Master's thesis

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

Isak Grande Bjørnstad

Deep Reinforcement Learning for Autonomous Vehicles in Simulated Environments

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

NTNU
Norwegian University of
Science and Technology

Isak Grande Bjørnstad

Deep Reinforcement Learning for Autonomous Vehicles in Simulated Environments

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

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

Abstract

In conjunction with deep learning, reinforcement learning has had several breakthroughs in recent years, causing a surge in research interest. Combined with the increased availability of realistic and open-source car simulators such as Carla, this means there has never been a better time to research reinforcement learning based autonomous vehicle systems.

In this thesis, an end-to-end autonomous vehicle system is trained with deep reinforcement learning in two different simulators with differing levels of realism. An implementation of the Proximal Policy Optimization algorithm is shown to learn good driving policies in both environments with only minor implementation differences. We find that designing and tuning the reinforcement learning based autonomous vehicle system in a simple simulator allowed faster experimentation resulting in a better tuned implementation to be deployed in the more complex simulator.

We develop a low-fidelity 3D environment that generates random roads on the fly in front of a car as it drives. Reinforcement learning experiments in this environment show that encoding visual observations with Variational Autoencoders result in better policies in terms of performance metrics like mean distance and episode success rate, but can have unintended side effects such as more uncomfortable driving policies being learned.

The *reality gap* between simulator and the real world causes difficulties when attempting to deploy a policy trained in a simulator in the real world. We demonstrate a moderately successful policy transfer over an analogous "simulator gap" between two different simulators that differ significantly in graphical fidelity and environment dynamics, such as vehicle physics. A model trained only in a simple Unity-based simulator is shown to achieve an episode success rate of 60 % in the Carla simulator.

Sammendrag

Sammen med dyp læring har Reinforcement Learning (forsterkningslæring) hatt flere gjennombrudd de siste årene, noe som har økt forskningsinteressen. Kombinert med økt tilgjengelighet av realistiske og open-source bilsimulatorer som Carla, har det aldri vært et bedre tidspunkt for forskning på autonome bilsystemer basert på RL.

I denne oppgaven blir et autonomt kjøretøysystem trent ved hjelp av ende-til-ende dyp RL i to forskjellige simulatorer med ulikt nivå av realisme. Simuleringene viser at en implementasjon av algoritmen Proximal Policy Optimisation lærer effektive kjørepolitikker i begge miljøer med kun små forskjeller i implementasjonsdetaljer. Vi finner at det å designe og finjustere det RL baserte autonome kjøretøysystemet muliggjør raskere utføring av eksperimenter, som igjen resulterer i en mer finjustert implementasjon til å bli plassert ut i den mer komplekse simulatoren.

Vi utvikler et enkelt 3D-miljø som genererer tilfeldige veier foran bilen mens den kjører. RL-eksperimenter i dette miljøet viser at det å bruke komprimerte representasjoner av de visuelle observasjonene ved å benytte en Variational Autoencoder, resulterte i bedre politikker målt ved ytelsesmetrikker som gjennomsnittlig distanse og andel vellykkede episoder. Ulempen er at dette også kan ha bivirkninger som at den lærte kjørepolitikken blir mer "ukomfortabel".

Realitetsgapet mellom simulator og den virkelige verden skaper problemer når det blir forsøkt å plassere politikker som er trent i simulator, ut i den virkelige verden. Vi demonstrerer en moderat vellykket politikkoverføring over et tilsvarende "simulatorgap" mellom to simulatorer som varierer signifikant i grafikkrealisme og miljødynamikk, slik som kjøretøyfysikk. En modell trent kun i en enkel Unity-basert simulator viser seg å oppnå en andel vellykkede episoder på 60 % i Carla simulatoren.

Preface

This thesis was written as part of the NTNU Autonomous Perception Lab (NAPLab) research group at the Norwegian University of Science and Technology.

I am thankful to my supervisor Frank Lindseth for giving me the opportunity to write about this exciting field and for providing invaluable feedback throughout the project's duration. I would also like to thank Gabriel Kiss for his helpful feedback in the early stages of the project.

I want to extend additional thanks to Frank Lindseth and Jan Grønsberg for setting up and providing access to a remote desktop virtual machine with a GPU.

Some computations were performed on resources provided by the NTNU IDUN/EPIC computing cluster.

Isak Grande Bjørnstad
Trondheim, June 28, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Description

Autonomous driving has become an active field of research in recent years, and some limited forms of autonomous vehicles already exist today. According to the World Health Organization, the worldwide death toll caused by road traffic accidents is 1.35 million deaths each year, and these accidents are the leading cause of death for children and young adults aged 5 - 29 years [WHO, 2020]. In Norway, traffic accidents claimed the lives of 93 people in 2020 [SSB, 2021]. It is clear that an autonomous vehicle driving in a safe manner has the potential to prevent many premature deaths.

Saving time is another motivation for autonomous vehicles. People with long commutes by car could for instance start their workday during the commute, increasing productivity and/or spare time. Self-driving cars could also result in a more efficient traffic flow which could save even more time. There is also the exciting prospect of autonomous taxis and ridesharing which would remove the need for individuals to own their own car. Society would need fewer cars and huge amounts of space that is today wasted on parking lots and other car infrastructure could be freed up.

Deep Reinforcement Learning has shown impressive results in recent years. In 2016 a computer program known as AlphaGo trained with DRL became the first-ever computer program to beat a professional Go player [Silver et al., 2016]. Go had traditionally been a very challenging game for computers to play due to the large number of possible moves and the difficulty of evaluating how good a position is. The program was initially trained with supervised learning from expert games and then improved further with RL by playing games against itself. The following year the same team from DeepMind released AlphaZero which outperformed AlphaGo and was trained entirely from scratch utilizing DRL without any human knowledge [Silver et al., 2017]. The only knowledge given to the program was the rules of the game. AlphaZero is a general reinforcement learning algorithm that was also trained to play shogi and chess, subsequently defeating both games' respective state-of-the-art computer programs.

Schrittwieser et al. [2019] at DeepMind recently released an even more generalized iteration of the algorithm called MuZero which does not even need to know the rules of the game, as it learns them by itself.

Using Deep Reinforcement Learning to teach autonomous vehicles to drive with an end-to-end approach is advantageous because it requires less manual labor and domain knowledge compared to other approaches such as complex modular systems often used today. Combined with the use of simulators which allow models to be cheaply trained and tested, it has great potential in the future of AVs. A significant obstacle with this approach is closing the reality gap in order to deploy a model trained in a simulator to the real world.

1.2 Goals and Research Questions

This thesis seeks to explore the training of autonomous vehicle agents in simulators using Reinforcement Learning. Using Reinforcement Learning is promising because it has shown itself able to learn good policies without any expert example data. Training from scratch in this way eliminates any possibility of sub-optimal expert demonstrations negatively influencing the learned policy. The DeepMind publications mentioned earlier showed that training without any expert demonstrations can lead to better final policies. Two simulators of differing complexity will be used, allowing experiments to explore the "simulator gap" between them as an analogous problem to the reality gap between simulators and reality.

These goals will be addressed with the following research questions:

- **Research question 1:** How does using a pre-trained Variational Autoencoder to encode visual features influence both the training process and the resulting policy?

- **Research question 2:** How can a low fidelity simulator be used to accelerate the process of building and deploying a reinforcement learning based autonomous vehicle in a more realistic environment?

- **Research question 3:** To what extent can a driving policy learned in a low fidelity environment be deployed and drive successfully in an unseen high fidelity environment?

