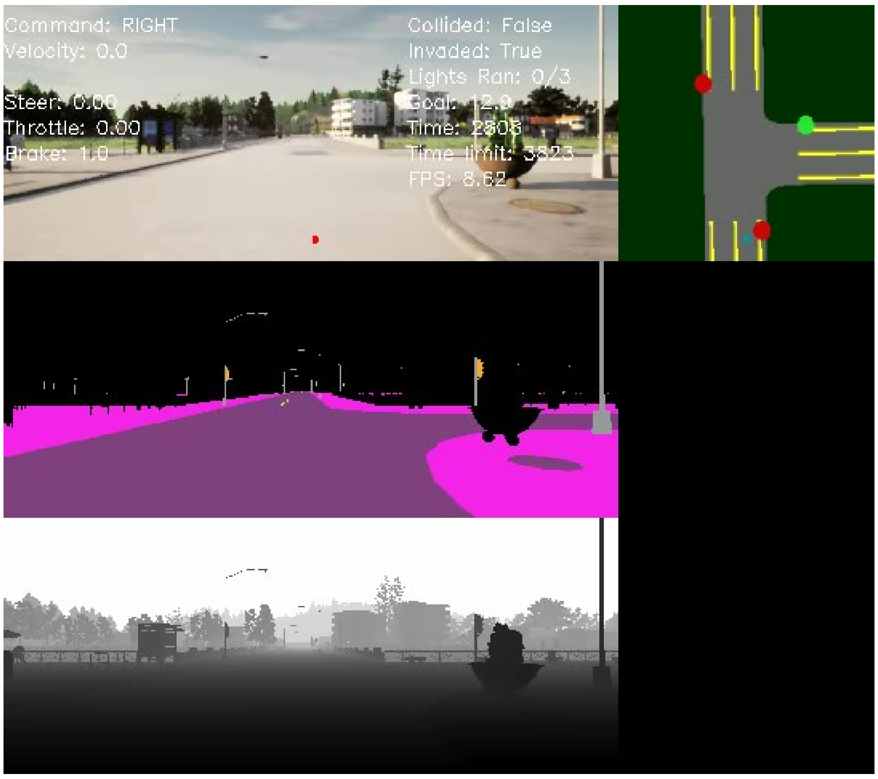
During the dense tasks, several failures occurred due to collisions with other vehicles. Furthermore, the number of timeouts for the dense task was higher than for the empty and regular tasks as well. Timeouts sometimes occurred because the agent did not reach the goal position in time. There were also some examples of the agent getting stuck in traffic scenarios involving multiple pedestrians and vehicles. For instance, there was a particular spot in Town02 where the agent often stopped completely. This spot was located in a turn curving to the right. It managed to follow the road when there were no vehicles in the oncoming lane, otherwise it would stop. This scenario is shown in Figure 4.3.

Task	Town	Weather	Success	Collision	Timeout	Lights ran
Empty			98 ± 1.0	0.0 ± 0.0	2.0 ± 1.0	4.6 ± 0.4
Regular	train	train	98.7 ± 0.6	0.7 ± 0.6	0.7 ± 0.6	3.5 ± 0.8
Dense			85.3 ± 3.8	12.3 ± 4.2	2.3 ± 0.6	12.5 ± 1.4
Empty			97.3 ± 1.2	0.0 ± 0.0	2.7 ± 1.2	5.1 ± 1.1
Regular	train	test	96.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	5.1 ± 1.6
Dense			89.3 ± 1.2	8.7 ± 1.2	2.0 ± 0.0	14.2 ± 2.3
Empty			100.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	7.0 ± 0.7
Regular	test	train	96.3 ± 1.5	2.7 ± 0.6	1.0 ± 1.0	9.2 ± 1.8
Dense			59.7 ± 3.8	16.7 ± 1.5	$23.7\pm, 4.5$	16.0 ± 1.0
Empty			100.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	6.7 ± 0.9
Regular	test	test	96.0 ± 0.0	2.7 ± 1.2	1.3 ± 1.2	11.3 ± 1.6
Dense			58.7 ± 3.1	19.3 ± 7.6	22.0 ± 8.0	18.5 ± 4.0

Table 4.4: Results of LBC-GTCV on the NoCrash benchmark. Shows the success rates, collision rates and timeout rates of each benchmark task. Collision and timeout rates constitute the total failure rate. The traffic light violation rate is also shown. All rates are presented as the estimated statistical means and standard deviations, calculated over three benchmark runs with different RNG seeds.

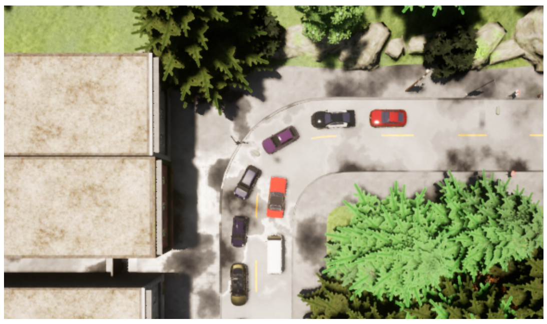
Weather	Success	Collision	Timeout	Lights ran
ClearNoon	89 ± 1.0	4.2 ± 1.5	6.0 ±2.0	8.3 ± 0.7
WetNoon	91.6 ± 2.1	4.2 ± 1.4	4.2 ± 1.0	7.6 ± 1.4
HardRainNoon	88.9 ± 1.0	7.3 ± 1.8	3.8 ± 1.0	8.3 ± 1.0
ClearSunset	88.4 ± 4.4	5.8 ± 4.3	5.8 ± 2.8	9.9 ± 0.5
WetSunset	90.2 ± 3.4	6.0 ± 2.3	3.8 ± 2.0	11.7 ± 1.9
SoftRainSunset	88.9 ± 2.3	4.9 ± 2.5	6.2 ± 0.8	8.1 ± 0.8

Table 4.5: A modified view of the results of LBC-GTCV on the NoCrash benchmark. Shows the success rates, collision rates and timeout rates for each weather type. Collision and timeout rates constitute the total failure rate. The traffic light violation rate is also shown. All rates are presented as the estimated statistical means and standard deviations, calculated over three benchmark runs with different RNG seeds. Weathers *ClearNoon*, *WetNoon*, *HardRainNoon* and *ClearSunset* were seen in training data, while weathers *WetSunset* and *SoftRainSunset* are unseen.

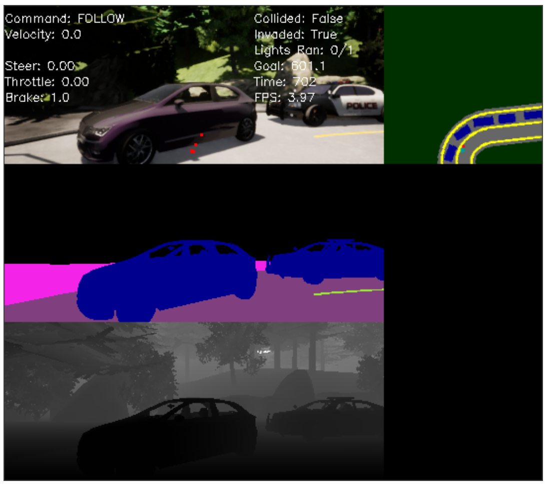


```
Command: RIGHT              Collided: False
Velocity: 0.0               Invaded: True
                            Lights Ran: 0/3
Steer: 0.00                 Goal: 12.9
Throttle: 0.00              Time: 2806
Brake: 1.0                  Time limit: 3823
                            FPS: 8.62
```

Figure 4.2: LBC-GTCV regularly stopped completely when it received a right HLC in this intersection during the empty task of the NoCrash benchmark in Town01.



(a) Overview



```
Command: FOLLOW             Collided: False
Velocity: 0.0               Invaded: True
                            Lights Ran: 0/1
Steer: 0.00                 Goal: 601.1
Throttle: 0.00              Time: 702
Brake: 1.0                  FPS: 3.97
```

(b) Model view

Figure 4.3: A spot where LBC-GTCV could get stuck in Town02 during the dense task of the NoCrash benchmark. This occured when there were vehicles on the opposite side of the road. The agent usually waited until all vehicles had passed before continuing along the route. This could result in a timeout failure.

4.4 Experiment 3: LBC with Trained Perception Models

This experiment consists of two parts. First, the results of the perception model evaluation is given. This section is divided into semantic segmentation results and monocular depth estimation results. Then follows the benchmark results of the combined sensorimotor network.

4.4.1 Perception Model Evaluation and Selection

The tables in this section are divided on the $Test1$ and $Test2$ datasets, which are described in Section 3.5.

Semantic Segmentation

The evaluation of the semantic segmentation models on the Test1 set is shown in Table 4.6. The results of the evaluation on the Test2 set is shown in Table 4.7. These tables show that the U-Net models have the highest scores on both the Mean IoU and Weighted IoU metrics. It can also be seen that the weighted U-Net models learned the traffic light class to a sufficient degree, while the unweighted U-Net did not.

The model speed comparison is shown in Table 4.8. This table shows that only the *Deeplabv3 w/ MobileNet* model were faster than the U-Net models on prediction speed.

The *U-Net w/ResNet50 tl-5* model was therefore selected to perform semantic segmentation for the sensorimotor network. This model was chosen over the *U-Net w/ResNet50 tl-2.5* model because it had higher scores on the Mean IoU metric.

