Master's thesis

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

Alexander Stensland Iversen Szewczyk

AI-agents Trained Using Deep Reinforcement Learning in the CARLA Simulator

Master's thesis in Computer Science
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2022

NTNU
Norwegian University of Science and Technology

Alexander Stensland Iversen Szewczyk

AI-agents Trained Using Deep Reinforcement Learning in the CARLA Simulator

Master's thesis in Computer Science
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2022

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

NTNU
Norwegian University of
Science and Technology

Abstract

In recent years, deep learning and reinforcement learning within the field of vision-based autonomous driving systems has had a large increase in interest, and additionally several breakthroughs. The creation of publicly available open-source tools such as the CARLA driving simulator [1], has spawned the creation of several highly capable autonomous driving systems, both using a reinforcement learning approach and imitation learning approach.

The World on Rails algorithm [2] is a reinforcement learning approach to training an autonomous agent to drive using the CARLA driving simulator. In the World on Rails algorithm, a forward model is first trained and used to predict the future states for different actions that an agent can choose at a given state, without actually performing them in the real world. This forward model is then used to supervise a visuomotor agent, by predicting the outcome of any potential driving trajectory.

In this thesis, we investigate the World on Rails algorithm by attempting to recreate the results that were presented in the World on Rails paper [2]. We only managed to create a dataset of approximately 100 thousand data frames, but still manage to create a model that performs remarkably well on the NoCrash benchmark, even outperforming the World on Rails paper's [2] recreation of Learning by Cheating [3] in terms of success rate.

We also investigate how the introduction of an autoencoder training step can help improve the performance of the World on Rails algorithm. We do this by pre-training a visual encoder backbone on a highly relevant dataset containing images from the CARLA driving simulator. We train the visuomotor agent with this visual encoder backbone in two different arrangements, one using the visual encoder with frozen weights, and the other with unfrozen weights. We found that using the pre-trained visual encoder backbone with unfrozen weights, did significantly help to improve the model's ability to understand traffic lights, and their states.

Sammendrag

De siste årene har dyp læring og forsterknings-læring innenfor synsbaserte autonome kjøresystemer hatt en stor økning i interesse, i tillegg til flere gjennombrudd. Opprettelsen av offentlig tilgjengelige verktøy, med åpen kildekode, som kjøresimulatoren CARLA [1], har skapt flere svært dyktige autonome kjøresystemer, både ved bruk av forsterknings-læring og imiterings-læring.

World on Rails-algoritmen [2] er en forsterknings-læring-tilnærming, som bruker kjøringssimulatoren CARLA til å trene en agent til å kjøre autonomt. I World on Rails-algoritmen trenes først en "forward"-modell, som brukes til å forutsi fremtidige tilstander for forskjellige handlinger som en agent kan velge i en gitt tilstand, uten å faktisk utføre dem i simulatoren. Denne "forward"-modellen brukes deretter til å trene en visuomotorisk agent, ved å forutsi utfallet til enhver potensiell kjørebane.

I denne oppgaven undersøker vi World on Rails-algoritmen ved å forsøke å gjenskape resultatene som ble presentert i World on Rails-artikkelen [2]. Vi klarte bare å lage et datasett på omtrent 100 tusen datarammer, men klarer likevel å lage en modell som yter bemerkelsesverdig godt på NoCrash-benchmarken, til og med bedre enn World on Rails-artikkelens [2] gjenskaping av Learning by Cheating [3], i forhold til suksessrate.

Vi undersøker også hvordan introduksjonen av et autoencoder-treningstrinn kan bidra til å forbedre ytelsen til World on Rails-algoritmen. Vi gjør dette ved å forhåndstrene en visuell enkoder på et svært relevant datasett som inneholder bilder fra kjøresimulatoren CARLA. Vi trener den visuomotoriske modellen med denne visuelle enkoderen i to forskjellige oppsett, der den ene bruker den visuelle enkoderen med frosne vekter, og den andre med ufrosne vekter. Vi fant ut at bruk av den forhåndstrente visuelle enkoderen med ufrosne vekter bidro betydelig til å forbedre modellens evne til å forstå trafikklys og deres tilstander.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

1.1 Background and Motivation

Ever since large breakthroughs within vision based autonomous driving back in the 1980s [4][5], the topic of cars being driven by artificial intelligence has been highly anticipated. Yet, large-scale implementation of fully self-driving cars is still not present at the time of writing. In contrast to how things were back in the 1980s, the requirement for highly expensive equipment and large budgets has faded, with the introduction of publicly accessible tools and information such as driving simulators, machine learning frameworks and code sharing platforms, making it easier to contribute towards the goal of large-scale full autonomy.

Within the world of developing autonomous vehicles, we generally have two main approaches. One being the modular approach, which is the approach used in most of today's most prominent autonomous driving systems. The modular approach separates the driving system into several independent modules, examples being mapping, perception, planning, and control modules. The end-to-end approach, on the other hand, aims to learn a driving policy that maps directly observations and vehicle actions, most commonly using an artificial neural network.

Currently, one of the most viable options for creating autonomous vehicles has been through the use of imitation learning as it only requires a large dataset, which can be collected by drivers all around the world. As most drivers usually travel significantly long distances without any significant incidents [6], a problem arises as autonomous vehicles are trained significantly less on how to handle safety-critical situations. This is where an approach using reinforcement learning can give an advantage over imitation learning, as the reinforcement learning process entails more exploration.

In this thesis, we investigate the World on Rails algorithm by trying to recreate the results as presented in their paper [2]. We also investigate how adding an autoencoder training step to pre-train a visual encoder backbone can affect the performance and training of the World on Rails algorithm. We use the pre-trained visual encoder backbone in two different implementations. The first implementation uses the visual encoder backbone with frozen weights, but with a small

trainable interpretation head. The second interpretation uses the visual encoder backbone with unfrozen weights. We validate our model using the CARLA driving simulator [1] and the NoCrash benchmark [7].

1.2 Research Goal and Research Questions

In this thesis, we will use the World on Rails [2] algorithm as a foundation to explore the use of an autoencoder training step, and how it can improve the performance of the World on Rails algorithm. Our research goal is formulated as the following: Improve training and performance in the World on Rails algorithm.

We intend to reach this goal by answering the following research questions:

- **Research question 1:** Can we recreate the results presented in the Learning to drive from a World on Rails paper [2]?
- **Research question 2:** How will pre-training a visual backbone on a highly relevant dataset affect performance in the world on rails algorithm?

Two experiments have been conducted to answer these research questions.

1.3 Contributions

This thesis serves as an exploration and comparison of different implementations of the World on Rails algorithm. The main contributions of the thesis are the following:

- We convert the World on Rails algorithm to be run on a newer version of PyTorch and CUDA, more specifically 1.10.2 and 11.3, respectively.
- An investigation on the performance of the World on Rails algorithm, by attempting to recreate the results that were presented in the World on Rails paper [2] with a limited dataset of approximately 100 thousand data frames on the NoCrash benchmark [7].
- An investigation on how the World on Rails algorithm can be improved using a visual encoder backbone pre-trained on a highly relevant dataset extracted from the CARLA simulator.

1.4 Thesis Structure

This thesis is structured into six chapters. These chapters are organized in the following order:

- **Chapter 1 - Introduction:** This chapter gives an introduction to the thesis, including the motivation and context.
- **Chapter 2 - Background and Related Work:** This chapter will cover some of the theoretical foundations of this thesis, including topics such as machine

learning, deep learning and reinforcement learning. This chapter will also cover some related works that are relevant to this thesis.

- **Chapter 3 - Methodology:** This chapter will describe how the work in this thesis has been executed. This chapter will describe the experiments, how these experiments were performed and which tools and resources were utilized.
- **Chapter 4 - Results:** This chapter will present the results from the experiments.
- **Chapter 5 - Discussion:** This chapter will discuss the results from the experiments
- **Chapter 6 - Conclusion and Future Work:** This chapter concludes the results and discussion of this thesis and proposes ideas for future work.

Chapter 2

Background and Related Work

This chapter will be covering the theoretical foundations and topics that are related to this thesis. The first section will be covering machine learning, second will be covering reinforcement learning and also concepts and algorithms related to reinforcement learning. This chapter will also be covering topics related to autonomous driving, such as computer vision and approaches to autonomous driving. The final section of this chapter will cover related works.

2.1 Machine Learning

"Artificial intelligence" and "machine learning" are two terms there can be confusion between. Machine learning is a sub-field within the field of AI, studying the ability to improve performance based on experience. AI systems can use machine learning to achieve intelligent behaviour but can achieve this with other methods than machine learning [8].

Learning can be done in different ways. An agent can learn from reading, studying others, performing actions or solving mathematical problems. What makes learning regarded as machine learning is when the agent is computer-based. Given observed data, a computer-based agent can build a model that can fit this observed data, which can be used as a hypothesis about the agent's world, to help it solve problems within its world [8].

2.1.1 Supervised and Unsupervised learning

In the field of machine learning, we have two main methods of training an agent. These two are *Supervised learning* and *Unsupervised learning*. Each with their use cases, the two mainly differ in terms of the data used to train the agents.

Supervised Learning

Supervised learning focuses on making an agent learn a function that maps from an input to a desired output. As an example, we could take the task of classifying

the animal displayed in the image, where the input is a camera image of an animal, and the output is the class of the animal in the image. In the case of this example, an animal classifying agent would have to be trained on a data set containing pairs of image input and their corresponding animal label. A training data set D_{sup}, can then be defined as:

$$D_{sup} = [[x_1, y_1], [x_2, y_2], ...[x_n, y_n]]$$

Where x_i is an input, and y_i is the corresponding label (or target value). The agent trains on this data set D to learn a function that maps an input x to an appropriate output y [8].

Unsupervised Learning

In unsupervised learning, the training data is unlabeled, and the agent is expected to learn patterns and gather knowledge from within the data itself without any explicit supervision in the form of target values or labels during training. Due to unsupervised learning not requiring labelled data, it circumvents the need for extensive labelling work which can be time and resource costly [8]. A more specific example of unsupervised learning (Autoencoders) is explained in section 2.3.4.

Since unsupervised learning doesn't require any labels y_i, a training data set D_{unsup} can be defined as:

$$D_{unsup} = [x_1, x_2, ...x_n]$$

Where x_i represents a data input, e.g. an image or generally a set of values.

2.1.2 Deep Learning

As machine learning is a sub-field within AI, deep learning is then again a sub-field within machine learning. Deep learning refers to the group of methods of machine learning involving the use of deep neural networks. Deep neural networks are networks comprised of multiple layers of simple, adjustable computing elements, e.g. weights and biases [8].

Deep neural networks were originally inspired by the structure and flow of the human brain and its brain cells. One key component that the neural net mimics, is the brain's neuron in the form of a perceptron. The perceptrons in a neural network are combined together in a connected and layered structure which is shown in figure 2.1. With these perceptrons, or nodes, their activation functions, weights between layers, and bias', the network is able to learn a mapping between an input X and an output Y [8]. More about different types of machine learning is discussed in section 2.1.1.

Being one of the most common approaches to machine learning, deep learning is a versatile and widely applicable method. It can effectively handle complex data, and also plays a significant role in this thesis due to its usage within reinforcement learning [8]. Within reinforcement learning, a deep neural network can act as both

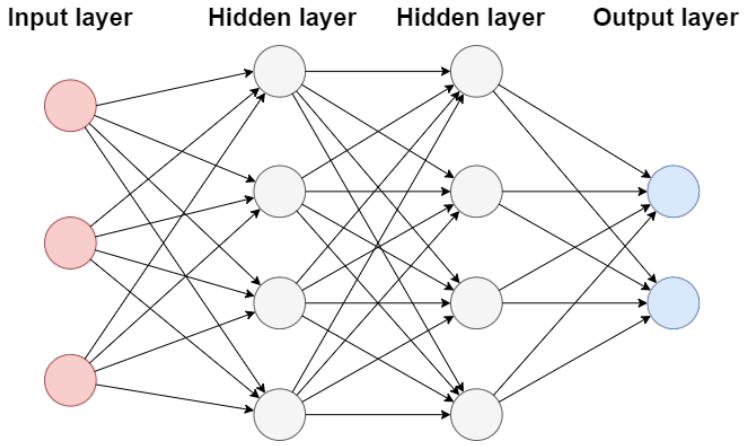


Figure 2.1: Figure of a neural network with two hidden layers

a policy mapping between state and action, and as a state-value mapping, which is explained further in section 2.2.

Activation Function

One of the explanations as to why deep learning is so powerful is the *universal approximation theorem*. This theorem implies that neural networks with a linear output layer with a minimum of one hidden layer consisting of enough neurons with a "squashing" activation function can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$ [9].

Choosing a fitting activation function is an important part of designing a neural network, and impacts the results of the training process and the rate of convergence of the network. Different activation functions can be optimal for different scenarios, and are not necessarily only placed on the neurons in the hidden layers. Below is a list of some common activation functions, where z denotes the sum of weighted input plus bias.

- **Rectified Linear Unit (ReLU):** ReLU is similar to the identity function, however if the input $z \leq 0$, the output from ReLU will be zero. ReLU is considered to be the default recommendation for most modern feed-forward neural networks [9].

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

- **Leaky Rectified Linear Unit (LReLU):** Similar to ReLU, except it allows for a small output when $z \leq 0$.

$$LReLU(z) = \begin{cases} z, & \text{if } z > 0, \\ 0.01z, & \text{otherwise.} \end{cases}$$

- **Sigmoid (σ):** Sigmoid outputs a values between zero and one.

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

- **Hyperbolic Tangent ($tanh$):** $tanh$ outputs values between minus one and one. In the equation below, σ is the sigmoid function.

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

- **Softmax:** Softmax is commonly used in the output layer of a neural network, as it limits the network to output an output vector where the sum of the vector is equal to one, making it a good alternative when the network is expected to handle classification problems, or generally output probability distributions.

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

  where z_i denotes one element in the vector z.

Loss Function

An important part of estimating the accuracy of a neural network is determining how much its output values deviate from the desired target values. In other words, the degree of error the network makes on every training data frame after propagating it through its layers, by comparing a network's output $\hat{y}$ against the desired target value y of that data frame. $\hat{y}$ and y are compared using a loss function, and the result from this comparison (the loss) is what fundamentally impacts how the network's parameters will be updated. The choice of a loss function is therefore an important decision to make during the designing of the network, and the optimal loss function can vary based on the network's architecture, training process and the training data.

Below is a list of different loss functions, where $\hat{y}$ refers to the network's output, and y refers to the target value. n is the number of outputs from the network based on a sample of n data frames in the training data.

- **Mean Squared Error (MSE):** MSE estimates the error by squaring the difference between prediction and target. Meaning that MSE strictly will not return negative numbers.

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

- **Root Mean Squared Error (RMSE):** Similar to MSE, except it outputs the root of MSE.

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

- **Cross Entropy (CE):** CE compares two probability distributions, $\hat{y}$ and y, by measuring the entropy between these two distributions, which makes it a good alternative for classification or logistic regression.

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

Back-propagation (backprop)

Feed-forward neural networks take in an input x as an initial flow of information that propagates *forward* throughout the network's hidden layers until it gets output as an output $\hat{y}$, which can be compared against a target y to produce a scalar loss $J(\theta)$. This information flow is altered throughout the network by the weights, biases and activation functions at each layer. From this we can form an optimization problem, as the activation functions commonly are immutable during training, the weights and biases in the network need to be altered to best minimize the loss [9].

In 1986, Rumelhart *et al.* [10] introduced the back-propagation algorithm, not to be confused with stochastic gradient descent. The back-propagation algorithm allows for the information gathered from the loss to be propagated backwards through the network, which ultimately computes a gradient. This gradient can then be used by another algorithm, such as stochastic gradient descent, to have the network update its parameters to learn accordingly [9].

The back-propagation algorithm works in two stages, where the first stage involves passing information forward in the network, which allows for the loss to be created. The second stage is the backwards pass, where we use the chain rule from calculus to compute the gradient from the loss. Ultimately, with gradient descent, we update all the weights and biases in the network, which can be formulated as:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial L}{\partial \theta},$$

where L is the loss function, and θ denotes all the weights and biases in the network. $\frac{\partial L}{\partial \theta}$ is the gradient computed in the backwards pass, and α is the learning rate. As this is gradient *descent*, the term including the gradient is negatively signed.

Issues related to training

- **Overfitting** is a problem that can occur in which the network too closely fits to its training dataset, and badly generalizes outside the dataset it trained on. This problem can occur in cases where the network has too many parameters or trains on a not adequately diverse or large enough dataset. Apart from resizing the network or modifying the dataset, one can add regularization such as dropout, L1 or L2, or add noise or augmentations to the data

being input to the network. On the other hand, underfitting is the scenario where the network becomes too simplistic to accurately represent the data.

- The **Vanishing gradient** problem can occur during back-propagation in many-layered feed-forward networks when the partial derivative of the error function with respect to a network weight ($\frac{\partial L}{\partial \omega}$) becomes vanishingly small, which is the result of multiplying many low-value terms. With a vanishing gradient, the weights of the network get insignificantly updated, and the network might become stuck in local minima. In relation, exploding gradients happen when gradients become too large, which is the result of multiplying multiple high-value terms, making the network unable to update its weights in a stable manner. This can occur when activation functions whose derivatives can have large values are used.

Training, Testing and Validation

A common practice in deep learning is to have three separate datasets, a training dataset, a testing dataset and a validation dataset. The training dataset is commonly the largest of the three, and is used to generate losses to perform back-propagation and ultimately train a network. The validation dataset is a dataset usually passed through the network after each epoch, in order to get an unbiased evaluation of the network's performance. The evaluation can be used to detect whether the network generalizes well from the training dataset (not overfitting), and can also be used for hyperparameter tuning. After training has finished, we want to get an unbiased evaluation of the performance of the network by passing data through the network that it has not been trained on. The predictions of the network are then compared to targets, and we can measure the network's performance in some desired metric.

2.2 Reinforcement Learning

As opposed to imitation learning [11], where an agent learns how to act based on a labelled dataset, reinforcement learning lets an agent interact with an environment and periodically receive rewards based on how it acts. This reward allows the agent to reflect on how it acts, and how it should change its policy. An example of a reward function could be in a simple game of tic-tac-toe, where a win results in a reward of 1, loss 0, and a draw $\frac{1}{2}$. In the case of imitation learning, an agent is not expected to perform better than what he is imitating, which in the field of autonomous motor vehicles often is a human being. Reinforcement learning on the other hand can in theory allow for an agent to explore and potentially reach a better than human performance [8].