1.3 Contributions

The main contributions of this thesis to the field of deep reinforcement learning for autonomous vehicles can be summarized as follows:

- A literature review on Deep Reinforcement Learning with a focus on state-of-the-art on-policy algorithms.
- A demonstration of Unity: ML-Agents' viability as a sandbox for creating Reinforcement Learning environments for autonomous vehicle research. This is demonstrated by our procedurally generated road environment for prototyping autonomous vehicle lane following systems.
- A Deep Reinforcement Learning based autonomous car lane following system that drives well in both a procedural road environment and in the Carla simulator.
- A deeper analysis on how using Variational Autoencoders to extract features affects the resulting policy.
- A moderately successful approach to sim-to-sim policy transfer as an analogous problem to the sim-to-real policy transfer.

1.4 Thesis Structure

This thesis is structured into six chapters. The chapters are organized in the following way:

Chapter 1: Introduction Introduces the thesis by specifying the problem description and the motivation behind it. It then details the goals and research questions that the rest of the thesis is oriented around before listing the contributions and thesis structure.

Chapter 2: Background and Related Work This chapter covers the relevant background and related work for Deep Reinforcement Learning in an autonomous vehicle context. First, it introduces reinforcement learning before listing a selection of different simulators that can be used with reinforcement learning. This is followed by an introduction to autonomous vehicles and a review on how they can be integrated with an RL system. Finally, some important papers relevant to this thesis are discussed.

Chapter 3: Methodology Covers the neural network architecture and reinforcement learning algorithm design decisions. Continues with an overview of both the custom simulator created for the thesis as well as the Carla simulator and how the RL framework was constructed around them to create RL environments. Finally it covers how a car trained in the Unity-based simulator was transferred to the Carla simulator.

Chapter 4: Experiments and Results The chapter begins with a brief explanation of each experiment and it's purpose in relation to the research questions. Each of the three experiments is then presented in the three parts *setup*, *results* and *discussion*.

Chapter 5: Discussion Provides a more comprehensive discussion based on all experiments and discusses how the findings provide answers to the research questions. Later compares the findings to results in related work before reflecting on the work done in this thesis.

Chapter 6: Conclusion and Future Work Draws conclusion based on the results and discussion and talks about intriguing directions for future work.

Chapter 2

Background and Related Work

This chapter starts with a deep dive into Reinforcement Learning, culminating with state-of-the-art on-policy Deep Reinforcement Learning algorithms. A selection of simulated environments that can be used with RL is then listed. This is followed by an introduction to autonomous vehicles before they are discussed in a reinforcement learning context. Lastly a review is done of related work.

2.1 Reinforcement Learning

Reinforcement Learning is a method that can be applied to problems where an agent needs to choose actions in an environment in order to receive an action-dependent reward. Such an environment is more generally referred to as a Markov Decision Process. The problem reinforcement learning aims to solve is to map environment-states to actions such that expected reward returned is maximized.

2.1.1 Markov Decision Process

A *Markov Decision Process* is a time-discrete stochastic-control process in which an agent acts in an environment and receives rewards. Both the environment and the rewards are influenced by the actions of the agent. An MDP is characterized by having the following properties:

- There is an environment which is observed by an agent and is represented by a state $s \in S$ where S is the set of all possible states.

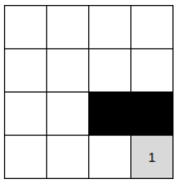


Figure 2.1: An example MDP environment. Non-terminal states are represented by white squares and the terminal state by a grey square. Each action that results in the agent being in a white square gives a reward of -0.1, while actions that result in the agent reaching the grey square give a reward of 1 and terminates the episode. The agent can move to any adjacent square but will stay in the same square if it hits a wall or tries to enter the black square. For simplicity, this example has deterministic control and is therefore not a true MDP. Stochasticity could be introduced by adding a random chance that the agent will move in a different direction than the chosen action dictates.

- In each state, the agent has to choose an action $a \in A(s), s \in S$ where $A(s)$ is the set of possible actions given the environment state.
- For each state-action pair $s \in S, a \in A(s)$, an immediate reward $r(s,a) \in \mathbb{R}$ is given to the agent.
- When an action is performed in a given state, the next state is given by the state transition function $P(s'|s,a), s \in S, s' \in S, a \in A(s)$. This function is a probability distribution over states given the current state s and chosen action a and s' being the next state.
- The process has the *Markov property*, meaning that future states depend only on the current state, and not the sequence of states leading to it.

Another requirement for an MDP is that the environment is *fully observable*, meaning that there is no hidden information in the state, and the agent is omniscient. Chess is an example of such an environment since there is no hidden information. Relaxing the fully-observable requirement results in a *Partially Observable Markov Decision Process*. Battleships is an example of a POMDP, since players do not know where their opponents' ships are. When modeling real-world processes it is helpful to model them as POMDPs since it is impossible to have perfect information about the real world.

A sequence of consecutive timesteps with states, actions and rewards is called a *trajectory* $\tau = \{(s_0, a_0, r_0), (s_1, a_1, r_1), ...\}$. Where r_t is the reward obtained at timestep t given by $r_t = r(s_t, a_t)$. The return $R(\tau)$ of the trajectory is given by the discounted sum of reward along the trajectory using a discount factor

$\gamma \in [0,1]$. The use of a discount factor ensures that returns are finite and causes rewards earlier in time to be more valuable and thus prioritised. The return of a trajectory is defined as:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t \cdot r_t \tag{2.1}$$

It can also be useful to calculate the return of a trajectory starting from a given timestep. The following formula defines the return $R_t(\tau)$ of trajectory τ starting from timestep t.

$$R_t(\tau) = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k} \tag{2.2}$$

Policy Evaluation

The agent chooses actions according to its policy π. The policy is modeled as a probability distribution over actions given a state. The probability that the agent chooses action a when in state s is $\pi(a|s)$.

The value of a state $V_\pi(s), s \in S$ given a policy is the expected discounted return if starting from that state and following policy π.

$$V_\pi(s) = \sum_{a \in A(s)} \pi(a|s) \cdot \left[r(s,a) + \gamma \sum_{s' \in S} V_\pi(s') \cdot P(s'|s,a) \right] \tag{2.3}$$

Since the reward function and state transition function are known in the MDP, it is possible to create a lookup table of the value function for a given policy if the state space and action space are small enough. This dynamic programming approach is done by setting the initial values to arbitrary values and then updating the value of each state in each iteration. This *iterative policy evaluation* will bring the values closer to the real value and will eventually converge to the true values.

When the value of each state is known, the policy can be improved by acting greedily with respect to the value function. *Policy iteration* is a technique that alternates between updating the value function and updating the policy. This converges to an optimal policy.

A closely related function to the value function is the action-value function also known as the Q-function $Q_\pi(s,a)$. The Q-function gives the expected return for the agent if it chooses action a when in state s and then continuing following the policy π.

