Master's thesis

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

Christian Seeberg Hådem

Autonomous Vehicle Control: Variational Autoencoders and Reinforcement Learning

Master's thesis in MIDT
Supervisor: Frank Lindseth
June 2020

NTNU
Norwegian University of
Science and Technology

Christian S. Hådem

Autonomous Vehicle Control: Variational Autoencoders and Reinforcement Learning

TDT4501 - Computer Science, Master Thesis
Supervisor: Frank Lindseth

Spring 2020

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

Abstract

In a time where artificial intelligence is making great progress, the biggest challenges are yet to be solved. One of these challenges is to make fully autonomous vehicles. Over the last decade, hardware has become incredibly fast, and countless algorithms has been invented and optimized on the software side. Better simulations are available making research and prototyping autonomous cars easier than ever. In this thesis, multiple aspects of autonomous vehicles will be explored, and a state-of-the-art reinforcement learning algorithm will be applied in a realistic simulation and combined with a method found in unsupervised learning: Autoencoders. Sensory input will be encoded to a small vector using a trained autoencoder. This encoded vector will then be used in a reinforcement learning algorithm to train a vehicle. Autoencoders learns to reconstruct images after compressing them to a very small vector and a car learns to drive in both a simple simulation and in a more sophisticated simulation to some extent. Autoencoders are shown to give a big performance boost in the long term without seeing much loss in terms of rewards and the reinforcement learning agent is able to use this to learn how to drive.

Abstrakt

I en tid hvor kunstig intelligens gjør stor fremgang gjenstår de største utfordringene. En av disse utfordringene er å lage helautomatiske kjøretøy. Over det siste tiåret har maskinvare blitt utrolig raskt og utallige algoritmer har blitt oppfunnet og optimisert på programvare siden. I denne avhandlingen vil flere aspekter rundt helautomatiske kjøretøy bli utforsket, og topp moderne algoritmer innenfor forsterkende læring vil bli anvendt i realistiske simuleringer og kombinert med metoder funnet i ikke-veiledet læring: Autokodere. Sensor data vil bli kodet til en liten vektor ved å bruke en trent autokoder. Denne vektoren vil så bli brukt i en algorithme i forsterket læring for å trene et kjøretøy. Autokodere lærer seg å rekonstruere bilder etter å ha komprimert de til en veldig liten vektor og en bil lærer seg å kjøre både i en enkel simulering og i en mer sofistikert simulering til en viss grad. Autokodere viser seg å gi en stor økning i ytelse på lang sikt uten å gi for mye tap i verdiene og forsterket læring agenten klarer å bruker dette til å lære seg å kjøre.

Preface

This project was done as a masters thesis at the department of computer science at NTNU. The supervisor for this project, Frank Lindseth, has been of great help by providing useful feedback throughout the lifespan of the project and especially so for feedback on the thesis. Håkon Hukkelås has also been helpful by doing the small favor of setting up a SSH server to a PC on campus during the lockdown.

Christian S. Hådem
Trondheim, June 14, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Driving a vehicle is a complex task for humans which requires many hours of supervised practice. After a drivers license has been acquired, there are still situations that practice hasn't covered, yet humans are still able to handle many of these situations without problems. Capturing the ability to use similar experiences in new unseen situations in a machine, has already been done to a certain extent [Aamodt and Plaza, 1994]. There has also been a lot of successful work done in the field of autonomous vehicles using machine learning [Kendall et al., 2018]. The environments used in existing work are for the most part very simple or done in simulations, so as of now, no vehicle can cover every scenario in the real world as good as humans.

Self driving cars are often seen in science fiction and many people want it to become a reality. In a perfect world, autonomous vehicles bring safety and efficiency to the roads. Autonomous vehicles theoretically only have to be more safe on average than a human driver, but this is not really the case in practice as deaths caused by a manufacturer or equipment error causes a lot of problems and controversy. Perfect vehicles can find the most optimal path to the destination with respect to the current traffic all on its own. Manually driven vehicles will never be totally replaced however as many people prefer driving themselves. Some people don't have the trust to let their car drive on their own, and other people might enjoy driving as a spare time activity or hobby, like racing. For many people however autonomous vehicles are the next step going into the future so the daily commute and other travel destinations can become automatic.

1.2 Goals and Research Questions

The overall goal of autonomous vehicles combined with reinforcement learning is to answer the following question:

Research question 1: *Is it possible to train an end-to-end reinforcement learning agent to drive a vehicle?*

There are already many papers that shows this is feasible [Kendall et al., 2018] [Chen et al., 2020]. Although this thesis might help answer question 1 better, it will also help to answer some sub-problems related to modern reinforcement learning in autonomous vehicles.

Research question 2: *How does autoencoders affect the performance of reinforcement learning agents?*

The use of autoencoders in reinforcement learning is rather new, but has the potential to greatly increase real-time performance. By learning about autoencoders and how they are used in reinforcement learning, better performance could be achieved in some areas. To answer this question, models with and without autoencoders will be tested and compared to each other to see how they eventually can improve reinforcement learning in combination with autonomous vehicles.

Research question 3: *How does an agents performance increase as the models capacity increases?*

This question seeks to answer what happens if a model is given more capacity. Capacity in this case means a larger model to handle the sensor data and more sensory information. Experiments will be conducted to see how much the performance increases, if any at all.

1.3 Research Method

In short, the method that will be used to answer the research questions is to lay a theoretical foundation of the topics followed by practical experiments. A theoretical foundation has to be made in the field of reinforcement learning and how they learn. One or more reinforcement learning algorithms has to be chosen for the practical part. An environment is also needed to simulate autonomous vehicles. Autoencoders must be explored to learn how they work, both alone and in conjunction with reinforcement learning. The experiments themselves will try to be both qualitative and quantitative to a certain extent.

1.4 Contributions

This thesis explores the usage of autoencoders in reinforcement learning for autonomous vehicles. The work provides an implementation for autoencoder and variational autoencoder, a powerful reinforcement learning algorithm, and code for a reinforcement learning environment in Carla. Qualitative experiments have been performed combining reinforcement learning and autoencoders to see how they perform compared to no autoencoder.

1.5 Thesis Structure

Chapter 1. Introduction: An introduction to the thesis. Contains the research questions this thesis seeks to answer, the motivation for it, and how the questions will be answered.

Chapter 2. Background: The background chapter dives into the theory necessary to understand the rest of the thesis. This chapter also explains some advantages and potential disadvantages of autonomous vehicles. Readers who are familiar with the field may skip parts of this chapter.

Chapter 3, 4 and 5. Experiments and Results: These chapters contains the experiments that were conducted. These include autoencoders, the Car-Racing environment and the Carla environment. Each of these chapters contains the setup, results and some discussion.

Chapter 6. Discussion: Overall discussion about the results. This chapter also discusses the research questions and includes some reflection on the thesis.

Chapter 7. Conclusion & Future Work: The final chapter concludes the thesis and discusses potential future work proceeding the thesis and the field in general.

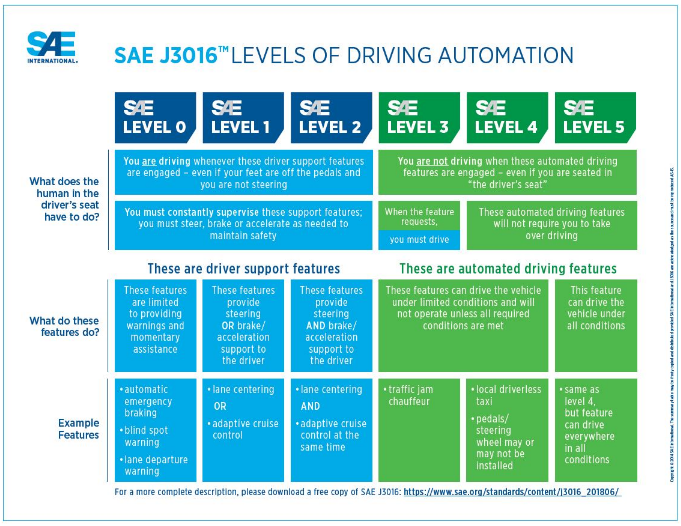
Chapter 2

Background

2.1 Autonomous Vehicles

Autonomous vehicles (AV), as implied by its name, are vehicles that can operate on their own without human intervention. More specifically, the dynamics controlling the speed and steering angle of the vehicle. Other systems, like controlling temperature, can be automated independently and does not count towards automated vehicles. However, some driving conditions might be easier to automate than others, leading to vehicles being partly autonomous. Because of this, SAE international has proposed a standard for defining autonomous vehicles [SAE International, 2018]. The standard defines 6 levels of driving automation. The lowest level, level 0, has no driving automation, after this the autonomy of the vehicle is gradually more and more capable until level 5 where the driving task is fully automated. A quick summary of this standard can be found in Figure 2.1. Up until level 2, the user is still considered driving the vehicle. After this the user is not considered driving the vehicle, but at level 3 the user must be able to intervene when the system requests it.

Autonomous vehicles has been a desirable feature for a long time. The drivers themselves might want to do other things when travelling rather than focusing on the road, but autonomous vehicles could also have other benefits. The vehicles could cooperate and be configured to prevent traffic jams. Safety could be improved as well since accidents on the road is often caused by an operator error [WHO, 2018]. Autonomous vehicles have to be safer than humans to be acceptable for commercial use. There has already been a few accidents related to self-driving cars, and these have caused a lot of controversy and fear. People don't trust a system that has a chance at failing at any point in time. This also leads to the ethical and moral issues with AV. Who is at fault when accidents do



SAE INTERNATIONAL

SAE J3016™ LEVELS OF DRIVING AUTOMATION

	SAE LEVEL 0	SAE LEVEL 1	SAE LEVEL 2	SAE LEVEL 3	SAE LEVEL 4	SAE LEVEL 5
What does the human in the driver's seat have to do?	**You are driving** whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			**You are not driving** when these automated driving features are engaged – even if you are seated in "the driver's seat"		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
	These are driver support features			These are automated driving features		
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering **OR** brake/ acceleration support to the driver	These features provide steering **AND** brake/ acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met		This feature can drive the vehicle under all conditions
Example Features	• automatic emergency braking • blind spot warning • lane departure warning	• lane centering **OR** • adaptive cruise control	• lane centering **AND** • adaptive cruise control at the same time	• traffic jam chauffeur	• local driverless taxi • pedals/ steering wheel may or may not be installed	• same as level 4, but feature can drive everywhere in all conditions

For a more complete description, please download a free copy of SAE J3016: https://www.sae.org/standards/content/J3016_201806/

Copyright © 2014 SAE International. The summary table may be freely copied and distributed provided SAE International and J3016 are acknowledged as the source and must be reproduced AS-IS.

Figure 2.1: SAE Internationals standard of driving automation summarised in a graphic.

happen? What if a vehicle has to decide between hitting a human or steering to the side and kill the driver instead. Autonomous vehicles still have a long way to go. These questions will not be discussed anymore in this thesis, but has to be addressed at some point. Another possible downside to AV is how it could affect the economy. Truck drivers, taxi drivers and bus drivers are some of the most common occupations in the world. Replacing those jobs entirely could have a massive impact on the worlds economy.

With recent advancements in technology, autonomous vehicles are closer than ever to being realised. The access to big data has led to huge advancements in the field of artificial intelligence (AI) and more specifically in the field of machine learning.