The main objective of a reinforcement learning algorithm is to maximize the expected sum of rewards. The environment is usually in the form of a Markov decision process (MDP), where the agent needs to choose an action based on the current state in the environment, where some states result in "better" or "worse"

rewards. The agent's policy (denoted as π) is what determines which action the agent chooses both outside of training, and during training (depending on the training being off-policy or on-policy). It is this policy π which the agent needs to optimize during training to achieve its ultimate goal of maximizing its expected sum of rewards [8].

A more extensive explanation of reinforcement learning is given by Stuart Russell [8] and Richard S. Button [12].

2.2.1 Markov Decision Process

A Markov decision process is a sequential decision problem in a stochastic environment with a transition model that specifies the probabilistic outcomes of any action and a reward function that specifies the reward from each state [8]. Given an agent in a current state s_t (in a state space $\mathbf{S}$) with its reward r_t, the agent can choose to perform an action a_t (from an action space $\mathbf{A}$, or if the state has limited actions $\mathbf{A_s}$), and based on a transition probability function p and a_t, the agent can in the next time step $t+1$ end up in a new state s_{t+1} with its reward r_t. p can be defined as:

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

where $p(s'|s,a)$ is the probability that action a in state s at time t will put the agent at the next state s' at time $t+1$. As the transition probability function only depends on a current state s, and an agents action a, and is independent from all previous states and actions, it satisfies the *Markov property*. From a given first state s_0 and to some final state s_n, and the actions chosen in between, we can define an agents trajectory τ, where $\tau = \{(s_0, a_0, r_0), ..., (s_n, a_n, r_n)\}$.

2.2.2 Bellman equation

In reinforcement learning, a common task is to estimate an action-value function $Q(s,a)$ for each state s in the environment. A usual approach is to let an agent act out a full episode in either an on-policy or off-policy manner, and retroactively update an action-value function based on the agent's trajectory, which can be achieved using the Bellman equation. The Bellman equation in a deterministic environment:

$$V(s) = \max_{a \in A(s)} \sum_{s'} [R(s,a) + \gamma V(s')], \tag{2.1}$$

where V is the value function that gives the value of a given state s, and γ is a discount factor. The Bellman equation computes the value of a state s, by looking at the reward of the next step, plus the expected value from the subsequent states [8]. The action-value function $Q(s,a)$ is also directly related to the value function as follows:

$$V(s) = \max_{a \in A(s)} Q(s,a), \tag{2.2}$$

2.2.3 Reward Function

As one of the fundamental parts of reinforcement learning is for an agent to maximize its expected reward, the use and design of a fitting reward function are critical for the success of the learning algorithm. As reinforcement learning usually implies that an agent starts with a policy that gives random actions, a good reward function is expected to give the agent a good indication as to what actions are "good" and which are "bad". In scenarios where rewards are too low, or too hard to earn, random actions might not be enough for the agent to start learning a good policy [12].

Given the problem of learning to drive in an urban environment, reward shaping can become complex. In cases of racing on a track, reward could be simply defined as the longer the distance driven, in the shortest amount of time, the higher the reward. In urban driving, there are traffic laws, pedestrians, other cars, traffic lights and signs that the agent needs to learn to act accordingly to. Reward can then be based upon several factors from the environment, such as collision with other pedestrians and running red lights resulting in a negative reward, or staying center lane and yielding for cars resulting in a positive reward.

A reward at a given time step t is usually defined as $r_t = R(s_{t+1}, s_t, a_t)$, where R is a reward function. Reward can also be defined over a trajectory τ, where we can sum all rewards over a finite and sequential amount of steps T in the environment, which can be written as:

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

Using trajectories, we can put rewards from states and actions in relation to each other in time, and not just evaluate the agent's policy at given isolated states and actions. Additionally, we can add a discount factor γ^t, so that we can value actions closer in time.

2.2.4 Deep Reinforcement Learning

Deep reinforcement learning is a method of reinforcement learning that includes the use of deep neural networks. The name stems from the combination of reinforcement learning and deep learning. Deep reinforcement learning algorithms usually use deep neural networks as an agent's policy, which maps states to actions. This allows agents to effectively handle much more complex and varied inputs, such as images, continuous state spaces and output into a more complex action space [12].

Policy Representation

The policy is, simply put, just a mapping between states and actions. After an agent is given a state from the environment, it chooses an action based on this state and its policy. The states in an environment can be represented in several ways, some can be represented in a table form, such as the environment in figure 2.2, and others can be represented with continuous values. Action spaces can also be represented as both discrete and continuous values.

Table policy: Given the problem of finding an optimal path from start to finish in a discretized environment such as the checkered board shown in figure 2.2, a good policy mapping can be represented as a table as well. In this table, the input to the mapping is the x and y position of the agent on the board, and the output is an action that is either up, down, left or right. In this case, both the state space and the action space are discrete.

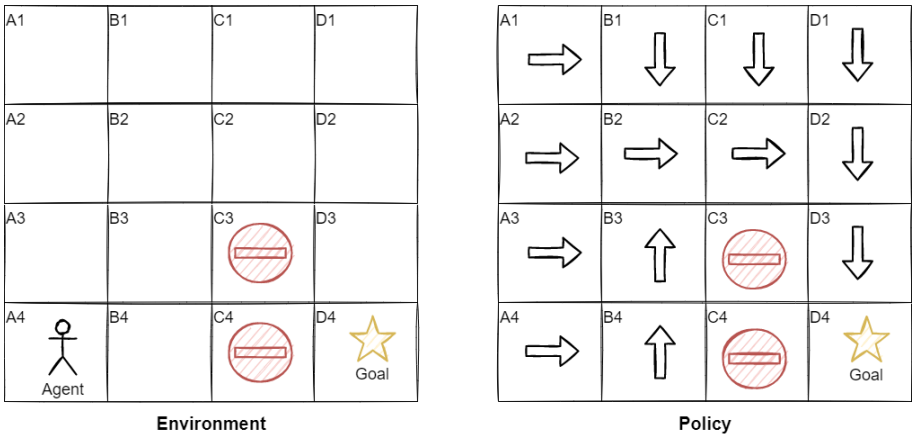


Figure 2.2: Figure of a optimal path problem environment to the left, and a possible optimal table policy to the right.

Policy networks: In scenarios where an agent is learning how to drive in urban environments, inputs from the environment can often come in the form of speeds, positions, accelerations and video/images, that come in the form of continuous, high-dimensional values. Continuous and high-dimensional spaces will in most cases be too complex to be handled efficiently by a table policy, and to be updated accordingly. Therefore, neural networks, or policy networks, can be used to map between states and actions.

Policy Gradient Methods

Policy gradient methods are a type of reinforcement learning techniques that aim to optimize policies with respect to expected reward using gradient descent. In regards to how the policy changes, the methods use an estimate of the gradient of the expected rewards to evaluate whether or not the changes made to the policy will be beneficial or not. For one of the more simple gradient methods, vanilla

gradient method (or REINFORCE policy gradient method), an estimated gradient is given by:

$$\hat{g} = \hat{E}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t], \tag{2.3}$$

where $\hat{A}_t$ is the estimator of the advantage function at a time step t, and $\hat{E}_t$ is the empirical average over a finite batch of samples, or trajectory, τ. The advantage function is a measure of how much a certain action in a given state is either "good" or "bad". It can be denoted as:

$$A(s,a) = Q(s,a) - V(s),$$

where $Q(s,a)$ is the expected reward of an action a from a state s, and $V(s)$ is the expected reward from state s prior to the action.

With equation 2.3, we can generate a loss function for a policy network with respect to its parameters denoted as θ:

$$L^{PG}(\theta) = \hat{E}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t], \tag{2.4}$$

Proximal Policy Optimization (or PPO) [13] is a policy gradient method that builds upon the work from Schulman *et al.* [14] which initially introduced Trust Region Policy Optimization (TRPO). PPO does have some of the benefits TRPO has, but is much simpler to implement, can be more easily applied to a wider range of problems, and has better sample complexity [13]. Schulman *et al.* [13] introduces a set of PPO methods, but in the case of this thesis, we will only be focusing on the method that uses a clipped surrogate objective. This clipped objective function is defined as the following:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{2.5}$$

where $r_t(\theta)$ denotes the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, which means that $r_t(\theta_{old}) = 1$. ϵ is the clipping parameter. $r_t(\theta)$, in combination with the clipping function, is used to discourage and limit large deviations from the original θ_{old}.

2.3 Computer Vision

The following section will cover some topics related to computer vision, and techniques created to help machines more easily interpret complex and high-dimensional image and video data. The section will cover the types of artificial neural networks known as CNN and ResNet, and will also cover semantic segmentation.

2.3.1 Convolutional Neural Networks (CNNs)

A convolutional neural network is a class of neural networks that employ one or more convolutional layers, and are common to use when analyzing image or other grid-like data. Looking at natural images, they usually contain statistical

properties that are invariant to spacial shifts. In practice, this means that an image containing motives, usually still remains an image with said motives even when the motives are moved a few pixels in a direction. The main advantage of the CNN is that they are shift invariant by using shared weights across multiple image locations. These shared weights are known as the kernel, and the process of applying this kernel across the image is known as a convolution. The method has proved to be highly successful in practical application [9].

For a two-dimensional image I as input, we can use a two-dimensional kernel K, this kernel K slides over the image to output a feature map S. If the convolution outputs into a two-dimensional feature map, we refer to a feature at position (i, j) as $S(i, j)$. The value in $S(i, j)$ is defined as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{2.6}$$

where K is a kernel with a size mxn. A visual representation of the convolution process can be seen in figure 2.3 where the kernel moves with a stride of one. After S is calculated, all the values in the feature space are passed through an activation function before being passed to the next layer.

In the examples above, the input I is a two-dimensional data structure, however, CNN can in theory handle data for any n-dimensional array of inputs. In the cases where CNNs handle RGB encoded images, the input is expected to be three-dimensional, as each pixel contains the three RGB values that represent its color. Another example of CNNs handling three-dimensional data are Volumetric CNNs, which can be used to classify objects in three-dimensional data [15].

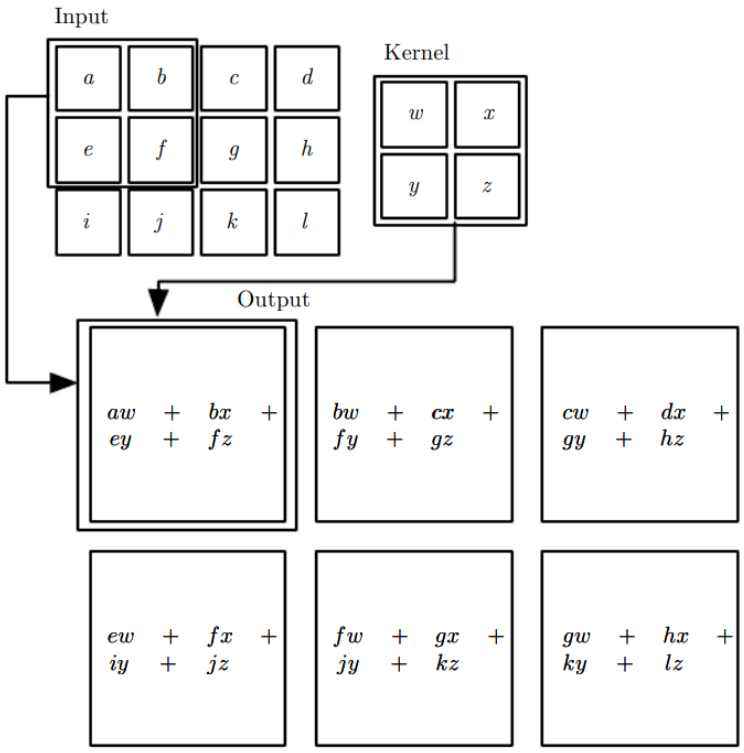
Max Pooling and Average Pooling

A common method used in CNNs after a convolution has been performed, is to apply a pooling function to further modify the output from the convolution. A pooling function can help to effectively down-scale the output from the convolution layer, by replacing the values of a rectangular area within the output with a summary statistic of the area. Two common pooling functions are max pooling and average pooling [9].

In the two-dimensional case of a $2x2$ pooling function over a $4x4$ data structure with a stride of two, the result from the pooling will be a $2x2$ data structure. As the pooling function is $2x2$ with a stride of two, the targeted regions in the original output don't overlap. If the max pooling function is being utilized, the output from each region after the pooling function is applied is the maximum value in that $2x2$ region. On the other hand, if the average pooling function is being used, then the average value from that region will be returned.

2.3.2 Residual Neural Networks (ResNet)

As neural networks grow deeper, the task to optimize them, and have them train reliably in a matter that make them more accurate, and able to store more know-



Source: [9]

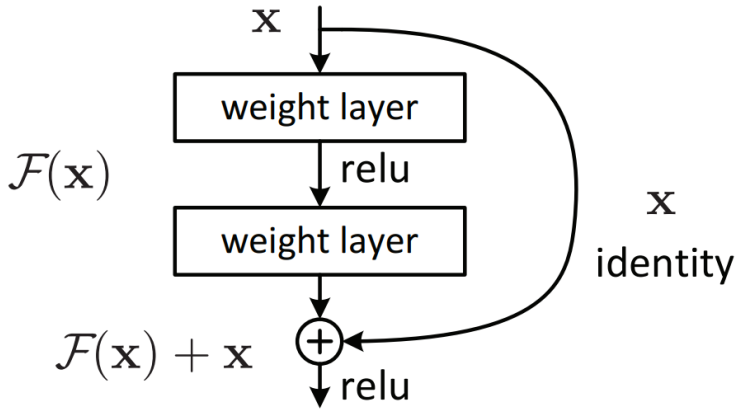
Figure 2.3: Figure illustrating how a two-dimensional kernel in applied to a two-dimensional data-structure to output into a two dimensional feature map.

ledge, proves to be a hard task to tackle. In addition, as traditional neural networks grow deeper, the problem of vanishing/exploding gradients become more prominent. In 2015, He *et al.* [16] introduced residual networks, which allowed for deeper neural network to more easily be optimized, and that gained accuracy with considerably increased depth [16]. The ResNet architecture has since become a widely used network architecture within the field of computer vision.

A residual network consists of several residual blocks. An example of a residual block can be seen in figure 2.4. A residual block receives an input x, and passes it through its multiple layers, as a mapping $F(x)$. Before being output from the block, its skip connection adds the original input x to its mapped value $F(x)$, which results in $F(x)+x$.

Consider a mapping $H(x)$ to be the underlying mapping that needs to be fit by a couple of layers of a network. By utilizing skip connections, we allow for the network to also fit the residual mapping. This means that instead of just $H(x)$, we let the layers fit $F(x) := H(x) - x$, giving $H(x) := F(x) + x$ [16].

There exists multiple different variants of ResNet, these include the ResNets having different amounts of learnable layers, like ResNet-34 having 34 learn-



Source: [16]

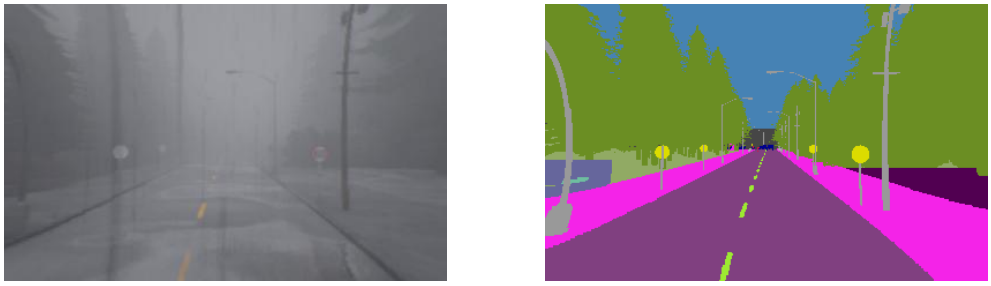
Figure 2.4: Figure showing a residual block.

able layers and ResNet-101 having 101 learnable layers. Related to ResNet, there also exists architectures which utilize skip connections such as HighwayNet [17] (weighted skip connections), and DenseNet [18].

2.3.3 Semantic Segmentation

Semantic segmentation is the process of assigning a label to every individual pixel in a given image, and is a popular tool in the field of computer vision. The main goal of the method is to help learning algorithms to better analyze, differentiate and classify objects and concepts in images. During training, semantic segmentations are most commonly used as target values for the network, where the original RGB image is the input. In figure 2.5, we can see an example of an RGB image, and its semantically segmented counterpart.

In a semantically segmented image, each individual pixel of an image has been given a label, usually in the form of an integer value ranging from zero to $n-1$, where n is the number of labels. In an urban driving environment, these labels could correspond to the five classes vehicles, pedestrians, road, sidewalk and "other", where "other" is a class relating to any class which is not within the first four classes. As a network is input RGB images from this driving environment, it needs to output into a similar amount of feature maps as the amount of categories of labels. In practice, if the original RGB image matrix is of the shape $3 \times 480 \times 360$, the resulting semantically segmented image matrix from the network will be of the shape $5 \times 480 \times 360$. With the use of a Softmax function across the label category dimension, we can ultimately get a probability distribution.



(a) RGB image **(b)** Semantically segmented image

Source: [2]

Figure 2.5: Figure showing an RGB image and its semantically segmented counterpart. The images are extracted from the one million frames World on Rails dataset [2].

2.3.4 Autoencoders and Dimensionality Reduction

In the field of statistics, the dimensionality of a dataset refers to how many attributes a dataset has. In the case of a dataset holding only two attributes (e.g. gender and hair color), the dataset is said to have a dimensionality of two. Datasets are usually of much higher dimensionality than two, and in the case of machine learning these datasets can often come closer to thousands or even millions of dimensions. The field of dimensionality reduction focuses on representing high-dimensional data in a more sensible way, for humans or even AI [8]. Dimensionality reduction can be performed using traditional methods such as PCA [19] or LDA [20], or through deep learning methods like autoencoders [8].

The autoencoder consists of two main parts, an encoder part and a decoder part, as can be seen in figure 2.6. The main goal during the training of a simple autoencoder is to fit a model such that the output y_i is similar to its input x_i, more specifically $y_i = f(x_i) \approx x_i$. This task is relatively trivial. However, the benefits of the autoencoder lies in its architecture. Since the autoencoder has a bottleneck between the input and the output layer, it needs to compress the input data as effectively as possible, at the bottleneck, in order to have it be reconstructed with as little loss as possible.

Post-training, the autoencoder can be divided into an encoder model and a decoder model. The encoder has learned to take an input x_i and encode it to a more condensed representation z_i, while the decoder has learned to take some condensed representation z_i and reconstruct it into some output $\hat{x}_i$, which should be similar to the original x_i [8]. In this thesis, mostly the properties of the encoder are utilised.