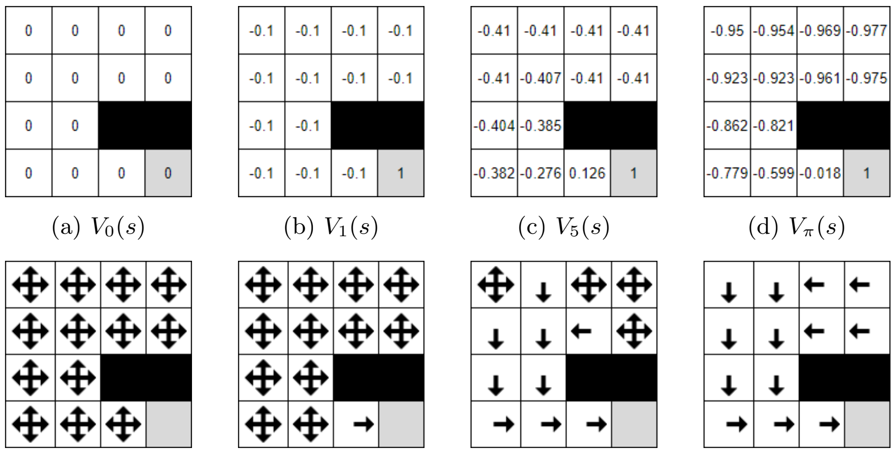


Figure 2.2: A few iterations of iterative policy evaluation. The policy being evaluated is the uniform random policy. All states are initialized to a value of 0. The figure shows the initial state values, and the state values after 1, 5 and ∞ iterations. The bottom parts show the corresponding greedy policy with respect to the value function. The policy acting with respect to the converged value function is optimal in this example.

Finally, the advantage function $A_\pi(s, a)$ says how much better than expected the return is when choosing action a when in state s, with the expected return being the value of the state.

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{2.4}$$

Challenges with complex environments

Most interesting MDP problems are too complex for dynamic programming approaches to be feasible for computing the value function. *Monte-Carlo simulations* is a way to estimate the value of a state s by sampling a number of trajectories originating from s and following the policy π. The value estimate is then derived by simply taking the average return from these samples.

Monte-Carlo works in *episodic environments*, where some terminal state is eventually reached. In *non-episodic environments*, the length of an episode is unbounded, which makes Monte-Carlo unfeasible. An alternative is to use a technique called *temporal difference*. TD computes the value by taking n steps in the environment and returning the discounted sum of rewards plus the discounted value of the state s'. An equation showing this calculation for TD-1, which looks one step into the future, is shown below.

$$V(s) \leftarrow r(s,a) + \gamma V(s') \tag{2.5}$$

When the state- and action spaces are too large to be stored in a table, function approximations can be used to estimate functions such as the value, policy and Q-function. A neural network with adjustable weights is typically used.

MDP environments are assumed to be *time discrete* environments. This means time is divided into discrete steps where the agent observes and acts before the next state and time step is reached. When modeling turn based games such as chess this makes sense, but the real world has no discrete time steps since time is continuous. One way to deal with this issue is to divide time into discrete time steps by choosing a frequency of how often the agent will observe and act. The real world can then be modeled as a time-discrete environment where each time step represents a fixed amount of time.

Reinforcement Learning

In a Markov Decision Process, the reward function and state transition function are both known. In a Reinforcement Learning problem, at least one is unknown, which means the agent needs to learn what behavior is good, and then try to maximize its reward. To achieve this, the agent can act randomly in the environment to explore and observe the consequences or rewards of its actions. After exploring the environment, it can exploit what it learned in order to obtain better rewards. This leads to the important tradeoff of *Exploration vs. Exploitation.* Exploration means that the agent performs random actions instead of what the agent believes is optimal in an attempt to find better actions. Exploitation means that the agent utilizes what the agent already knows and picks the action it believes is best. Too much exploration might mean it never reaches the best rewards since it might require many good actions in a row. Too much exploitation might mean good behaviors are missed because the agent focused on a suboptimal pattern of behavior early in the training process and got stuck in a local optimum.

2.1.2 Deep Reinforcement Learning

Deep Reinforcement Learning is a type of Reinforcement Learning where deep neural networks are used as function approximators. An overview of some important and recent deep reinforcement learning algorithms will be provided. A few reinforcement learning concepts that are important to know will be introduced first, followed by an explanation of advantage estimation.

Model-based vs. Model-free RL *Model-based* Reinforcement Learning attempts to create a model of the environment that can be used to help make decisions. The model can be taught to predict the next state depending on the action taken, and then the agent can consult this model to plan ahead and find good actions with for instance, a heuristic tree search algorithm like Monte Carlo tree search. A downside of a model-based approach is that any errors in the model compounds for every timestep, which can cause large errors when searching multiple steps ahead. *Model-free* Reinforcement Learning methods learn to act directly without a model of the world, meaning they try to map observations directly to actions.

Sample efficiency *Sample efficiency* is a term describing how much data an algorithm needs to learn. An algorithm with a high sample efficiency needs fewer samples to learn a good policy than an algorithm with a lower sample efficiency.

On-policy vs. Off-policy algorithms Reinforcement Learning algorithms can be separated into two groups based on whether they are on-policy or off-policy. An *on-policy* algorithm has the agent explore the environment using the policy it is trying to optimize. *Off-policy* algorithms can optimize the policy using experience collected with any policy. This allows off-policy algorithms to replay old experience many times to learn more which increases its sample efficiency. On-policy algorithms have to discard old experience since it was collected with an outdated policy and therefore violates the on-policy requirement.

Discrete vs. continuous action space An environment can have a discrete or a continuous action space. A *discrete action space* has a fixed number of actions that can depend on the state, such as movement in a grid or moving a piece on a chessboard. A *continuous action space* models actions as real numbers in a specified range such as $[-1, 1]$. This action can for instance be the normalized steering angle of the steering wheel in a car. Since a policy is a probability distribution over actions, this is handled differently for discrete and continuous action spaces. In the case of policies in discrete action spaces, outputting a probability for each action is often done, with the sum of all actions being probability 1. For continuous action spaces, the policies output a continuous probability distribution from which an action can be sampled. The neural network might predict a mean action used in a normal distribution used to sample the action.

Actor-Critic architecture Actor-Critic architectures are a type of reinforcement learning models that train a policy (actor) and value function (critic) independently.

Curriculum Learning *Curriculum learning* is a technique that can be used with reinforcement learning that starts by learning a simple task and then subsequently introducing new concepts in a specific order to teach the agent increasingly complex tasks. As an example, a car agent can start by learning lane following. Once the agent has mastered lane following, other cars and the possibility of overtaking other cars can be added to the environment. Curriculum learning is similar to how humans learn.

Advantage Estimation

Policy gradient methods need a way to know which actions were good and which actions were bad. Advantage Estimation is the task of estimating the advantage $A(s, a)$ of taking action a while in state s. Recall that the advantage function quantifies how much better the return from an action was than expected. Advantage estimation therefore needs to have an expectation for the return, which is typically estimated with the value function $V_\pi(s)$. One way to estimate the advantage is to compare the return $R_t(\tau)$ with the value estimate $V(s_t)$.

$$\hat{A}_t = R_t(\tau) - V(s_t) \tag{2.6}$$

This method takes into account the whole trajectory to compute the advantage estimate. This can be undesirable since the effect of the action is confounded with the effects of later actions in the trajectory, causing a high variance in the estimate. A different method is to use the temporal difference (TD) residual as the estimate, defined as

$$\hat{A}_t^{(1)} = \delta_t^V = -V(s_t) + r_t + \gamma V(s_{t+1}) \tag{2.7}$$

This estimates the advantage by looking a single step into the future. This approach has the downside of introducing bias to the estimate due to the value term at the end (the $-V(s_t)$ term does not introduce bias). Multiple δ terms can be summed together to create an estimate looking multiple steps into the future.

$$\hat{A}_t^{(2)} = \delta_t^V + \gamma \delta_{t+1}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \tag{2.8}$$

This is a telescoping sum as the intermediate value terms cancel out, and the sum can be generalized to look k steps into the future:

$$\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \tag{2.9}$$

Higher values of k reduce the bias as the $\gamma^k V(s_{t+k})$ term gets more discounted. *Generalized Advantage Estimation* [Schulman et al., 2015b] is a method of advantage estimation that uses a parameter $\lambda \in [0, 1]$ to control the tradeoff between bias and variance by using an exponentially weighted average of the k-step estimators. A more detailed explanation of how the formula is derived can found in the paper. The resulting Generalized Advantage Estimate is defined as:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+l}^V \tag{2.10}$$

The advantage estimate based on discounted returns is now a special case of GAE with $\lambda = 1$ and the 1-step return estimate is another special case with $\lambda = 0$.