2.2 Machine Learning

Machine learning (ML) is a subfield of artificial intelligence where computers use algorithms and statistical models to finds patterns. Machine learning is often used for tasks that are otherwise infeasible by conventional means. Generally ML can be split into three categories: supervised learning, unsupervised learning and reinforcement learning.

In supervised learning agents are learning a function that maps a given input to a given output. These agents are trained using labeled datasets containing (x_i, y_i) pairs. Classification, object detection and speech recognition are some of the applications that use supervised learning. Unsupervised learning attempts to find pattern in unlabeled datasets. These are tasks like clustering, modelling probability densities or autoencoders which will be covered in Section 2.2.2. Finally, reinforcement learning is where agents observe, act and receive rewards in an environment. These agents doesn't use any datasets and their goal is to maximize their cumulative reward. Reinforcement learning will be covered in more detail in Section 2.3.

2.2.1 Neural Networks and Deep Learning

Neural networks (NN) is a well known technique found in machine learning and that works as universal function approximators. Even though neural networks first appeared in the 1950s they could only approximate linear functions [Wang and Raj, 2017]. It wasn't until many years later that backpropagation was discovered which enabled non-linear function approximation. The true potential of neural networks was shown when Lecun et al. [1998] introduced LeNet, a convolutional neural network. The last decade, neural networks has seen an significant rise in popularity as more access to data and more computational power has become available.

Neural networks are inspired by biological neurons in the brain. A neural network is commonly split into multiple layers where each layer has multiple perceptrons (or neurons). In each perceptron, input x from the previous layer is received, multiplied with weights w, subtracted a bias b and summed together. This sum z is sent through an activation function, producing output a, before being sent to the next layer where the same process is repeated. The whole process for calculating the output can be written as

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} - b_j^l\right) \tag{2.1}$$

Where connections are going from node k in the current layer to node j in the next layer. The way a neural network learns is to update its weights and biases through backpropagation. The loss is calculated as a measure of how far off the network output is from the answer. After the loss is calculated using a loss function L the loss is minimized using an iterative optimization algorithm like gradient descent. In such algorithms the gradients $\frac{\partial L}{\partial w^l_j k}$ and $\frac{\partial L}{\partial b^l_j}$ are calculated using the chain rule and multiplied with a learning rate to control the speed of the learning. The learning rate is usually only one of many hyperparameters found in nerual networks which impacts the way the network learns a function. Finding the optimal set of hyperparameters is an open optimization problem itself, and searching for it is often time consuming and computationally expensive.

2.2.2 Autoencoders

Autoencoders are neural networks trained to learn the task of encoding and decoding data. Autoencoders consists of two parts: an encoder and a decoder. The encoder is a function that compresses the input data into a more efficient representation, also known as the hidden-space or latent-space, represented as z. The decoder attempts to reconstruct the original data using only the compressed representation. The whole process can be described as $\hat{x} = g(f(x))$ where f is the encoder and g is the decoder, see Figure 2.2. Since the output should resemble the input as much as possible, the error between the original image and the processed image should be minimized. By setting up a loss function $L(x, g(f(x)))$, the problem is now an optimization problem where f and g can be approximated using a neural network. Typically in autoencoders, the loss function is either the mean squared error (MSE) or binary cross entropy (BCE). MSE (Equation 2.2) simply measures how close the reconstruction error is from the original image by calculating the mean of the squared errors. BCE (Equation 2.3) measures how far off each reconstructed pixel is from the original value by calculating log probabilities.

$$MSE(x, \hat{x}) = \frac{1}{N} \sum_i (x_i - \hat{x}_i)^2 \tag{2.2}$$

$$BCE(x, \hat{x}) = -\frac{1}{N} \sum_{i=1}^{N} x_i \log \hat{x}_i + (1 - x_i) \log(1 - \hat{x}_i) \tag{2.3}$$

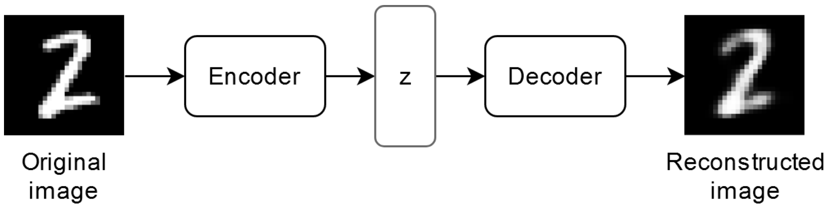


Figure 2.2: The flow of a basic autoencoder

Autoencoders can be learned to denoise images as well. During training, if noise is added to the input image before passing it through the autoencoder, and the loss is still calculated using the original image without noise, the autoencoder will eventually learn to denoise images. The difference of autoencoders over existing compression algorithms is that autoencoders are data-specific. Since the autoencoders are neural networks, they are trained using a dataset, as such the autoencoder will be specialized at compressing data similar to the dataset it was trained on. This gives autoencoders the opportunity to be very good at compressing a specific set of data and nothing else. Autoencoders will however always be a form of lossy compression due to the nature of neural networks.

Having sparse weights in an autoencoder can increase the performance on classification tasks [Makhzani and Frey, 2013]. Sparse autoencoders can be achieved by adding a term to the loss function

$$L(x, \hat{x}) + \Omega(h) \tag{2.4}$$

Where h are the activations. The most common way to achieve sparsity is by using L1 or L2 regularization which is defined in Equation 2.5 and 2.6 respectively.

$$\Omega(h) = \lambda \sum_j |h_j| \tag{2.5}$$

$$\Omega(h) = \lambda \sum_j h_j^2 \tag{2.6}$$

Here the sum of the activations will be added to the loss function, so in order to minimize it, the network is encouraged to adjust the weights so they produce lower activation values. Lambda is a hyperparameter which is used to regulate how much the regularization term should be prioritized.

2.2.3 Variational autoencoders

Variational autoencoders (VAE) represents its latent-space as the mean and standard deviation of distributions instead. In the final part of the encoder, the network will split into two parts. One part predicting the means μ, and one part predicting the standard deviations σ. These are then used to sample numbers using separate normal distributions $\mathcal{N}(\mu, \sigma)$ for each element which will form the latent-vector.

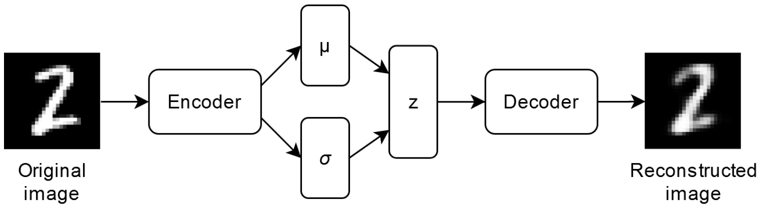


Figure 2.3: The flow of a basic variational autoencoder

The advantage of variational autoencoders is that the latent space has the ability to contain independent high-level features of the data. In the MNIST dataset, which is a dataset with thousands of handwritten digits, these high-level features could include size and rotation of the digits. VAE can also generate new images since it samples from a distribution. This is not really possible in normal autoencoders as the latent space vector in normal autoencoders are rarely continuous due to no sampling.

VAE has one problem though, sampling is an operation with no gradient during backpropagation. Luckily this can be solved using the **reparametrization trick**. By introducing a new variable $\epsilon \sim \mathcal{N}(0, 1)$ and instead compute $z = \sigma * \epsilon + \mu$, all of the nodes except ϵ will be deterministic as shown in Figure 2.4, which allows for backpropagation.

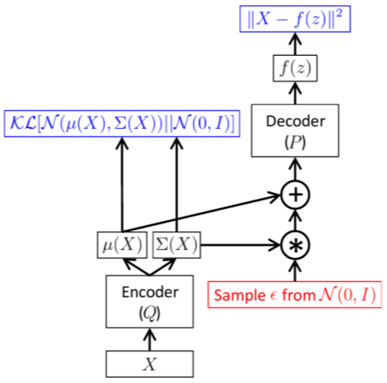

$\|X - f(z)\|^2$
$f(z)$
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)]$
Decoder (P)
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
Sample ϵ from $\mathcal{N}(0, I)$
X

Figure 2.4: The reparametrization trick. Figure from the work of Doersch [2016].

To reduce the potential sparsity of the sampling, a Kullback Leibler (KL) divergence loss (Equation 2.7) is suplemented to the loss function. The KL divergence is a measure of how similar two probability distributions are. By minimizing the difference between the generated normal distribution and the standard normal distribution, the VAE will become much better at handling new unseen data. Combining the reconstruction loss and the KL divergence loss, the final loss function, using MSE as the reconstruction loss, can be seen in Equation 2.8.

$$KL(x, \hat{x}) = \frac{1}{N} \sum_n \hat{x}_n \cdot (\log \hat{x}_n - x_n) \tag{2.7}$$

$$L_{\text{total}}(x, \hat{x}) = MSE(x, \hat{x}) - KL(x, \hat{x}) \tag{2.8}$$

2.3 Reinforcement Learning

As mentioned in Section 2.2, reinforcement learning (RL) is where an agent acts in an environment to learn a task. At each discrete timestep t, a state s is observed by the agent, the agent uses this state to generate an action a and performs it in the environment. Every timestep the agent also receives a reward. This cycle continues until the environment terminates, in which case an episode is finished. Using the reward, the agent can learn whether the action was good or bad. This process, as seen in Figure 2.5, can be modeled as a Markov Decision Process (MDP). In many cases the states of the environment will only be partially observable, making the MDP a partially observable MDP (POMDP).

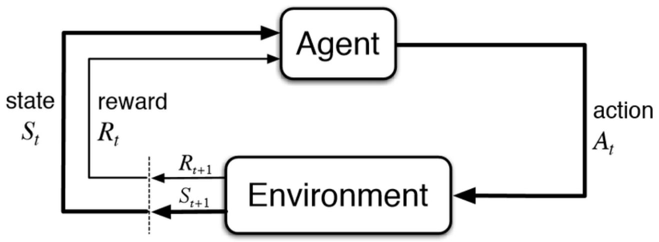


Figure 2.5: The process of reinforcement learning

The advantage this has over other machine learning methods is that it doesn't require a dataset. In games like chess where the possible state space is too big to make a dataset out of, reinforcement learning thrives[1] compared to supervised learning as it learns by doing, much like human beings. Up until now, reinforcement learning has proven to be very efficient at learning games and robotic tasks [Schulman et al., 2017] [Fujimoto et al., 2018].

Agents in reinforcement learning want to achieve maximum cumulative reward $R(\tau) = \sum_{i=t}^{T} \gamma^i r_t$, also called return. Here τ is a trajectory defined as state-action pairs $(s_0, a_0, s_1, a_1, ...)$ and $\gamma \in (0, 1]$ is the discount factor which decides how much to focus on short-term rewards compared to long-term rewards. Very often, the best rewards will come from a specific sequence of actions, and finding these sequences can be difficult. This is known as exploration vs exploitation in reinforcement learning. The trade-off between exploiting known rewards, and exploring new and possibly better rewards. Different types of algorithms does this in different ways.

Rewards received from the environment are gathered from a reward function $R(s, a)$. Designing a good reward function for an environment can be a difficult task. It can make or break the agents performance, or in some cases even make it dangerous [Amodei et al., 2016]. Getting an agent to maximize its return involves finding the optimal policy function. The policy function $\mu(s) : \mathcal{S} \rightarrow \mathcal{A}$ is a function that maps states to actions. In cases where the environment is stochastic, the notation changes to $\pi(a_t|s_t)$ to accommodate for the stochasticity when performing actions. An agent doesn't necessarily need to create a policy function explicitly, but it needs a way to map states into actions. In general, this is referred to as the policy of the agent.