The encoder part of an autoencoder can be beneficial to use in certain reinforcement learning scenarios. In cases where an agent observes the environment through very high-dimensional observations, such as images, it can take both longer, and be more challenging for the agent to create a good internal rep-

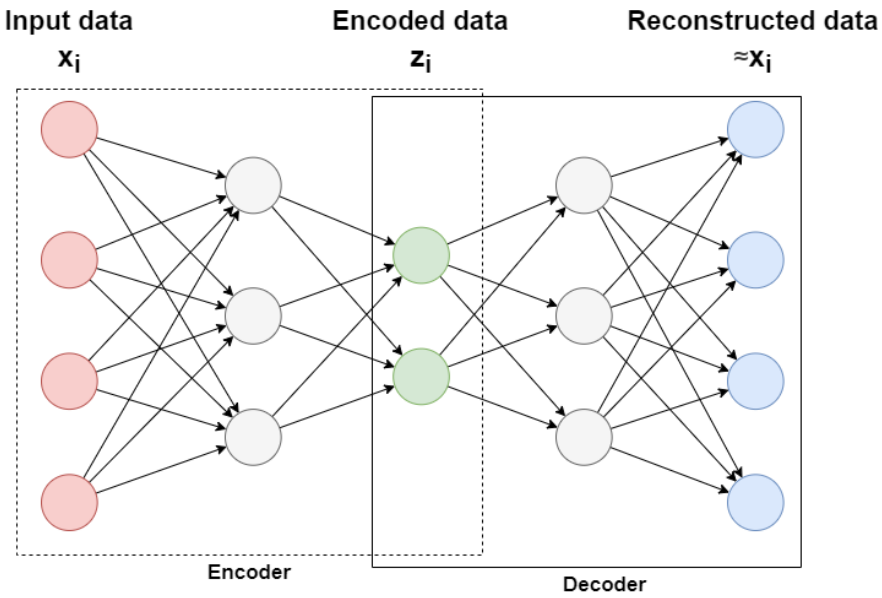


Figure 2.6: Figure of an autoencoder with a latent space, or bottleneck, of 2 nodes

resentation of the data, whilst also trying to learn a good policy for acting in the environment. Using a pre-trained encoder from an autoencoder, which can have the agent train on fewer input parameters, and possible a better representation of the observation, which can be beneficial [21].

2.4 Approaches to autonomous driving

This section will cover the two main approaches to autonomous driving, modular and end-to-end, including some of their advantages and disadvantages. A more extensive explanation of approaches to autonomous driving can found in Janai *et al.* [22].

2.4.1 Modular

The modular approach implies that the autonomous system that operates the car consists of separate modules that individually perform separate tasks that are needed for the car to operate in its environment. Modules could be human-made or be based on machine learning. Modules are usually structured in a pipe-lined manner. The tasks these modules can handle might be navigation, lane orientation, perception, vehicle control, object detection etc. The modular approach is considered to be the standard approach in the autonomous driving industry [22].

An advantage of using a modular approach is that the pipeline allows for human interpretable intermediate representations to be extracted from between modules, which allows for humans to gain insights into possible reasons as to why a system is failing, given system failure [22]. The modular approach does require additional domain knowledge in comparison to an end-to-end approach. However, in a commercial setting, the modular nature of the system allows for separate teams to work and specialize in different aspects of the driving system simultaneously.

2.4.2 End-to-end

In contrast to a modular approach, the end-to-end approach focuses on learning a driving policy that maps directly between observations from the environment to vehicle actions. This driving policy usually comes in the form a deep neural network, that outputs either discrete or continuous vehicle control values. The neural network can be train in either an imitation learning manner, or in a reinforcement learning manner. However, there are also approaches that use a combination of both reinforcement learning and imitation learning, which can combine the benefits of both exploration and expert data [23].

A large problem related to end-to-end approaches, is that the neural network acts as a "black box" which makes it harder for humans to understand the reasoning behind the network's inferences. In cases where the system's actions are faulty, it is therefore harder to understand why this/these actions were chosen. For end-to-end systems trained in a reinforcement learning manner, the approach is expected to act in a simulated environment during training, as early actions in the reinforcement learning process can be considered to be random and dangerous for real life situations. In contrast to the modular approach, the end-to-end approach does not necessarily require the equivalent amount of domain knowledge.

2.5 Technology

This section will cover some of the technology related to this thesis. This includes the Car Learning to Act (CARLA) driving simulator, some important benchmarks used in CARLA to evaluate performance of autonomous driving systems, and machine learning frameworks.

2.5.1 Car Learning to Act (CARLA)

Car Learning to Act (CARLA) [1] is an open-source urban driving simulator used for research within the field of autonomous driving. The system has been built from the ground up to support the development and training of autonomous driving systems, and also for the systems to be evaluated through various benchmarks.

A more extensive explanation of the CARLA simulator can found in Dosovitskiy *et al.* [1].

The CARLA simulator is by it self hosted on a port in a given computing system, and the user can communicate and interact with simulator via a Python API. In the CARLA simulator, the user can choose to load different maps that traffic can be spawned within. These maps are also referred to as towns. For this thesis we will refer to eight different publicly available towns for the CARLA simulator. These towns, and a short description (as presented in the CARLA simulator docs [24]) of each, can be found in the list below:

- **Town01:** A basic town layout with all "T junctions".
- **Town02:** Similar to Town01, but smaller.
- **Town03:** The most complex town, with a 5-lane junction, a roundabout, unevenness, a tunnel, and much more. Essentially a medley.
- **Town04:** An infinite loop with a highway and a small town.
- **Town05:** Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes.
- **Town06:** Long highways with many highway entrances and exits. It also has a Michigan left.
- **Town07:** A rural environment with narrow roads, barely non traffic lights and barns.
- **Town10:** A city environment with different environments such as an avenue or a promenade, and more realistic textures.

The CARLA simulator also allows for the weather and lighting conditions to be changed in the town that is currently being used. The user can change the weather and lighting conditions by augmenting certain parameters such as cloudiness, precipitation, sun altitude angle or fog density to name a few. 14 different weather and lighting condition combinations are however also defined as a set of presets. Examples of these weathers are ClearNoon, CloudyNoon, WetNoon and WetCloudyNoon. In this thesis, we will refer to weather and lighting conditions as just weather.

NoCrash Benchmark

Evaluation in the CARLA simulator can be performed using different benchmarks. Examples of these benchmarks can be the CARLA benchmark [1] or the NoCrash benchmark [7]. For this thesis, we will only focus on the NoCrash benchmark, as it is the only benchmark utilized for agent evaluation in the two experiments conducted.

The NoCrash benchmark is a high-demanding benchmark that requires an agent to adapt to dynamic urban driving scenarios, with different levels of traffic density, as well as a different types of weather. The different levels of traffic density, also referred to as tasks, are *empty*, *regular* and *dense*. The *empty* task does not spawn any pedestrians or other vehicles, the *regular* task spawns some pedestrians and vehicles, and the *dense* task spawns a higher level of pedestrians and

vehicles than *regular*.

During training and evaluation for the NoCrash benchmark, the agent is initially trained to act in *Town01*, and is evaluated in both *Town01* and *Town02*. The NoCrash benchmark does also utilize six different weathers, where four are used for training, and all six are used for evaluation. The training weathers are ClearNoon, WetNoon, HardRainNoon and ClearSunset, and the last two testing weathers being WetSunset and SoftRainSunset.

In each town, there are 25 different routes that the agent is evaluated on. For each route, there are three different tasks, and six different weathers, meaning that the agent is ultimately evaluated on $2 \times 25 \times 3 \times 6 = 900$ different episodes. Throughout an episode, the agent receives high-level commands that tell the agent which path to follow. Episodes end with the agent either reaching the end of the route, collision, deviating too far from the route, or by the time limit of the route being reached. Agents can be evaluated on several different metrics, one of the most common being success rate, which is defined as the number of times the agent completed an episode compared to the total number of episodes.

2.5.2 Machine Learning frameworks

Two common machine learning frameworks in the research and development industry are TensorFlow [25] and PyTorch [26], developed by Google Brain and Facebook's AI Research lab, respectively. The frameworks are open-source tools that aim to give simpler interfaces that allow researchers and developers to build and deploy machine learning models much faster, easier and reliably, than beginning the development process from scratch [25] [26].

The open-source machine learning framework PyTorch allows for an easy set-up and installation process, and allows for the creation of machine learning models down to a layer-to-layer scale. PyTorch also allows for accelerating training with both AMD and Nvidia GPUs. For Nvidia's GPUs, the CUDA (Compute Unified Device Architecture) library allows for software to utilize GPUs and their parallel processing capabilities, which in machine learning can be remarkably more efficient than the CPU. The cuDNN (CUDA Deep Neural Network) also allows the use of optimized GPU-accelerated primitives for the use and training of deep neural networks.

Torchvision is a library that is part of PyTorch. The Torchvision library includes an array of popular datasets, model architectures and image transformations. Some model architectures can also be fetched that have been pre-trained on popular datasets such as ImageNet [27] or COCO [28].

2.6 Related Work

This section will cover the related work that has had major influence on the area of study that this thesis exists within. The works are presented in chronological order in respect to publication date.

2.6.1 Alvinn: An autonomous land vehicle in a neural network (1989)

Pomerleau [4] is one of the first instances of autonomous driving using an artificial neural network. The fully-connected feed-forward network consisted of an input layer, an output layer and a hidden layer, totalling only three layers. The input to the network where divided into three main types:

- **30x32 Video Input Retina:** Video camera input from a road scene, where the activation level is proportional to the intensity of the blue color band of the corresponding path of the image. Blue was chosen since it provided the highest contrast between road and non-road.
- **8x32 Range Finder Input Retina:** Laser range finder input, where the activation level is proportional to the proximity of the corresponding area in the image.
- **1x1 Road Intensity Feedback Unit:** Indicates whether the road is lighter or darker than the non-road in the previous image.

These values totalled 1217 input units to the neural network, which ultimately outputs into 46 units where 45 of these is a linear representation of the turn curvature which the vehicle should follow in order to move towards the center of the road. The middle units represented no turn, and left and right from the center represented successively sharper left and right turns.

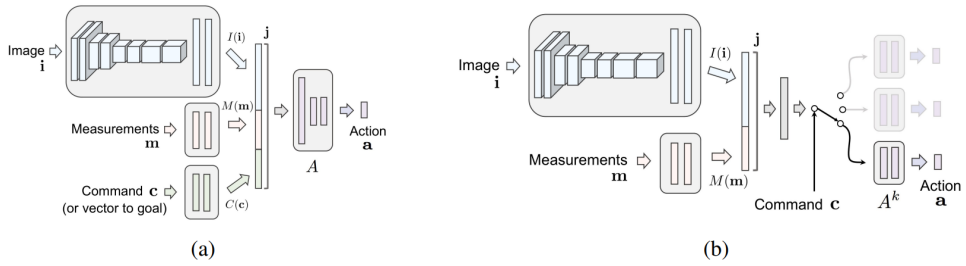
The system was trained for 40 epochs on 1200 data samples, and managed to output turn curvatures which were within two units of the correct answer approximately 90% of the time on test examples. ALVINN ultimately managed to accurately drive 400 meters through a wooded area under sunny fall conditions. Due to the limited processing power of the on-board Sun computer that ALVINN was equipped with [4], it could only drive at a speed of 0.5 meters per second.

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

Codevilla *et al.* [29] proposes conditioning imitation learning on high-level command input, and introduced a novel imitation learning approach called conditional imitation learning, which learns a driving policy that acts as a chauffeur that can handle sensorimotor coordination, but continues to act upon given high-level navigational commands. The approach aims to remove the ambiguity in situations that present multiple actions, i.e. arriving at an intersection where one can choose between multiple directions (straight, left, right). The work presented by Codevilla *et al.* [29] has ultimately had a large impact in the field of end-to-end learning for autonomous vehicles, and has been used in several projects [2] [30] [3].

In order to remove ambiguity from traffic situations, each observation that is to be fed into the agent is also accompanied by a high-level command, which specifies a navigational direction that the agent is expected to move towards. During training, each data point in the training set therefore consists of an observation, a high-level command (encoded as a one-hot vector), and a target action. This

does require the system to have a separate navigational resource to constantly feed navigational commands to the agent at any given time outside of training.



Source: [29]

Figure 2.7: Figure showing the two network architectures for command-conditional imitation learning presented in Codevilla *et al.* [29], where (a) shows the architecture where the high-level command in processed as input by the network along with the image and other measurements. (b) shows the branched network, where the command is used as a switch to select which sub-part of the network the image and measurement data will be passing through.

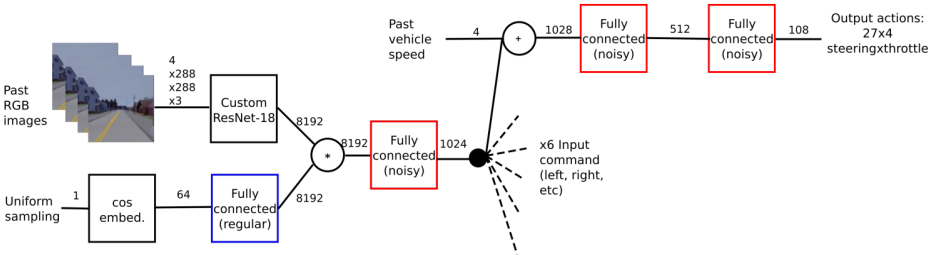
Codevilla *et al.* [29] explores two different network architectures in regards to how an end-to-end system should handle a high-level command input, these two networks are shown in figure 2.7. Figure 2.7 (a) shows the *command input network* architecture which handles an image, measurements and a high-level command as input in separate modules, from which the results are concatenated into a single vector, to be processed further in the network. 2.7 (b) shows the *branched network* handles image and measurements as input in separate modules, and the high-level command as switch to different branches, where each branch represents and is specialized to perform a single high-level command.

After testing the different network architectures in both a real world scenario using a toy truck, and in the CARLA simulator, Codevilla *et al.* [29] showed that the *branched network* performed notably better than the *command input network*. The results also showed that performing augmentations on the training data, such as augmenting image hue, contrast and brightness, randomly dropping pixels and applying Gaussian blur and noise, notably improved the agent's performance and generalization.

2.6.3 End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances (2019)

Toromanoff *et al.* [30] presents an RL algorithm for urban driving that, at the time, was the first RL algorithm to successfully handle the complex nature of the CARLA environment, being able to handle lane keeping, pedestrians, vehicle avoidance and traffic light detection. Toromanoff *et al.* [30] also managed to win the track "Camera Only" in the CARLA challenge.

Toromanoff *et al.* [30] use a large custom ResNet-18 encoder which is trained using supervised learning from data collected from driving around in the CARLA simulator using CARLA's autopilot. The encoder is trained using loss from tasks relevant to autonomous driving, such as semantic segmentation, traffic light presence, intersection presence and lane positioning.



Source: [30]

Figure 2.8: Figure showing overview of the end-to-end model-free reinforcement learning for urban driving using implicit affordances (MaRLn) architecture.

During RL training, the encoder's weights are frozen, and the output from the encoder is what ultimately is used to train the rest of the network, including its conditional branches (similar to Codevilla *et al.* [29]). Figure 2.8 shows an overview of the architecture used in Toromanoff *et al.* [30].

2.6.4 Learning to Drive From a World on Rails (2021)

Chen *et al.* [2] presents the model-based approach "Learning to drive from a world on rails", which at its time of publication achieved the best score on the CARLA leaderboard, using 40 times less training data than the next ranking model, "End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances" [30]. It also managed to achieve new state of the art results on the NoCrash benchmark. The authors also shows the approach's ability to generalize by achieving good results and still being sample-efficient on the ProcGen benchmark.

Chen *et al.* [2] starts with training a forward model of the world which is used to simulate the actions that an agent takes, without the them actually being executed in CARLA. The forward model is used to estimate an action-value function via the use of Bellman equations. This action-value function is then used to supervise a visuomotor driving policy, which only takes RGB images and the current agent speed as input.

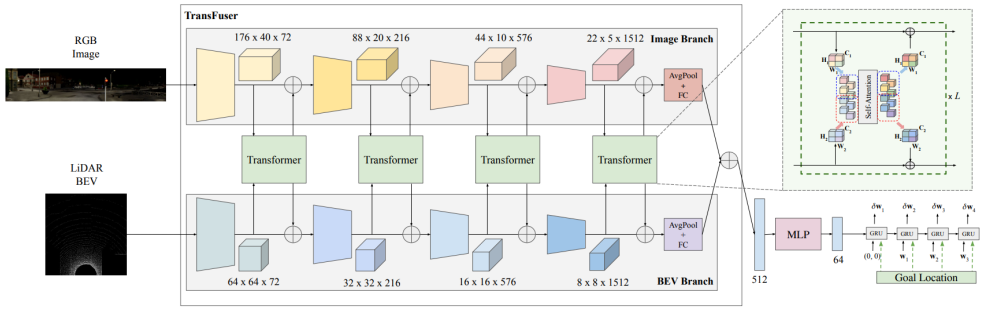
A main part of the algorithm presented in Chen *et al.* [2], is the assumption that "the world is on rails". This means that the actions the agent chooses, does not influence the environment, but only itself (ego). With this assumption, the state space is significantly simplified, meaning that the action-value function can be stored in a tabular form, and that the use of Bellman equations with a learned

forward model is adequate.

As this thesis builds upon the work of World on Rails [2], a more descriptive explanation of the algorithm can be found in section 3.3.

2.6.5 TransFuser (2022)

Chitta *et al.* [31] investigates how information from multiple sensors should be fused together in end-to-end driving systems. Chitta *et al.* [31] introduce a mechanism called TransFuser, which integrates RGB images and LiDAR representations using transformer modules at multiple resolutions. This allows for a perspective view (RGB image) and a bird's eye view (LiDAR scan) feature maps to be fused together using self-attention. An overview of the TransFuser architecture can be seen in figure 2.9.



Source: [31]

Figure 2.9: Figure showing an overview of the TransFuser architecture.

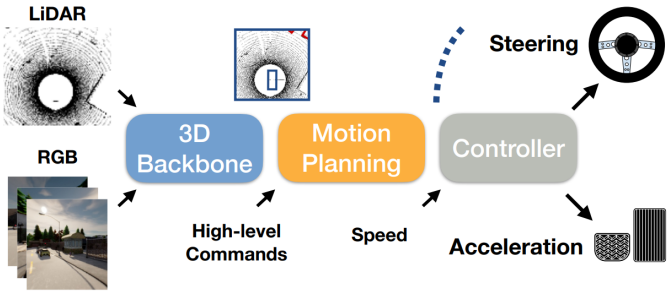
The TransFuser architecture consists of two main branches, with transformer modules connected between them for intermediate information to be shared between the two branches. From figure 2.9, we also see that this fusion is applied at different resolutions, and is ultimately combined to a 512 dimensional vector and passed on a multi-layer perceptron network, before ultimately being passed on to a waypoint prediction network.