Deep Q Learning

Mnih et al. [2015] trained agents to play 49 different Atari games using only pixel values and the game score as inputs to a Deep Q network. The agents managed to reach human-level performance on many Atari games and superhuman performance on some. The same architecture and hyperparameters were used for every Atari game. The Deep Q network used a Convolutional Neural Network to map the image input to a discrete set of Q values corresponding to each possible action, and the resulting Q function was used to choose actions. A human professional games tester was used to obtain reference human-level scores.

Deep Q Learning aims to approximate the optimal Q function

$$Q^*(s, a) = \max_\pi E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s = s_t, a = a_t, \pi] \tag{2.11}$$

Approximating the Q function with neural networks had been known to be unstable or even diverging. Some causes of this are that minor changes in the Q function can cause a significant change in the policy, and that there are correlations between the Q function and the target values used to train the Q function. The authors address the first problem by using experience replay with randomly shuffled data to smooth out the transitions between different Q functions and remove the correlation in the observation sequence. They address the second problem by only periodically updating the Q function used for target values, as this reduces the correlation between target Q value and actual Q value. The Q function is optimized using the following loss function:

$$L_i(\theta_i) = (r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a, \theta_i))^2 \tag{2.12}$$

where s, a, s' and r are the state transitions from the replay buffer. The weights θ_i are trained while using the old weights θ_i^- in the target function. The weights of the target function get updated periodically. This algorithm can only be used in environments with discrete action spaces.

Policy gradient methods and Vanilla policy gradient

Policy gradient methods are a class of on-policy methods first introduced by Williams [1992] that model the policy directly as a function to be optimized. This method of modeling the policy differs from Q-learning methods such as Deep Q-learning, where the policy is implicitly given by the Q-function. These methods use an estimate of the gradient of the expected reward with respect to the policy parameters to optimize the policy with gradient ascent. This estimate is obtained by sampling trajectories from the environment using the current policy hence the on-policy categorization. Since this estimate needs trajectories sampled with the current policy, the trajectories are no longer valid after a policy update and have to be discarded. This means that experience replay cannot be used, and sample efficiency is lower.

The general idea of policy gradient methods is to compute the advantages of a batch of collected trajectories to determine which actions were better or worse than expected. The policy is then updated such that the probability of actions that were better than expected is increased and the probability of actions worse than expected is decreased.

The estimated gradient is given by

$$\hat{g} = \hat{E}_t[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.13}$$

This is averaged over a batch of samples in the experience buffer and gives the following loss function:

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.14}$$

It is essential only to perform one gradient step when updating the policy, as performing multiple steps has been shown empirically to lead to large policy updates that are harmful to performance.

Trust Region Policy Optimization

Trust-Region Policy Optimization [Schulman et al., 2015a] is an algorithm that allows multiple policy updates per batch of samples without destroying the policy. This is achieved thanks to a trust-region constraint that prevents the policy from changing too much. The objective function used in TRPO becomes:

$$\max_\theta \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t] \tag{2.15}$$

Algorithm 1: Vanilla policy gradient, Actor-Critic style

```
Initialize policy parameter $\theta$, value parameter $\phi$;
for iteration=1,2,... do
    Collect a a set of trajectories by executing the current policy
    Compute the advantage estimates $\hat{A}_t$ at each timestep for each
      trajectory based on $V_\phi$ and any advantage estimation algorithm.
      (e.g. discounted returns or GAE)
    Update the value function, by minimizing the mean square error
      $||V(s_t) - R_t||^2$ summed over all trajectories and timesteps by using
      gradient descent
    Update the policy, using a policy gradient estimate $\hat{g}$, given by the
      sum of terms $\nabla_\theta log\pi(a_t|s_t, \theta)\hat{A}_t$
end
```

subject to following constraint:

$$\hat{E}_t[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] < \delta \tag{2.16}$$

where θ_{old} is the policy parameters used when the samples were collected. The *Kullback–Leibler divergence* measures how different the two probability distributions $\pi_{\theta_{old}}$ and π_θ are.

This is a hard constraint on the policy update. The TRPO paper found that the theory justifies using a KL-divergence *penalty* on the objective function instead of a constraint, but that it was difficult to choose a robust penalty coefficient in practice. This is why the hard constraint of δ is used instead.

Proximal Policy Optimization

Proximal Policy Optimization is an algorithm introduced by Schulman et al. [2017] that, like Trust-Region Policy Optimization, enables multiple policy updates per batch of samples by limiting how much the policy is allowed to change. Instead of a hard constraint on KL-divergence like TRPO, PPO limits policy updates using a clipped objective function that disincentivizes large changes in the probability ratios of actions between the old and new policy. The probability ratio $r_t(\theta)$ between the new and the old policy is defined as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. The objective function from TRPO, which PPO is derived from, can then be written as

$$L^{TRPO}(\theta) = \hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t] = \hat{E}_t[r_t(\theta)\hat{A}_t] \tag{2.17}$$

Algorithm 2: Proximal Policy Optimization, Actor-Critic Style, adapted from Schulman et al. [2017]

```
for iteration=1,2,... do
    for actor=1,2,...,N do
        Run policy $\pi_{\theta_{old}}$ in an environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
    end
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size
      $M \leq NT$
    $\theta_{old} \leftarrow \theta$
end
```

The objective function of PPO uses a probability ratio that is clipped to the range $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter, typically $\epsilon = 0.2$, that controls how much the policy can change compared to the old policy. The clipped objective is obtained by replacing the probability ratio in L^{TRPO} with the clipped probability ratio. The objective function used in PPO is then constructed by performing a *min* operation on the clipped and unclipped objectives:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.18}$$

This minimization means that L^{CLIP} is a lower bound of the unclipped objective and that changes in probability ratios are ignored only when they would make the objective improve too much. Changes in probability ratios that would make the objective worse are still included. Consider an example where $\epsilon = 0.2$. This means that the probability ratio is clipped at 1.2 if the advantage is positive or clipped at 0.8 if the advantage is negative. In other words, the objective function will not try to increase the probability of an action that was better than expected to more than 1.2 times the original probability. Vice versa it will not try to decrease the probability of an action that was worse than expected to less than 0.8 times the original probability.

The objective function can be augmented by adding an entropy bonus to incentivize exploration and the mean squared-error loss between values and target values for the value function.

A significant advantage of PPO is that it is much easier to implement since there are no constraints that complicate the optimization of the objective function. Optimizing the PPO objective is very straightforward as it can be done with a standard gradient descent optimizer. Empirical observations also show that PPO performs as well or better than TRPO in most environments.

Phasic Policy Gradient

When implementing PPO and earlier methods, a choice would have to be made between using a shared network for the policy and value function or using separate networks. Using a shared network has the advantage of having fewer parameters to optimize, being computationally cheaper, and potentially being better at feature extraction since both the value and policy objectives share the same base network. However, it has the significant drawback that the value and policy objectives can interfere and inhibit the learning process. By using separate policy and value networks the objectives cannot interfere, but each network needs to optimize its own set of parameters, and the policy network cannot use features learned by the value network.