To be able to learn, the policies are usually parametrized in some way. A policy π parametrized by parameters θ is denoted π_θ. The policy is iteratively optimized using either temporal-difference (TD) learning or monte-carlo (MC)

[1]Chess AIs are more often using minimax algorithms rather than reinforcement learning, although reinforcement learning AIs have beaten some of the best minimax chess AIs.

learning. Temporal difference updates the parameters every timestep, while monte-carlo learning updates them every episode.

Value functions are an important part of reinforcement learning. Mainly there are three types of functions. The value function $V(s) = \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)|s_t = s]$ gives the expected return when starting in state s and following the policy π after. The action-value function $Q(s,a) = \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)|s_0 = s, a_0 = a]$ gives the expected return when starting in state s, taking action a and following the policy π after. Finally there is the advantage function $A(s,a) = Q(s,a) - V(s)$ which gives the advantage of being in state s and taking action a relative to the other actions. The value and action-value functions can be decomposed into immediate reward plus discounted future reward:

$$V(s) = \mathbb{E}[R_{t+1} + \gamma V(s_{t+1})|s_t = s] \tag{2.9}$$

$$Q(s,a) = \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a\sim\pi} Q(s_{t+1}, a)|s_t = s, a_t = a] \tag{2.10}$$

These functions are recursive and can be decomposed further to create update functions for the value functions, also called the *Bellman functions*:

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) Q_\pi(s,a) \tag{2.11}$$

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} V_\pi(s') \tag{2.12}$$

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} V_\pi(s') \right) \tag{2.13}$$

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} \sum_{a\in\mathcal{A}} \pi(a'|s') Q_\pi(s',a') \tag{2.14}$$

$P^a_{ss'}$ is the transition probability function which is almost always unknown in model-free approaches. Due to this, the equations cannot be solved directly, but the Bellman functions lays the foundation for many algorithms. Looking at reinforcement learning algorithms, there are two main types: model-free algorithms where the agent learns the model of the environment itself, and model-based algorithms where a model of the environment is given. This thesis will only focus on model-free algorithms. Most model-free algorithms can again be split into two parts: Policy-based algorithms and value-based algorithms. Policy-based algorithms optimize directly on the policy function, which requires the agent to have a policy function explicitly defined. Mathematically this can be written as:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)] \tag{2.15}$$

Value-based algorithms are based on the value functions $V(s)$ and $Q(s,a)$. These algorithms uses the Bellman equations to find the optimal action-value function and creates a policy out of it by taking the action that gives the highest value (Q-value):

$$a_t = \arg\max_{a_{t+1}\in\mathcal{A}} Q^*(s, a_{t+1}) \tag{2.16}$$

In modern reinforcement learning, a combination called actor-critic algorithms are often used. The actor is the policy, mapping states to actions. And the critic evaluates the actions that are taken.

2.3.1 Twin Delayed Deep Deterministic Policy Gradient

Even though many environments are stochastic, deterministic policy functions will still work just fine. Having a deterministic policy function will only give deterministic actions when given a state, while stochastic policies sample from a set of actions. By setting up the objective function with a deterministic policy function

$$\begin{aligned} J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) r(s, \mu_\theta(s))\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\mu}[r(s,\mu_\theta(s))] \end{aligned} \tag{2.17}$$

where $\rho^\mu(s)$ is the discounted state distribution. Silver et al. [2014] shows that the gradient $\nabla J(\mu_\theta)$ becomes

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s)\nabla_\theta\mu_\theta(s)\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\mu}\left[\nabla_\theta\mu_\theta(s)\nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right] \end{aligned} \tag{2.18}$$

This lays the foundation of deterministic policy gradient algorithms. The deep deterministic policy gradient algorithm (DDPG) is one such algorithm. It was introduced by Lillicrap et al. [2015] and takes inspiration from the simple algorithm introduced in the deterministic policy gradient paper and methods in deep Q-learning Mnih et al. [2013].

DDPG starts by initializing 4 networks: an actor network $\mu_\theta(s)$, a critic network $Q_\theta(s,a)$, and target networks μ' and Q' with same weights as their counterparts initially. Since value and action-values are calculated every timestep, updates are performed on target networks instead to ensure stability during training. The actor and critic networks are then slowly tracking the target networks during training using the following update rule:

$$\begin{aligned}\theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}\end{aligned} \tag{2.19}$$

Every timestep DDPG will select an action using the current policy (not the target network) $a_t = \mu_\theta(s)+\epsilon$, where ϵ is the action noise. DDPG uses a Ornstein-Uhlenbeck process for generating action noise [Uhlenbeck and Ornstein, 1930]. The action is then executed in the environment and the transition (s_t, a_t, r_t, s_{t+1}) is stored in a replay buffer $\mathcal{R}$. A minibatch is then sampled from the replay buffer (indexed i) and the loss functions for the critic is defined to be

$$L = \frac{1}{N}\sum_i (y_i - Q_\theta(s_i, a_i))^2 \tag{2.20}$$

where

$$y_i = r_i + \gamma Q'_\theta(s_{i+1}, \mu'_\theta(s_{i+1})) \tag{2.21}$$

The actor loss is defined to be

$$L = -\frac{1}{N}\sum_i Q_\theta(s_i, \mu_\theta(s_i)) \tag{2.22}$$

The loss is then minimized by backpropogating the gradients back into the network parameters θ which is commonly done using automatic differentiation in modern implementations. After the network parameters are updated, the target networks are updated using Equation 2.19. Fujimoto et al. [2018] shows that DDPG suffers from function approximation errors. The main way they address this is by using double Q-learning on the critic function [van Hasselt et al., 2015]. By creating two critics, each with their own target network, and setting $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ where $\tilde{a} \leftarrow \mu_{\theta'}(s') + \epsilon$. The critics can be updated using

$$\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s,a))^2 \tag{2.23}$$

Other improvements that were made were to swap out Ornstein-Uhlenbeck exploration noise for a normal distribution sample $\mathcal{N}(0, \sigma)$ where σ is a constant number. The actor is updated less frequently than the critic to ensure less repeated updates if the critic haven't changed. And finally the action noise during the mini-batch update is clipped. These improvements convincingly beat many other state-of-the-art algorithms in RL on OpenAI Gym environments.

2.3.2 Autoencoders in Reinforcement Learning

The purpose of using autoencoders in reinforcement learning is to reduce the observation space. In environments where observations contains sizeable sensor data like cameras, autoencoders can help reduce that observation to increase the performance of the RL agent. The autoencoder is trained separately before the reinforcement learning model is trained.

First a dataset of observations is gathered from the environment (e.g. using random actions). Then the autoencoder is trained to encode and reconstruct the observations in the dataset. When the autoencoder has finished training, the encoder part is used to encode observations which are then used to train the RL model.

2.3.3 Reinforcement Learning in Autonomous Vehicles

Reinforcement learning in autonomous vehicles is still a relatively new topic. While there exists many papers and success stories covering end-to-end learning in autonomous vehicles, reinforcement learning still has a long way to go. The feasability of reinforcement learning in autonomous vehicles is no longer a question. Kendall et al. [2018] has demonstrated that the data-efficiency has come a long way and that a reinforcement learning agent can learn the basics of driving in under a day. One issue related to the training of these agents is the safety. Reinforcement learning agents learn by doing. Realistic simulations is a common starting point for many methods, only limited by computational power. While most scenarios can be added in a simulation, the transferability of models to the real world can be problematic. Real world environments are also limited, mostly due to safety reasons, but also due to the cost.

2.4 Related Work

The preceding work for this thesis [Hådem, 2019] explores the different reinforcement learning algorithms more in-depth. Both discrete and continuous algorithms are covered and compared to each other to a certain extent. The specialization project also covers the basics of autonomous vehicles and how it is done in practice with reinforcement learning. Vergara [2019] has done much in the same domain while focusing on accelerating training of reinforcement learning agents in the field of autonomous vehicles.

The work of [Ha and Schmidhuber, 2018] takes a lot of inspiration from the human cognitive system. Humans build an internal, temporal representation of the world based on sensor input and performs action that are based only on this internal representation. The model uses a variational autoencoder, and a large portion of the paper is based on the success of learning the OpenAI Gym CarRacing environment which is a rather simplified driving game.

Kendall et al. [2018] showed that continuous reinforcement learning models can learn lane-following very quickly. They also show that using a VAE can drastically improve the data efficiency during training. The reward function, defined as forward speed and terminates upon infringing traffic rules, is very simple while giving a trained agent the desired features.

As for the transferability of models from simulation to reality, Osiński et al. [2019] did 9 similar experiments using Carla and reinforcement learning to test the transferability of models by testing them out in the real world. One technique they use to generate more stable policies in general and for transferring is domain randomization which is to randomize the rendering in a simulation. This technique is explained more detailed in the work of Tobin et al. [2017]. They did not use autoencoders at all however.

Chapter 3

Autoencoder Experiments and Results

3.1 Experimental Plan

To get a better understanding of the capabilities of autoencoders, a series of experiments will be conducted on the MNIST and CIFAR10 datasets. The images found in MNIST are simple and have a lot of information that can be thrown away like the dark background. CIFAR10 images are much more complex since they are images taken from real scenes. Knowing what is the background in these images can be a much harder task. MNIST and CIFAR10 have 70 000 and 60 000 images respectively.

The first experiment will be on the simplest form of an autoencoder, a fully-connected single-layer encoder and decoder. After this, more depth and complexity will be added to see if it can improve the autoencoders ability to reconstruct images. Experiments using CNNs are expected to perform better than fully-connected ones, especially for CIFAR10 images as they are more complex compared to the digits found in MNIST.

The implementation of the autoencoders are written in Python 3.5 using PyTorch. The optimizer used is the Adam optimizer with default parameters except for the learning rate found in the paper by Kingma and Ba [2014]. For the loss function, both mean squared error and binary cross entropy were experimented with slightly. The reconstructed images were pretty much identical so BCE loss was chosen as the loss function. The ReLU activation function was used after every layer except for the last layer in the decoder where the sigmoid activation function was used to force values between 0 and 1. An overview of the hyperparameters used can be found in Table 3.1. Batch size, number of epochs, latent-vector size and network architecture differ from experiment to experiment and is specified in the individual experiments.

Learning rate	0.001
Loss function	BCE
β_1	0.9
β_2	0.999
ϵ	10^{-8}

Table 3.1: Common hyperparameters for all autoencoder experiments. β and ϵ are parameters found in the Adam optimizer.

All code used for the experiments can be found here: `https://gitlab.com/chrissha/master_thesis_autoencoder`. The experiments was performed on a desktop computer running Ubuntu 18.04 with Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz and GeForce GTX 1080 Ti. The time it took to run the different experiments can be found in the corresponding results in Section 3.2.

3.2 Results

The first set of experiments are done on the simplest possible autoencoder where the input is directly mapped into the latent-vector with a fully-connected layer as seen in Figure 3.1. Training a one-layer fully-connected autoencoder on MNIST gave the results found in Figure 3.2. The network was trained for 1 epoch (60 000 images) with a batch size of 64. The figures contains 10 random samples from the test set with the original image on top, and the reconstructed image at the bottom. These three experiments took about 4 seconds to train each.