Model	Mean IoU	Weighted IoU	Traffic Light IoU
FCN w/ ResNet101	0.5468	0.9277	<u>0.5732</u>
DeepLabv3 w/ ResNet50	0.5526	0.9296	0.5511
DeepLabv3 w/ ResNet101	0.5497	0.9314	0.5373
DeepLabv3 w/ MobileNet	0.4561	0.9066	0.4562
U-Net w/ ResNet50	<u>0.6072</u>	**0.9421**	0.0
U-Net w/ ResNet50 tl-2.5	0.5963	0.9404	**0.6003**
U-Net w/ ResNet50 tl-5	**0.6359**	<u>0.9420</u>	0.5575

Table 4.6: Evaluation results of the semantic segmentation models on the Test1 set. Test1 consists of images from Town02 with training weathers. All models are fine tuned or trained on Town01 images with training weathers. The FCN and DeepLabv3 models are pretrained on COCO train 2017, while the U-Net backbones are pretrained on ImageNet. For the U-Net models, *tl-2.5* and *tl-5* signifies that the traffic loss weight has been weighted 2.5 or 5 times higher, respectively.

Model	Mean IoU	Weighted IoU	Traffic Light IoU
FCN w/ ResNet101	0.5340	0.9221	<u>0.5674</u>
DeepLabv3 w/ ResNet50	0.5368	0.9224	0.5529
DeepLabv3 w/ ResNet101	0.5330	0.9237	0.5387
DeepLabv3 w/ MobileNet	0.4419	0.9003	0.4580
U-Net w/ ResNet50	<u>0.5842</u>	<u>0.9320</u>	0.0
U-Net w/ ResNet50 tl-2.5	0.5769	0.9305	**0.5734**
U-Net w/ ResNet50 tl-5	**0.6141**	**0.9323**	0.5255

Table 4.7: Evaluation results of the semantic segmentation models on the Test2 set. Test2 consists of images from Town02 with test weathers. All models are fine tuned or trained on Town01 images with training weathers. The FCN and DeepLabv3 models are pretrained on COCO train 2017, while the U-Net backbones are pretrained on ImageNet. For the U-Net models, *tl-2.5* and *tl-5* signifies that the traffic loss weight has been weighted 2.5 or 5 times higher, respectively.

Model	Time per prediction [s]	FPS [Hz]
FCN w/ ResNet101	0.0485	21
DeepLabv3 w/ ResNet50	0.0400	25
DeepLabv3 w/ ResNet101	0.0617	16
DeepLabv3 w/ MobileNet	**0.0106**	**94**
U-Net w/ ResNet50	<u>0.0132</u>	<u>76</u>

Table 4.8: Results of the speed evaluation of the semantic segmentation models. Frames per second is denoted by FPS.

Monocular Depth Estimation

The evaluation of the depth estimation models on the Test1 set is shown in Table 4.9. The results of the evaluation on the Test2 set is shown in Table 4.10. The tables show that the U-Net models significantly outperformed the MiDaS models on all metrics. The difference between the two U-Net models on these metrics was insignificant.

The model speed comparison is shown in Table 4.11. This table shows that the *U-Net w/ ResNet34* model was approximately twice as fast as the vanilla *U-Net* model. The *U-Net w/ ResNet34* model was therefore selected to perform depth estimation for the sensorimotor network.

Model	σ_1	σ_2	σ_3	RMSE
MiDaS-small	0.2160	0.3394	0.5426	0.3030
MiDaS-large	0.2083	0.3432	0.5777	0.2822
U-Net	**0.9115**	<u>0.9687</u>	<u>0.9846</u>	**0.0536**
U-Net w/ ResNet34	<u>0.9101</u>	**0.9689**	**0.9850**	<u>0.0559</u>

Table 4.9: Evaluation results of the monocular depth estimation models on the Test1 set. Test1 consists of images from Town02 with training weathers. The accuracy within threshold metric is calculated with $\sigma_1 = 1.25^1$, $\sigma_2 = 1.25^2$ and $\sigma_3 = 1.25^3$.

Model	σ_1	σ_2	σ_3	RMSE
MiDaS-small	0.2090	0.3179	0.5156	0.3097
MiDaS-large	0.2039	0.3216	0.5440	0.2942
U-Net	**0.9077**	**0.9671**	**0.9835**	**0.0583**
U-Net w/ ResNet34	<u>0.9046</u>	<u>0.9665</u>	**0.9835**	<u>0.0610</u>

Table 4.10: Evaluation results of the monocular depth estimation models on the Test2 set. Test2 consists of images from Town02 with test weathers. The accuracy within threshold metric is calculated with $\sigma_1 = 1.25^1$, $\sigma_2 = 1.25^2$ and $\sigma_3 = 1.25^3$.

Model	Time per prediction [s]	FPS [Hz]
MiDaS-small	<u>0.0108</u>	<u>93</u>
MiDaS-large	0.0487	21
U-Net	0.0206	49
U-Net w/ ResNet34	**0.0093**	**108**

Table 4.11: Results of the speed evaluation of the depth estimation models. Frames per second is denoted by FPS.

4.4.2 Results of LBC-TCV

The results of LBC-TCV on the NoCrash benchmark with comparisons to other approaches is given in Table 4.1. More detailed results of LBC-TCV can be seen in Table 4.12. Table 4.13 shows the benchmark results with respect to weather types.

LBC-TCV beat the state-of-the-art approach, *Rails*, on the empty and regular tasks in the Town02 with training weathers. It generally performed worse than LBC-R in Town01, with exception of the dense tasks. However, it performed better than LBC-R in Town02. It also performed worse than LBC-GTCV on every task. LBC-TCV had fewer traffic light violations than LBC-R. The rates were slightly lower for Town01, and significantly lower for Town02. It had much worse results in the test weather *WetSunset* compared to the other weather conditions.

The agent failed in a variety of scenarios, many of which were similar to LBC-R and LBC-GTCV. Like the other agents, LBC-TCV sometimes failed to continue driving after having stopped at a red traffic light. This happened even though the traffic light box was clearly identified in the semantic segmentation images.

Another issue was that it tried to drive around the car ahead. This could lead to dangerous behavior where it almost entered the oncoming lane. This behavior happened more frequently when it drove behind a small car, e.g., the BMW Isetta.

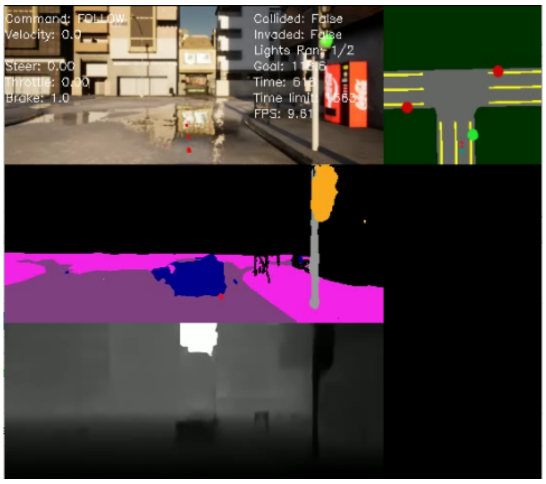
The semantic segmentation and depth estimation models would sometimes find objects which were not present. For example, they could perceive non-existing vehicles or humans on the road. This issue occurred more often in the wet environments. The models struggled with puddles, which have a reflective and unique surface. This could cause the agent to pause and wait indefinitely for the objects to move. This is also reflected by the success and timeout rates of the *WetSunset* weather in Table 4.13. Two examples of such incidents can be seen in Figure 4.4.

Task	Town	Weather	Success	Collision	Timeout	Lights ran
Empty			85.0 ± 3.0	0.0 ± 0.0	15.0 ± 3.0	4.9 ± 0.2
Regular	train	train	92.0 ± 2.6	3.3 ± 1.5	4.7 ± 1.5	6.9 ± 1.7
Dense			75.7 ± 0.6	18.0 ± 5.0	6.3 ± 4.5	15.5 ± 0.6
Empty			70.0 ± 2.0	0.0 ± 0.0	30.0 ± 2.0	5.7 ± 1.9
Regular	train	test	81.3 ± 2.3	0.7 ± 1.2	18.0 ± 2.0	7.6 ± 0.4
Dense			79.3 ± 4.2	10.7 ± 4.2	10.0 ± 0.0	18.2 ± 3.0
Empty			99.3 ± 0.6	0.7 ± 0.6	0.0 ± 0.0	12.7 ± 1.0
Regular	test	train	95.7 ± 0.6	3.3 ± 0.6	1.0 ± 1.0	16.0 ± 2.2
Dense			52.3 ± 4.0	23.7 ± 2.1	24.0 ± 2.6	17.6 ± 3.1
Empty			74.7 ± 1.2	0.0 ± 0.0	25.3 ± 1.2	16.6 ± 1.6
Regular	test	test	71.3 ± 3.1	4.7 ± 1.2	24.0 ± 2.0	12.9 ± 1.6
Dense			38.0 ± 3.5	18.0 ± 3.5	44.0 ± 3.5	19.3 ± 2.2

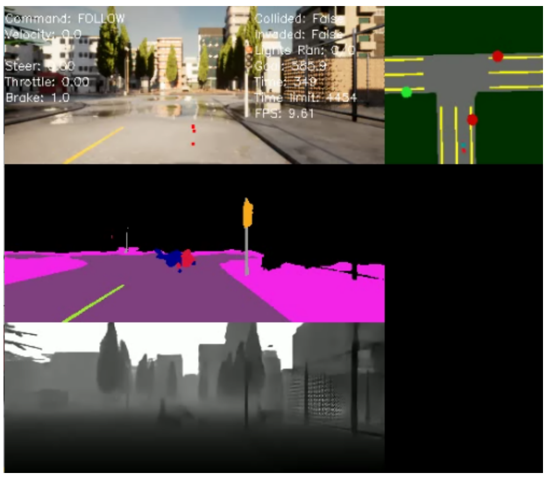
Table 4.12: Results of LBC-TCV on the NoCrash benchmark. Shows the success rates, collision rates and timeout rates of each benchmark task. Collision and timeout rates constitute the total failure rate. The traffic light violation rate is also shown. All rates are presented as the estimated statistical means and standard deviations, calculated over three benchmark runs with different RNG seeds.