Phasic Policy Gradient by Cobbe et al. [2020] attempt to improve the training process by splitting training into two distinct phases. The first phase advances training, and the second phase distills features. PPG tries to create a best of both worlds scenario where the policy can be trained independently in the first phase. In the second phase, an auxiliary value head on the policy network is trained to ensure value-related features are learned. This auxiliary value head has no other purpose than to help the policy network learn useful features. It is not used for advantage estimation, as that is done by the separate value network. To prevent destroying the policy when training the auxiliary value, a KL-divergence term is added to the auxiliary loss function to prevent the policy from changing too much.

The paper showed experiments where the policy and value networks were trained with varying numbers of epochs E_π and E_V. In PPO, the number of epochs is typically kept the same for the value and policy network. The experiments showed that training with only one epoch E_π was almost always optimal or near-optimal. This contrasts with PPO, where the optimal number of epochs was three, given the other hyperparameters in their tuned implementation. This finding suggests that the extra epochs in PPO only improve performance because they offer more training to the value function.

The authors compared the algorithm to a highly tuned PPO implementation and found that PPG had a better sample efficiency than PPO in all 16 environments in the Procgen Benchmark described in Cobbe et al. [2019]. This benchmark contains environments similar in style to Atari games which have commonly been used in Reinforcement Learning benchmarks, but with randomly generated levels to prevent overfitting and instead reward generalization. The researchers conjecture that the high dimensional input space of visual domains contributes to the importance of sharing parameters between the value and policy network, and therefore believe that visual environments are the most likely domain to see PPG outperform PPO.

Algorithm 3: Phasic Policy Gradient, adapted from Cobbe et al. [2020]

```
for phase=1,2,... do
    Initialize empty buffer $B$
    for iteration=1,2,...,$N_\pi$ do
        Collect experience under the current policy $\pi$
        Compute value function target $\hat{V}_t^{targ}$ for each state $s_t$
        for epoch=1,2,...,$E_\pi$ do
            Optimize $L^{clip} + \beta_S S[\pi]$ wrt $\theta_\pi$
        end
        for epoch=1,2,...,$E_V$ do
            Optimize $L^{value} + \beta_S S[\pi]$ wrt $\theta_V$
        end
        Add all $(s_t, \hat{V}_t^{targ})$ to $B$
    end
    Compute and store current policy for $\pi_{\theta_{old}}(\cdot, s_t)$ for all states $s_t$ in $B$
    for epoch=1,2,...,$E_{aux}$ do
        Optimize $L^{joint}$ wrt $\theta_\pi$, on all data in $B$
        Optimize $L^{value}$ wrt $\theta_V$, on all data in $B$
    end
end
```

2.2 Simulated Environments

This section will list some relevant simulators that can potentially be used to create environments used with Deep Reinforcement Learning for Autonomous Vehicles.

2.2.1 TORCS

TORCS or The Open Racing Car Simulator is an open source 3D racing simulator [Wymann et al., 2015]. The simulator has an interface that allows communication with the environment via UDP sockets to control the cars. It also includes a built-in AI that can drive. The environment can provide sensor readings such as distance to the edge of the track at various angles as well as camera images. The simulator attempts to provide realistic racing physics by for instance including different friction profiles for different types of tires on different types of ground. The physics engine also includes a simple aerodynamics model that includes slip-streaming effects. A wide selection of racing tracks is included in the simulator. A disadvantage of TORCS is that the graphics are basic and not photorealistic.

2.2.2 Nvidia ISAAC

Nvidia ISAAC is a platform built on Nvidia Omniverse that includes a Software Development Kit and simulator for robotics. It offers photorealistic rendering and comes with sensors like camera, LiDAR and semantic segmentation maps. It also includes GPU-optimized algorithms for navigation and path planning. The use cases are mostly tailored towards warehouse robotics, such as controlling robot arms and robots that move pallets around.

Nvidia ISAAC includes an example environment of a robot arm with a suction cup that has the goal of stacking boxes on a pallet. Nvidia ISAAC is, as of writing this thesis, not cross-platform and requires an Nvidia GPU + Ubuntu 18.04 or a proprietary Nvidia device such as Nvidia Jetson Nano to work. Alternatively, it can be run in Docker containers on a cloud service.

Nvidia DRIVE is another platform powered by Omniverse that specializes in autonomous vehicle development. It features photorealistic graphics thanks to the Nvidia RTX platform. Nvidia Drive is not yet available to the public.

2.2.3 LGSVL

LGSVL Rong et al. is an open source simulator made by LG Electronics R&D Center. The simulator is built on the Unity game engine and comes with photo-realistic graphics. The simulator includes a small detailed test map and a larger, less detailed San Francisco map. LGSVL also comes with many different sensors such as Camera, Fisheye Camera, Depth camera (LiDAR), RADAR, Semantic Segmentation camera and 3D bounding boxes. It is possible to import supported map formats into the simulator, and a premium commercial version of the simulator exists with more features than the open source version.

2.2.4 CARLA

CARLA (Dosovitskiy et al. [2017]) or Car Learning to Act is a car simulator with photorealistic graphics and a great weather system. The simulator is built on Unreal Engine and mainly focuses on urban driving situations, which is reflected in the included maps and traffic engine. Interaction with the environment can be done through the included Python API, where most aspects of the simulator can be controlled, including controlling the cars. The PythonAPI also comes with a waypoint navigation system and autopilot, making it easy to simulate traffic and collect driving data. Carla comes with a wide suite of different sensors, including RGB camera, depth sensor (LiDAR) and semantic segmentation camera sensors. Several maps are included with different types of driving environments. CARLA has been used in previous theses, which means results can be compared to those of previous authors. This also provides an expectation of what results should be achievable.

The Carla paper also includes a benchmark in which the three autonomous vehicle pipelines *modular*, *imitation learning* and *reinforcement learning*, are tested. The setup for running the benchmark is included in the Python API so others can use it. The benchmark includes different tasks in increasing difficulty ranging from driving straight ahead to navigating to random positions on the map with dynamic objects such as other cars and pedestrians on the map. The benchmark runs tests in different weathers and maps to measure how well the model generalizes. The first test is done with the same types of weather and the same map as during training. The other tests evaluate the agent in a new map, unseen weather conditions and a combination of both new map and unseen weather conditions.

Codevilla et al. [2019] proposes another Carla benchmark *NoCrash* which is more difficult than the original Carla benchmark.

2.2.5 Unity: ML-Agents

Unity: ML-Agents is a machine learning framework for the Unity game engine that facilitates the creation of agents that learn to act through both reinforcement learning and imitation learning. The framework includes a Unity package and a Python package.

In ML-Agents, agents are defined in Unity by giving them a *behavior*. This makes them actors in the environment. The behavior controls how the agent acts, which in Reinforcement Learning is referred to as the policy.

The framework contains many example environments such as balancing a ball on top of a cube that the different RL algorithms can be tested on. The included reinforcement learning algorithms include Proximal Policy Optimization and Soft Actor Critic.

Using the included Python trainer allows ML-Agents to automatically create the neural network architecture for the agent based on a configuration file specifying the hyperparameters. This handles both visual and vector observation data. A script has to be created in Unity that parses the action outputs and uses it to control the actor in the environment. The automation of this framework makes it easy to prototype agents and environments with the framework, and the Python API can always be used to interact directly with the Unity environment allowing custom RL implementations. Camera sensors and Raycast sensors are included, but semantic segmentation sensors and depth/LiDAR sensors are not included when this thesis was written.