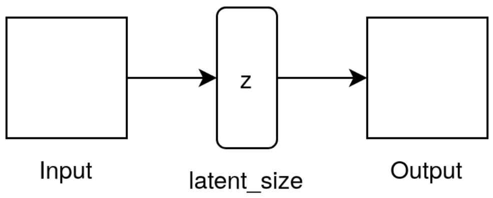
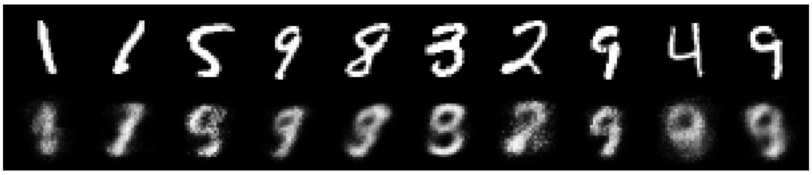
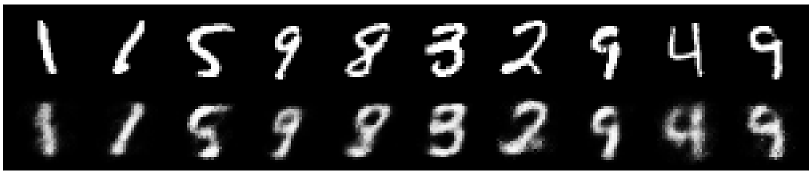


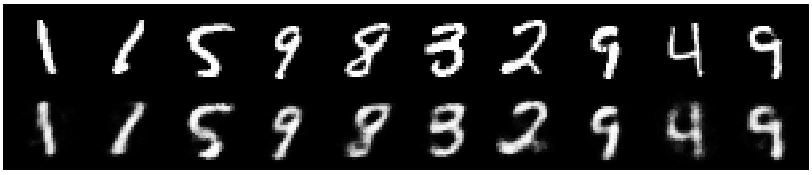
Figure 3.1: Basic fully-connected autoencoder



(a)



(b)



(c)

Figure 3.2: Results of a single-layered fully-connected autoencoder with a latent-vector size of 16, 32 and 64 are found in (a), (b) and (c) respectively.

Sticking to a latent-vector size of 32, but swapping to a deeper fully-connected network shown in Figure 3.3 should give the network more capacity. The network consists of 3 hidden layers, each with 400, 250 and 100 nodes respectively. Training for 1 epoch with a batch size of 64 gave the results found in Figure 3.4 (a) while training for 10 epochs with a batch size of 256 gave the results found in Figure 3.4 (b). One epoch in both of these experiments took about 5,5 seconds to train which means about 55 seconds for 10 epochs.

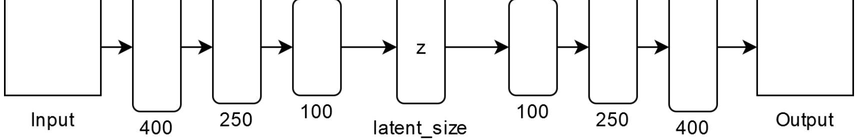
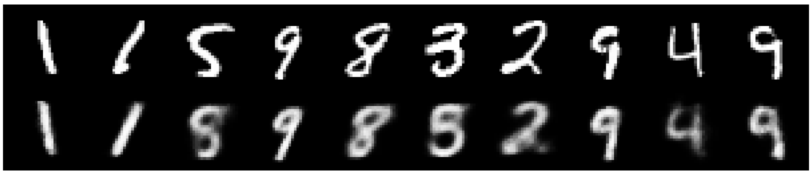
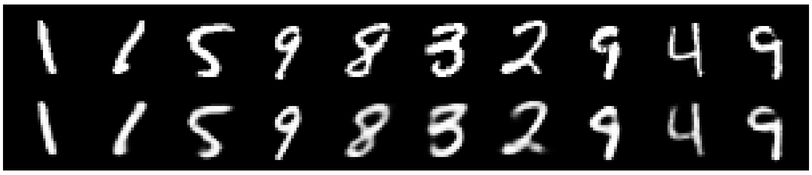


Figure 3.3: Deep fully-connected autoencoder



(a)



(b)

Figure 3.4: Results of fully-connected autoencoder with 3 hidden layers. (a) was trained for 1 epoch with 64 batch size, (b) was trained for 10 epochs with 256 batch size.

Moving on to the CIFAR10 dataset which contains much more complex images, the fully connected network will no longer suffice. This is demonstrated by an experiment shown in Figure 3.5 where the same deep fully-connected network from the previous experiment was used (Figure 3.3). Even though the input images are slightly bigger (32x32x3 instead of 28x28x1), the training time is about the same as the previous experiment. The colors are mostly correct, but every reconstructed image is significantly washed out and blurry. To improve this, networks with more capacity are required.

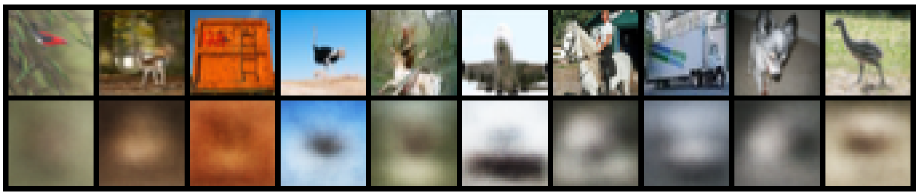


Figure 3.5: CIFAR10 dataset trained on a deep fully-connected autoencoder.

Changing to a VAE as well as using CNNs should give the capacity needed for the CIFAR10 experiments to improve. The architecture chosen is a typical CNN architecture which gradually scales down the image size while increasing the amount of channels in the tensor. The architecture can be found in Figure 3.6. For more details on the architecture see Appendix A.1. The creation of the μ and

σ vectors are done using a fully-connected layer. The last layer in the decoder is also a fully-connected layer. This did not increase the training time much at all. It took around one minute to train for 10 epochs with a batch size of 64. The results of the CNN VAE network using MNIST and CIFAR10 can be seen in Figure 3.7.

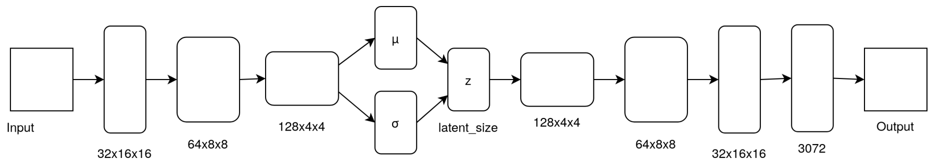
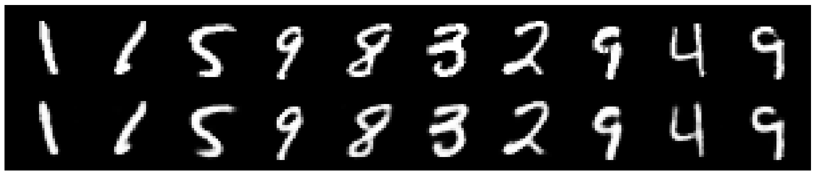
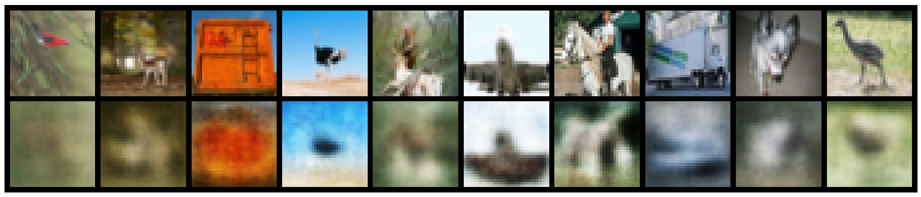


Figure 3.6: CNN VAE network used to train an autoencoder on MNIST and CIFAR10



(a)



(b)

Figure 3.7: CNN VAE network trained on MNIST and CIFAR10

Finally the size of the latent-vector can be increased to 64 to give slightly better reconstruction results as shown in Figure 3.8.

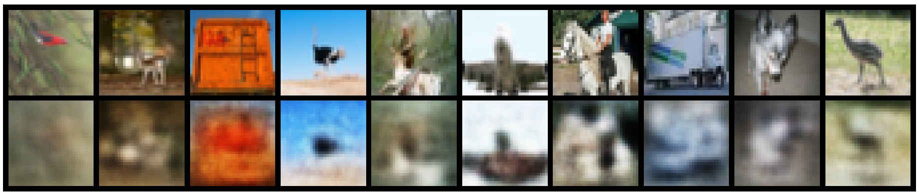


Figure 3.8: CNN VAE network with latent-vector size of 64

3.3 Discussion

Looking at the differences in latent-vector size in Figure 3.2, the differences between 16 and 32 nodes is much greater than the difference between 32 and 64 nodes. This pattern is likely to continue when going for even larger latent-vector sizes as more information can be encoded. The desired size of the latent-vector in this case is as small as possible so that the observation space in reinforcement learning can be smaller. Depending on the use case, as long as the encoded data can still be distinguished from another, a very low latent-vector size can be chosen. Deeper networks does make a difference as shown in Figure 3.4, especially when trained for a longer time. Training for 10 epochs with a latent-vector size of 32 gives much better results than training for 1 epoch with size 64.

One metric that was not included in these experiments was the average loss each epoch. Visually inspecting the reconstruction results is a good indicator for how much information that is kept during the compression, but a reinforcement learning algorithm is going to use the encoded data, not a human, so the loss value could be a great metric to have besides this.

Overall autoencoders are much better at encoding images where parts of the image are the same color, like in cartoons. Realistic images are harder to encode as they may contain more detailed information that is necessary to keep.

Chapter 4

CarRacing Experiments and Results

CarRacing is a reinforcement learning environment that can be found in OpenAI Gym [Brockman et al., 2016]. In this environment a car is controlled using steer, gas and brake which are all continuous. A reward is given each time a track tile is entered while a small negative reward is given each frame. An episode ends whenever a lap is complete, the car is outside the play field (gives a large negative reward) or the max number of timesteps has been reached. The observations in this environment are RGB images from a top-down perspective as seen in Figure 4.1. The authors claim that this is their easiest continuous control task to learn from pixels.

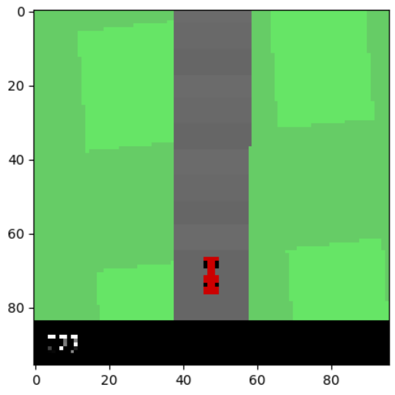


Figure 4.1: An observation in the CarRacing environment.

4.1 Experimental Plan

To test out how autoencoders can assist reinforcement learning, the autoencoder must first be trained separately. Reinforcement learning doesn't require a dataset, but training an autoencoder does. Such a dataset can be acquired by collecting observations in the environment with random or human controlled actions. The dataset used here will be collected using random actions. During reinforcement learning training, the trained autoencoder is not updated anymore.

Ha and Schmidhuber [2018] have done some experiments using a VAE on the CarRacing environment. Their model uses a VAE followed by a RNN layer into a single fully-connected layer. They chose to go with a very simple network to map states to actions and instead focus more on preprocessing the observations resulting in very few parameters in the final network. They trained it using CMA-ES which is an evolutionary algorithm. In this thesis a much more sophisticated algorithm will be used to learn the policy function, more specifically, the twin delayed DDPG (TD3) introduced in Section 2.3.1. The reason for choosing this algorithm is covered more deeply in the preceding work to this thesis [Hådem, 2019].