Weather	Success	Collision	Timeout	Lights ran
ClearNoon	83.1 ± 2.7	8.2 ± 1.9	8.7 ± 1.2	12.0 ± 1.0
WetNoon	79.8 ± 1.7	8.2 ± 2.0	12.0 ± 1.8	12.0 ± 2.8
HardRainNoon	88.9 ± 1.7	7.1 ± 0.8	4.0 ± 1.2	10.9 ± 1.1
ClearSunset	81.6 ± 1.9	9.1 ± 0.4	9.3 ± 1.8	13.2 ± 1.4
WetSunset	55.3 ± 2.4	5.1 ± 0.8	39.6 ± 2.3	14.5 ± 0.9
SoftRainSunset	82.9 ± 1.7	6.2 ± 1.7	10.9 ± 0.4	11.8 ± 1.5

Table 4.13: A modified view of the results of LBC-TCV on the NoCrash benchmark. Shows the success rates, collision rates and timeout rates for each weather type. Collision and timeout rates constitute the total failure rate. The traffic light violation rate is also shown. All rates are presented as the estimated statistical means and standard deviations, calculated over three benchmark runs with different RNG seeds. Weathers *ClearNoon*, *WetNoon*, *HardRainNoon* and *ClearSunset* were seen in training data, while weathers *WetSunset* and *SoftRainSunset* are unseen.



(a)



(b)

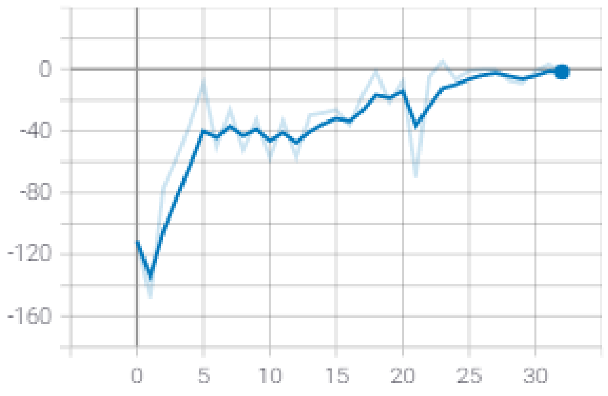
Figure 4.4: Example of how the semantic segmentation and depth models of LBC-TCV could perceive non-existing objects in the puddles of wet environments. This caused the agent to wait indefinitely. (a) LBC-TCV perceives parts of the puddle as a vehicle. (b) LBC-TCV perceives both the vehicle and human class in the puddle.

4.5 Experiment 4: LBC with Reinforcement Learning

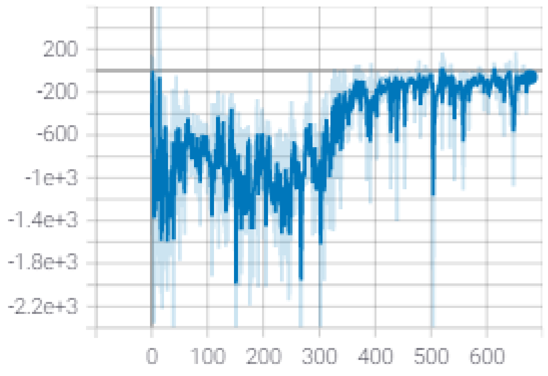
None of the agents trained in the RL experiment managed to improve from the initial pretrained weights. The performance degraded in all cases.

The results from a single run of PPO is shown here. This run consisted of 250 000 time steps. During this run, all waypoints were sampled individually with the same standard deviation value. This run consisted of 32 episodes before it was terminated. Each episode consisted of 20 rollouts. After enough episodes, the agent would slow down and end up stopping in all rollouts.

Figure 4.5a shows the average accumulated sum of rewards for each episode. Figure 4.5a shows the accumulated sum of rewards for each rollout. These graphs shows that the rewards increased over time. Figure 4.6 shows the loss of the critic for each epoch of training. The graph shows that the loss decreases as training progresses.



(a)



(b)

Figure 4.5: Return of rewards for a single run of PPO. In both graphs the darker line is smoothed. (a) Graph showing the average accumulated sum of rewards from each episode. Each episode consisted of 20 rollouts. (b) Graph showing the accumulated sum of rewards for every single rollout.

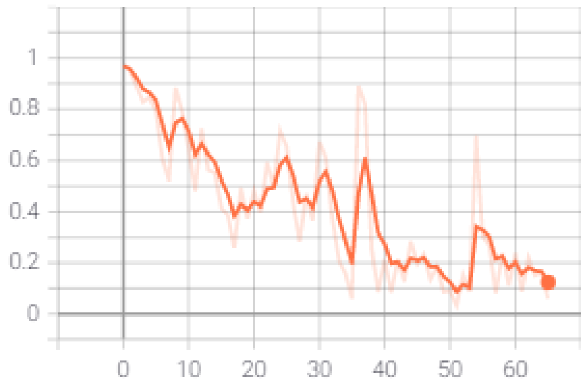


Figure 4.6: The average loss of the critic as training progressed. The darker line is smoothed.

Chapter 5

Discussion

This chapter discusses the results and the findings of the experiments. The experiments are discussed sequentially. In Section 5.5, some of the shortcomings in the work are discussed.

5.1 Experiment 1: LBC Reproduction

The privileged network never achieved validation loss below 0.005. A well-trained privileged network should be expected to go below this loss, according to the LBC GitHub README. This might have been caused by simulator differences. It might also have been caused by the issue the privileged agent had with the yellow traffic lights after entering an intersection. If there were situations like this in the dataset, and the network failed to learn the correct behavior, the network would get a higher loss.

The benchmarking of the privileged agent indicated that it was well-behaving in general. With the exception of the empty routes, the reproduced privileged agent got slightly better results than the privileged agent in the original LBC approach. We therefore believe that the privileged network did not negatively affect the training of LBC-R, LBC-GTCV, and LBC-TCV. The privileged network was used to train all sensorimotor networks in Experiment 1, 2 and 3.

The LBC-R results on the NoCrash benchmark was better than what LBC† achieved, which was trained in CARLA version 0.9.10. This implies that some deviation in the reproduction approach from the original approach made a difference. This could be caused by two changes. The first change is the correction of the PyTorch and NumPy random number generation, which had caused identical batches and batch augmentations during training. The second change is the PID controller tuning, whether a better set of PID parameters was found.

The fact that the PID controller parameters have an effect on model performance is a downside with the LBC approach. Finding a set of PID parameters that work well can cover up the fact that the model did not learn to output waypoints well enough. The manual search for good PID parameters was time-consuming. PID tuning also deviates from the main idea of end-to-end learning, where the model itself learns how to handle every aspect of the task. It is difficult to tell if there exists a set of PID parameters which give better performance of the model. It could be beneficial to implement an automatic PID tuner.

Many routes in the NoCrash benchmark failed because of unfortunate environment circumstances. Because of too high traffic, the agent could fail to reach the goal position in time, even with flawless driving. It is unfortunate that routes are failed purely due to unlucky spawn positions of the other vehicles. Another aspect is the unpredictability of other vehicles. They can ignore traffic lights, which leads to complex situations to handle for the agent. However, this can allow a good agent to prove itself. To make the randomness have less influence over the results, the benchmark needs to be run several times, which is time-consuming.

LBC-R performed similarly to the original LBC, with the exception of slightly worse performance in Town02 with training weathers. These numbers are not from the same CARLA versions however. Compared to LBC†, which was benchmarked in CARLA version 0.9.10, LBC-R had higher performance. It can therefore safely be said that LBC can be reproduced in the newest version of CARLA, which answers the first research question.

5.2 Experiment 2: LBC with Ground Truth Computer Vision

The results show that LBC-GTCV performed much better than LBC-R on the NoCrash benchmark. This clearly shows that the performance of LBC can be improved by providing it with ground truth semantic segmentation and depth images, which answers the second research question. This result proves that the intermediate representations can help improve performance in the LBC approach, which was the foundation for Experiment 3.

The model beat the state-of-the-art results on multiple tasks. These improvements were the highest on the test weathers. This is logical, because the agent had learned to drive using the semantic segmentation and depth images. Because these images were perfect in all conditions, the weather obfuscation on the RGB images had very little effect. Consequently, the agent did not experience the shadow problem that LBC-R experienced.

Because the agent gets the semantic segmentation and depth images directly from the simulator, it does not compare fairly to other approaches. However, achieving such high performance confirms that LBC is a very capable learning procedure. It also confirmed the results of the paper by Zhou et al. [12].

The agent had much fewer traffic light violations compared to LBC-R, especially in the test town. The traffic light state is not included in the semantic segmentation image, but the traffic light box is. This showed that the agent had learned to look at the corresponding location in the RGB image in these situations, and that including the traffic light class was a successful measure.

The PID controller parameters for this agent did not have to be tuned. As discussed in Section 5.1, this might mean that the model had learned its task better.

5.3 Experiment 3: LBC with Trained Perception Models

LBC-TCV performed better than LBC-R on all tasks in the test town, and on the dense tasks in the training town. However, it performed worse than LBC-R on the empty and regular tasks in the training town. This confirms that it is possible to improve LBC by using semantic segmentation and depth estimation images from trained perception models, which is the answer to the third research question.