2.3 Autonomous Cars

An *autonomous vehicle* is a vehicle that can control itself autonomously without human supervision. The main focus of this thesis will be self-driving cars which is an active area of research within autonomous vehicles.

2.3.1 Introduction to Autonomous Cars

Self-driving cars need to observe the surrounding environment and send control signals to the car controller. They might need to take into account high-level commands such as navigational instructions. There are several different levels of autonomy for autonomous cars, where higher levels are progressively more difficult. Today, some cars can already help with lane-keeping and controlling speed, which is a form of autonomy, but this still requires a driver to pay attention to the road. Ideally, an autonomous vehicle would not even need a steering wheel and could drive entirely on its own.

SAE International defines six levels of autonomy for self-driving vehicles in the J3016 standard, where each level is an improvement over the previous [SAE, 2021]. In the first three levels, the human is monitoring the environment, and in the latter three, the system is monitoring the environment. All but the last level are specific to *driving modes*, where a driving mode is a specific driving situation such as driving on a highway.

- **Level 0:** No automation.

- **Level 1:** Driver assistance. The car can use information about the environment to send steering or acceleration/brake signals. Adaptive Cruise Control, which can adjust the car's speed depending on the distance to other cars, is an example of level 1 autonomy. The driver is expected to be able to intervene immediately.
- **Level 2:** Partial automation. The car can use information about the environment to send both steering and acceleration/brake signals. A system that can autonomously follow lanes and accelerate/brake when needed is considered a level 2 system. The driver is still expected to be able to intervene immediately.
- **Level 3:** Conditional automation. The system monitors the environment and handles all control. The driver is expected to intervene if the system requests it.
- **Level 4:** High automation. The system monitors the environment and handles all control just like level 3, but without the requirement for a human to be able to intervene. This requirement needs to be met for *some* driving modes, such as driving on the highway.
- **Level 5:** Full automation. Same requirement as level 4, but for *all driving modes.* A car with level 5 autonomy would not need a steering wheel.

2.3.2 Sensors used in autonomous vehicles

Autonomous vehicles can use many different kinds of sensors to observe and gain information about the environment. This is a list detailing a few sensors that are useful in an autonomous vehicle.

Camera A camera provides a visual image of the environment. Cameras are also called RGB sensors since they provide color images with red, green and blue channels. Alternatively, they can also provide grayscale images. Two camera sensors can be used together to create a stereoscopic image that can be used to obtain depth information. Camera sensors are cheap, but they can be unreliable in conditions such as fog, snowstorms or during nighttime.

LiDAR LiDAR is a sensor that uses electromagnetic waves (usually infrared) from a laser to determine the distance to an object or a surface by illuminating it and measuring the reflection time. This laser is scanned over a scene to produce a depth image or a 3d point cloud of the environment. A significant drawback of LiDAR sensors is that a quality long-range one as required to be useful for an autonomous vehicle is very expensive. Some companies such as Tesla do not use them and instead rely on cameras. The cost issue might be about to change as cheaper LiDARs such as solid-state LiDARs have become an active field of research and development in recent times, largely motivated by their applications in autonomous vehicles.

Radar Radar is a system that uses Radio waves to detect objects and determine how far away they are based on their reflection. Radar is similar to LiDAR but uses radio waves instead of infrared waves.

Ultrasonic sensor Ultrasonic sensors measure the distance to the nearest obstacle using sound waves. These sensors have a relatively short range of a few meters. These sensors are often used in parking assistants and for detecting vehicles in the driver's blind spot. The sensors themselves are very cheap.

GPS GPS uses satellites to obtain global localization to find the vehicle's location on earth to an accuracy on the order of meters. This sensor requires line-of-sight to satellites which means it will not work when the sky is blocked, such as in a tunnel. A system with GPS can be augmented with a gyroscope and accelerometer, which can sense change in velocity and rotation. This information can then be used to estimate the vehicle's position even after the GPS signal is lost.

2.3.3 Modular vs. end-to-end approach

Modular approach

A modular approach to autonomous cars means that the system is separated into modules that each perform specific tasks. Modules include mapping and localization modules, perception modules, prediction modules and planning and control modules. These modules can be hand-crafted by an engineer or trained with machine learning. The system uses these modules to create a model of the world that it can then use to plan the vehicle's actions. Most commercial solutions for autonomous vehicles at the time this thesis was written use modular systems.

One disadvantage of a modular system is that it requires a lot of domain knowledge and careful engineering and tuning by humans. Another disadvantage is that the performance of the system is limited by the model. How will the system react if a football rolls into the road and the model is not designed to handle it? The system's performance is largely dependent on and limited by decisions made by the engineers who designed it.

End-to-end approach

An end-to-end approach to autonomous cars creates a system that directly maps observations about the environment to vehicle control signals such as steering angle, throttle and brake signals. It is called end-to-end because it does not create an explicit intermediate representation of the world.

One advantage of an end-to-end approach is that it does not require humans to engineer an internal world model and enables learning algorithms such as Imitation Learning and Reinforcement Learning to learn the entire system on its own. A major disadvantage of end-to-end systems is that they act as a black box, meaning one does not know its inner workings. One only knows the input and output. This makes it hard to attribute the cause of a failure as opposed to a modular system where you might see system logs showing that for instance the car detection module failed to detect a vehicle.

2.3.4 End-to-end Learning for autonomous vehicles

A nice property of end-to-end systems is that they can be trained directly using Imitation- or Reinforcement Learning.

Imitation Learning

Imitation learning is a form of supervised learning that uses a dataset labeled with the correct action. Imitation Learning is essentially learning from demonstrations. An agent trained with IL tries to *imitate* the behavior it has been shown during training. IL requires large datasets of expert driving data to train the system, which can be easy to collect from real drivers.

A disadvantage of imitation learning is that it suffers from *distributional shift*, meaning that the state distribution in training is different than in test. This happens because the future states depend on the chosen action, which will differ in test. This phenomenon is described in detail in de Haan et al. [2019]. *Causal misidentification* is a consequence of distributional shift that can be very destructive to the driving policy. Using the same example as in the paper, let us say a braking lamp on the dashboard lights up whenever the car brakes, and that this is included in the input features of the system. The system might then learn to brake when the braking light is on, as this might give a low training error. The model then only brakes when the braking light is on. The model misidentified the braking light as the cause of braking.

Reinforcement Learning

Reinforcement Learning does not need a dataset of demonstrations and instead learns by trial and error. Naturally, this means the system needs to be able to explore the environment in order to learn. Collecting experience in the real world can be expensive and dangerous. Using a simulator alleviates these problems but introduces the reality gap problem: a policy trained in a simulator might not generalize well to the real world since a simulator will never be able to perfectly simulate the real world.

2.3.5 Deep Reinforcement Learning for Autonomous Vehicles

Some important design decisions need to be taken when designing a DRL system for autonomous vehicles. These decisions include defining the reward function, defining the action space, deciding what observations to use as input to the system and designing the architecture of the neural network.

Reward function design

Choosing the reward function is one of the most important decisions when designing a reinforcement learning system as it is what dictates how the agent will behave. If rewards are sparse and difficult to reach initially by random behaviour the agent might never reach them and therefore never learn.

RL algorithms improve their (explicit or implicit) policy by slightly adjusting its parameters based on the experience it has collected. We can imagine the training process as the policy (parameters) taking small steps towards a desired policy in a way such that the expected reward of the policy increases almost monotonically along the way. It is therefore important that such a path of monotonically increasing expected rewards exists in the policy parameter space.