The autoencoder will be trained similarly to the one in Chapter 3, but the network will be the same CNN that Ha and Schmidhuber [2018] used which can be seen in Figure 4.2. Full details about the VAE can be found in Appendix A.2. The trained autoencoder will then be used to compress observations during reinforcement learning to efficiently train a TD3 model. Some hyperparameter tuning may have to be done until a good model has been found. The goal of these experiments is to put together a model using an autoencoder to see how it performs before going forward with Carla experiments.

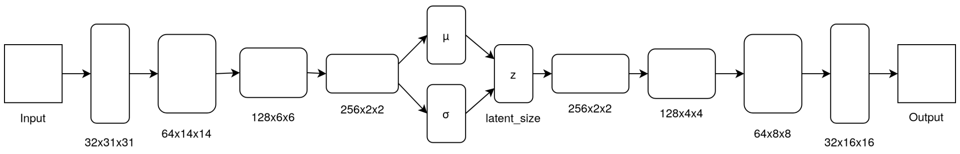


Figure 4.2: Architecture of the VAE used in the CarRacing experiments

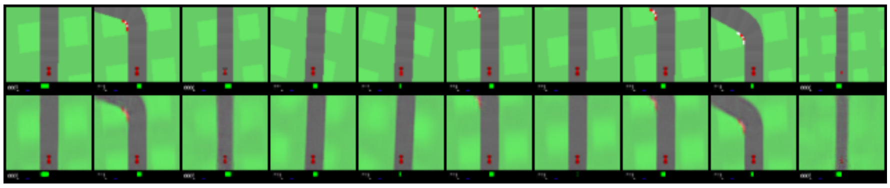
4.2 Implementation

The same implementation of the variational autoencoder used in Chapter 3 was used in these experiments as well. The implementation of the TD3 algorithm is also written in Python 3.5 and started out as an implementation of DDPG since TD3 is an extension of DDPG (see Section 2.3.1). After DDPG was implemented and tested properly, the extension to TD3 was very simple. The correctness of the implementations were tested by running OpenAI Gym environments [Brockman et al., 2016] similarly to how the algorithms were tested in the original papers. A wrapper class, supported by OpenAI Gym, was used to modify the CarRacing environment so it would return observations processed by the autoencoder.

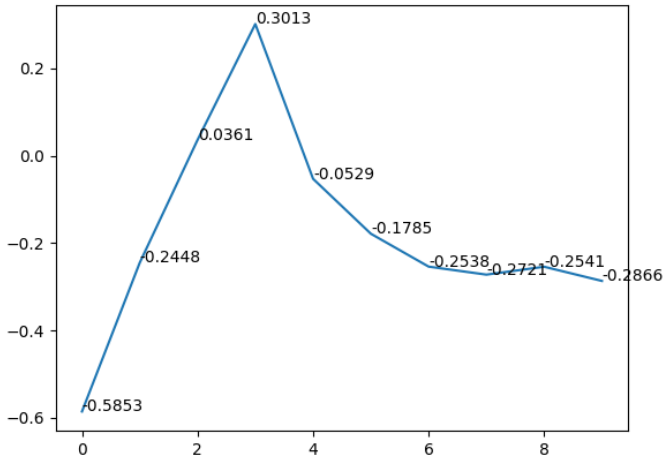
The code for the TD3 algorithm and the CarRacing experiments can be found in the following repositories respectively `https://gitlab.com/chrissha/td3` and `https://gitlab.com/chrissha/master_thesis_car_racing`. These experiments were done on a dekstop Ubuntu computer running AMD Ryzen 5 3600 Processor with a Gigabyte GeForce GTX 1070 G1 Gaming GPU.

4.3 Results

Training the VAE for CarRacing was done using a dataset with 10 000 images of size 64x64x3 collected using random actions. The images were resized to 64x64 as the original observations from the environment are 96x96. A batch size of 64 and a learning rate of 0.001 was used. It was trained for 10 epochs. The results of the VAE training can be seen in Figure 4.3



(a)



(b)

Figure 4.3: VAE experiment on CarRacing. *a)* contains reconstruction samples with the original images on the top row. *b)* shows the average loss each epoch.

Training a TD3 reinforcement learning model with the same hyperparameters (except for replay buffer size which was set to 50 000) as in the TD3 paper (see Table 4.1) the agent did not manage to learn anything meaningful when using a VAE and training for 100 000 timesteps. All of the experiments that Fujimoto et al. [2018] performed on TD3 where trained for 1 000 000 timesteps. Since these CarRacing experiments will only be trained for 100 000 timesteps due to limited resources, it would be reasonable to change some of the hyperparameters. The size of the replay buffer was set to 5000 and τ was increased by a factor of 10 as it controls the speed of which the target networks track the trained networks. These changes makes it so the agent is able to learn the driving task to some extent as seen in Figure 4.4. The figures shows the average reward over

10 episodes together with min-max shading. These rewards are gathered during evaluation phases where the TD3 model evaluates the agent by sampling actions without using any exploration noise or performing any learning while doing the evaluation. Thus the last evaluation shown in the figures should be a pretty good indicator for how the trained model performs.

Hyper-parameter	**TD3 default value**
Critic Learning Rate	10^{-3}
Critic Regularization	None
Actor Learning Rate	10^{-3}
Actor Regularization	None
Optimizer	Adam
Target Update Rate (τ)	$5 \cdot 10^{-3}$
Batch Size	100
Iterations per time step	1
Discount Factor	0.99
Reward Scaling	1.0
Normalized Observation	False
Gradient Clipping	False
Exploration Policy	$\mathcal{N}(0, 0.1)$
Replay buffer size	50 000

Table 4.1: Default TD3 hyperparameters except for replay buffer size. Taken from the TD3 paper.

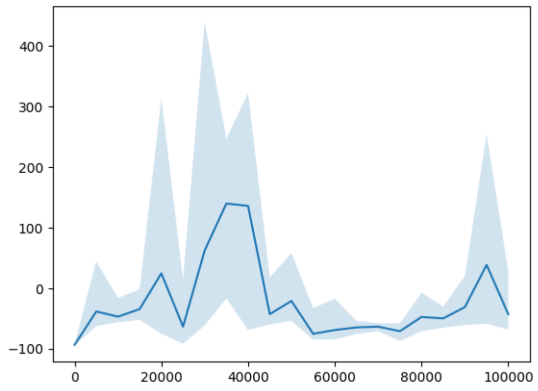


Figure 4.4: Results of training after reducing replay buffer size and increasing τ

Repeating this experiment however gave high variance in the obtained rewards. Running the experiments with the same hyperparameters, but lowering the learning rate to $1e-4$ generally increased the performance of the agent and made it much more consistent, one example of this can be seen in Figure 4.5.

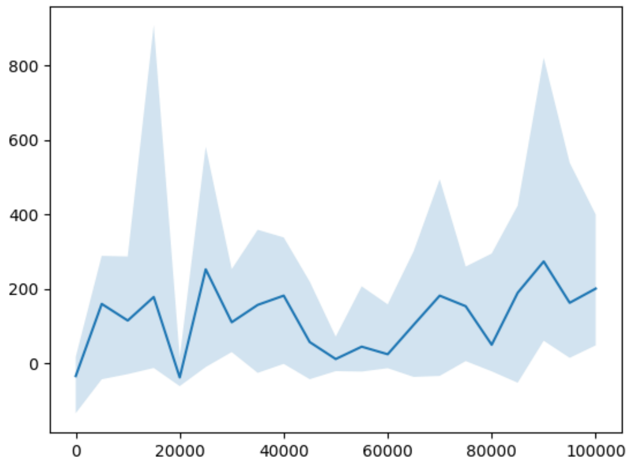
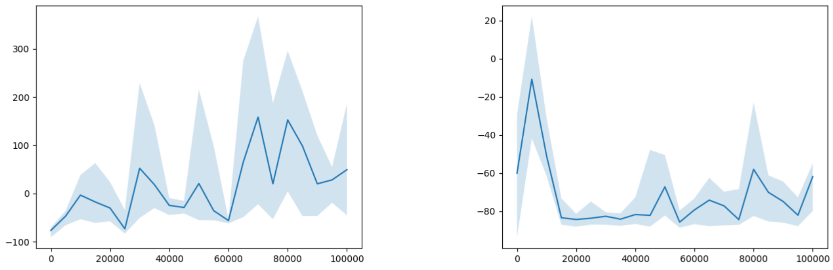


Figure 4.5: Results after lowering the learning rate by a factor of 10

Lowering the learning rate even more didn't seem to improve the training. It would also get stuck sometimes near the worst possible reward, which is obtained by doing almost no actions at all. Two examples of training using a learning rate of $1e-5$ can be found in Figure 4.6.



(a) A good attempt with $1e-5$ learning rate (b) A bad attempt with $1e-5$ learning rate

Figure 4.6: Two attempts using a learning rate of $1e-5$

DDPG uses a lower learning rate on its critic, some attempts using an actor learning rate of $5e-4$ and a critic learning rate of $5e-5$ gave the results found in Figure 4.7.

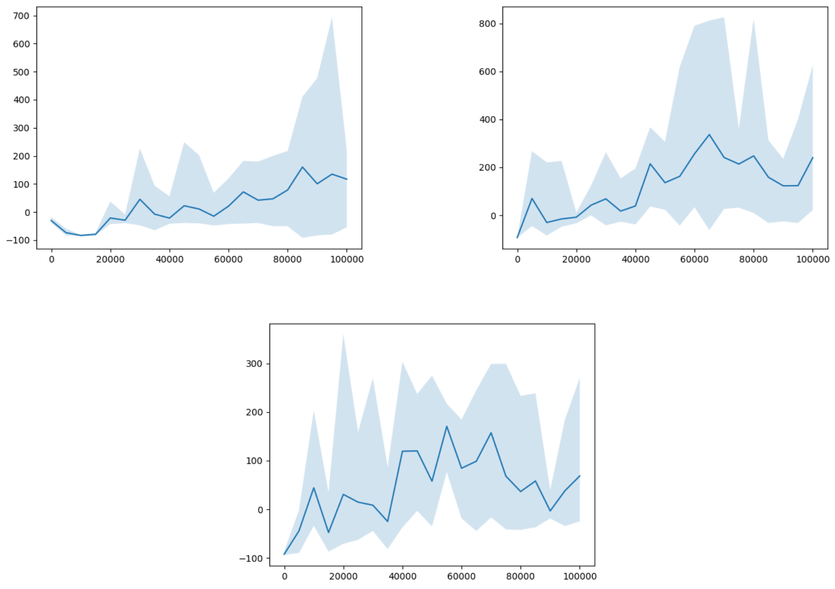


Figure 4.7: Some attempts using a lower critic learning rate than actor learning rate.

A video has been made to demonstrate one of the agents in action. The agent used in the video is the same agent that produced the bottommost figure in Figure 4.7. To see the video visit `https://youtu.be/yZP922ukDsQ`. The video features 3 different cases. In the first clip the agent fails the first turn because it has too much speed and the turn is very sharp. In the second clip the agent performs very well and manages to recieve a high reward. However it fails later on when it reaches some sharp turns after having built up a lot of speed. In the third clip the agent gets stuck at the first turn. This could be because the autoencoder gives an unusual encoding which the reinforcement learning algorithm does not understand.