The perception models had never seen the test environment during training, but LBC-TCV still performed better than LBC-R in the test environment. It can therefore be said that the model generalized better. This finding is in line with the research by Zhou et al. [12]. LBC-TCV had a lower traffic light violation rate compared to LBC-R in almost all tasks, especially in the test town. This also confirms that LBC-TCV generalizes better. It also confirms that having a traffic light box class was a successful measure, as was found in Experiment 2.

It also beat the state-of-the-art results on the empty and regular tasks in the test town with training weathers. Because the model was only trained in the training environment, these results can be regarded as fair and legitimate. However, the model backbones were pretrained on real life datasets, but this is the case for many of the state-of-the-art models as well.

A valid critique of the approach would be that the perception models were trained on perfect images supplied by the simulator. This would not be possible in real life. But with high quality datasets available, such as Mapillary [44] and Cityscapes [43], training well-performing perception models for real environments is possible as well [50]. And similar depth image datasets can be created using cars with stereo cameras. This approach is therefore just as relevant for the physical world as the original LBC approach is. The sensorimotor networks are not required to be run in a simulator.

As in Experiment 2, no PID tuning was performed for this model. As for LBC-R and LBC-GTCV, there is a possibility that finding a better set of PID parameters could have improved the the results of some benchmark tasks. However, it did not seem necessary during initial testing. LBC-R required extensive PID tuning, while LBC-GTCV and LBC-TCV did not require any tuning. This might mean that agents equipped with intermediate representations are better at mimicking the privileged agent.

The performance of LBC-GTCV can be seen as a upper limit of how good LBC-TCV can become. It is likely that LBC-TCV could have performed better if the quality of the intermediate representations were better. To get better perception models, we could have trained the models on more varied datasets or found even better architectures. Unfortunately, the state-of-the-art architectures were found to be hard to use for custom datasets. We also trained the perception models with the medium augmentation setting. It could have been beneficial to train with harder augmentations.

After training the first iteration of LBC-TCV, it was found that the intermediate representations were of low quality. This agent performed poorly, and it was decided that the perception models had to be retrained. The second iteration had better results. In contrast to LBC-GTCV, the semantic segmentation images were not perfect, so the agent could not solely rely on the intermediate representations. We believe that the agent relies on the RGB images if the quality of the perception models are too low. Therefore a driving network pretrained on ImageNet was also used for the second iteration. This could also have affected the performance.

The visualizations of the intermediate representations gave better interpretability. For some of the failures of the agent, the intermediate representations perceived objects which were not present, and the agent handled accordingly as if the objects were present. By looking at the live view, it was clear why the agent had acted the way it did. Figure 4.4a shows an example of this situation.

5.4 Experiment 4: LBC with Reinforcement Learning

In Section 3.6.4 two methods for sampling waypoints were proposed. The first method used the same standard deviation for all waypoints. In the second method, every subsequent waypoint was sampled with a higher standard deviation than the preceding waypoint.

The waypoints are used to create a circular arc that indicate the trajectory of the vehicle. When all waypoints are sampled individually, they do not necessarily form a curve. This resulted in chaotic exploration. The graphs shown in Figure 4.5a and Figure 4.5b shows a run of PPO with this method of sampling. As can been seen in these graphs, the agent managed to increase the rewards over time. This is because it learned that the most optimal action was to output waypoints that resulted in zero velocity. After enough episodes, the agent would eventually stop driving during all rollouts. The agent would never continue driving, because the sampling technique would not sample actions that allowed the agent to increase its velocity. This means that this form of waypoint sampling is chaotic, inefficient, and results in a degradation of the performance of the actor.

The second method of sampling waypoints gave more natural trajectories during exploration. However, this sampling technique caused the PPO objective function to often be close to zero. In the PPO implementation, the log-likelihood of $\pi_\theta(a_t|s_t)$ is required to compute the following ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

However, the denominator in the log-likelihood function (Equation 2.4) gets small with this sampling method, which results in the ratio $r_t(\theta)$ being close to zero. Therefore, this sampling technique was not satisfactory either.

In Section 2.4 several advantages of using waypoints as an output modality were described. Waypoints have been used as the output modality in IL, but not in RL approaches. The difficulties that we experienced with waypoint sampling puts into question whether this form of output modality is suitable for RL algorithms that use stochastic policies. There may exist a more inventive way of sampling waypoints that results in better exploration of the environment.

Figure 4.6 shows a graph of the loss of the critic. Recall that the critic evalutes states based on semantically segmented BEV images. Since the loss decreases as training progresses, it might indicate that the critic managed to learn an approximation of the value function. However, as the actor policy changes during training, this will result in a different distribution of trajectories being collected in the next episode. Therefore, observing the loss of the critic does not necessarily indicate that it has learned a good approximation. The fact that the actor learned to stop driving, might indicate that the critic managed to output sensible state values. But, it is impossible to draw any concrete conclusions on this matter.

The reward signal was probably too unbalanced, which made it difficult for the critic to approximate the value function. Additionally, the BEV images should have contained information about static objects, such as poles and benches. This is because the actor could collide into static objects, but the critic would not be able to see these objects in the BEV image. Compared to other approaches [16, 52], the actor has significantly more tunable parameters. This might make learning more difficult. It was also believed that using pretrained weights would put the actor policy at a reasonable starting point. Because of the difficulties with sampling waypoints, it did not matter that the initial policy was better than a randomly initialized policy. Furthermore, the PPO training should have continued for many more timesteps. The combination of all these factors probably contributed to the poor results.

We were not able to improve the performance of the sensorimotor network with an additional RL stage, which answers the fourth research question. However, we can not conclude that this is not possible. Perhaps with more computational resources, more hyperparameter tuning, or a different RL algorithm, the sensorimotor network could be further improved.

5.5 Shortcomings of the Thesis

This section reflects over the shortcomings of our research.

Problems Reproducing LBC
During the initial planning phase of the project, the first experiment was estimated to be the easiest and quickest to conduct. This is because it would consist of running scripts that had already been written by the authors of LBC. Updating the LBC code for CARLA 0.9.11 and fixing other issues required more work than anticipated.

These updates are described in Section 3.2. It included correcting the spawn points for the NoCrash benchmark in CARLA 0.9.11, fixing the random number generator issue that caused duplicate batch augmentations, fixing the multiprocessing issue preventing training on the Windows PCs, and modifying the replay buffer for DAgger.

Almost every time a new error was encountered, some part of the pipeline had to be redone. For instance, the first time the *driving dataset* was collected, the spawn points for CARLA 0.9.6 was used. Using the incorrect spawn points might have resulted in a dataset with an undesirable distribution of instances. To ensure that the reproduction of LBC went as intended, the data collection procedure had to be redone. Every time data collection was performed, it took at least four days of continuous computation to complete. With every fixed error, the three phases of training the sensorimotor network had to be conducted again. With the original replay buffer, phase 2 required approximately seven real days to complete. Every time a sensorimotor network had finished training, it always performed worse than expected in testing. It was difficult to understand if this was due to simulator differences, an overlooked error, or if the PID parameters had to be tuned further.

Experiment 1 ended up taking much longer than anticipated, and occupied most of the available computational resources for a significant amount of time. While the computational resources were occupied, we prepared for the next experiments as well as we could. After modifying the replay buffer for DAgger, this reduced phase 2 to under 14 hours. With all errors corrected, Experiment 2 and 3 went relatively straightforward in comparison.

Trained Perception Problems
The first time perception data was collected, the sensors were mounted 3 meters above ground on all vehicles in the simulation. This was in contrast to the camera placement of the LBC approach, which was 1.4 meters above ground. When the camera was placed at

this lower height, the camera would sometimes be placed inside the body of the vehicles. It was therefore decided to place the camera 3 meters above ground. This allowed for spawning sensors on all vehicles, which sped up the data collection process.

It was hypothesized that the height difference would not cause any major negative effects. When the perception models were trained with this dataset, it ended up being detrimental for the performance of the sensorimotor network. This meant that new perception data had to be collected. For next iteration of data collection, vehicles were specifically chosen to ensure that sensors could be safely mounted at 1.4 meters. To ensure that the data was varied, sensor data was retrieved every 40 seconds. The data collection procedure now took much longer to complete, but it ensured varied data with the correct height position.

RL Experiment
Due to the time delays for Experiment 1, 2 and 3, the RL experiment was postponed significantly. This reduced the time for experimenting with different network architectures, and hyperparameter tuning. Tuning hyperparameters in RL is a long and integral part of the process. These delays also left us with less time to figure out an efficient technique for sampling waypoints.

Functionality for running several actor threads in parallel was not implemented. This was because it was considered to be a difficult and time-consuming task. Implementing this functionality would have sped up the training process. However, it would require significantly better hardware.

Chapter 6

Conclusion and Future Work

This chapter is divided into two sections. Section 6.1 summarizes the experiments that have been conducted in this thesis, and presents the significant findings. Section 6.2 discusses ideas for further work.

6.1 Conclusion

The research goal of this thesis was to improve the performance of neural networks trained with the Learning by Cheating (LBC) [14] approach. Through four experiments, this goal was achieved. By using semantic segmentation and depth images, the performance of the neural networks improved.

In the first experiment, the LBC approach was reproduced in the newest version of CARLA, namely version 0.9.11. Through a more recent paper [18], the authors of LBC benchmarked LBC in CARLA 0.9.10, which is similar to CARLA 0.9.11. The same paper states that the newer CARLA versions are harder environments than CARLA 0.9.6. The reproduction in this thesis significantly outperformed the 0.9.10 reproduction, and matched the results from version 0.9.6.