At the time of its original submission, TransFuser outperforms significantly outperformed other models on the CARLA leaderboard in terms of driving score, and at the time of writing currently stands among the top three on the CARLA leaderboard. The paper also introduce a new challenging benchmark *Longest6*, which uses long routes with high traffic density and pre-crash scenarios [31].

2.6.6 Learning from All Vehicles (2022)

Learning from All Vehicles (LAV) [32] introduces a driving system, that during training not only trains on the experiences from the ego-vehicle, but also all the other vehicles that it observes. Without using the other vehicles' sensors, the sys-

tem trains on predicting trajectories of the other vehicles in the ego-vehicle vicinity. An overview of a LAV agent's inference pipeline can be seen in figure 2.10.



Source: [32]

Figure 2.10: Figure showing an overview of the inference pipeline to a LAV agent.

Chen and Krähenbühl [32] aims to train a deterministic driving model that outputs control commands based on RGB images, LiDAR scans, high-level commands, and speed. The agent's pipeline consists of three main modules; perception module, motion planner, and low-level controller. The perception model aims to learn a mapping between a combination of RGB images and LiDAR scans, and a two-dimensional map-view representation of the near vicinity of the ego-vehicle. The motion planner uses these map-view's to predict the ego-vehicles future trajectory, but does also, during training, predict the future trajectories for close-by vehicles.

At the time of their submission, LAV was at first place, but at the time of writing is currently ranked second best on the CARLA leaderboard, with still the best route completion of any other entry.

Chapter 3

Methodology

This chapter covers the methodology of the project. The first section will cover the technological choices and the resources used in this thesis. Section 3.2 will cover some changes made in terms of software versions, and section 3.3 will give a deeper description of the World on Rails [2] algorithm. The last two sections will cover the experiments conducted in this thesis.

3.1 Tools and Resources

This section will cover some of the technological choices made in regards to the project, this includes the choice of simulator, the machine learning framework, and hardware resources.

3.1.1 Software

Simulator

The simulator chosen for this project was the CARLA driving simulator [1]. The main reasoning behind this choice was that this project builds upon the work of Chen *et al.* [2], which does part of their experiments in the CARLA simulator. The simulator is also built with a focus on training and validating autonomous driving systems, and provides several free assets such as car models, pedestrian models, buildings, and maps which can be used for creating realistic simulated environments.

Machine Learning Framework

For the machine learning framework, PyTorch was chosen. PyTorch was chosen due to it being the main machine learning framework utilized in the World on Rails [2] project. PyTorch is also a highly popular framework within the research community, and allows the use of Torchvision, which contains pre-made models that can be fetched as either non-pre-trained or pre-trained on popular datasets.

3.1.2 Hardware

For this thesis there were four hardware resources available:

- **Personal laptop** with an Intel Core i5-8265U CPU, 8GB RAM, and an Nvidia GeForce GTX 1650 GPU with 4GB VRAM.
- **Personal desktop** with an AMD Ryzen 5 3600 CPU, 32GB RAM, and an Nvidia Geforce RTX 3060 Ti GPU with 8GB VRAM.
- **Virtual machine** hosted by IDI Horizon with an Intel Xeon Gold 6342 CPU, 51GB RAM, and an Nvidia A40-16Q GPU with an allocated VRAM amount of 16GB.
- **Idun cluster** hosted by NTNU High Performance Computing Group. Specifications for this resource can be found on the Idun cluster webpage (`https://www.hpc.ntnu.no/idun/`)

Due to the resource-demanding task of utilizing the CARLA driving simulator, and using the models and methods used in the World on Rails [2] algorithm, none of the personal resources were utilized. As the World On Rails [2] repository proved to be difficult to implement by itself, only the Virtual machine hosted by IDI Horizon was utilized to produce the results in this thesis. Additionally, for ease of implementation and not having to abide by the queuing times that are expected when using the Idun cluster, we did not utilize the Idun cluster. The virtual machine hosted by IDI Horizon had approximately 540GB of local storage, and a 10TB separate storage, which allowed for multiple large datasets to be stored and moved simultaneously.

3.1.3 Working Environment

Throughout the span of the experiments being performed, we mainly utilized the IDI Horizon VM. By using the VMware Horizon Client, we could access the VM with a desktop GUI from any device connected to the NTNU campus network, or connected to the network via VPN. We also had the ability to access the VM via secure shell (SSH) using a Ubuntu terminal.

For management of packages and libraries for our python projects, we also created conda environments using Anaconda [33]. Additionally, we used the open-source code editor Visual Studio Code (VS Code) [34] as it did make navigating through large projects easier, as well as being compatible with conda environments.

During training and evaluation of the agents throughout this project, we also used the machine learning platform known as Weights & Biases (W&B or wandb) [35], which allowed us to store relevant data at given steps throughout processes. Information could be easily accessed either via local files on the VM, or online at W&B's webpages[1]. Using W&B, we were also able to record videos from the agent's perspective during evaluation in the NoCrash benchmark.

[1] `https://wandb.ai/`

3.2 Converting to PyTorch 1.10.2 and CUDA 11.3

As both the virtual machine 3.1.2 and the personal desktop 3.1.2 both utilize GPUs built with Nvidia's Ampere architecture which initially requires applications to be built using CUDA version 11.0 or later [36]. The `environment.yml` in the World on Rails GitHub repository specifies that the project requires the use of CUDA version 10.1, and PyTorch version 1.4.0. The latest CUDA version that PyTorch 1.4.0's is compiled against is CUDA 10.1 [37].

Using PyTorch 1.4.0 with CUDA 10.1 initially created issues related to running the World on Rails [38] project. The specific issues that occurred were not investigated in-depth or documented. The choice to move versions from PyTorch 1.4.0 with CUDA 10.1 to 1.10.2 with CUDA 11.3, was therefore made to circumvent these issues and to also potentially improve performance. Some other python package versions were also changed. Changes made in terms of this conversion can be viewed in our GitHub repository [2]. A list of package versions used for this project can be found in appendix A.

Using CARLA version 0.9.10 on a GPU with the Ampere architecture with CUDA 11.3 did prove to have some issues. First being that the simulator would randomly crash, which required regular supervision to ensure that processes had not stopped, and to restart them if they had. Secondly, rendering in the CARLA simulator launched with `-quality-level=Low` (lower graphical settings) showed a lot of artefacts in the form of black squares, but was not as present in the simulator when launched with `-quality-level=Epic` (higher graphical settings). The artefacts did however also occur on other sensor data collected from the CARLA simulator, such as the semantic segmentations, even with `-quality-level=Epic`. An example of the artefacts occurring in the renderings can be seen in figure 3.1. To avoid having the visuomotor agent learn with RGB images containing many artefacts, CARLA was only launched with `-quality-level=Epic`.

3.3 World on Rails

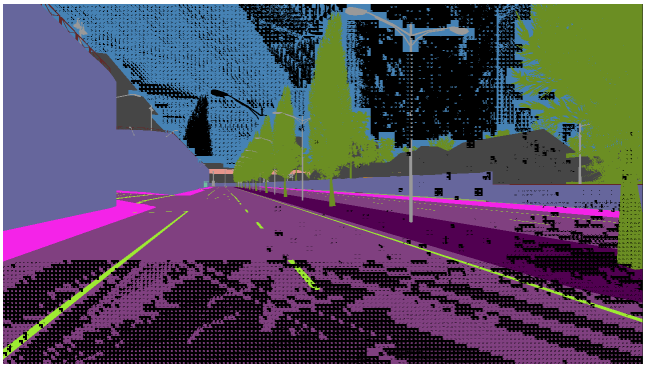
This section describes the World on Rails algorithm in more detail. The algorithm can be divided into five sequential operations:

- **data_phase0:** Collect data in the CARLA simulator that is to be used in the training of the forward model. The resulting dataset is a small subset of trajectories consisting of 2400 frames. This dataset is generated by an agent performing random actions.
- **train_phase0:** Train the forward model with the dataset collected in the previous phase, `data_phase0`. The forward model takes as inputs an action a_t from an agent and a current driving state L_t in the form of a two-dimensional location x_t, y_t, orientation θ_t, and speed v_t, and outputs the next driving state L_{t+1} as $x_{t+1}, y_{t+1}, \theta_{t+1}, v_{t+1}$.

[2] `https://github.com/MorningClub/master-thesis`

(a) Low settings

(b) High settings



(c) Semantic segmentations

Figure 3.1: Figures showing how renderings in CARLA would have artefacts on different graphical settings, and how the semantic segmentations would also have artefacts. The semantic segmentation image was captured in CARLA launched with `-quality-level=Epic`.

- **data_phase1:** Collect data for the visuomotor training by using a privileged autopilot driving around on a predefined set of routes in the CARLA simulator, collecting RGB images, semantic segmentations, high-level commands, and labels in the form of proximal information such as pedestrians, vehicles, waypoints etc.
- **data_phase2:** Label each data frame in the collected `data_phase1` using bellman equation evaluation and the forward model from *phase0*, where labels come in the form of action value tables for steering and throttle, and one additional bin for braking.
- **train_phase2:** Train a visuomotor agent on the dataset from `data_phase2`, with an additional semantic segmentation loss.

For the CARLA leaderboard, Chen *et al.* [2] collected one million data frames in `data_phase1` which corresponded to 69 hours of simulation time. The dataset was collected across the 8 publicly available towns in CARLA. The RGB images collected consist of front-facing wide images and front-facing narrow images that are collected from cameras placed at $x = 1.5$m and $z = 2.4$m on the ego-vehicle. Examples of these images can be seen in figure 3.2. The wide image is an im-

age stitched up from three individual cameras with a 60°field of view, where the side cameras are angled at 55°angles. The narrow image comes from a telephoto camera, with a 50°field of view. The main reasoning behind using an additional telephoto camera is that some cities in CARLA's public towns have intersections with traffic lights placed on the other side of the intersection, which can be hard to distinguish in the wide camera. Figure 3.2 shows how even in dark-lit scenarios, traffic lights from across an intersection can be hard to distinguish in the wide image. For dataset augmentation, additional cameras are mounted with a similar setup as the four previously mentioned, however, these cameras are rotated 30°, giving each data frame in the CARLA leaderboard dataset a total of six RGB images. These three different camera orientations will be referred to as the three camera yaws.

(a) Wide image

(b) Narrow image

Source: [2]

Figure 3.2: Figure showing images from the wide (3.2a) and narrow camera (3.2b). The images are extracted from the one million frames World on Rails dataset [2].

For the NoCrash benchmark, Chen *et al.* [2] collected 270 thousand frames. These frames were collected in the training town *Town01*, and the model was tested in both *Town01* and *Town02* in different weather as is the procedure in the NoCrash benchmark. Since *Town01* and *Town02* don't contain any intersection with agents having to read traffic light states on the other side of the intersection,

the model in the NoCrash benchmark does not utilize the telephoto camera, and therefore only takes as input the stitched-together wide images.

3.3.1 Bellman Evaluation

The main goal of `data_phase2`, is to compute action-values that will be used to supervise the visuomotor training. Visualisations of the computed value functions and action-value functions for given frames can be seen in figure 3.3. The value function V_t is a $N_H \times N_W \times N_v \times N_\theta$ four-dimensional tensor, where $N_H \times N_W$ corresponds to the bins that are the discretized 24m^2 area surrounding the vehicle, where $N_H = N_W = 96$. N_v are the velocity bins, that are in the range $[0, 8]$ m/s where $N_v = 4$. N_θ are the orientation bins, and are in the range $[-95, 95]$, where $N_\theta = 5$. Value maps were also generated using 5 Bellman updates, predicting 1.25s into the future.

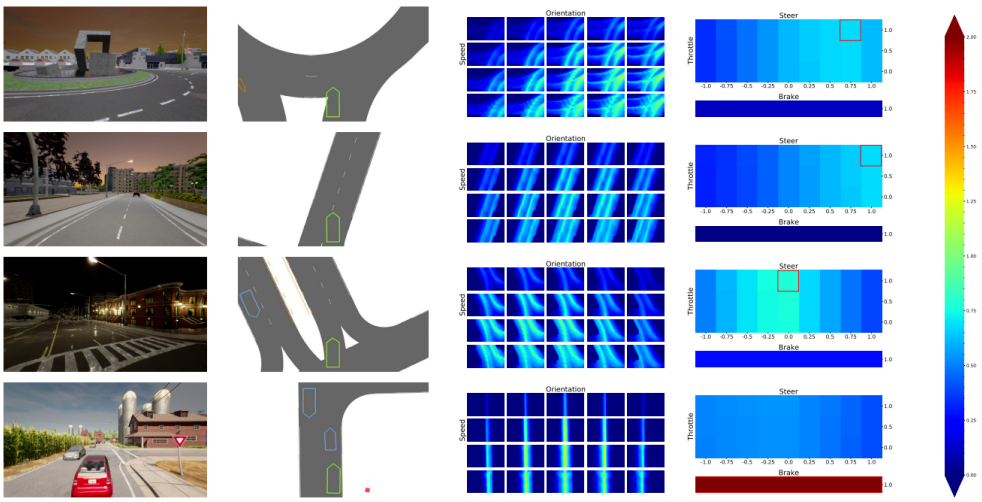
The actions for the visuomotor agent have also been discretized into $M_s \times M_t$ table of bins, with an additional bin for braking. Using $M_s = 9, M_t = 3$, the number of actions the agent can choose from are 28, which includes the braking action. Using the values from the value function, they generate action-values for each of the 28 bins as shown in figure 3.3d, implying that throttle and steering are chosen in pairs, and no other action is performed when the agent chooses to brake. During `data_phase2`, action-value tables were stored for each of the six different high-level commands and the four speed bins covering the speed range $[0, 8]$ m/s, and for each of the camera yaws.

3.3.2 Reward Function

The reward function utilized during the Bellman evaluations is a function that considers the current ego-vehicle state, the current high-level command, the world state, and an action. The reward is based upon the following:

- Position, speed and orientation in the target lane. The agent is rewarded +1 for staying in the desired position, orientation and speed in the target lane. Deviation from this desired state smoothly reduces the reward to 0.
- Speed in zero-speed regions (proximity to other vehicles and pedestrians, or red light at intersections). The agent is rewarded for standing still in zero-speed regions. In red light regions, the agent is also penalized for not keeping orientation.
- Braking in zero-speed regions. Agent receives +5 for braking in zero-speed regions.

Rewards from zero-speed regions are scaled by a factor $r_{stop} = 0.01$, so rewards from staying in the target lane are not overshadowed. Chen *et al.* [2] mention that they restrict the braking reward in the sense that they can not be accumulated, which they argue avoids agents chasing braking zones. Their reward design using zero-speed and braking zones also removed the necessity to implement any penalty for collision.



(a) RGB camera (b) Map (c) Value map (d) Action-value

Source: [2]

Figure 3.3: Figure showing a visualisation of the computed value functions and action-value functions for given data frames. The figure shows the RGB images (a), bird's-eye view maps of the area surrounding the ego-vehicle (b), the value maps (c), and the action-values for the current frame (d). In the maps the ego-vehicle is centered, but for visualisation the area behind the ego-vehicle has been removed.

3.4 Experiment 1: Reproducing the results from World on Rails in the NoCrash benchmark

This section will introduce the first experiment of this thesis, which involved reproducing the results presented in the World on Rails paper [2]. For additional information regarding the World on Rails [2] algorithm, see section 3.3.

3.4.1 Data Collection

In the World on Rails [2] algorithm, there are two phases where data is collected in the CARLA simulator. The initial data collection, referred to as `data_phase0`, and the second data collection, referred to as `data_phase1`. There is also an additional phase which labels the data collected in `data_phase1` with action-values, and this phase is referred to as `data_phase2`.

Collection in data_phase0

In `data_phase0`, data was collected in *Town04* using an agent performing random actions. The actions consisted of performing steering or throttling simultaneously, or just braking. Steering and throttle were uniformly sampled between ranges [-

1,1] and [0,1], respectively. The chances of braking P_B were based on the following equation:

$$P_B = \frac{1}{M_s \times M_t + 1},$$

where M_s and M_t respectively denote the number of steering actions and throttle actions, using $M_s = 9$, $M_t = 3$. In this phase, the agent's trajectories are collected in the form of positions, orientations, speeds and actions. For this phase, we collected 2079 frames, in comparison to the original amount of 2400 that the World on Rails paper [2] collected.

Collection and Labeling in data_phase1 and data_phase2

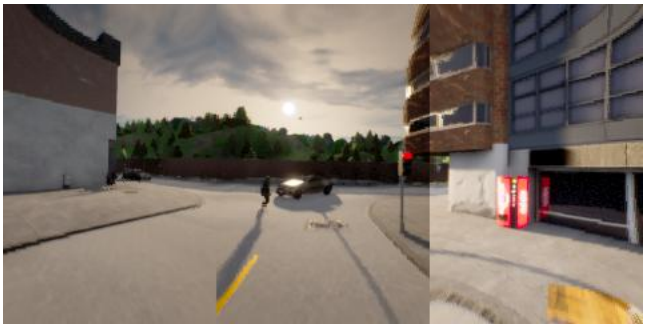
For `data_phase1` we collected in *Town01*, which is the training town for the NoCrash benchmark. The agent used for collecting data is the `q_collector` agent, which uses privileged information from the CARLA simulator and the *forward model* to calculate the best action using Bellman equation evaluations. During `data_phase1` we collected wide RGB images, their semantic segmentation, the current high-level command, and labels which were used for Bellman evaluation.

The wide RGB images consist of three side-by-side images collected from three cameras mounted at $x = 1.5$m, $z = 2.4$m on the ego-vehicle, producing images that are 480×240 pixels. One camera is pointed straight forward, and the two others are rotated 55°in each direction from the straight forward-facing camera. Each camera has a field of view of 60°. An example of a wide RGB image can be seen in figure 3.2a. As described in Chen *et al.* [2], for data augmentation purposes, there are also additional cameras set up to take similar wide images, but rotated in ±30. These three rotations are referred to as the three camera yaws.

The labels collected include information that is proximal to the agent. The information is from a birds-eye view, where the agent is placed in the center. These labels are roads, lanes, stops, vehicles, pedestrians, traffic and waypoints. An example of the road, pedestrian and vehicle labels can be seen layered on top of each other in figure 3.4b.