4.4 Discussion

The dataset used to train the VAE was collected using random actions. However every possible orientation of turns are not accounted for in the dataset. Random actions will, on average, not turn the vehicle very much and not reach that many turns each episode. For the most part the autoencoder did very well when encoding and decoding images as the most important part was to differentiate between what is road and grass. Another edge case is at the very beginning of the simulation when the camera is very far away and about to zoom in closer (see Figure 4.8), although the agent mostly want to just accelerate during this part anyways. Diversifying and creating a balanced dataset could be beneficial for the VAE, but would require manual driving and handpicking images to cover most cases.

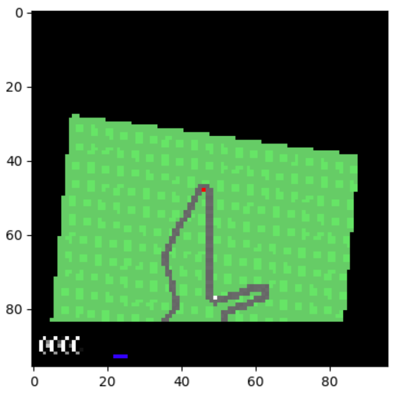


Figure 4.8: The first observation of the CarRacing environment

As for the results, the agents seen in Figure 4.7 did perform quite well when tested out. It is a little inconsistent sometimes, but it manages most of the turns. The sharpest turn, which can be seen in Figure 4.8 is very hard for the agent to do properly as it usually has a lot of speed when entering the turn. This problem also occurs in the work of Ha and Schmidhuber [2018], but there it manages to recover more often. In most of the cases where the agent did not achieve a good reward, it simply accelerates too fast and fails to make the first turn. When the vehicle fails a turn and lands in the middle of the grass it is not able to make its way back. Since there is a large negative reward received when driving outside of the map, the agent simply stops driving if it cannot find the road.

To compare this to an agent that doesn't use a VAE, let's look at the differences. When using a VAE, the CNN part of the model lies in the encoder which is trained beforehand for only 10 000 timesteps. The actor and critic can then be swapped out for fully-connected networks. Since the reinforcement learning part is trained for much longer (100 000 timesteps), training fully-connected layers will be much faster than training CNN layers. Training an agent without a VAE moves the CNN networks to the actor and critic, which will significantly slow down the training. This will be experimentally tested towards the end of the next chapter.

Chapter 5

Carla Simulator Experiments and Results

Carla [Dosovitskiy et al., 2017] is a highly customizable, realistic driving simulator designed to work with machine learning and artificial intelligence. The simulator is built on top of Unreal Engine and uses a Python interface for configuring the environment, reading input and performing actions.

5.1 Experimental Plan

The model used in the Carla experiments will work similarly to the model used in the CarRacing Experiments (see Section 4.1). An autoencoder followed by a TD3 model consisting of an actor and critic with fully-connected networks. The same CNN network for the VAE will be used which inputs $64 \times 64 \times 3$ images.

The first step will be to train the autoencoder and specialize it for Carla input images. A dataset will have to be collected from Carla using random actions. The Carla simulator will be set up to be as simple as possible to begin with, meaning only clear and sunny weather. The map used in all experiments was the default ”Town01” which contains simple 3 way road junctions that can be found almost anywhere. The downside to this map is that it doesn't have that many turns, and the turns that it has are all $90°$.

The latent-vector size will be 32 and the CNN used will be the same as the one used in CarRacing experiments. The aim is to have an autoencoder where the loss value is as low as possible. Remember that the loss function also contains the KL divergence loss and thus, can be negative (see Equation 2.8). After the autoencoder has been trained, the TD3 model can be trained similarly to the model used in CarRacing. The initial experiments training the autoencoder will be done using the same hyperparameters as CarRacing (see Section 4.3) as a basis. This means the base will be 10 epochs, a batch size of 256 and a learning rate of 0.001.

To answer research question 2, a comparison has to be made to models that don't use autoencoders. In this case, such a model will be a standard TD3 agent with a custom CNN actor and critic which can be seen in Figure 5.1. The CNN is similar to the encoder from the previous experiments. The output of the actor will be the continuous action vector while the critic needs to input the action as well as the observation and output a single value.

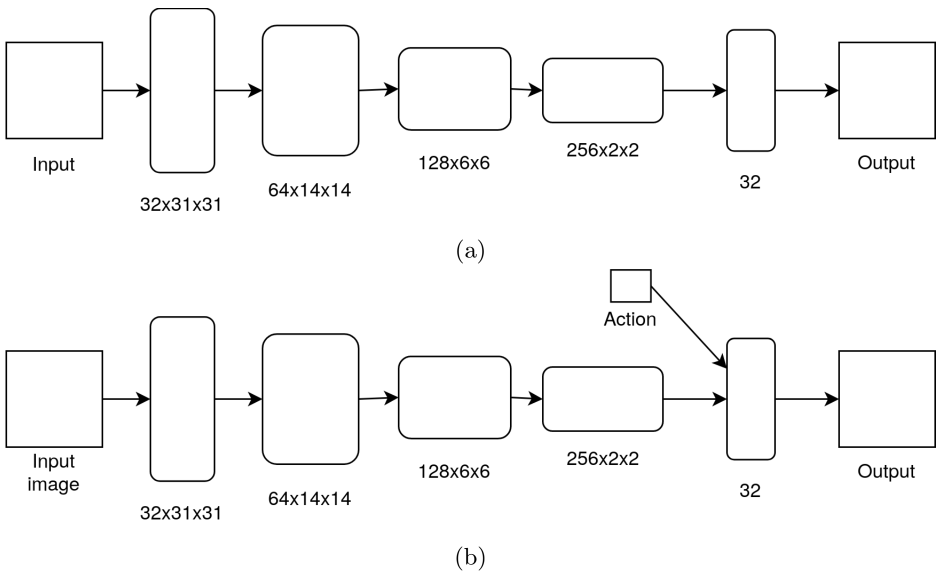


Figure 5.1: CNN architecture for actor (a) and critic (b) used in the TD3 model without VAE.

5.2 Implementation

The autoencoder part was the same as the one used in the CarRacing experiments (see Section 4.2). A custom OpenAI Gym environment was created around Carla, hereby referred to as the ”Carla environment”. This environment loads a given map, spawns an Audi A2 at a random location and attaches a RGB camera sensor at the front of the vehicle. No other vehicles or pedestrians are spawned.

Observations are collected using the RGB camera. The environment can be configured to collect observations every n frames instead of every frame to increase real-time performance. If this is the case, then the environment will repeat the previous action during the skipped frames. Observations can be of any size, but in these experiments they are set to 64x64 RGB images. The three actions defined are throttle, steer and break. Throttle and break are real values between 0 and 1 while steer is between -1 and 1. The reward given each frame is defined as the forward velocity of the vehicle. If the speed is above the speed limit or the car has collided, the episode ends.

The dataset used to train the VAE was collected using the Carla environment. During the collection, the value of the break action was permanently set to be 0 otherwise the car wouldn't move out of its initial spot. All code used to perform these experiments can be found here: `https://gitlab.com/chrissha/master_thesis_carla`. Similarly to the autoencoder experiments, the experiments was performed on a desktop computer running Ubuntu 18.04 with Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz and GeForce GTX 1080 Ti.

5.3 Results

The dataset was collected at ”Epic” quality since this best simulates reality. Three random frames collected from this dataset can be observed in Figure 5.2. The reward numbers are displayed the same way as in CarRacing, every point is from an evaluation phase where no action exploration or learning is performed. As such the last evaluation should be a decent indicator for how the model performs in practice. The values plotted are average reward with min-max shading.

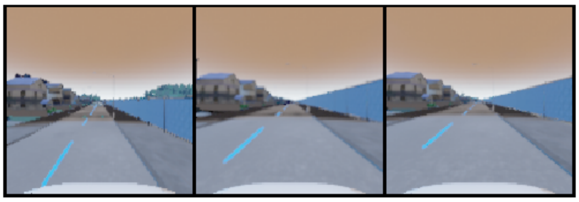


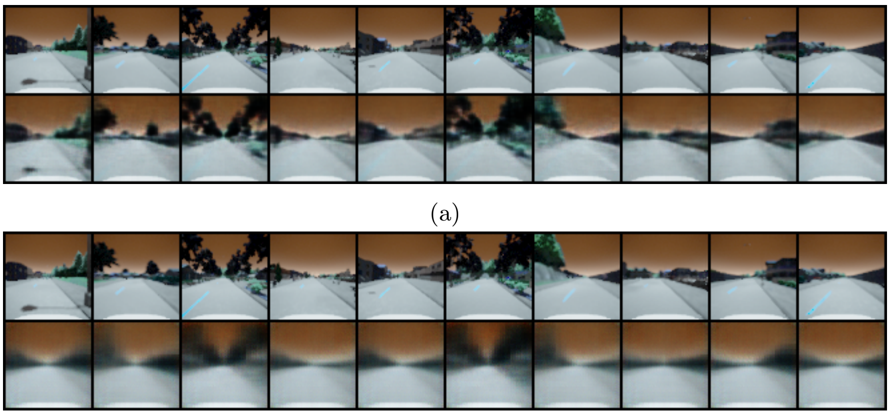
Figure 5.2: Three random samples from the dataset collected in Carla

To minimize the loss function as much as possible on the VAE, a hyperparameter search was performed with various batch size and learning rate combinations. Table 5.1 contains all experiments with the different hyperparameters used together with the final loss value and training time. Initially experiments 1-4 was done to develop a better baseline. As a lower batch size and slightly higher learning rate seemed to indicate a lower loss value, more experiments (5-7) were created around these results. The final experiment (8) was done with 1 epoch to see how it would compare to the rest.

Experiment	**Epochs**	**Batch size**	**Learning rate**	**Loss**	**Time (seconds)**
1	10	256	0.001	1.4070	360
2	10	256	0.01	-0.3197	362
3	10	256	0.0001	0.7135	360
4	10	64	0.001	-2.4760	401
5	25	64	0.001	-2.1980	1005
6	10	64	0.005	-0.6981	402
7	10	256	0.005	-0.5103	362
8	1	64	0.001	-1.9279	31

Table 5.1: VAE experiments on Carla dataset

Experiment 4 gave the lowest loss value. Experiment 5 is a good contender, but it was trained for 25 epochs and could suffer from some overfitting. Experiment 8 could be a decent choice as well since Ha and Schmidhuber [2018] used a VAE trained only 1 epoch. Taking a more visual look at the reconstruction results for experiment 4 and 8:


(a)

(b)

Figure 5.3: Results from VAE experiment 4 (a) and experiment 8 (b)

The VAE obtained from experiment 4 seems to be one of the best performing ones. The loss graph over time for the VAE in experiment 4 can be found in Figure 5.4. This is the VAE that will be used in Carla experiements going forward.