In the second experiment, the agent was provided with perfect semantic segmentation and depth images, directly from the simulator. The results clearly show that when the agent was trained with these representations, it generalized better to new environments and achieved higher task performance. This confirms the results by Zhou et al [12]. The agent also had lower collision and traffic light violation rates than the RGB agent.

In the third experiment, perception models that perform semantic segmentation and monocular depth estimation were trained and evaluated. An agent was trained with the intermediate representations created by these perception models. This agent performed better in the test environments than the reproduced agent, but slightly worse in the training environments. It beat the state-of-the-art approach on the empty and regular tasks in the test town with training weathers in the NoCrash [11] benchmark. However, it performed worse on every task compared to the agent trained with perfect representations. The intermediate representations made the model more interpretable.

In the final experiment, it was attempted to further improve agent performance by using reinforcement learning. However, due to time constraints and difficulties regarding the output modality, the experiment was unsuccessful.

6.2 Future Work

The results of this work were promising, but it is likely that they can be improved even further. The performance of LBC-GTCV was significantly better than LBC-TCV, which leads us to believe that there is more to gain by using intermediate representations. Higher quality semantic segmentation and depth estimation images could be attainable by exploring other perception model architectures and improving the training datasets.

There is also the possibility of using additional intermediate representations, such as optical flow and albedo. These were explored in the work of Zhou et al. [12], but were found to be of less importance than semantic segmentation and depth images.

There are many improvements to be made on the reinforcement learning phase in Experiment 4. We are not aware of other works using RL with waypoints, and we failed to get promising results ourselves. A natural first step is therefore to establish if RL with waypoints is possible, either mathematically or experimentally. Then follows the search for working hyperparameters. As training is slow, it would be advantageous to implement more than one actor during PPO rollouts. Using better hardware would also be beneficial.

If the RL phase looks promising, it should use the best weights from LBC phase 2 as the initial policy. It might also be worth exploring whether the RL phase can improve agents with perception models.

The agents could also be submitted to the *CARLA Autonomous Driving Leaderboard*[1], which is a newer benchmark than the NoCrash benchmark. This benchmark was made to ensure that the evaluations are fair and reproducible. The website contains an updated table over submitted models, and contains information over evaluation metrics, which sensors were used, and other details about each submitted model.

As with most research regarding autonomous vehicles, it would be interesting to evaluate the model in real-life situations. The sensorimotor network is not tied to simulation, and should in principle function in the physical world. Due to real-life datasets such as Mapillary [44] and Cityscapes [43], we know that it is possible to create well-performing perception models for urban driving. Transfer of the complete model is also made possible by sim-to-real methods [71, 72, 73].

Our code is available at `https://github.com/jostl/masters-thesis`.

[1]`https://leaderboard.carla.org/`

Bibliography

[1] United Nations. Statistics of road traffic accidents in europe and north america, volume lv. page 8, 2019. `https://unece.org/DAM/trans/main/wp6/publications/2020_INLAND_TRANSPORT_STATISTICS.pdf`.

[2] WHO. Road traffic injuries, Feb, 2020. `https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries`.

[3] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. 2020. `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9046805&tag=1`.

[4] Zeyu Zhu and Huijing Zhao. A survey of deep rl and il for autonomous driving policy learning. 2021. `https://arxiv.org/pdf/2101.01993.pdf`.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. `https://arxiv.org/abs/1312.5602`.

[6] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019. `https://arxiv.org/abs/1912.06680`.

[7] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. `https://arxiv.org/abs/1712.01815`.

[8] Ardi Tampuu, Tambet Matiisen, Maksym Semikin, Dmytro Fishman, and Naveed Muhammad. A survey of end-to-end driving: Architectures and training methods. *IEEE Transactions on Neural Networks and Learning Systems*, page 1–21, 2020. `https://arxiv.org/pdf/2003.06404.pdf`.

[9] Yi Xiao, Felipe Codevilla, Akhil Gurram, Onay Urfalioglu, and Antonio M. Lopez. Multimodal end-to-end autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, page 1–11, 2020. `http://dx.doi.org/10.1109/TITS.2020.3013234`.

[10] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. Nov 2017. https://arxiv.org/abs/1711.03938.

[11] Felipe Codevilla, Eder Santana, Antonio M. Lopez, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. 2019. https://arxiv.org/abs/1904.08980.

[12] Brady Zhou, Philipp Krähenbühl, and Vladlen Koltun. Does computer vision matter for action? May 2019. https://arxiv.org/pdf/1905.12887v2.pdf.

[13] Éloi Zablocki, Hédi Ben-Younes, Patrick Pérez, and Matthieu Cord. Explainability of vision-based autonomous driving systems: Review and challenges, 2021. https://arxiv.org/abs/2101.05307.

[14] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. 2019. https://arxiv.org/abs/1912.12294.

[15] Xiadon Liang, Tairui Wang, Luona Yang, and Eric Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving. 2018. https://arxiv.org/abs/1807.03776.

[16] Marin Toromanoff, Emilie Wirbel, and Fabien Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances. 2020. https://arxiv.org/abs/1911.10868.

[17] Tanmay Agarwal, Hitesh Arora, and Jeff Schneider. Affordance-based reinforcement learning for urban driving. 2021. https://arxiv.org/abs/2101.05970.

[18] Dian Chen, Vladlen Koltun, and Philipp Krähenbühl. Learning to drive from a world on rails, 2021. https://arxiv.org/pdf/2105.00636.pdf.

[19] Stuart Russel and Peter Norvig. *Artificial Intelligience: A Modern Approach, Third Edition.* Pearson Education Limited, 2016.

[20] Ahmed Hussein andd Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. Apr 2017. https://dl.acm.org/doi/10.1145/3054912.

[21] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. 2011. https://www.ri.cmu.edu/pub_files/2011/4/Ross-AISTATS11-NoRegret.pdf.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction, second edition.* The MIT Press, 2018.

[23] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018. https://spinningup.openai.com/en/latest/.

[24] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018. https://arxiv.org/abs/1506.02438.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[26] Michael Nielsen. *Neural Networks and Deep Learning.* Determiniation Press, 2015. `http://neuralnetworksanddeeplearning.com/`.

[27] Aurèlien Gèron. *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow.* O'Reilly Media, Inc, 2019.

[28] Joel Janai, Fatma Güney, Aseem Behl, and Andreas Geiger. Computer vision for autonomous vehicles: Problems, datasets and state of the art, 2019. `https://arxiv.org/abs/1704.05519`.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. `https://arxiv.org/abs/1502.03167`.

[30] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2019. `https://arxiv.org/pdf/1805.11604.pdf`.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2017. `https://arxiv.org/pdf/1412.6980.pdf`.

[32] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2016. `https://arxiv.org/pdf/1511.07122.pdf`.

[33] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs, 2017. `https://arxiv.org/abs/1606.00915`.

[34] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation, 2017. `https://arxiv.org/pdf/1706.05587.pdf`.

[35] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017. `https://arxiv.org/abs/1502.05477`.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. `https://arxiv.org/abs/1707.06347`.

[37] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019. `https://arxiv.org/abs/1509.02971`.

[38] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. 2015. `https://arxiv.org/pdf/1505.04597.pdf`.

[39] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017. `https://arxiv.org/pdf/1704.04861.pdf`.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. `https://arxiv.org/pdf/1512.03385.pdf`.

[41] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. https://www.image-net.org/index.php.

[42] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015. https://arxiv.org/abs/1405.0312.

[43] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding, 2016.

[44] Gerhard Neuhold, Tobias Ollmann, Samuel Rota Bulò, and Peter Kontschieder. The mapillary vistas dataset for semantic understanding of street scenes. In *International Conference on Computer Vision (ICCV)*, 2017.

[45] Raul de Queiroz Mendes, Eduardo Godinho Ribeiro, Nicolas dos Santos Rosa, and Valdir Grassi. On deep learning techniques to boost monocular depth estimation for autonomous navigation. *Robotics and Autonomous Systems*, 136:103701, 2021. https://www.sciencedirect.com/science/article/pii/S0921889020305418.

[46] Ibraheem Alhashim and Peter Wonka. High quality monocular depth estimation via transfer learning, 2019. https://arxiv.org/pdf/1812.11941.pdf.

[47] Katrin Lasinger, René Ranftl, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *CoRR*, abs/1907.01341, 2019. https://arxiv.org/pdf/1907.01341.pdf.

[48] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation, 2015. https://arxiv.org/pdf/1411.4038.pdf.

[49] Audun Wigum Arbor, Even Dalen, and Frank Lindseth. Autonomous driving in simulation using domain-independent perception. 2020. https://www.semanticscholar.org/paper/Autonomous-Driving-in-Simulation-using-Perception-Arbo-Dalen/23a3caae03dc3516b1c7670ad49f586b2f36e74a.

[50] Jeffrey Hawke, Richard Shen, Corina Gurau, Siddharth Sharma, Daniele Reda, Nikolay Nikolov Przemysław Mazur, Sean Micklethwaite, Nicolas Griffiths, Amar Shah, and Alex Kendall. Urban driving with conditional imitation learning. Dec 2019. https://arxiv.org/abs/1912.00177.

[51] Felipe Codevilla, Matthias Müller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. *End-to-End Driving via Conditional Imitation Learning.* Oct 2017. https://arxiv.org/abs/1710.02410.

[52] Eshed Ohn-Bar, Aditya Prakash, Aseem Behl, Kashyap Chitta, and Andreas Geiger. Learning situational driving. 2020. https://www.semanticscholar.org/paper/Learning-Situational-Driving-Ohn-Bar-Prakash/45d7a12b198bdd74d91b8a69bc88b1d9bec2091f.