Consider a racing track environment where the agent controls a car with the goal of driving around the track within a time limit. If the time limit expires, the car receives a negative reward. If the car completes a lap, it receives a positive reward. An RL algorithm is run in this environment with a randomly initialized policy. Since it is extremely unlikely that the randomly initialized policy will successfully complete a lap and receive a positive reward, the expected reward will always be the same negative reward. In this environment there is no reasonably sized step the policy can take that increases the performance (expected reward) of the policy. It is therefore very unlikely that the training finds a good policy. *Reward shaping* tackles this problem by adding extra rewards that serve the purpose of guiding the agent towards the real intended reward. Reward shaping comes with its own set of difficulties though, as a poorly designed reward function might be exploited by the agent in an unexpected and sub-optimal way resulting in the agent never reaching the real intended reward.

Local optima is another problem that can hinder learning. This happens when the policy parameters reach a point where they locally maximize the policy performance. There might exist a much better policy, but no path of monotonically increasing performance to it exists meaning there is no chance of actually finding this policy. The policy will then most likely continue to be stuck in its local optimum. Reusing the racing track example, let us assume the reward is based on how fast the car completes a lap in the track and that a locally optimal policy that completes the lap has been found. Imagine that the track has a difficult to maneuver shortcut that can significantly improve the lap time (i.e., there exists a much better policy). The only way to improve the policy would be to drive through the shortcut, but this would require learning to navigate the shortcut, which would require policy steps that degrade performance. There would be no incentive for the agent to perform these policy steps since they make the policy (temporarily) worse and the agent has no way of knowing this will eventually lead to a better policy.

An often used reward function that can work pretty well for lane following is to give a reward proportional to the forward speed of the car, and negative rewards for driving out of the lane or crashing. It is common to add multiple objectives to the reward function by linearly combining multiple reward functions with a weight that controls the importance of each objective. One might for example add a small penalty term based on erratic steering and velocity behavior to reduce discomfort experienced by a human in the vehicle.

Action space

The action space defines the output of the neural network model. A straightforward action space is to output a throttle, braking and steering signal. A more abstract option is to output a waypoint and target speed that the car should steer towards and adjust to. This requires a low-level controller that converts the abstract action into throttle, brake and steering angle control signals for the car. This low-level controller can be programmed by an engineer or even trained with reinforcement learning.

In the context of autonomous vehicles, it is preferable to use a continuous action space rather than a discrete action space as it can result in smoother policies. When using a continuous action space, actions can be sampled from a probability distribution such as a normal distribution with a mean predicted by the network. This ensures exploration.

Observations

Autonomous vehicles can utilize many sensors to observe the environment. Perhaps the most important sensor is the camera sensor. A forward-facing camera gives a visual image of the road, and multiple cameras can be combined to create a wider field of view. Processed images such as semantic segmentation maps can also be used as observations. Semantic segmentation maps classify each pixel of the image into classes like Road, Vehicles, Lane Markings, Vegetation and so on.

Input from multiple sensors is usually required, and these need to be combined in a neural network that outputs an action. This is commonly done by concatenating the features from each sensor into a common feature vector that is passed through the rest of the network. Visual input is not a vector, so it needs to be converted to one by encoding it through a visual encoder such as a CNN. This visual encoder can then be trained as part of the neural network, or a pre-trained encoder can be used. If multiple types of image observations are used such as RGB images, LiDAR depth images and semantic segmentation maps, a choice has to be made on where these images should be merged together in the network. They can be concatenated together channel-wise before or in the middle of the CNN visual encoder, or they can each have their own CNN visual encoder and let their feature vector be concatenated afterwards. The neural network architectures used in this thesis will be explained in more detail in section 3.1.1.

2.4 Related Work

2.4.1 Implementation Matters in Deep RL: A Case Study on PPO and TRPO

Most implementations of RL algorithms such as PPO contain ”code-level optimizations” which are types of algorithm augmentations. These are often only described as implementation details of secondary importance in the literature. Engstrom et al. [2020], however, find that these details have a major impact on the behavior of the agent. Not only are they responsible for most of the improvement PPO has over TRPO, but they also fundamentally change how the algorithm operates.

Since the code-level optimizations PPO uses are algorithm agnostic in the way that they can trivially be applied to any policy gradient method, they can also be added to the PPO predecessor TRPO. In an attempt to correctly attribute the success of PPO the researchers create the following variants of PPO and TRPO:

- **TRPO+**: TRPO with the addition of the code-level optimizations used in PPO.
- **PPO-M**: A core PPO implementation without the code-level optimizations.
- **PPO-NoClip**: Standard PPO implementation with code-level optimizations but without the clipping mechanism.

Their experiments showed that varying the use of code-level optimizations had a significantly higher impact on the performance than varying whether PPO or TRPO was used. They also found that PPO-NoClip achieved similar benchmark performance to PPO without using any clipping mechanism at all.

While it is widely believed that the success of PPO comes from the clipping mechanism, these results suggest that the success actually comes from the implementation details such as the code-level optimizations.

2.4.2 What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study

Similarly to the previous article, Andrychowicz et al. [2020] argue that implementation details are crucial for the good performance of many state-of-the-art RL implementations while often not being discussed extensively in the literature. This makes it hard to attribute progress in RL algorithms since it is hard to know if the improvement comes from the algorithm itself or the implementation details.

The paper continues by investigating the effect different low-level implementation details have on the performance of on-policy RL using PPO. The authors train over 250 000 agents where they vary >50 different design choices. The findings give insights into which design choices matter the most, and provides recommendations for these design choices when implementing on-policy algorithms. One surprising finding they made was that the policy initialization scheme had a large impact on the performance of the policy while rarely even being mentioned in RL publications.

2.4.3 Survival-Oriented Reinforcement Learning Model: An Efficient and Robust Deep Reinforcement Learning Algorithm for Autonomous Driving Problem

Ye et al. [2017] aims to improve the safety of reinforcement learning approaches by prioritizing survival rather than maximizing reward. This is done by a Negative-Avoidance or danger function $D(s, a)$ which predicts a danger score of taking action a in state s. This danger function is trained by giving state-action pairs leading to early termination of an episode a danger score.

The authors argue that defining an optimal policy is a bad idea, as the real goal is to learn a policy to safely control the speed and lane position, not staying in some arbitrarily defined optimal speed or a specific position in the lane. It is also hard to learn to respond to rare accident scenarios with a small number of samples.

Their experiments involve lane-keeping and collision avoidance and are done in the TORCS simulator. They find that their model converges faster and is less sensitive to the reward function than the equivalent model without the Negative-Avoidance aspect.

2.4.4 Learning to Drive in a Day

Kendall et al. [2018] create and tune a model using a custom-made simulator with procedural generation creating a country road. They found that using Variational Autoencoders to learn a compressed latent space of the RGB image input and using this to encode the image input greatly increased sample efficiency. After finding a good set of hyperparameters for the simulated environment, they trained a model from scratch in the real world using the same hyperparameters and managed to successfully perform lane-keeping on a 250m stretch of road after only 11 episodes of training. The Deep Deterministic Policy Gradient algorithm was used. As DDPG is an off-policy algorithm it can use experience replay and therefore have a higher sample efficiency compared to on-policy algorithms.

2.4.5 Driving Policy Transfer via Modularity and Abstraction

Mûller et al. [2018] divide the end-to-end driving task into modules that abstract away domain-specific details such that a policy trained in this abstract space can be transferred to the real world.

The first module is the perception module that translates an RGB image into a semantic segmentation map. The middle module is the driving policy module which predicts waypoints that the car should steer towards based on the segmentation map. The last module is a low-level controller that translates the waypoints into control signals such as acceleration/braking and steering angle for the vehicle. The first and last modules are domain-specific, so that the middle one can be domain-independent.