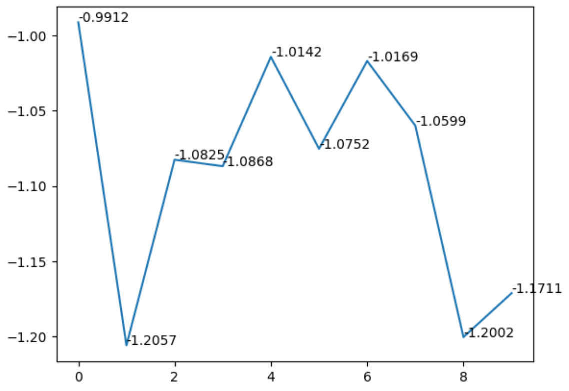
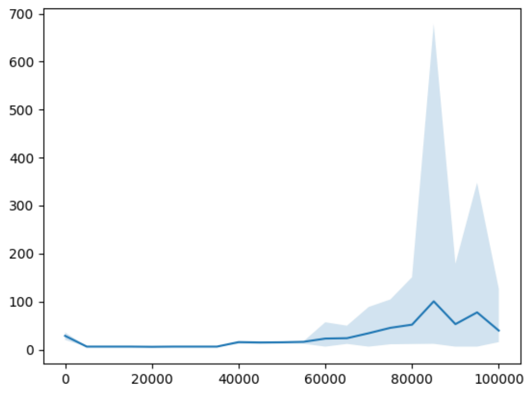
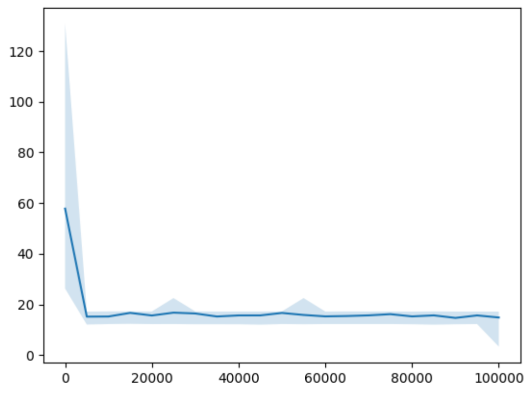


Figure 5.4: The loss over time for experiment 4

When training Carla for 100 000 timesteps, the same adjustments were made to τ and the replay memory size as in CarRacing. Two attempts using these parameters did not yield anything significant as seen in Figure 5.5. The training took 6 hours due to some technical issues, but it is estimated to take around 2 hours in normal conditions. Most of the hyperparameters were set to the default values except τ and the replay buffer size which were set to 0.05 and 5000 respectively. For a complete list of the default values for TD3, refer to Table 4.1.



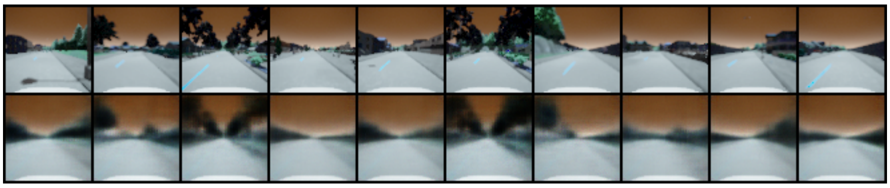
(a)



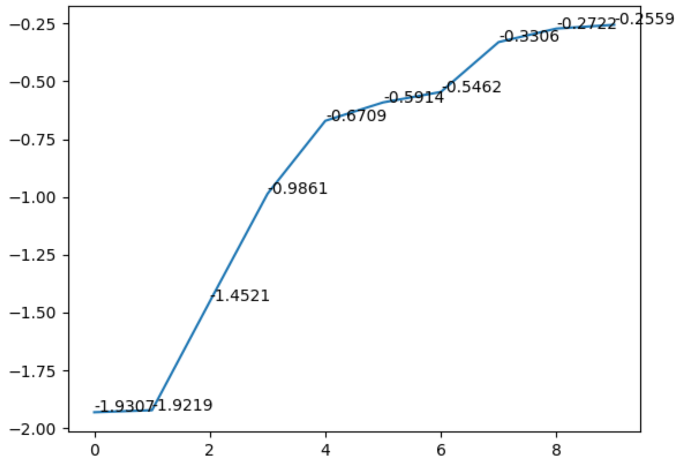
(b)

Figure 5.5: Results from two basic Carla experiments using a VAE

The current results are not very successful. Giving the model more capacity in terms of input image size and latent-vector size could improve the agent. A combination of two changes are done in an attempt to improve the model: Changing the input images to 128x128 and the latent-vector size to 64. For this, a new separate dataset had to be collected for 128x128 images. The VAEs were trained similarly to experiment 1 in Table 5.1. The results for the VAE training can be found in Figures 5.6 through 5.8. For the experiment with 64 image size and 32 latent-vector size, refer to experiment 1 in Table 5.1.

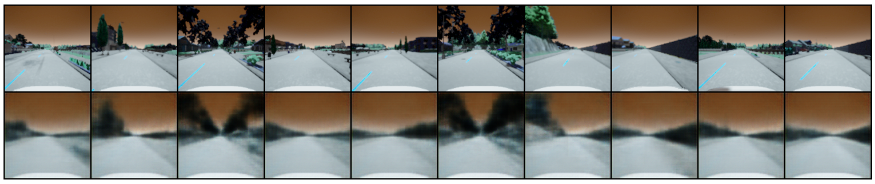


(a)

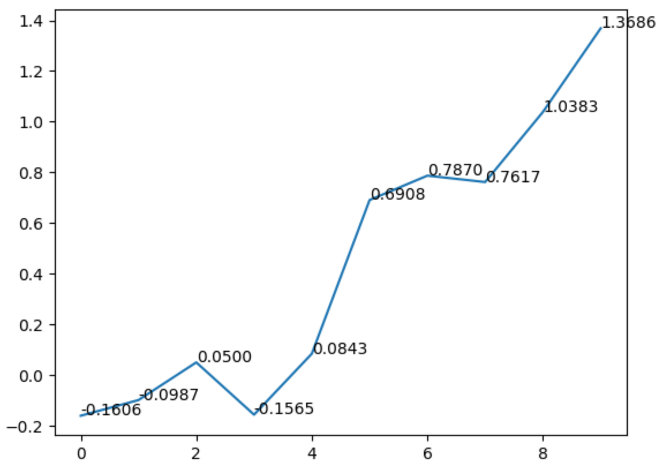


(b)

Figure 5.6: VAE experiment with 64 image size and 64 latent-vector size

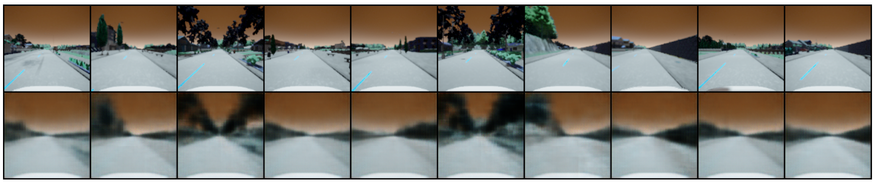


(a)

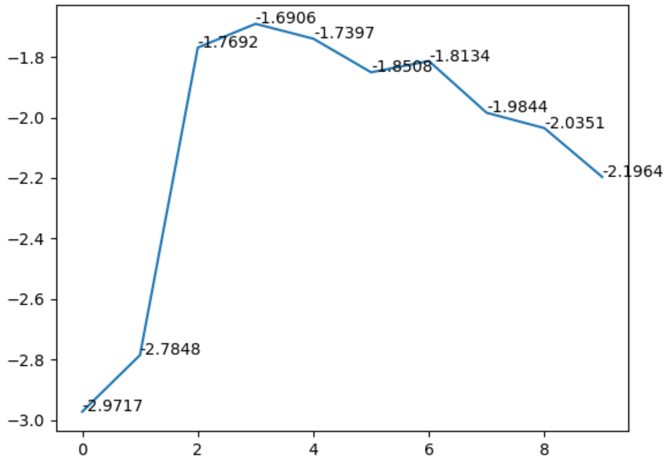


(b)

Figure 5.7: VAE experiment with 128 image size and 32 latent-vector size



(a)



(b)

Figure 5.8: VAE experiment with 128 image size and 64 latent-vector size

Attempting to train a TD3 model using these trained VAEs gave the results found in Figure 5.9, 5.7 and 5.11

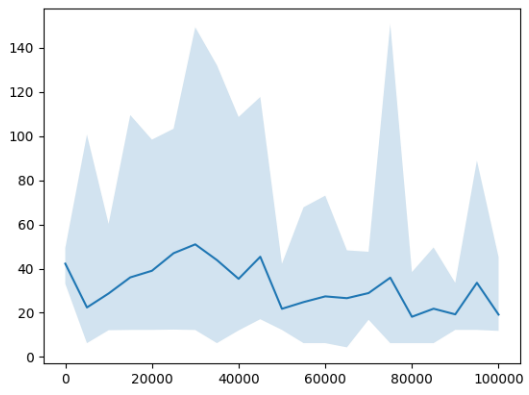


Figure 5.9: Model using 64x64 images encoded into 64-size latent vector

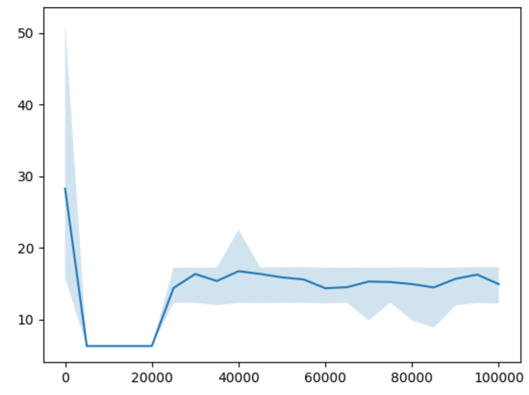


Figure 5.10: Model using 128x128 images encoded into 32-size latent vector

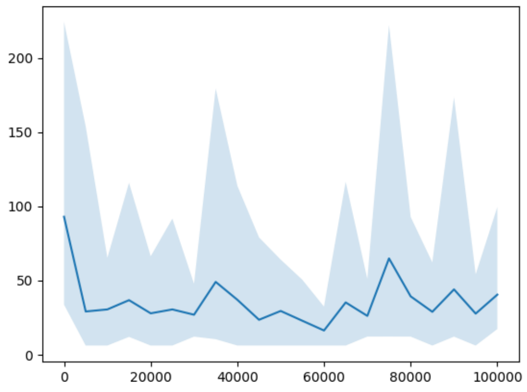


Figure 5.11: Model using 128x128 images encoded into 64-size latent vector

The Training with a TD3 model without using a VAE took much longer as it processes the raw 64x64 (or 128x128) image every optimization step in the reinforcement learning algorithm. The results can be seen in Figure 5.12. The training consisted of 100 000 timesteps which took about 10 hours.

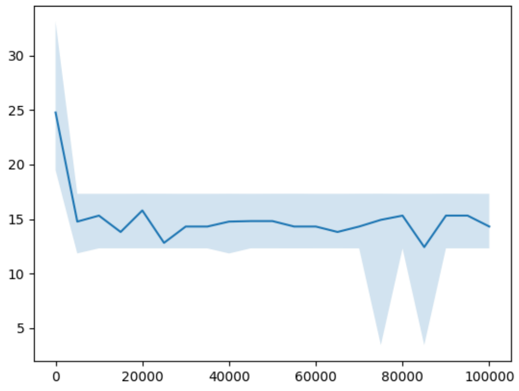


Figure 5.12: Results from a TD3 model without VAE

A test of how the trained agents performed when driving in the city was not performed. Even though the evaluation phase in TD3 does not use any action noise, it would definitely be useful to see how the agent performs when driving around. A reward of 100 in this specific implementation is very difficult to visualize without anything else.