[53] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments, 2018. https://arxiv.org/abs/1806.06498.

[54] Dean A. Pomerleau. *ALVINN: An Autonomous Land Vehicle in a Neural Network*, page 305–313. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989. https://papers.nips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

[55] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann LeCun. Off-road obstacle avoidance through end-to-end learning. 2005. http://papers.nips.cc/paper/2847-off-road-obstacle-avoidance-through-end-to-end-learning.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[57] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. https://www.tensorflow.org/.

[58] François Chollet et al. Keras. https://keras.io, 2015.

[59] Pavel Yakubovskiy. Segmentation models pytorch. https://github.com/qubvel/segmentation_models.pytorch, 2020.

[60] Kiam Heong Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005. https://ieeexplore.ieee.org/document/1453566.

[61] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. Apr 2016. https://arxiv.org/abs/1604.07316.

[62] Florence Carton, David Filliat, Jaonary Rabarisoa, and Quoc Cuong Pham. Using Semantic Information to Improve Generalization of Reinforcement Learning Policies for Autonomous Driving. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops*, Hawaii (on line), United States, January 2021.

[63] Albert Zhao, Tong He, Yitao Liang, Haibin Huang, Guy Van den Broeck, and Stefano Soatto. Sam: Squeeze-and-mimic networks for conditional visual driving policy learning. 2020. https://arxiv.org/abs/1912.02973.

[64] Horace He. The state of machine learning frameworks in 2019, 2019. https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.

[65] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure, 2019. https://www.hpc.ntnu.no/.

[66] Alexander B. Jung, Kentaro Wada, Jon Crall, Satoshi Tanaka, Jake Graving, Christoph Reinders, Sarthak Yadav, Joy Banerjee, Gábor Vecsei, Adam Kraft, Zheng Rui, Jirka Borovec, Christian Vallentin, Semen Zhydenko, Kilian Pfeiffer, Ben Cook, Ismael Fernández, François-Michel De Rainville, Chi-Hung Weng, Abner Ayala-Acevedo, Raphael Meudec, Matias Laporte, et al. imgaug. https://github.com/aleju/imgaug, 2020. Online; accessed 01-Feb-2020.

[67] Andrew Tao, Karan Sapra, and Bryan Catanzaro. Hierarchical multi-scale attention for semantic segmentation, 2020. https://arxiv.org/pdf/2005.10821.pdf.

[68] Shariq Farooq Bhat, Ibraheem Alhashim, and Peter Wonka. Adabins: Depth estimation using adaptive bins, 2020. https://arxiv.org/pdf/2011.14141.pdf.

[69] Clément Godard, Oisin Mac Aodha, Michael Firman, and Gabriel J. Brostow. Digging into self-supervised monocular depth prediction. October 2019. https://arxiv.org/pdf/1806.01260.pdf.

[70] Nikhil Barhate. Minimal pytorch implementation of proximal policy optimization. https://github.com/nikhilbarhate99/PPO-PyTorch, 2021.

[71] Matthias Müller, Alexey Dosovitskiy, Bernard Ghanem, and Vladlen Koltun. Driving policy transfer via modularity and abstraction, 2018. https://arxiv.org/abs/1804.09364.

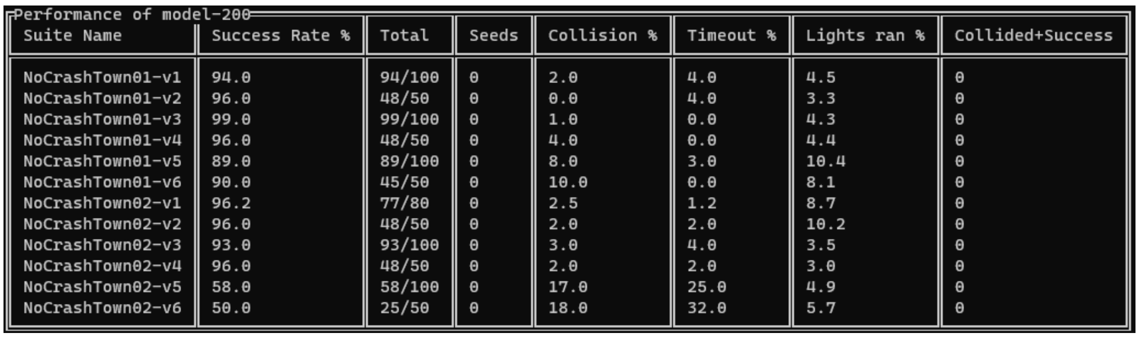
[72] Alex Bewley, Jessica Rigley, Yuxuan Liu, Jeffrey Hawke, Richard Shen, Vinh-Dieu Lam, and Alex Kendall. Learning to drive from simulation without real world labels, 2018. https://arxiv.org/pdf/1812.03823.pdf.

[73] Jingwei Zhang, Lei Tai, Peng Yun, Yufeng Xiong, Ming Liu, Joschka Boedecker, and Wolfram Burgard. Vr-goggles for robots: Real-to-sim domain adaptation for visual control, 2019. https://arxiv.org/abs/1802.00265.

Appendices

Appendix A

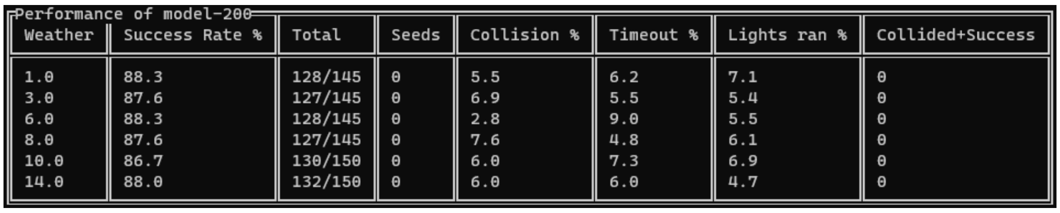
Benchmark Results



```
Performance of model-200
Suite Name         Success Rate %   Total    Seeds  Collision %  Timeout %  Lights ran %  Collided+Success

NoCrashTown01-v1   94.0             94/100   0      2.0          4.0        4.5           0
NoCrashTown01-v2   96.0             48/50    0      0.0          4.0        3.3           0
NoCrashTown01-v3   99.0             99/100   0      1.0          0.0        4.3           0
NoCrashTown01-v4   96.0             48/50    0      4.0          0.0        4.4           0
NoCrashTown01-v5   89.0             89/100   0      8.0          3.0        10.4          0
NoCrashTown01-v6   90.0             45/50    0      10.0         0.0        8.1           0
NoCrashTown02-v1   96.2             77/80    0      2.5          1.2        8.7           0
NoCrashTown02-v2   96.0             48/50    0      2.0          2.0        10.2          0
NoCrashTown02-v3   93.0             93/100   0      3.0          4.0        3.5           0
NoCrashTown02-v4   96.0             48/50    0      2.0          2.0        3.0           0
NoCrashTown02-v5   58.0             58/100   0      17.0         25.0       4.9           0
NoCrashTown02-v6   50.0             25/50    0      18.0         32.0       5.7           0
```

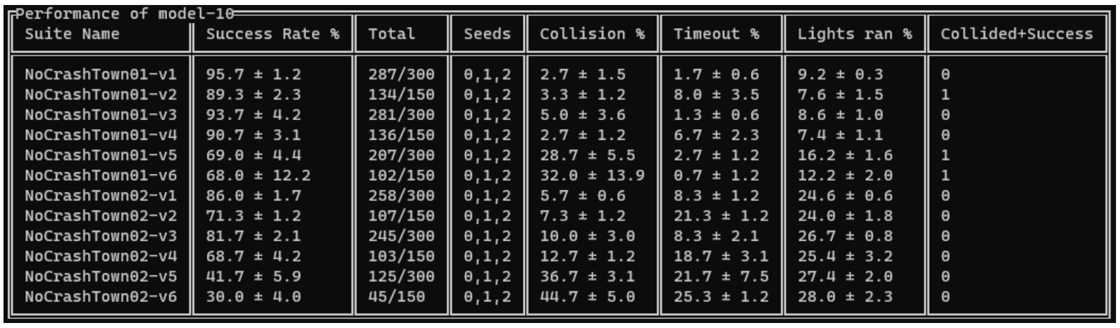
Figure A.1: The benchmark results of the privileged network in a raw format.



```
Performance of model-200
Weather  Success Rate %  Total    Seeds  Collision %  Timeout %  Lights ran %  Collided+Success

1.0      88.3            128/145  0      5.5          6.2        7.1           0
3.0      87.6            127/145  0      6.9          5.5        5.4           0
6.0      88.3            128/145  0      2.8          9.0        5.5           0
8.0      87.6            127/145  0      7.6          4.8        6.1           0
10.0     86.7            130/150  0      6.0          7.3        6.9           0
14.0     88.0            132/150  0      6.0          6.0        4.7           0
```