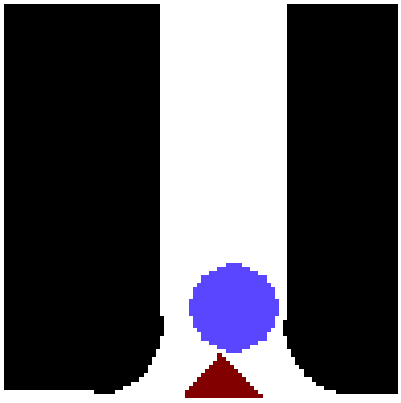
The total amount of data frames collected in `data_phase1` was 105 503, which is significantly lower than the original World on Rails paper [2] which used 270 000 frames for their NoCrash benchmark agent. We decided to use a lower amount due to the performance limitations of the virtual machine (3.1.2) and the general instability of the CARLA simulator. With 16GB of VRAM, we could have two parallel data collection processes running simultaneously. In terms of single process performance, we got a simulation time to wall-clock time ratio of approximately 0.15 to 0.20, meaning that simulating one second in the simulator took anywhere from 5 seconds to 6.7 seconds. With simulator crashes, collecting 100 000 frames would take approximately 4 days.

During `data_phase2` each data frame from the dataset collected in `data_phase1` was labeled with action-values using Bellman equation evaluations (as described



Source: [2]

(a) Wide image



(b) Stacked road, vehicle and pedestrian labels

Figure 3.4: Figure showing a wide RGB image 3.4a, and the stacked road, vehicle and pedestrian labels for a given data frame. In 3.4b, the agent's position is at the center. The pedestrian label is coloured blue, vehicle is coloured red, and the road is white.

in 3.3.1). Labels stored for each data frame come in a data structure with the dimensions $6 \times 4 \times 28$, which corresponds to the six different commands that can be active in a given moment, the four speed bins that the agent could be in, and the 28 different actions that the agent can choose between, and labels were created for each of the three camera yaws. These dense labels allow for the algorithms' high sample efficiency.

The original World on Rails project does not store labels to the dataset until all frames had been labelled, this was not possible with the limited 51GB of RAM the virtual machine (3.1.2) had, and the labelling process therefore had to be split up into 50 thousand frame batches.

3.4.2 Network Architectures

This section will cover the network architectures of the two networks used in the recreation of the World on Rails [2] results. The first being the forward model, which is used to predict future states based on a current world state and an action. The second network is the visuomotor network, which is the resulting autonomous driving agent.

Forward Model

As the forward model τ_{ego} was a key component during the prediction of future states for each data frame collected, a simple model needs to be utilized to make quick and accurate predictions. Building on the work of Polack *et al.* [39], World on Rails [2] use a parameterized kinematic bicycle model of a vehicle, where the front and rear wheels are in theory collapsed into one front and one rear wheelbase (f_b, r_b) at the front center and rear center of the vehicle. When assuming that only f_b can be steered, the kinematics of the bicycle model can then be written as:

$$\frac{dx}{dt} = vcos(\theta + \beta) \tag{3.1}$$

$$\frac{dy}{dt} = vsin(\theta + \beta) \tag{3.2}$$

$$\frac{dv}{dt} = a \tag{3.3}$$

$$\frac{d\theta}{dt} = \frac{v}{r_b} sin(\beta) \tag{3.4}$$

$$tan(\beta) = \frac{r_b}{f_b + r_b} tan(\phi) \tag{3.5}$$

where a is the acceleration corresponding to either the throttle action multiplied by a constant t_{accel}, or the breaking action multiplied by a constant b_{accel}, based on if the vehicle is throttling or braking. ϕ is the front wheel steering angle, which is the applied steering action s multiplied by a steering constant s_{gain}. $t_{accel}, b_{accel}, s_{gain}, r_b$ and f_b are the learnable parameters of the model, and loss is generated using L1 loss and stochastic gradient descent, based on the predicted new two-dimensional position $(\hat{x}_t, \hat{y}_t)$, and predicted orientation $\hat{\theta}_t$ of the vehicle.

Visuomotor Network

World on Rails' NoCrash visuomotor network consists of a ResNet34 visual backbone, a conditionally branched action head, and a semantic segmentation head for generating the additional semantic segmentation loss. An overview of the NoCrash system can be seen in figure 3.5.

In comparison to World on Rails' [2] leaderboard agent, which uses an additional ResNet18 visual backbone to parse images from the telephoto camera which

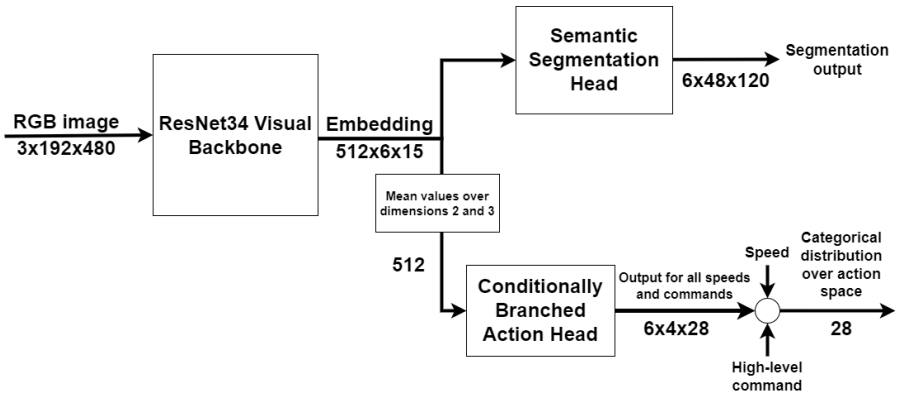


Figure 3.5: Figure showing the visuomotor network architecture.

is concatenated with the output from the ResNet34, the NoCrash agent only uses the ResNet34 visual backbone to parse the wide RGB images it takes as input. The ResNet34 backbone is pre-trained on the ImageNet [27] dataset prior to training. The pre-trained model is fetched from Torchvision [26]. The original wide RGB image has the dimension $3 \times 240 \times 480$, but following the configuration of World on Rails, is cropped by 48 pixels from the top of the image, making the input to the ResNet34 have the dimension $3 \times 192 \times 480$. The output of the ResNet34 backbone is an embedding following the dimension $512 \times 6 \times 15$, and is passed further on to both the semantic segmentation head and the conditionally branched action head.

The semantic segmentation head takes as input the embedding from the ResNet34, and by using transpose convolutions and regular convolutions, outputs a $6 \times 48 \times 120$, which corresponds to the 6 labels that the network trains semantic segmentation on. These 6 labels are pedestrians, road lines, road, vehicles, traffic lights and "other". Before generating loss by comparison with ground truth semantic segmentation images, the $6 \times 48 \times 120$ semantic segmentations are scaled up to $6 \times 48 \times 120$ using interpolation.

From the ResNet34 visual backbone, the image embedding is first collapsed on the second and third dimensions, by taking the average values across these dimensions. The resulting 512 parameter vector is then passed on to the conditionally branched action head, which ultimately outputs a categorical distribution over the action space, similar to the action-value table as presented in figure 3.3d, for each of the six high-level commands and four speed bins, resulting in a $6 \times 4 \times 28$ data structure. Outside of training, the agent requires the current high-level command and the agent's current speed, to choose the appropriate categorical distribution.

3.4.3 Visumotor Network Training

The visuomotor network was trained on the labelled dataset with 105 503 data frames. With three different camera yaws, this amounted to $3 \times 105\,503$ instances

that the network trained on. For each training instance, the network was trained on both semantically segmenting the wide RGB images with the semantic segmentation head, and generating categorical action distributions from the action head. For training, we used the Adam optimizer with a learning rate of 0.0003 over the course of 20 epochs. For an overview of the training parameters used during the training of the visuomotor network, see appendix B.

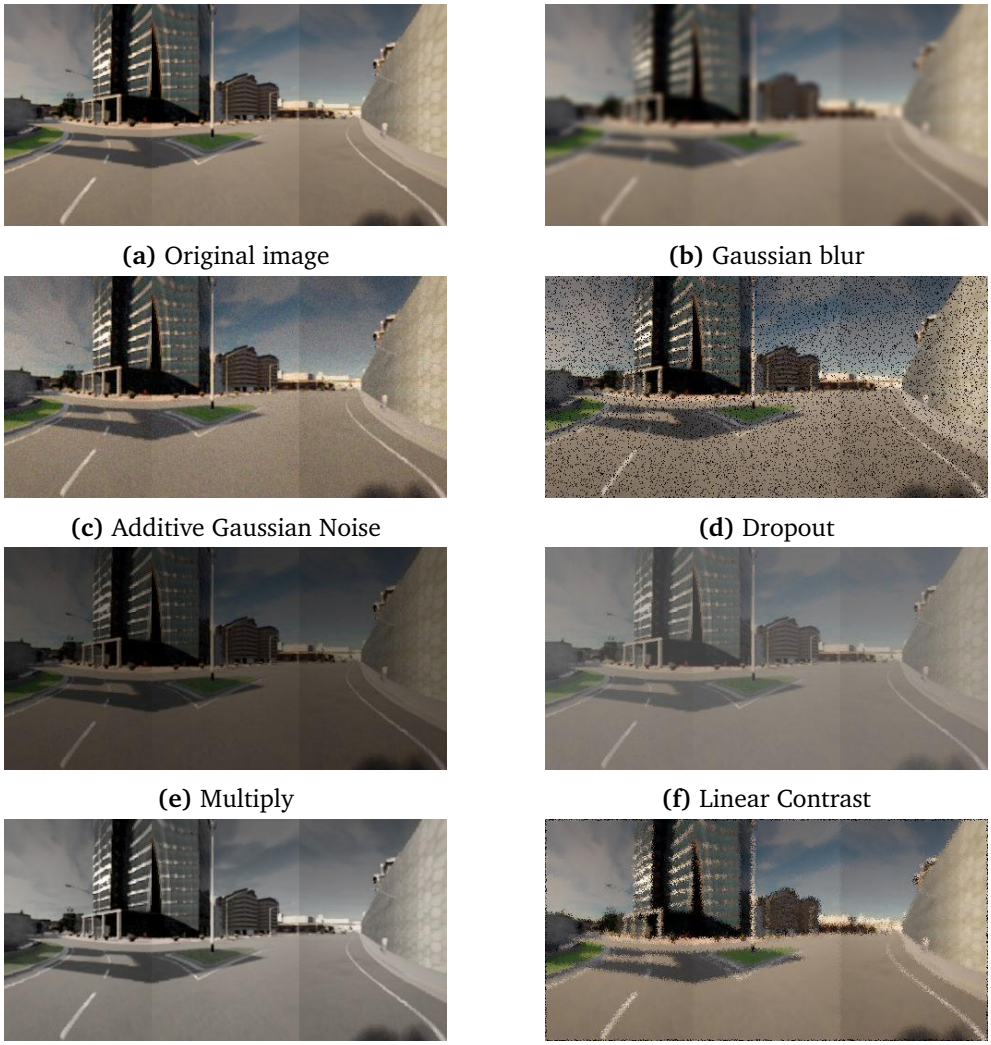
Image augmentations were also performed on the data similarly to the Learning by Cheating [3] algorithm. Augmentations were performed using the *imgaug* [40] python library. These augmentations were:

- **Gaussian Blur:** Blurs images using gaussian kernels.
- **Additive Gaussian Noise:** Adds gaussian noise to an image.
- **Dropout:** A fraction of the images pixels in an image are set to the value zero.
- **Multiply:** Multiplies all pixels in an image with a specific value, which makes the images lighter or darker.
- **Linear Contrast:** Adjusts the contrast by scaling each pixel in an image.
- **Grayscale:** Grayscales an image. Given a certain percent range α, the original image can also be overlaid with a random percent of strength within the range of α.
- **Elastic Transformation:** Moves pixels around in local areas using displacement fields.

Each image is passed through these augmentations in a random order sequence. There is a 50% chance of each of these augmentations being applied to the image, meaning that also none of the augmentations can be applied. The different augmentations can be shown applied in figure 3.6. As the Resnet34 visual backbone is pre-trained on images from the ImageNet [27] dataset, images are also normalized using the mean and standard deviation values calculated from ImageNet [27] images, commonly referred to as ImageNet normalization. The values for mean and standard deviation during ImageNet normalization are respectively $mean = [0.485, 0.456, 0.406]$ and $std = [0.229, 0.224, 0.225]$, where the three floating point numbers in each list correspond to the three RGB channels.

The collapsed 512 parameter vector embedding from the ResNet34 image embedding was taken as input to the conditionally branched action head, which output categorical action distributions for each of the high-level commands, and each of the four speed intervals that the agent could be in. As labels were categorical action distributions for each high-level command, and speed interval, we used *Kullback-Leibler divergence* to calculate loss between targets and predictions.

The output from the semantic segmentation head was scaled up by a factor of four, using interpolation, prior to being compared to the ground truth semantic segmentations. We used the cross-entropy loss function to generate loss for the task of semantic segmentation. The semantic segmentation loss was also scaled by a weight factor $seg_{weight} = 0.05$ before being added to the action loss and backpropagated. As mentioned in section 3.2, due to artefacts occurring in the



(a) Original image **(b)** Gaussian blur

(c) Additive Gaussian Noise **(d)** Dropout

(e) Multiply **(f)** Linear Contrast

(g) Grayscale **(h)** Elastic Transformation

Figure 3.6: Figure showing different types of image augmentations. For visualisation purposes the augmentations in sub-figures 3.6b, 3.6e and 3.6f were given more extreme augmentation values than what was used during the training of the visuomotor network.

semantic segmentation images collected from CARLA, the dataset generated in `data_phase1` collected semantic segmentation images with these artefacts. The semantic segmentation head did also learn to predict these artefacts during `train_phase2`. An example of these predictions can be seen in figure 3.7.

The visuomotor network was trained over the course of 20 epochs, and a model was saved after each epoch. Due to time limitations, and the time it takes took evaluate a single model, the decision was made to qualitatively inspect the performance of the model saved at epoch 10, and epoch 20. The two models were

(a) Ground truth **(b)** Prediction

Figure 3.7: Figure showing the ground truth semantic segmentation 3.7a, and the visuomotor network predictions 3.7b with predicted artefacts.

evaluated on the map *Town01* with testing weather as used in the NoCrash benchmark. The model from epoch 20 showed better driving capabilities than the model from 10, and was therefore chosen for further benchmarking.

3.5 Experiment 2: Integrate Semi-Supervised Learning into World on Rails

As a second experiment, we propose an additional autoencoder training step, that uses a semi-supervised approach to pre-train a visual encoder backbone that is used to encode RGB images into a condensed representation. The visual encoder is used in two separate implementations, where one utilizes the encoder with frozen weights, and the other with unfrozen weights.

3.5.1 Dataset for Semi-Supervised learning

We trained the autoencoder on a subset of the World on Rails one million data frames dataset [2] which is publicly available via the World on Rails GitHub repository. The dataset contains all the one million frames that were collected for the training of their CARLA leaderboard agent, which was collected across 8 publicly available towns for the CARLA simulator. Due to time limitations, and processing limitations, the decision was made to only use a subset of the dataset, which corresponded to one-third of the dataset.

As one data frame contains images in three different camera orientations, each data frame effectively has three training instances for the autoencoder to train on, meaning that when we sampled one-third of the dataset, we sampled one million training instances with their corresponding semantic segmentation, from a set of three million training instances. The subset was randomly selected using a set seed. We also selected a random subset of 15 000 training instances for validation.

It is important to note that from the one million training instances extracted, we did not actively filter out training instances that were from *Town02*, which puts into question whether or not *Town02* should be considered a testing town or not. Ideally, we would have these training instances filtered out, but did not find this to be realistic to perform due to time limitations as the dataset did not

contain labels pertaining to which towns the training instances are from. We also did not filter out training instances with certain weathers, as the dataset also did not contain labels pertaining to current weather.

3.5.2 Autoencoder Architecture

The encoder used in both implementations was taken from the same autoencoder consisting of three modules, the encoder, decoder, and a semantic segmentation head. An overview of the Autoencoder architecture can be seen in figure 3.8.

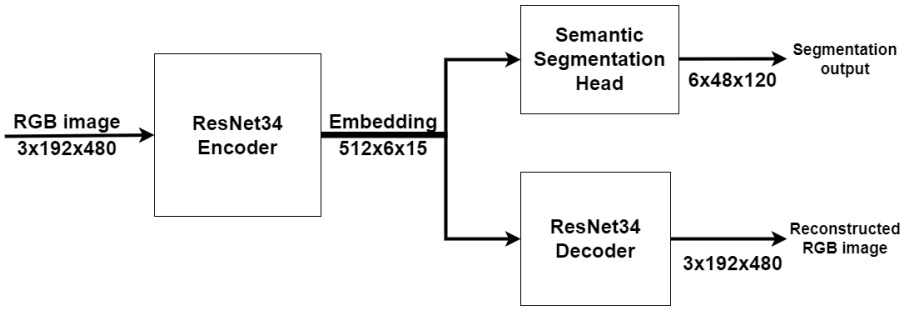


Figure 3.8: Figure showing the autoencoder architecture.

Taking inspiration from the visuomotor agent from 3.4, we used a similar ResNet34 network as our encoder module, taking in a wide RGB image as input, and embedding it into a $512 \times 6 \times 15$ latent space. The encoder is also fetched as pretrained on the ImageNet [27] dataset from Torchvision, but since we were fitting the encoder to another dataset which the images we had not calculated the new mean and standard deviation values for, we normalized RGB values in the range $[0, 1]$ by simply dividing by the maximum colour value, 255. This also implies that we used the same normalisation for later stages for all implementations in this experiment.

With the intention to fit more knowledge in the encoder module, we also use the same semantic segmentation head as was used in the visuomotor training phase in 3.4, however, did not weigh the semantic segmentation loss during training.

For the decoder module, we decided to use a correspondingly reversed ResNet34, which we originally took inspiration from a publicly available GitHub repository [3] for a variational ResNet18 autoencoder. We did however remove the variational aspects of this decoder architecture and expanded the decoder to be ResNet34. Given that images are normalized between the range $[0, 1]$, the decoder outputs on a last layer that uses a sigmoid activation function. The architecture for the ResNet34 decoder can be seen in PyTorch code form in appendix C.

[3] `https://github.com/julianstastny/VAE-ResNet18-PyTorch`

3.5.3 Autoencoder Training

We trained the autoencoder using the Adam optimizer with a learning rate of 0.0003, across 12 epochs where we stored models at the end of each epoch. For the image reconstructions, we used the *RMSE* loss function to generate loss, which we combined with the loss from the task of semantic segmentation. The task of semantic segmentation was done similarly to the semantic segmentation training that is described in 3.4. Losses were combined without weighting before being backpropagated. We did not weight losses, as we viewed the task of image reconstruction to be relatively easy to learn with ResNet34 modules, and wanted to give more focus on which elements were present in a given image.

There was also an intention to include several other tasks during this training phase, including traffic light state, presence and distance prediction, junction presence and distance prediction, and agent lane positioning and orientation prediction, similar to what was proposed by Toromanoff *et al.* [30]. Due to time limitations, these additional losses were not prioritised to be implemented.

We also performed image augmentations on the wide RGB images prior to being input to the autoencoder. These augmentations are identical to the ones used in 3.4. These augmentations include gaussian blur, additive gaussian noise, dropout, multiply, linear contract, grayscale and elastic transformation. The augmentations are applied in a random sequence on a given image, with a 50% chance of an augmentation being applied.