5.4 Discussion

Although VAE experiment 4 has the lowest loss value, the concept of overfitting is not really investigated here. Ha and Schmidhuber [2018] only trains their VAE for 1 epoch (with 10 000 images), but doing the same in this scenario leads to very blurry reconstruction images which leads to losing valuable information such as lane lines. For these reasons, the VAE were trained for multiple epochs, were 10 seemed to give reasonable reconstruction images. Training for even more epochs did seem to trend towards a lower loss value, but since overfitting is not measured, 10 epochs was chosen as sufficient.

Thaking a look at the results of increasing the VAE models capacity, the final loss values did not exceed the one of experiment 4 in Table 5.1. The VAE using 128 image size and 64 latent-vector size came the closest with -2.1964. Another interesting observation is how all of them have their lowest loss value after the first epoch. This could be the reason as for why Ha and Schmidhuber [2018] only choose to train for one epoch, but the comparison between one epoch and 10 epochs as shown in Figure 5.3 shows that one epoch gives poor reconstructions. Further experiments should be made to see which one gives the better result when combining the VAE with a reinforcement learning model.

Increasing the models capacity both in terms of latent-vector size and image resolution did improve the agents performance. Looking at the results from using 32 latent-vector size and comparing them to the results of using a 64 latent-vector size, it seems that the latent-vector does play a significant role in the performance of the final model. The results shown in Figure 5.10 gave no more reward than the results in Figure 5.5 where both averages no more than 20.

Chapter 6

Discussion

6.1 Overall Discussion

Reinforcement learning models are definitely capable of learning to drive a vehicle as seen in the CarRacing experiments. However the complexity quickly increases as the environments gets more realistic. A VAE model that was more than capable of following a road in CarRacing, did not have the necessary capacity to successfully learn the same task in Carla. This was however fixed by increasing the latent-vector size so that more information could be kept when encoding images. In practice this comes at the cost of real-time performance, so depending on the hardware available in the vehicle, the latent-vector size could be increased even further.

There are many more reinforcement algorithms that can tackle continuous environments such as autonomous driving. At the time of writing this thesis, TD3 is a state-of-the-art algorithm for general reinforcement learning. PPO is a great contender, supporting both discrete and continuous action spaces. PPO uses stochastic sampling of actions instead of discrete which may affect how autonomous vehicle agents operate.

In general it seems that reinforcement learning models in autonomous vehicles require a lot of capacity. However, when the capacity is good enough, these models are quick learners. Combine this with a properly designed reward function and the end result is a flexible agent that can drive smoothly for long distances.

6.2 Research Questions

Starting with research question 2: "How does autoencoders affect the performance of reinforcement learning agents?". Out of the experiments that were conducted in both CarRacing and Carla, it can with confidence be said that autoencoders reduces the training time of reinforcement learning agents. Even though the autoencoder has to be trained separately and a dataset might have to be collected, using an autoencoder reduces the training time for the reinforcement learning algorithm. The longer the reinforcement learning agent is trained for, the bigger the difference will be in training time.

For a direct comparison taken from Section 5.3, training a VAE and a reinforcement learning model together took around 2 hours, while training a model without VAE took around 10 hours. Both of these models were trained for 100 000 timesteps. If the model were trained for even longer, like 1 million timesteps, the difference would be even more noticeable. Ignoring the results obtained by the trained agents, this increased performance while training can be very useful in real-life situations were a vehicle needs to learn real-time on limited hardware.

The average reward obtained by the agents using a VAE were very close to the ones without. A slight advantage can be found when using 64 latent-vector size as seen in Figure 5.9. These results are of very low quantity however and since the architecture is slightly different when not using a VAE, conclusions cannot be taken from this comparison.

Moving on to the third research question which asked the question of how an agents performance increases as the models capacity increases. There weren't many experiments to highlight this, but the increase in image size and latent-vector size gave an indicator. When only the image size was increased the results so little to no improvement compared to increasing the latent-vector size. This indicated that 64 image size might contain enough information, and that the model is missing something elsewhere instead to observe and perform the right action. Increasing the latent-vector size did exactly this.

Encoding the input images down to a latent-vector containing 32 numbers forced the encoder to drop important information necessary to separate images from another, whether the original image was 64x64 or 128x128. So when the latent-vector size was doubled to 64, the agent had the necessary information required to separate images and perform the correct action.

Going back to the main question of the thesis, can end-to-end reinforcement learning drive a vehicle? This was already known to be true as specified right under the question. However, autoencoders has shown to be an important asset to the field and will definitely play a big role in the future of reinforcement learning and autonomous vehicles.

6.3 Reflection

Looking back at what has been done, there are many things that could have been done better. The experiments should have been much more quantitative to avoid any "lucky" and "unlucky" training sessions. A larger hyperparameter search should have been done as well to try and optimize the training better. The CarRacing experiments took a lot of time away from Carla experiments so maybe the CarRacing part should have simply been skipped entirely.

Programming everything everything to work together was a lot more time consuming than first anticipated. Python was definitely the best language to go for here. It made certain things much simpler to implement as some helper libraries such as Pytorch and OpenAI Gym were very flexible and well designed. Carla is where most issues appeared, probably due to it not having a reinforcement learning environment. Creating a custom environment to work correctly with the TD3 algorithm proved to be difficult.

Overall the thesis has been very educational. Learning about autoencoders and combining them with reinforcement learning has been challenging and very satisfying. It has also been very interesting to get so much insight into the current state of autonomous vehicles within reinforcement learning.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Autoencoders are a valuable asset in training complex reinforcement learning models. Combining the research done on autoencoders with reinforcement learning could give big breakthroughs. The future of reinforcement learning is looking bright. Headlines in computer science are often related to AI, and more specifically reinforcement learning these days. The amount of research done in both reinforcement learning and autonomous vehicles has been growing incredibly fast the last few years. Tech giants are also doing much research on their own, and as of now the race is on to develop the first commercially available, fully self-driving car.

7.2 Future Work

The work following this thesis could explore numerous themes within autonomous vehicles. The main point being to expand the reinforcement learning models and the learning environment. Including temporal information is a *must* in expert autonomous driving systems as the agent needs information about the current speed and direction of the vehicle. Temporal information can be supplied to the agent in multiple ways:

- Using RNNs
- Stacking frames
- Concatenating velocity and acceleration values to the observation

Exploring which option works best overall, if any, could be a topic to explore. Larger experiments with higher capacity models should be conducted such that moderate results can be achieved. This includes longer training times (for example 1e6 training steps), a wider hyperparameter search and more variation in the architectures of the VAE, actor and critic.

In practice, these models have to be able to adapt and drive in many different situations, as such the models should be generalized more. To do this, the training environments should include more randomness in roads, daytime and weather so that the agent can be exposed to more extreme situations from time to time. An agent who has experience driving in these various conditions will know the common denominator of driving a vehicle and knows what it needs to be careful of when it detects for example that the road is slippery, similarly to human drivers.

Bibliography

Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues,methodological variations, and systemapproaches.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete problems in ai safety.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Chen, J., Li, S. E., and Tomizuka, M. (2020). Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning.

Doersch, C. (2016). Tutorial on variational autoencoders.

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

Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods.

Geneva: World Health Organization (2018). Global status report on road safety 2018. Licence: CC BY-NC-SA 3.0 IGO.

Ha, D. and Schmidhuber, J. (2018). World models.

Hådem, C. S. (2019). Reinforcement learning in self-drivingvehicles. Project report in TDT4501), Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology.

Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. (2018). Learning to drive in a day.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

Makhzani, A. and Frey, B. (2013). k-sparse autoencoders.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Osiński, B., Jakubowski, A., Miłoś, P., Zięcina, P., Galias, C., Homoceanu, S., and Michalewski, H. (2019). Simulation-based reinforcement learning for real-world autonomous driving.

SAE International (2018). *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. SAE Standard J3016, Rev. June. 2018.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China. PMLR.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world.

Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Phys. Rev.*, 36:823–841.

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning.

Vergara, M. L. (2019). Accelerating training of deep reinforcement learning-based autonomous driving agents through comparative study of agent and environment designs. Master's thesis, NTNU.

Wang, H. and Raj, B. (2017). On the origin of deep learning.

Appendix A

Autoencoder architectures

The lists describing the architecture are function names in PyTorch style. Linear translates to a fully-connected layer.

A.1 MNIST and CIFAR VAE

This is the architecture used in the VAE CNN for CIFAR10. The encoder part can be summarized as following:

Conv2d(in_channels=3, out_channels=32, kernel_size=4, padding=1, stride=2)
ReLU
Conv2d(in_channels=32, out_channels=64, kernel_size=4, padding=1, stride=2)
ReLU
Conv2d(in_channels=64, out_channels=128, kernel_size=4, padding=1, stride=2)
ReLU

Table A.1: Encoder part of the VAE used on CIFAR10

After this, the tensor is flattened and split into to parts with a fully-connected layer $\text{Linear}(\text{in\_features} = 128*4*4, \text{out\_features} = 32)$. This produces one vector for the mean μ and one vector for the standard deviation σ. In the decoder, the latent-vector goes through a fully-connected layer $\text{Linear}(\text{in\_features} = 32, \text{out\_features} = 128*4*4)$ and is reshaped to $128*4*4$. Then it is the same as the encoder, but using transpose convolutions instead. Padding, stride and kernel size remains the same. At the end of the decoder there is a fully connected layer $\text{Linear}(\text{in\_features} = 32*32*3, \text{out\_features} = 32*32*3)$.

The MNIST version of this network is mostly the same, except that the convolutional layers are slightly different to account for the reduced image size on MNIST images.

Conv2d(in_channels=1, out_channels=32, kernel_size=4, padding=1, stride=2)
ReLU
Conv2d(in_channels=32, out_channels=64, kernel_size=4, padding=1, stride=2)
ReLU
Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1, stride=2)
ReLU

Table A.2: Encoder part of the VAE used on MNIST

The mean and standard deviation vectors are still 32 while the last fully-connected layer is lowered from $32 * 32 * 3$ to $28 * 28 * 1$.

A.2 Carracing and Carla VAE

The VAE that was used for CarRacing and Carla takes 64x64x3 images as input. The latent vector can be of varying size but was chosen to be 32. Table A.3 shows the details of the encoder part.

Conv2d(in_channels=3, out_channels=32, kernel_size=4, padding=0, stride=2)
ReLU
Conv2d(in_channels=32, out_channels=64, kernel_size=4, padding=0, stride=2)
ReLU
Conv2d(in_channels=64, out_channels=128, kernel_size=4, padding=0, stride=2)
ReLU
Conv2d(in_channels=128, out_channels=256, kernel_size=4, padding=0, stride=2)
ReLU

Table A.3: Encoder part of the VAE used in CarRacing and Carla

After the 4 convolution layers turn a 64x64x3 image into a 256x2x2 tensor, the tensor is flattened before going into two separate fully-connected layers. One for creating the mean and one for the standard deviation, the same way as in Appendix A.1. The fully-connected layer that makes the mean and standard deviation are defined as $\text{Linear}(\text{in\_features} = 256 * 2 * 2, \text{out\_features} = 32)$. The decoder part starts with a fully-connected layer $\text{Linear}(\text{in\_features} = 32, \text{out\_features} = 256*2*2)$. The tensor is then reshaped into a 256x2x2 tensor. This tensor is then scaled up to a 64x64x3 image using transpose convolution. The parameters of the transpose convolution layers are the same as in Table A.3 only backwards. The last activation layer is replaced with a sigmoid function to force values between 0 and 1.