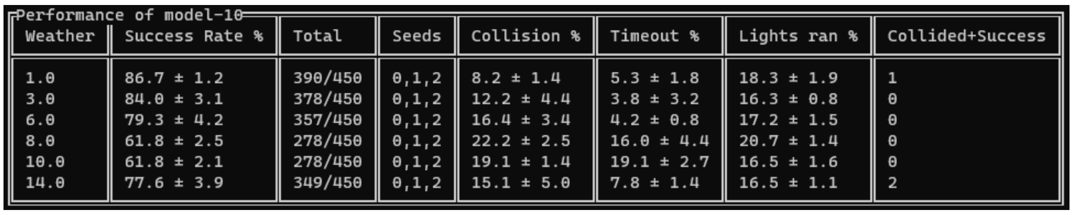
Figure A.2: The benchmark results of the privileged network with respect to weather types in a raw format.



```
Performance of model-10
Suite Name         Success Rate %  Total    Seeds  Collision %   Timeout %    Lights ran %  Collided+Success

NoCrashTown01-v1   95.7 ± 1.2      287/300  0,1,2  2.7 ± 1.5     1.7 ± 0.6    9.2 ± 0.3     0
NoCrashTown01-v2   89.3 ± 2.3      134/150  0,1,2  3.3 ± 1.2     8.0 ± 3.5    7.6 ± 1.5     1
NoCrashTown01-v3   93.7 ± 4.2      281/300  0,1,2  5.0 ± 3.6     1.3 ± 0.6    8.6 ± 1.0     0
NoCrashTown01-v4   90.7 ± 3.1      136/150  0,1,2  2.7 ± 1.2     6.7 ± 2.3    7.4 ± 1.1     0
NoCrashTown01-v5   69.0 ± 4.4      207/300  0,1,2  28.7 ± 5.5    2.7 ± 1.2    16.2 ± 1.6    1
NoCrashTown01-v6   68.0 ± 12.2     102/150  0,1,2  32.0 ± 13.9   0.7 ± 1.2    12.2 ± 2.0    1
NoCrashTown02-v1   86.0 ± 1.7      258/300  0,1,2  5.7 ± 0.6     8.3 ± 1.2    24.6 ± 0.6    0
NoCrashTown02-v2   71.3 ± 1.2      107/150  0,1,2  7.3 ± 1.2     21.3 ± 1.2   24.0 ± 1.8    0
NoCrashTown02-v3   81.7 ± 2.1      245/300  0,1,2  10.0 ± 3.0    8.3 ± 2.1    26.7 ± 0.8    0
NoCrashTown02-v4   68.7 ± 4.2      103/150  0,1,2  12.7 ± 1.2    18.7 ± 3.1   25.4 ± 3.2    0
NoCrashTown02-v5   41.7 ± 5.9      125/300  0,1,2  36.7 ± 3.1    21.7 ± 7.5   27.4 ± 2.0    0
NoCrashTown02-v6   30.0 ± 4.0      45/150   0,1,2  44.7 ± 5.0    25.3 ± 1.2   28.0 ± 2.3    0
```

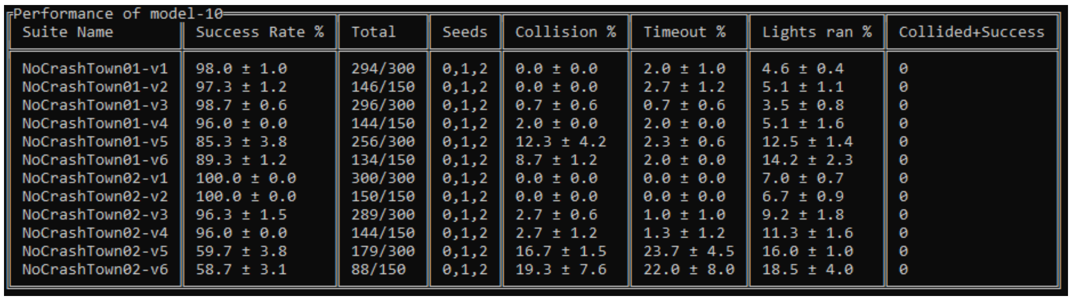
Figure A.3: The benchmark results of LBC-R in a raw format.



```
Performance of model-10
 Weather   Success Rate %   Total     Seeds   Collision %   Timeout %    Lights ran %   Collided+Success

 1.0       86.7 ± 1.2       390/450   0,1,2   8.2 ± 1.4     5.3 ± 1.8    18.3 ± 1.9     1
 3.0       84.0 ± 3.1       378/450   0,1,2   12.2 ± 4.4    3.8 ± 3.2    16.3 ± 0.8     0
 6.0       79.3 ± 4.2       357/450   0,1,2   16.4 ± 3.4    4.2 ± 0.8    17.2 ± 1.5     0
 8.0       61.8 ± 2.5       278/450   0,1,2   22.2 ± 2.5    16.0 ± 4.4   20.7 ± 1.4     0
 10.0      61.8 ± 2.1       278/450   0,1,2   19.1 ± 1.4    19.1 ± 2.7   16.5 ± 1.6     0
 14.0      77.6 ± 3.9       349/450   0,1,2   15.1 ± 5.0    7.8 ± 1.4    16.5 ± 1.1     2
```

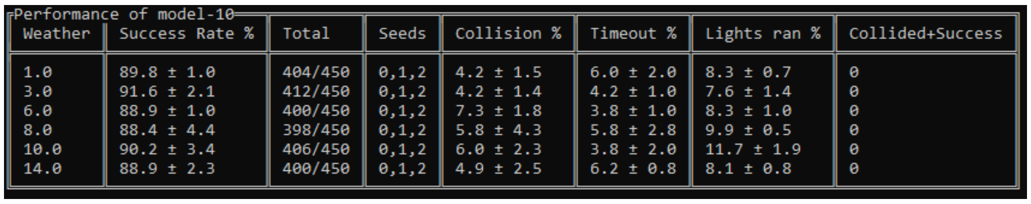
Figure A.4: The benchmark results of LBC-R with respect to weather types in a raw format.



```
Performance of model-10
 Suite Name         Success Rate %   Total     Seeds   Collision %   Timeout %    Lights ran %   Collided+Success

 NoCrashTown01-v1   98.0 ± 1.0       294/300   0,1,2   0.0 ± 0.0     2.0 ± 1.0    4.6 ± 0.4      0
 NoCrashTown01-v2   97.3 ± 1.2       146/150   0,1,2   0.0 ± 0.0     2.7 ± 1.2    5.1 ± 1.1      0
 NoCrashTown01-v3   98.7 ± 0.6       296/300   0,1,2   0.7 ± 0.6     0.7 ± 0.6    3.5 ± 0.8      0
 NoCrashTown01-v4   96.0 ± 0.0       144/150   0,1,2   2.0 ± 0.0     2.0 ± 0.0    5.1 ± 1.6      0
 NoCrashTown01-v5   85.3 ± 3.8       256/300   0,1,2   12.3 ± 4.2    2.3 ± 0.6    12.5 ± 1.4     0
 NoCrashTown01-v6   89.3 ± 1.2       134/150   0,1,2   8.7 ± 1.2     2.0 ± 0.0    14.2 ± 2.3     0
 NoCrashTown02-v1   100.0 ± 0.0      300/300   0,1,2   0.0 ± 0.0     0.0 ± 0.0    7.0 ± 0.7      0
 NoCrashTown02-v2   100.0 ± 0.0      150/150   0,1,2   0.0 ± 0.0     0.0 ± 0.0    6.7 ± 0.9      0
 NoCrashTown02-v3   96.3 ± 1.5       289/300   0,1,2   2.7 ± 0.6     1.0 ± 1.0    9.2 ± 1.8      0
 NoCrashTown02-v4   96.0 ± 0.0       144/150   0,1,2   2.7 ± 1.2     1.3 ± 1.2    11.3 ± 1.6     0
 NoCrashTown02-v5   59.7 ± 3.8       179/300   0,1,2   16.7 ± 1.5    23.7 ± 4.5   16.0 ± 1.0     0
 NoCrashTown02-v6   58.7 ± 3.1       88/150    0,1,2   19.3 ± 7.6    22.0 ± 8.0   18.5 ± 4.0     0
```

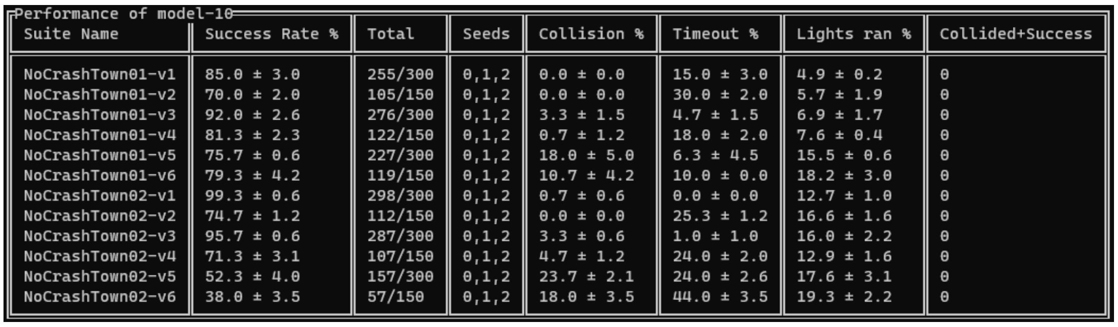
Figure A.5: The benchmark results of LBC-GTCV in a raw format.



```
Performance of model-10
 Weather   Success Rate %   Total     Seeds   Collision %   Timeout %   Lights ran %   Collided+Success

 1.0       89.8 ± 1.0       404/450   0,1,2   4.2 ± 1.5     6.0 ± 2.0   8.3 ± 0.7      0
 3.0       91.6 ± 2.1       412/450   0,1,2   4.2 ± 1.4     4.2 ± 1.0   7.6 ± 1.4      0
 6.0       88.9 ± 1.0       400/450   0,1,2   7.3 ± 1.8     3.8 ± 1.0   8.3 ± 1.0      0
 8.0       88.4 ± 4.4       398/450   0,1,2   5.8 ± 4.3     5.8 ± 2.8   9.9 ± 0.5      0
 10.0      90.2 ± 3.4       406/450   0,1,2   6.0 ± 2.3     3.8 ± 2.0   11.7 ± 1.9     0
 14.0      88.9 ± 2.3       400/450   0,1,2   4.9 ± 2.5     6.2 ± 0.8   8.1 ± 0.8      0
```