After each epoch, we stored training losses and validation losses. Additionally, when we validated the model, we stored the input images and reconstructed images with ground truth semantic segmentations and their semantic segmentation predictions, for qualitative inspection of the autoencoder's performance during training. Training loss and validation loss throughout the 12 epochs can be seen in figure 3.10. An input image from validation during epoch 12, and its respective reconstruction along with its semantic segmentations can be seen in figure 3.9. To see all wide RGBs, reconstructed wide RGBs, and semantic segmentations collected at each epoch, see appendix D. Based on the losses recorded throughout training, we decided to use the model from epoch 12, as it showed the best performance and reconstruction capabilities, with no initial sign of over-fitting to the training dataset.

3.5.4 Implementation A: Visuomotor network with frozen visual encoder weights

In the first implementation, we intended to study how well a visuomotor agent using a pre-trained visual backbone trained on highly relevant data performs when the visual backbone has frozen weights. We studied this implementation's performance both during visuomotor training and during evaluation in the NoCrash benchmark.

For the first implementation, we load the weights from the encoder from the autoencoder, and freeze them during the final visuomotor training phase, referred

Figure 3.9: Figure showing input wide RGB, and its ground truth semantic segmentation on the left, and reconstructed wide RGB and predicted semantic segmentation image on the right. Images are taken from the validation step at epoch 12. The predicted semantic segmentation image is shown in its state prior to upscaling using interpolation.

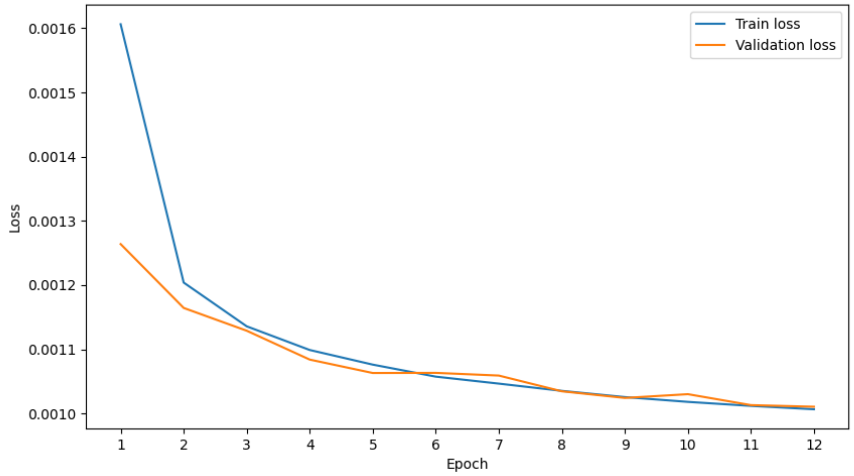


Figure 3.10: Figure showing training loss and validation loss throughout the 12 epochs of training.

to as `train_phase2`. As shown in figure 3.5 in the first experiment 3.4, the image embedding was first collapsed by taking the average values across the second and third dimensions before being sent to the action head. Since weights are frozen in our encoder, we remove this step and rather introduce a latent space head which took in the $512 \times 6 \times 15$ image embedding as input, and output a 512 parameter vector which is passed on to the action head. The latent space head is a simple CNN network, consisting of two convolutional layers with ReLU activation, a flattening operation, and a feed-forward layer with ReLU activation. Following the PyTorch

syntax, the latent space head with its parameters is shown in appendix C.

Since the encoder also is trained on semantic segmentation, we remove the segmentation head from the visuomotor agent in this implementation. The action head remains the same as presented in the first experiment 3.4. An overview of implementation A's visuomotor agent can be seen in figure 3.11.

Figure 3.11: Figure showing the visuomotor architecture in implementation A.

3.5.5 Implementation B: Visuomotor network with unfrozen visual encoder weights

In the second implementation, we intend to study how pre-training a visual backbone on a more specific and relevant dataset, can affect the performance of the visuomotor agent, both during training and during evaluation on the NoCrash benchmark.

For the second implementation, we load the weights from the encoder from the autoencoder but do not freeze them during the visuomotor training phase. We additionally did not remove the task of semantic segmentation during training and did not implement a latent space interpretation head, meaning that for this implementation we performed visuomotor training as described in 3.4, with the only difference being that we did not perform ImageNet normalization.

Chapter 4

Results

This chapter will present the results of the experiments in this thesis. The first section will cover the results related to the visuomotor training for all the experiments and implementations in the form of action loss collected throughout `train_phase2`. In the first section, we also present the success rates and traffic light infraction rates for all the different models trained throughout this thesis, including the results from the original World on Rails paper [2]. For the last two sections, we present more specific results from the NoCrash benchmark for each of the experiments.

For this chapter we will refer to the different models by their acronyms as described below:

- **WOR:** Original World on Rails model trained for the NoCrash benchmark as presented in the World on Rails paper [2]
- **WOR-R:** Our recreation of the World on Rails paper as presented in 3.4.
- **WOR-EF:** Our visuomotor implementation using an visual encoder backbone with frozen weights, as described in 3.5.4
- **WOR-EU:** Our visuomotor implementation using a visual encoder backbone pre-trained on a relevant dataset with unfrozen weights, as described in 3.5.5.
- **LBC:** World on Rails' model from the recreation of the Learning by Cheating algorithm in CARLA 0.9.10.

We also created a video showing examples of the models WOR-R, WOR-EF and WOR-EU driving around in the CARLA simulator, which can be accessed via the following link: `https://youtu.be/rOCWftL7kFU`.

4.1 Comparison of models on the NoCrash benchmark

In this section, we compare the models WOR, WOR-R, WOR-EF and WOR-EU on their performance in the NoCrash benchmark. We compare the results from WOR against the other WOR-R, WOR-EF and WOR-EU, where the latter three have performed `train_phase2` with a limited 100 thousand data frame dataset.

Task	Town	Weather	WOR	WOR-R	WOR-EF	WOR-EU
Empty			**98**	91	61	92
Regular	train	train	**100**	95	68	91
Dense			**96**	86	65	81
Empty			94	**95**	84	92
Regular	test	train	89	87	81	**91**
Dense			**74**	59	51	56
Empty			**90**	70	62	80
Regular	train	test	**90**	68	60	76
Dense			**84**	82	64	78
Empty			78	**84**	60	76
Regular	test	test	**82**	70	58	78
Dense			**66**	42	40	50
Average values			**86.75**	77.42	62.83	78.42

Table 4.1: Table showing the success rates for different models in the NoCrash benchmark. Where train town and test town refer to *Town01* and *Town02*, respectively.

We compare models on two main metrics on the NoCrash benchmark, which are success rate and traffic light infractions per hour.

Throughout experiments 1 and 2, we also collected the loss generated from the action head during `train_phase2`, when training models WOR-R, WOR-EF and WOR-EU. This loss can be seen in figure 4.1. The models were trained on 105 503 data frames with three different camera yaws, giving a total of 316 509 training instances. With 20 epochs and a batch size of 128, we saved action losses from every 100th batch, which equalled 494 saves.

In table 4.1, we see the average success rates of each of the key models for this thesis. We calculated average values across different settings of tasks (traffic and pedestrian density), towns and weather. Looking at the overall averages for each of the models, we see that WOR has the highest average score of 86.75, followed by 78.42, 77.42 and 62.83 for WOR-EU, WOR-R and WOR-EF, respectively.

In table 4.2, we see the average number of traffic light infractions per hour for different settings from the NoCrash benchmark. The overall averages for the traffic light infractions per hour per model are 6.41 (WOR), 12.51 (WOR-R), 25.89 (WOR-EF) and 7.34 (WOR-EU).

4.2 Results from Experiment 1

With a limited dataset containing less than half of the number of data frames used to train the WOR model, we still saw a remarkable performance from the WOR-R model. The model also performs overall better than World on Rails' recreation of the Learning by Cheating (LBC) algorithm. A comparison between the success

Task	Town	Weather	WOR	WOR-R	WOR-EF	WOR-EU
Empty			0.00	0.57	6.17	0.73
Regular	train	train	0.43	1.94	8.90	1.50
Dense			2.61	2.74	9.76	4.86
Empty			10.68	8.77	44.61	7.56
Regular	test	train	6.95	18.11	43.80	11.12
Dense			12.90	17.27	33.99	14.41
Empty			0.00	1.29	18.70	0.98
Regular	train	test	0.00	1.39	15.06	1.04
Dense			4.29	5.06	15.32	3.38
Empty			14.46	27.62	41.71	9.85
Regular	test	test	11.30	37.23	35.88	13.65
Dense			13.28	28.17	36.77	19.03
Average values			**6.41**	12.51	25.89	7.34

Table 4.2: Table showing the average number of traffic light infractions per hour for different models in the NoCrash benchmark.

rates in the NoCrash benchmark between WOR-R and LBC can be seen in table 4.3.

The WOR-R model did show good driving capabilities, and for the majority of the time during evaluation, it did show a relatively good understanding of the traffic light states at intersections. The most common errors encountered during evaluation on the NoCrash benchmark was that the agent would misunderstand the traffic light state at given intersections, and end up either staying stuck at intersections expecting the light to be red, or running a red light misunderstanding it to be green. Running red lights would also create scenarios where the agent would either collide or get stuck face-to-face with other vehicles.

4.3 Results from Experiment 2

4.3.1 Implementation A

From table 4.1 we see that WOR-EF was the worst performer of the three models from experiments 1 and 2. The agent would handle turns and intersections relatively well, and also stop for pedestrians and other vehicles driving in front of the agent, but did not show a good understanding of traffic light states. In these cases, the agent would often become stuck in intersections, or run red lights. These observations are reflected in both table 4.1 and 4.2. During evaluation, the WOR-EF model would perform some turns in a less-than-desirable manner, outputting fluctuating steering commands, making the turn seem wobbly.

Looking at the action loss in figure 4.1, we see that WOR-EF did manage to fit relatively well to the dataset generated in `data_phase2`, but not as good as WOR-R and WOR-EU. WOR-EF had an average action loss above 0.005 towards the end of

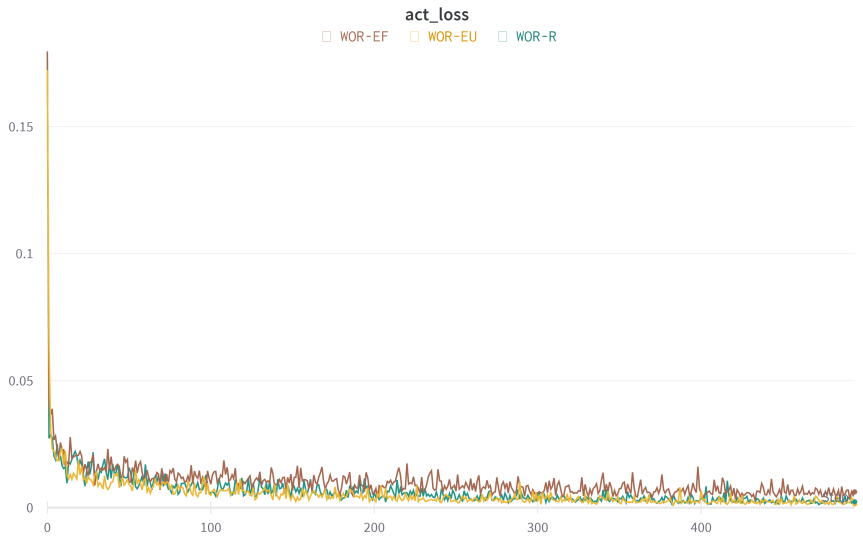


Figure 4.1: Figure showing the action loss generated by the action head during `train_phase2` for WOR-R, WOR-EF and WOR-EU.

`train_phase2`, and looking at the graphs' volatility, did indicate a more unstable convergence in the training process. On average, WOR-R and WOR-EU would stay at an action loss of approximately 0.002 towards the end of their visuomotor training. WOR-EF did also tend to miss turns at intersections under some weather conditions, mistaking water patches and reflections for something else than road. An example of such a turn can be seen in figure 4.2.

Figure 4.2: Image from an intersection with a weather condition where the WOR-EF model was unable to turn.

Task	Town	Weather	LBC	WOR-R
Empty			89	91
Regular	train	train	87	95
Dense			75	86
Empty			86	95
Regular	test	train	79	87
Dense			53	59
Empty			60	70
Regular	train	test	60	68
Dense			54	82
Empty			36	84
Regular	test	test	36	70
Dense			12	42

Table 4.3: Table showing the success rates for the LBC and WOR-R models in the NoCrash benchmark.

4.3.2 Implementation B

Looking at the average value of the success rates, as seen in table 4.1, of each of the three models WOR-R, WOR-EF and WOR-EU, we see that the WOR-EU performed best, marginally better than WOR-R. Taking into account the traffic light infraction rate, as presented in table 4.2, we see a significant improvement over both the WOR-EF and the WOR-R models. As a common reason for WOR-R, WOR-EF and WOR-EU to get failed runs were that they would get stuck at intersections constantly assuming that a traffic light is red, models ignoring red lights would sometimes benefit from it in the form of better success rates. We can see a good example of this from the performance of WOR-EF on the testing town with training weather, the agent has one of its highest instances of success rate, but also the highest instances of traffic light infraction rates.

During evaluation of WOR-EU, we saw good driving capabilities, similar to the WOR-R model. WOR-EU performed turns in a more smooth manner compared to WOR-EF. The WOR-EU model did however show a greater understanding of traffic lights and their states, less frequently ending up running red lights, or staying stuck at intersections not being able to interpret when the traffic light is green.

From the action losses observed in figure 4.1, we do observe some indication that WOR-EU's action loss more quickly converges and fits to the training dataset than the WOR-R model that uses a visual backbone network only pre-trained on the ImageNet [27] dataset. This is best observed in the first 100 losses saved. Towards the end of the training period, the two models WOR-EU and WOR-R do however converge to similar loss values of approximately 0.002.

Chapter 5

Discussion

This chapter will discuss the results presented in this thesis in relation to the research questions presented in section 1.2. The last section will discuss shortcomings of this thesis.

5.1 Experiment 1

For our first research question, we set out to recreate the results presented in the Learning to drive from a World on Rails paper [2]. Because of a generally unstable training environment, with frequent simulator crashes, and a shorter time frame than initially expected, due to converting PyTorch versions and utilizing graphics cards with the Ampere architecture, we were not able to fully recreate the results that were presented in the World on Rails paper [2]. The original paper specifies that they performed the NoCrash visuomotor training with a training dataset containing 270 000 data frames, while we only managed to generate approximately 100 000 data frames for visuomotor training.

Although we were unable to fully recreate the results, we still managed to create a WOR-R model which was trained with a smaller dataset, that still managed to outperform World on Rails' LBC recreation, which does to some extent validate the World on Rails algorithm's high sample efficiency. From only one data frame, it calculates values for the different speeds that the agent could be in, and the different high-level commands the agent could be given, making single data frames highly densely informative.

During rendering in CARLA 0.9.10, we did experience some artefacts occurring in the form of black shapes showing in the renderings, which were significantly apparent in the semantic segmentations. Even though these artefacts were present in the training dataset, it did not indicate a significant performance drop for our WOR-R model, considering that the model was trained on less than half of the amount the original World on Rails paper used on their NoCrash model. The model did also learn a rough estimation of the patterns that the artefacts would occur in, as can be seen in figure 3.7.

5.2 Experiment 2

This section will discuss the results from the second experiment in relation to the second research question presented in section 1.2. Discussion related to the results for implementation A can be seen in 5.2.1, and for implementation B in 5.2.2.

5.2.1 Implementation A

Looking at the results from the NoCrash benchmark we see that WOR-EF, which uses a visual backbone image encoder with frozen weights, performed significantly worse than the other models we compared it to, which does to some extent give an answer to our second research question, when following this implementation. Looking at the model's architecture, we do see that using a ResNet34 visual backbone with a smaller, 3-layered, interpretation head was not an optimal configuration. From the action loss generated by the action head throughout `train_phase2`, we also saw a higher and more volatile loss amount compared to the other models WOR-R and WOR-EU, which could indicate some information being lost in the encoding, or that the interpretation head and action head hold too few parameters to be able to reasonably fit a good mapping between input images and the large output that the action head outputs.

To answer our second research question, we initially wanted to include more losses during autoencoder training in our second experiment, taking inspiration from the work of Toromanoff *et al.* [30]. Compared to the ResNet34 visual backbone from the first experiment, it learns mainly based on the loss generated from action prediction and semantic segmentation, and not image reconstruction, while we mainly make the encoder in the autoencoder training learn to encode an image to a smaller embedding space, along with the task of semantic segmentation, which means it in practice does not learn anything related to performing actions, or how to encode an image in the context of performing an action. To get a better image encoding using implementation A, we expect that more driving relevant losses during autoencoder training could potentially increase performance in the NoCrash benchmark. Additional losses could be intersection presence prediction, intersection distance prediction, traffic light state prediction, or traffic light distance prediction.

From what was observed during evaluation on the NoCrash benchmark, we still saw some issues in terms of misinterpreting puddles, or similar distortions on the ground for not being road or being obstacles. This observation was most prominent in turns, making the agent miss a turn and ultimately fail the run. This does beg into question, the necessity to create loss in regards to image reconstruction, as it also requires the system to learn to reconstruct these puddles or visual distortions, information regarding these phenomena would also have to be present in the image embedding. Given an autonomous driving problem with a bigger focus on traction control, a visual encoder could potentially benefit from image reconstruction in this manner.

5.2.2 Implementation B

From the results presented in this thesis, we see that the WOR-EU model generally performed better than both the WOR-EF model and the WOR-R model, which does give more answer to our second research question. This indicates that even though the visual encoder has been trained on tasks other than action prediction, it still manages to retain knowledge relevant to the visual interpretation of the environment even after being trained on action prediction in `train_phase2`.

Although showing much broader generalizability as can be seen in tables 4.1 and 4.2, it is important to note that WOR-EU uses a visual encoder that most likely had been trained on images from *Town02* and also test weathers. This does show itself in how the traffic light infraction rates differ from one another in any other NoCrash benchmark configuration other than train town and train weather. Although interesting to see how familiarising a network with relevant images does translate well into other tasks, we would still ideally want to test this implementation with a dataset that does not contain images from *Town02* or images with testing weather.

The action losses observed in figure 4.1 did indicate some quicker convergence for the WOR-EU model in comparison to the WOR-R model, implying that the autoencoder training process does help the WOR-EU more quickly fit to the training dataset after being familiarized with images and semantic segmentations from the CARLA simulator.

5.3 Shortcomings of this Thesis

Throughout the duration of this project, we have mainly utilized a VM hosted by IDI Horizon. During a short initial period we used a VM with an Nvidia Quadro RTX 6000/8000 GPU, but later moved to another configuration which allowed us to use a more powerful Nvidia A40-16Q GPU, due to the processing power and VRAM required to parallelize processes which required multiple CARLA simulators to run efficiently. This transition did however pose a couple of issues and delays throughout some of the lifetime of this project. Most prominent was the issue regarding not having enough Nvidia Grid licenses for all of IDI Horizon's users, making the occurrence of sudden GPU access loss quite frequent. We did also experience that the VMWare client GUI would randomly crash, which required us to reboot the VM to fix the issue. These issues did impact progression in the initial time period of the project.

A substantial amount of time was spent on making the World on Rails algorithm run, and converting PyTorch versions. An initial issue occurred when trying to run an evaluation script following the same environment as described on the World on Rails GitHub repository, this issue was related to the PyTorch version not being compiled to a newer version of CUDA which is initially supported by newer graphics cards using the Ampere architecture. The decision was therefore made to change versions of PyTorch, which created several issues at each step of the World

on Rails algorithms. Not all issues were fixed, as some were circumvented using checkpoints throughout the steps, to restart the step from where it crashed. We did also experience random CARLA simulator crashes frequently, where we again used checkpoints to not have to restart the process from the beginning. Frequent simulator crashes could have been circumvented by converting the World on Rails algorithm to use one of the newer versions of CARLA, which at the time of writing is CARLA 0.9.13. We did not perform this conversion as we expected not to have enough time to allocate to perform such a conversion.

Due to the time limitations of this project, we were unable to fully perform a hyperparameter search, and could have spent more time better configuring the different neural networks. We were also not able to perform many evaluations on the different models saved at different epochs throughout `train_phase2`, which means that our chosen model, with our parameters, might not be the optimal solution for the proposed task.

In retrospect, we did observe that when choosing to build upon the work of others, it is important to investigate which technology and resources the work has been created with, and what resources we require or currently have access to. Building upon work using older resources might require time for conversion and debugging when this could be time spent to further evolve the work we are building upon. In contrast, with state-of-the-art submissions on the CARLA benchmark, not all of the newest submissions have published their code which does discourage building upon their work, as coding a state-of-the-art project from scratch can be both challenging and time-consuming.

Chapter 6

Conclusion and Future Work

This chapter will cover the conclusion and future work of the thesis. Section 6.1 will cover the main findings from the experiments that were conducted in this thesis, and section 6.2 will present some ideas for future work.

6.1 Conclusion

The research goal of this thesis was to improve the training and performance of the World on Rails [2] algorithm. By performing two experiments, with one experiment having two different implementations, this goal was partially reached, but due to issues encountered throughout the project, not fully reached.

For our first research question, we were not able to fully recreate the results that are presented in the World on Rails [2] paper, as we were unable to create a same-size dataset for visuomotor training. We were however able to get fairly decent results on the NoCrash benchmark, using a visuomotor training dataset of less than half the data frames that the World on Rails [2] paper specified, even outperforming their Learning by Cheating [3] recreation in terms of success rate.

For our second research question, we did manage to greatly improve performance in terms of lowering traffic lights infraction rate with our second implementation in the second experiment, WOR-EU. The model also did manage to decrease loss quicker than our model from the first experiment, WOR-R, showing quicker convergence. It is important to note that we were unable to filter out images from the testing town and weathers used in the NoCrash benchmark during autoencoder training, which did give WOR-EU a questionable advantage.

For our first implementation in the second experiment, the WOR-EF model, we saw a significant drop in performance using a visual encoder with frozen weights. The performance drop was large in both success rate and traffic light infraction rate in the NoCrash benchmark. The implementation was however not implemented as we originally intended it to be, as it only trained on the two tasks of image reconstruction and semantic segmentation during autoencoder training, so we do not regard an approach using a visual encoder with frozen weights as a sub-optimal solution.

6.2 Future Work

For future work, we suggest converting the World on Rails algorithm to use CARLA 0.9.13 instead of using CARLA 0.9.10. From this conversion, we expect to experience fewer random crashes during simulation when using newer graphics cards using the Ampere architecture. In addition to less frequent crashes, the conversion could potentially remove the issue regarding artefacts occurring in the renderings from the CARLA simulator, including the semantic segmentations.

Although being one of the worst-performing implementations from the experiments performed in this thesis, our WOR-EF used a visual encoder that was not pre-trained in the manner we initially planned. For future work we suggest augmenting the autoencoder training step in our second experiment, to add more tasks for the autoencoder to train on. Examples of potential tasks could be intersection presence prediction, intersection distance prediction, traffic light state prediction, or traffic light distance prediction. Implementing such tasks in the autoencoder training could be beneficial, as was explored by Toromanoff *et al.* [30].

As the second experiment does perform autoencoder training using a dataset containing images from NoCrash's test town and test weather, we propose performing the second experiment with a dataset that does not contain such images. It would then be possible to more accurately compare the WOR-EU model's ability to generalize in comparison to WOR-R.

We would also like to see the World on Rails algorithm be implemented in a real-world urban driving scenario, and whether or not it performs well. Following the World on Rails algorithm, the initial forward model could be trained on a closed course, using highly accurate positioning and navigational systems, including the vehicle's speed and steering sensors. During data collection in the visuomotor training phase, a human expert driver can be used to collect data. Data could be collected using positioning and navigational systems, as well as traffic light data. A LiDAR sensor can be used additionally to accurately position vehicles and pedestrians nearby.

Our code is available at: `https://github.com/MorningClub/master-thesis`.

Bibliography

[1] A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López and V. Koltun, 'CARLA: an open urban driving simulator,' *CoRR*, vol. abs/1711.03938, 2017. arXiv: 1711.03938. [Online]. Available: http://arxiv.org/abs/1711.03938.

[2] D. Chen, V. Koltun and P. Krähenbühl, *Learning to drive from a world on rails*, 2021. arXiv: 2105.00636 [cs.RO].

[3] D. Chen, B. Zhou, V. Koltun and P. Krähenbühl, *Learning by cheating*, 2019. arXiv: 1912.12294 [cs.RO].

[4] D. Pomerleau, 'Alvinn: An autonomous land vehicle in a neural network,' in *Proceedings of (NeurIPS) Neural Information Processing Systems*, D. Touretzky, Ed., Morgan Kaufmann, Dec. 1989, pp. 305–313.

[5] M. Xie, L. Trassoudaine, J. Alizon, M. Thonnat and J. Gallice, 'Active and intelligent sensing of road obstacles: Application to the european eureka-prometheus project,' in *1993 (4th) International Conference on Computer Vision*, 1993, pp. 616–623. DOI: 10.1109/ICCV.1993.378154.

[6] B. Tefft, 'Rates of motor vehicle crashes, injuries and deaths in relation to driver age, united states, 2014-2015,' in AAA Foundation for Traffic Safety, 2017.

[7] F. Codevilla, E. Santana, A. M. López and A. Gaidon, *Exploring the limitations of behavior cloning for autonomous driving*, 2019. DOI: 10.48550/ARXIV.1904.08980. [Online]. Available: https://arxiv.org/abs/1904.08980.

[8] P. N. Stuart Russell, *Artificial Intelligence: A Modern Approach (4th Edition) (Pearson Series in Artifical Intelligence)*, 4th ed. Language: English, 2020, ISBN: 0134610997; 9780134610993.

[9] A. C. Ian Goodfellow Yoshua Bengio, *Deep Learning*. The MIT Press, 2016.

[10] D. E. Rumelhart, G. E. Hinton and R. J. Williams, 'Learning representations by back-propagating errors,' *Nature*, vol. 323, pp. 533–536, 1986.

[11] A. Hussein, M. M. Gaber, E. Elyan and C. Jayne, 'Imitation learning: A survey of learning methods,' *ACM Comput. Surv.*, vol. 50, no. 2, Apr. 2017, ISSN: 0360-0300. DOI: 10.1145/3054912. [Online]. Available: https://doi.org/10.1145/3054912.

[12] A. G. B. Richard S. Button, *Reinforcement Learning, Second Edition*, 2nd ed. Bradford Book, 2018, ISBN: 0262039249; 978-0262039246.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[14] J. Schulman, S. Levine, P. Moritz, M. I. Jordan and P. Abbeel, *Trust region policy optimization*, 2017. arXiv: 1502.05477 [cs.LG].

[15] C. Ruizhongtai Qi, H. Su, M. Niessner, A. Dai, M. Yan and L. Guibas, 'Volumetric and multi-view cnns for object classification on 3d data,' Apr. 2016.

[16] K. He, X. Zhang, S. Ren and J. Sun, *Deep residual learning for image recognition*, 2015. DOI: 10.48550/ARXIV.1512.03385. [Online]. Available: https://arxiv.org/abs/1512.03385.

[17] R. K. Srivastava, K. Greff and J. Schmidhuber, 'Highway networks,' *CoRR*, vol. abs/1505.00387, 2015. arXiv: 1505.00387. [Online]. Available: http://arxiv.org/abs/1505.00387.

[18] G. Huang, Z. Liu, L. Van Der Maaten and K. Q. Weinberger, 'Densely connected convolutional networks,' in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243.

[19] S. Wold, K. Esbensen and P. Geladi, 'Principal component analysis,' *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1, pp. 37–52, 1987, Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists, ISSN: 0169-7439. DOI: https://doi.org/10.1016/0169-7439(87)80084-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0169743987800849.

[20] S. Balakrishnama and A. Ganapathiraju, 'Linear discriminant analysis—a brief tutorial,' vol. 11, Jan. 1998.

[21] S. Lange and M. Riedmiller, 'Deep auto-encoder neural networks in reinforcement learning,' in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 2010, pp. 1–8. DOI: 10.1109/IJCNN.2010.5596468.

[22] J. Janai, F. Güney, A. Behl and A. Geiger, *Computer vision for autonomous vehicles: Problems, datasets and state of the art*, 2017. DOI: 10.48550/ARXIV.1704.05519. [Online]. Available: https://arxiv.org/abs/1704.05519.

[23] R. Chekroun, M. Toromanoff, S. Hornauer and F. Moutarde, 'GRI: general reinforced imitation and its application to vision-based autonomous driving,' *CoRR*, vol. abs/2111.08575, 2021. arXiv: 2111.08575. [Online]. Available: https://arxiv.org/abs/2111.08575.

[24] CARLA, *3rd. maps and navigation*, https://carla.readthedocs.io/en/0.9.10/core_map/, Accessed: 06-06-2022, 2022.

[25] Martın 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, Y. 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*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, 'Pytorch: An imperative style, high-performance deep learning library,' in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, 'Imagenet: A large-scale hierarchical image database,' in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[28] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick and P. Dollár, *Microsoft coco: Common objects in context*, 2014. DOI: `10.48550/ARXIV.1405.0312`. [Online]. Available: `https://arxiv.org/abs/1405.0312`.

[29] F. Codevilla, M. Müller, A. López, V. Koltun and A. Dosovitskiy, *End-to-end driving via conditional imitation learning*, 2018. arXiv: `1710.02410 [cs.RO]`.

[30] M. Toromanoff, E. Wirbel and F. Moutarde, *End-to-end model-free reinforcement learning for urban driving using implicit affordances*, 2020. arXiv: `1911.10868 [cs.LG]`.

[31] K. Chitta, A. Prakash, B. Jaeger, Z. Yu, K. Renz and A. Geiger, *Transfuser: Imitation with transformer-based sensor fusion for autonomous driving*, 2022. DOI: `10.48550/ARXIV.2205.15997`. [Online]. Available: `https://arxiv.org/abs/2205.15997`.

[32] D. Chen and P. Krähenbühl, *Learning from all vehicles*, 2022. DOI: `10.48550/ARXIV.2203.11934`. [Online]. Available: `https://arxiv.org/abs/2203.11934`.

[33] Anaconda, *Anaconda*, `https://www.anaconda.com/products/distribution`, Accessed: 06-06-2022, 2022.

[34] Microsoft, *Visual studio code*, https://code.visualstudio.com/, Accessed: 06-06-2022, 2022.

[35] Weights Biases, *Weights biases*, https://wandb.ai/site, Accessed: 06-06-2022, 2022.

[36] NVIDIA. 'Nvidia ampere gpu architecture compatibility guide for cuda applications.' (2022), [Online]. Available: https://docs.nvidia.com/cuda/archive/11.4.0/ampere-compatibility-guide/index.html (visited on 19/05/2022).

[37] PyTorch. 'Installing previous versions of pytorch.' (2022), [Online]. Available: https://pytorch.org/get-started/previous-versions/ (visited on 19/05/2022).

[38] D. Chen, V. Koltun and P. Krähenbühl, *World on rails*, https://github.com/dotchen/WorldOnRails, 2022.

[39] P. Polack, F. Altché, B. d'Andréa-Novel and A. de La Fortelle, 'The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?' In *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[40] A. B. Jung, K. Wada, J. Crall, S. Tanaka, J. Graving, C. Reinders, S. Yadav, J. Banerjee, G. Vecsei, A. Kraft, Z. Rui, J. Borovec, C. Vallentin, S. Zhydenko, K. Pfeiffer, B. Cook, I. Fernández, F.-M. De Rainville, C.-H. Weng, A. Ayala-Acevedo, R. Meudec, M. Laporte *et al.*, *imgaug*, https://github.com/aleju/imgaug, Online; accessed 01-Feb-2020, 2020.

Appendix A

Environment

Code listing A.1: environment.yml file generated from conda environment used for this thesis.

```
name: world_on_rails
channels:
  - pytorch
  - conda-forge
  - anaconda
  - defaults
dependencies:
  - cudatoolkit=11.3.1
  - pip=21.2.2
  - python=3.7.9
  - pip:
    - dictor==0.1.9
    - imageio==2.16.0
    - imageio-ffmpeg==0.4.5
    - imgaug==0.4.0
    - lmdb==1.1.1
    - matplotlib==3.5.1
    - moviepy==1.0.3
    - numpy==1.21.5
    - opencv-python==4.5.5.62
    - py-trees==0.8.3
    - pygame==2.1.2
    - pyyaml==6.0
    - ray==1.11.0
    - shapely==1.8.0
    - tabulate==0.8.9
    - torch==1.10.2+cu113
    - torchaudio==0.10.2+cu113
    - torchvision==0.5.0
    - tqdm==4.62.3
    - wandb==0.12.10
    - xmlschema==1.9.2
```

Appendix B

Training Parameters

This appendix presents the different training parameters used during training of the main networks in this thesis.

Parameter	Value
Batch size	128
Learning rate	1e-2
Epochs	100

Table B.1: Training parameters used for training the forward model.

Parameter	Value
Batch size	64
Learning rate	3e-4
Weight decay	3e-5
Epochs	12
Segmentation loss scale	1

Table B.2: Training parameters used for training the autoencoder network.

Parameter	Value
Batch size	128
Learning rate	3e-4
Weight decay	3e-5
Epochs	20
Segmentation loss scale	5e-2

Table B.3: Training parameters used for training the visuomotor networks.

Appendix C

PyTorch Code

This appendix will present some PyTorch code relevant to this thesis.

Code listing C.1: The Resnet34 decoder architecture in PyTorch code form.

```
class ResizeConv2d(nn.Module):

    def __init__(self, in_channels, out_channels, kernel_size, scale_factor, mode='
        nearest'):
        super().__init__()
        self.scale_factor = scale_factor
        self.mode = mode
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
            padding=1)

    def forward(self, x):
        x = F.interpolate(x, scale_factor=self.scale_factor, mode=self.mode)
        x = self.conv(x)
        return x


class BasicBlockDec(nn.Module):

    def __init__(self, in_planes, stride=1):
        super().__init__()

        planes = int(in_planes/stride)
        self.in_planes = in_planes

        self.conv2 = nn.Conv2d(in_planes, in_planes, kernel_size=3, stride=1,
            padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(in_planes)

        if stride == 1:
            self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=1,
                padding=1, bias=False)
            self.bn1 = nn.BatchNorm2d(planes)
            self.shortcut = nn.Sequential()
        else:
            self.conv1 = ResizeConv2d(in_planes, planes, kernel_size=3,
                scale_factor=stride)
            self.bn1 = nn.BatchNorm2d(planes)
            self.shortcut = nn.Sequential(
```

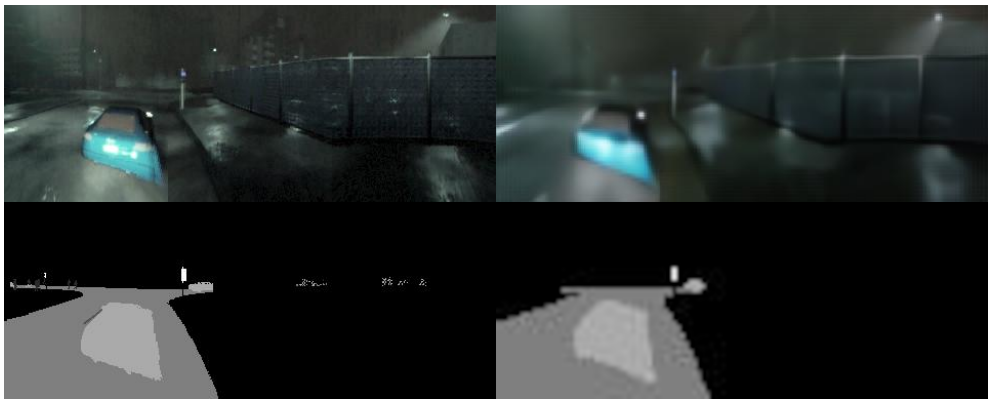
```
                ResizeConv2d(in_planes, planes, kernel_size=3, scale_factor=stride)
                    ,
                nn.BatchNorm2d(planes)
            )

    def forward(self, x):
        out = torch.relu(self.bn2(self.conv2(x)))
        out = self.bn1(self.conv1(out))
        out += self.shortcut(x)
        out = torch.relu(out)
        return out

class ResNet34Dec(nn.Module):

    def __init__(self, num_Blocks=[3, 4, 6, 3], z_dim=10, nc=3):
        super().__init__()
        self.in_planes = 512

        self.upsample1 = nn.Upsample(scale_factor=2)

        self.layer4 = self._make_layer(BasicBlockDec, 512, num_Blocks[3], stride=2)
        self.layer3 = self._make_layer(BasicBlockDec, 256, num_Blocks[2], stride=2)
        self.layer2 = self._make_layer(BasicBlockDec, 128, num_Blocks[1], stride=2)
        self.layer1 = self._make_layer(BasicBlockDec, 64, num_Blocks[0], stride=1)
        self.conv1 = ResizeConv2d(64, nc, kernel_size=3, scale_factor=2)

    def _make_layer(self, BasicBlockDec, planes, num_Blocks, stride):
        strides = [stride] + [1]*(num_Blocks-1)
        layers = []
        for stride in reversed(strides):
            layers += [BasicBlockDec(planes, stride)]
        self.in_planes = planes
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.layer4(x)
        x = self.layer3(x)
        x = self.layer2(x)
        x = self.layer1(x)
        x = self.upsample1(x)
        x = torch.sigmoid(self.conv1(x))
        x = x.view(x.size(0), 3, 192, 480)
        return x
```

Code listing C.2: Latent space head in PyTorch code form.