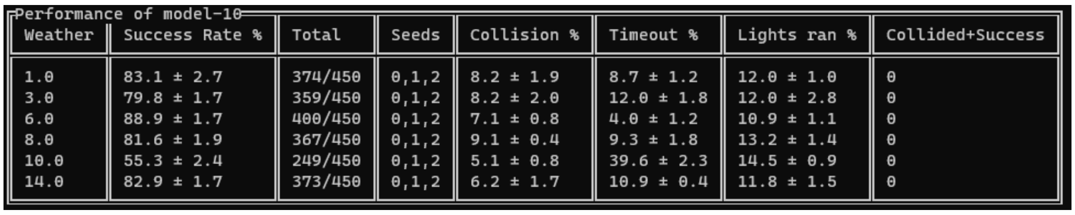
Figure A.6: The benchmark results of LBC-GTCV with respect to weather types in a raw format.



```
Performance of model-10
 Suite Name         Success Rate %   Total     Seeds   Collision %   Timeout %    Lights ran %   Collided+Success

 NoCrashTown01-v1   85.0 ± 3.0       255/300   0,1,2   0.0 ± 0.0     15.0 ± 3.0   4.9 ± 0.2      0
 NoCrashTown01-v2   70.0 ± 2.0       105/150   0,1,2   0.0 ± 0.0     30.0 ± 2.0   5.7 ± 1.9      0
 NoCrashTown01-v3   92.0 ± 2.6       276/300   0,1,2   3.3 ± 1.5     4.7 ± 1.5    6.9 ± 1.7      0
 NoCrashTown01-v4   81.3 ± 2.3       122/150   0,1,2   0.7 ± 1.2     18.0 ± 2.0   7.6 ± 0.4      0
 NoCrashTown01-v5   75.7 ± 0.6       227/300   0,1,2   18.0 ± 5.0    6.3 ± 4.5    15.5 ± 0.6     0
 NoCrashTown01-v6   79.3 ± 4.2       119/150   0,1,2   10.7 ± 4.2    10.0 ± 0.0   18.2 ± 3.0     0
 NoCrashTown02-v1   99.3 ± 0.6       298/300   0,1,2   0.7 ± 0.6     0.0 ± 0.0    12.7 ± 1.0     0
 NoCrashTown02-v2   74.7 ± 1.2       112/150   0,1,2   0.0 ± 0.0     25.3 ± 1.2   16.6 ± 1.6     0
 NoCrashTown02-v3   95.7 ± 0.6       287/300   0,1,2   3.3 ± 0.6     1.0 ± 1.0    16.0 ± 2.2     0
 NoCrashTown02-v4   71.3 ± 3.1       107/150   0,1,2   4.7 ± 1.2     24.0 ± 2.0   12.9 ± 1.6     0
 NoCrashTown02-v5   52.3 ± 4.0       157/300   0,1,2   23.7 ± 2.1    24.0 ± 2.6   17.6 ± 3.1     0
 NoCrashTown02-v6   38.0 ± 3.5       57/150    0,1,2   18.0 ± 3.5    44.0 ± 3.5   19.3 ± 2.2     0
```

Figure A.7: The benchmark results of LBC-TCV in a raw format.



```
Performance of model-10
 Weather   Success Rate %   Total     Seeds   Collision %   Timeout %    Lights ran %   Collided+Success

 1.0       83.1 ± 2.7       374/450   0,1,2   8.2 ± 1.9     8.7 ± 1.2    12.0 ± 1.0     0
 3.0       79.8 ± 1.7       359/450   0,1,2   8.2 ± 2.0     12.0 ± 1.8   12.0 ± 2.8     0
 6.0       88.9 ± 1.7       400/450   0,1,2   7.1 ± 0.8     4.0 ± 1.2    10.9 ± 1.1     0
 8.0       81.6 ± 1.9       367/450   0,1,2   9.1 ± 0.4     9.3 ± 1.8    13.2 ± 1.4     0
 10.0      55.3 ± 2.4       249/450   0,1,2   5.1 ± 0.8     39.6 ± 2.3   14.5 ± 0.9     0
 14.0      82.9 ± 1.7       373/450   0,1,2   6.2 ± 1.7     10.9 ± 0.4   11.8 ± 1.5     0
```

Figure A.8: The benchmark results of LBC-TCV with respect to weather types in a raw format.

Appendix B

LBC-TCV Full Model Architecture

```
FullModel(
  (depth_model): Unet(
    (encoder): ResNetEncoder(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
      (layer1): Sequential(
        (0): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (1): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (2): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (layer2): Sequential(
        (0): BasicBlock(
          (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (downsample): Sequential(
            (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          )
        )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (3): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (4): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (5): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
)
(decoder): UnetDecoder(
  (center): Identity()
  (blocks): ModuleList(
    (0): DecoderBlock(
      (conv1): Conv2dReLU(
        (0): Conv2d(768, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (attention1): Attention(
        (attention): Identity()
      )
      (conv2): Conv2dReLU(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (attention2): Attention(
        (attention): Identity()
      )
    )
    (1): DecoderBlock(
      (conv1): Conv2dReLU(
        (0): Conv2d(384, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
      )
      (attention1): Attention(
      (attention): Identity()
    )
    (conv2): Conv2dReLU(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention2): Attention(
      (attention): Identity()
    )
  )
  (2): DecoderBlock(
    (conv1): Conv2dReLU(
      (0): Conv2d(192, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention1): Attention(
      (attention): Identity()
    )
    (conv2): Conv2dReLU(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention2): Attention(
      (attention): Identity()
    )
  )
  (3): DecoderBlock(
    (conv1): Conv2dReLU(
      (0): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention1): Attention(
      (attention): Identity()
    )
    (conv2): Conv2dReLU(
      (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
```

```
        )
        (attention2): Attention(
          (attention): Identity()
        )
      )
      (4): DecoderBlock(
        (conv1): Conv2dReLU(
          (0): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU(inplace=True)
        )
        (attention1): Attention(
          (attention): Identity()
        )
        (conv2): Conv2dReLU(
          (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU(inplace=True)
        )
        (attention2): Attention(
          (attention): Identity()
        )
      )
    )
  )
  (segmentation_head): SegmentationHead(
    (0): Conv2d(16, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Identity()
    (2): Activation(
      (activation): Sigmoid()
    )
  )
)
(semseg_model): Unet(
  (encoder): ResNetEncoder(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
  )
  (decoder): UnetDecoder(
    (center): Identity()
```

```
(blocks): ModuleList(
  (0): DecoderBlock(
    (conv1): Conv2dReLU(
      (0): Conv2d(3072, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention1): Attention(
      (attention): Identity()
    )
    (conv2): Conv2dReLU(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention2): Attention(
      (attention): Identity()
    )
  )
  (1): DecoderBlock(
    (conv1): Conv2dReLU(
      (0): Conv2d(768, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention1): Attention(
      (attention): Identity()
    )
    (conv2): Conv2dReLU(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (attention2): Attention(
      (attention): Identity()
    )
  )
  (2): DecoderBlock(
    (conv1): Conv2dReLU(
      (0): Conv2d(384, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
  )
  (attention1): Attention(
    (attention): Identity()
  )
  (conv2): Conv2dReLU(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (attention2): Attention(
    (attention): Identity()
  )
)
(3): DecoderBlock(
  (conv1): Conv2dReLU(
    (0): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (attention1): Attention(
    (attention): Identity()
  )
  (conv2): Conv2dReLU(
    (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (attention2): Attention(
    (attention): Identity()
  )
)
(4): DecoderBlock(
  (conv1): Conv2dReLU(
    (0): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (attention1): Attention(
    (attention): Identity()
  )
  (conv2): Conv2dReLU(
    (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
            (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
          )
          (attention2): Attention(
            (attention): Identity()
          )
        )
      )
    )
    (segmentation_head): SegmentationHead(
      (0): Conv2d(16, 9, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): Identity()
      (2): Activation(
        (activation): Softmax(dim=1)
      )
    )
  )
  (normalize_rgb): NormalizeV2()
  (image_model): ImagePolicyModelSS(
    (conv): ResNet(
      (conv1): Conv2d(13, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
      (layer1): Sequential(
        (0): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (1): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (2): BasicBlock(
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (downsample): Sequential(
    (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(1): BasicBlock(
  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(2): BasicBlock(
  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(3): BasicBlock(
  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(4): BasicBlock(
  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(5): BasicBlock(
  (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=1000, bias=True)
  )
  (rgb_transform): NormalizeV2()
  (deconv): Sequential(
    (0): BatchNorm2d(640, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (1): ConvTranspose2d(640, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
    (2): ReLU(inplace=True)
    (3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
    (5): ReLU(inplace=True)
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (7): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
      (8): ReLU(inplace=True)
    )
    (location_pred): ModuleList(
      (0): Sequential(
        (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Conv2d(64, 5, kernel_size=(1, 1), stride=(1, 1))
        (2): SpatialSoftmax()
      )
      (1): Sequential(
        (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Conv2d(64, 5, kernel_size=(1, 1), stride=(1, 1))
        (2): SpatialSoftmax()
      )
      (2): Sequential(
        (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Conv2d(64, 5, kernel_size=(1, 1), stride=(1, 1))
        (2): SpatialSoftmax()
      )
      (3): Sequential(
        (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Conv2d(64, 5, kernel_size=(1, 1), stride=(1, 1))
        (2): SpatialSoftmax()
      )
    )
  )
)
```

Jostein Lilleløkken, Martin Hermansen