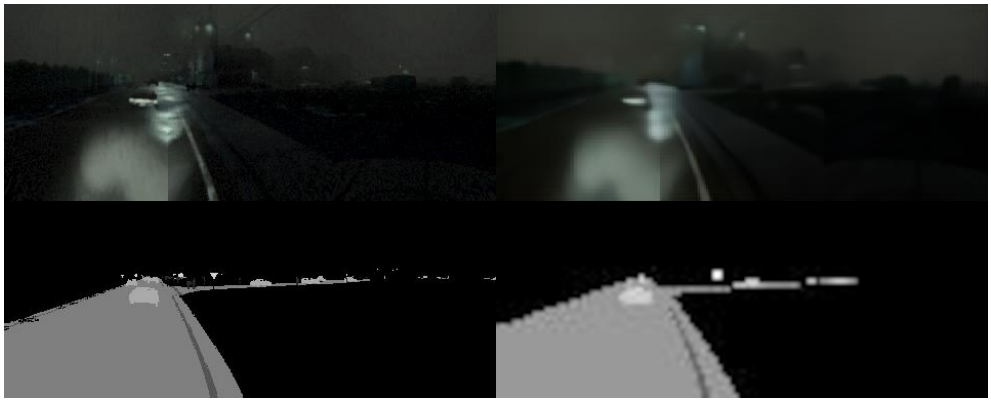
```
    self.latent_space_head = nn.Sequential(
        nn.Conv2d(512, 256, kernel_size=3, stride=1, padding=1),
        nn.ReLU(True),
        nn.Conv2d(256, 64, kernel_size=3, stride=2),
        nn.ReLU(True),
        nn.Flatten(),
        nn.Linear(896, 512),
        nn.ReLU(True)
    )
```

Appendix D

Autoencoder Reconstructions

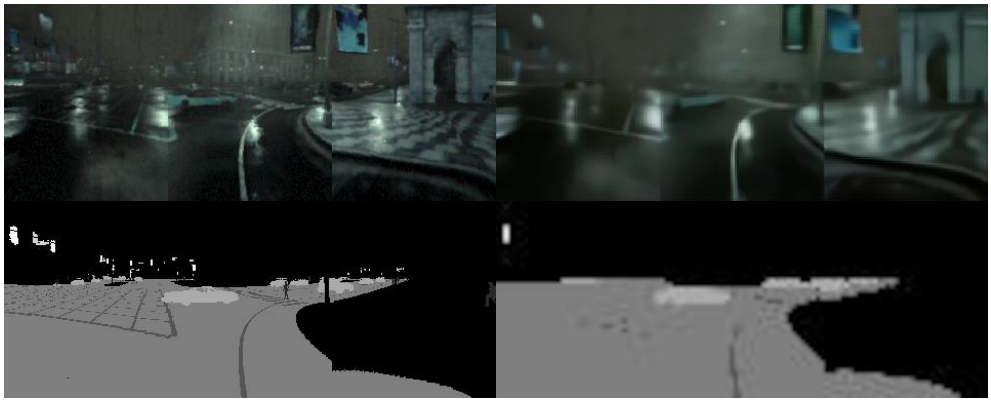


(a) Epoch 1

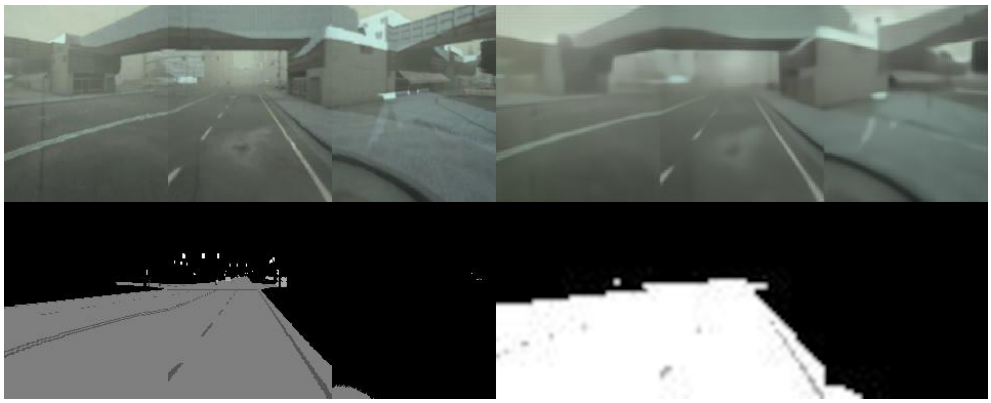


(b) Epoch 2

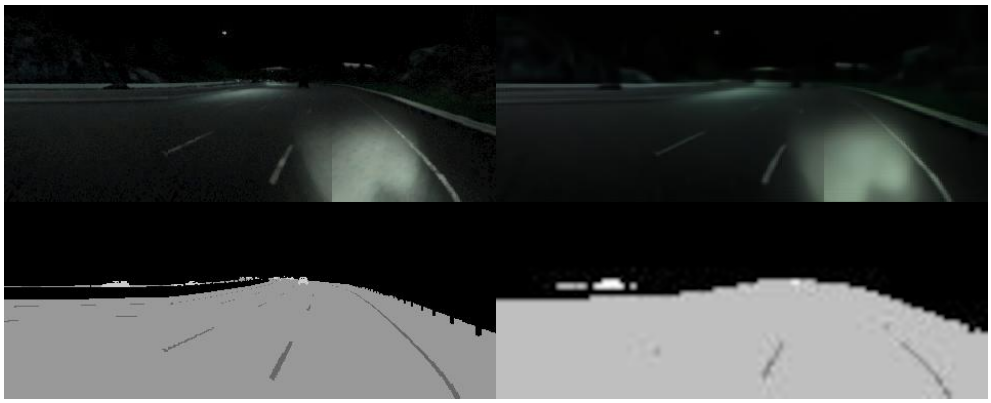
Figure D.1: Figure showing an example of image reconstruction from each epoch of the autoencoder training.



(c) Epoch 3

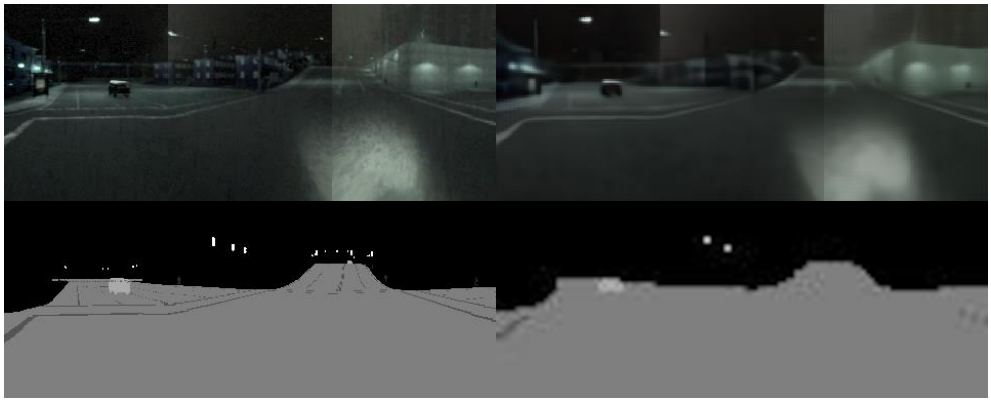


(d) Epoch 4

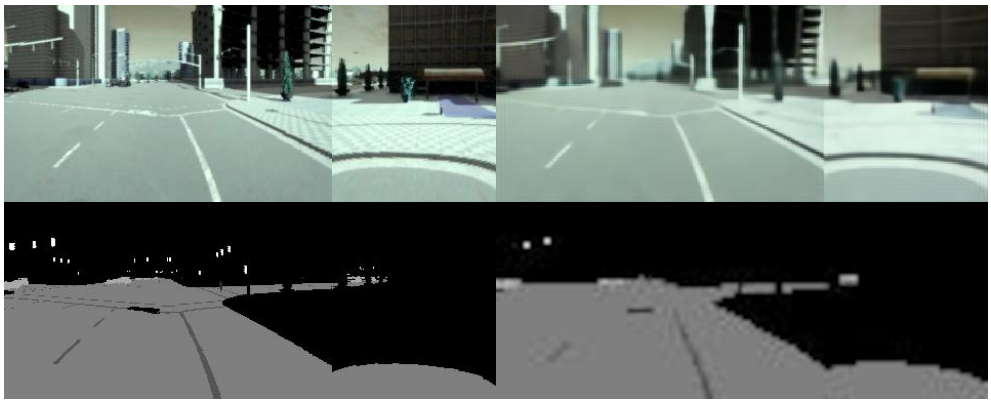


(e) Epoch 5

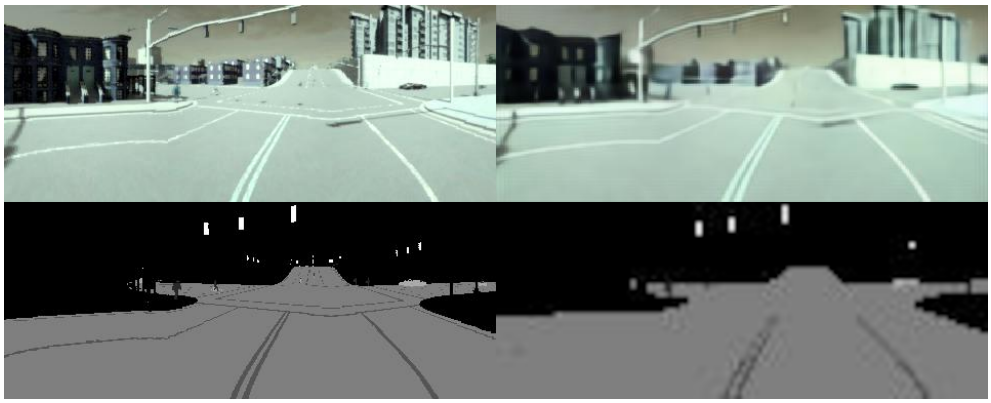
Figure D.1: Figure showing an example of image reconstruction from each epoch of the autoencoder training.



(f) Epoch 6

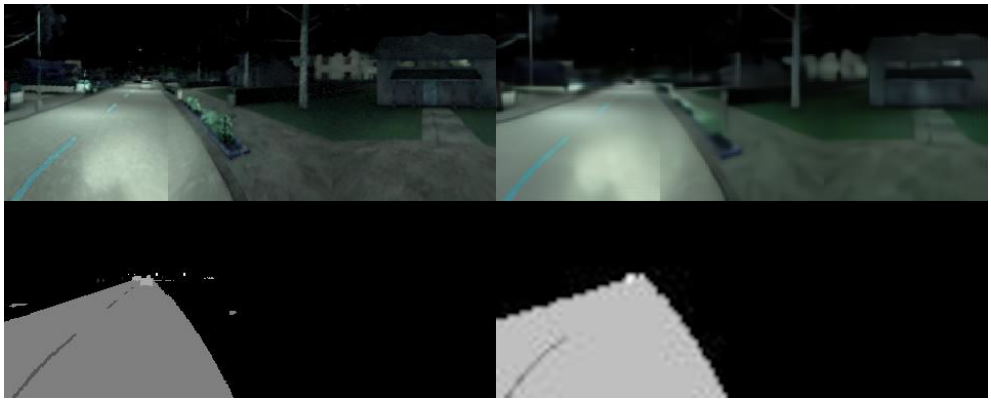


(g) Epoch 7

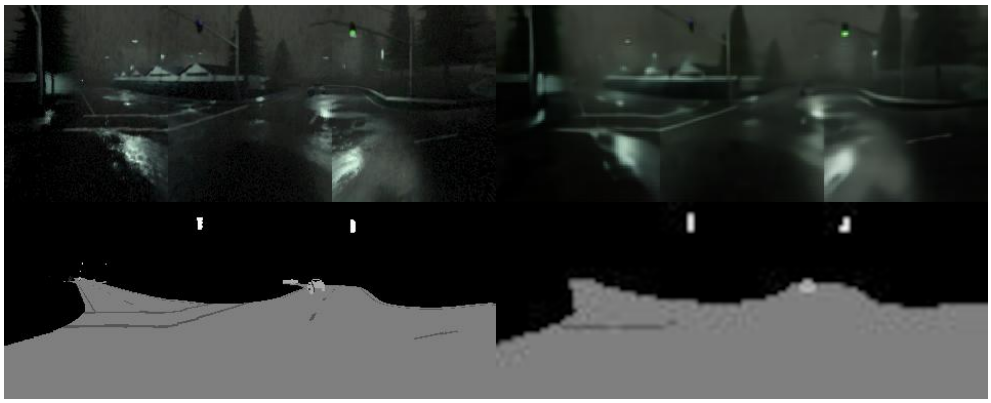


(h) Epoch 8

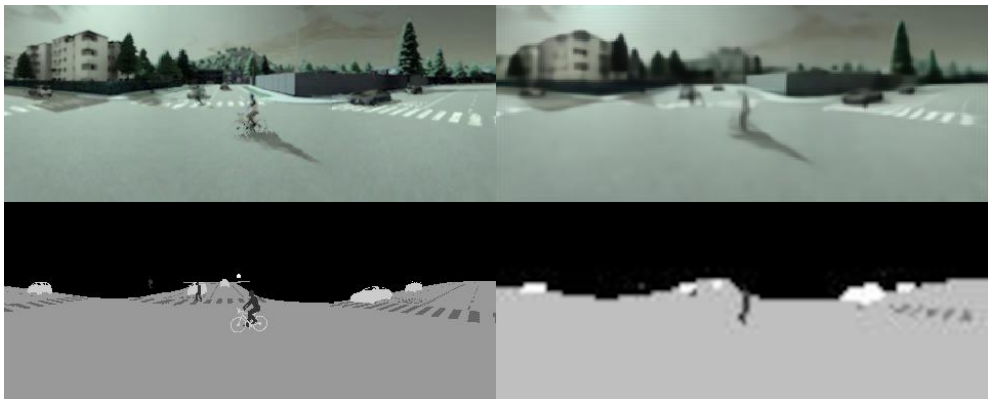
Figure D.1: Figure showing an example of image reconstruction from each epoch of the autoencoder training.



(i) Epoch 9



(j) Epoch 10

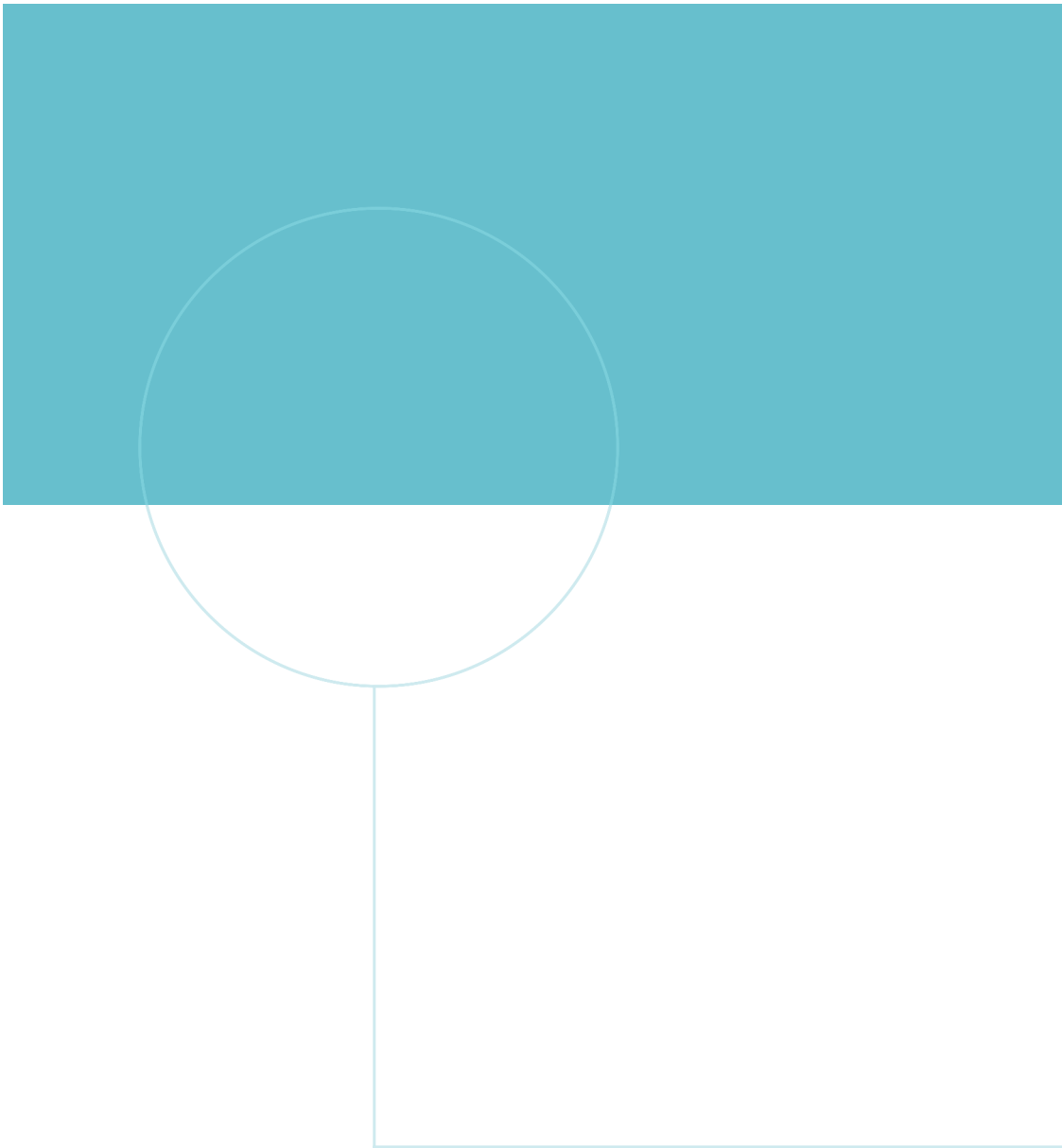


(k) Epoch 11

Figure D.1: Figure showing an example of image reconstruction from each epoch of the autoencoder training.

(l) Epoch 12

Figure D.1: Figure showing an example of image reconstruction from each epoch of the autoencoder training.



NTNU

Norwegian University of
Science and Technology