The authors found that this architecture allowed them to transfer a policy learned exclusively in the Carla simulator to a 1:5 scale robotic truck in the real world. The truck was able to complete multiple routes with lengths up to 1.1 kilometers, albeit with some missed turns.

2.4.6 RL-CycleGAN: Reinforcement Learning Aware Simulation-To-Real

Rao et al. [2020] perform a sim-to-real transfer on robotics grasping tasks by using a CycleGAN to translate the image to a realistic style. The network is trained with an RL-Scene consistency loss which ensures consistent Q-values for both the simulated and generated image. This helps the GAN keep important information relevant to the RL task and prevents the GAN from making the generated image too different such that the action outcome would be changed. This method addresses only the visual "reality gap," and it does not address the physics gap between simulated and real-world physics.

This method requires off-policy data from the real world to first train the initial CycleGAN, and then to train the RL model off policy. The model is later improved in simulation with realistic images generated by the CycleGAN.

The authors suggest in future work that Augmented CycleGAN (Almahairi et al. [2018]) could be used to extend RL-CycleGAN produce stochastic image outputs.

2.4.7 CIRL: Controllable Imitative Reinforcement Learning for Vision-based Self-driving

Liang et al. [2018] use Imitation Learning to initialize the action exploration into a reasonable space. They argue that this reduces the chance that the policy gets stuck in a bad local optimum. They then continue training using the Deep Deterministic Policy Gradient algorithm to improve the policy. The architecture uses a gating mechanism to select the network head to use based on a high-level command, allowing the car to respond to navigation commands. The four different high level commands are $\{TurnLeft, TurnRight, Straight, Follow\}$. This allows the model to learn specialized policies for each high-level command, and also allows the authors to use different reward functions for each policy head. The model was tested in the Carla simulator using the Carla benchmark and outperformed all other methods at the time in terms of the percentage of episodes successfully completed.

Chapter 3

Methodology

3.1 Implementation details

3.1.1 Neural network architecture

Several neural network architectures are evaluated in this thesis. The first is inspired by the architecture used in ML-Agents Release 10. This allowed us to verify that the PPO algorithm was implemented correctly, as the results could be compared to those from the specialization project, where the ML-Agents trainer was used in the same environment.

Once the initial implementation performs at least as well as the ML-Agents implementation, tweaks can be made in an attempt to improve the performance. It seems reasonable to assume that the ML-Agents implementation is designed to perform well in a wide variety of environments, and that more optimizations can be made for our specific use case.

A revised second architecture based on recommendations from Andrychowicz et al. [2020] is also made. The first and second architectures will henceforth be referred to as Implementation A and Implementation B, respectively.

The policy function and value function used by PPO are approximated by using deep neural networks as function approximators. These networks take an image from a camera sensor and a vector of real numbers containing values such as the car's velocity as input. Output is an action vector for the policy function and a real number for the value function.

Each neural network consists of a backbone network that is identical for both the policy and value network. The backbone network extracts features from the visual and vector observations into a 512-dimensional feature vector. This vector is fed through the value or policy head to complete the value and policy networks, respectively.

Backbone network

The image is passed through an encoder resulting in a vector of extracted features that can be concatenated with the vector input. This encoder can be trained from scratch, but it is also possible to use the encoder from a pre-trained Variational Autoencoder. Both approaches are compared in this thesis. The concatenated vector is then run through two fully connected layers and then the head of the policy or value network. It is possible to use the same backbone network for the policy and value function and attach both heads at the end in parallel, but this paper uses separate networks for both as this results in more stable training. Implementation A uses the Swish activation function after each fully connected layer, while implementation B uses the tanh activation.

Although the value and policy networks share the same backbone architecture shown in figure 3.1, they are separate networks and do not share parameters. This approach is recommended in Andrychowicz et al. [2020] as they find it leads to better performance in 4 out of 5 tested environments. The ML-Agents implementation of PPO also uses separate networks.

Policy network head

The policy network head starts with a fully connected layer reducing the 512-dimensional feature vector from the backbone to a 2-dimensional action-mean vector. The network then samples from a normal distribution with the given action-mean and learnable standard deviation. This standard deviation is per action and global, meaning it isn't state-dependent. The details of this action sampling vary between implementations:

Implementation A The standard deviations are initialized to 1. The sampled action is clipped into the range $[-3, 3]$, and then divided by 3 resulting in an action in the desired range [-1, 1].

Implementation B The standard deviation parameters are initialized to 1, but a constant is subtracted such that the resulting initial standard deviation after a softplus operation is 0.5. The value after the softplus operation is used when sampling from the action distribution, and the actions are then passed through a tanh activation to confine the values to the [-1, 1] range. An illustration showing the policy head from this implementation can be seen in figure 3.2.

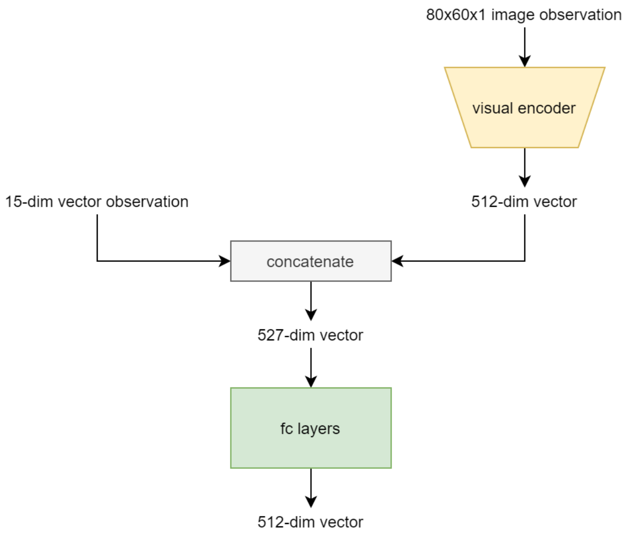


Figure 3.1: Overview of the backbone of the neural network architecture used for the policy and value networks. The visual observation is encoded with a CNN-based encoder that is either trained from scratch or taken from a pre-trained Variational Autoencoder. The encoded vector is then concatenated with the vector observation that contains the velocity of the car and the previous actions, and is then fed through two fully connected layers. The illustrated network uses a 512-dimensional vector for the encoded image, but lower dimensionalities are also used.

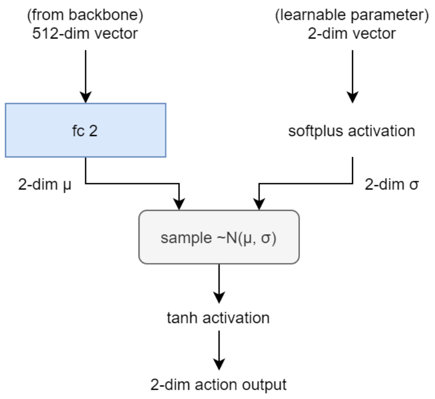


Figure 3.2: The policy head used in implementation B using the Proximal Policy Optimization algorithm. The output of the backbone network is passed through a fully connected layer with no activation function, outputting a 2-dimensional vector used as the mean of a normal distribution. A learnable parameter is passed through a softplus activation before being used as the standard deviation for the normal distribution. The softplus activation ensures the standard deviation will never be negative. An action is sampled from the normal distribution and passed through a tanh activation to force the values into the $[-1, 1]$ range resulting in a sampled action output. This architecture is based on the recommendations by Andrychowicz et al. [2020].

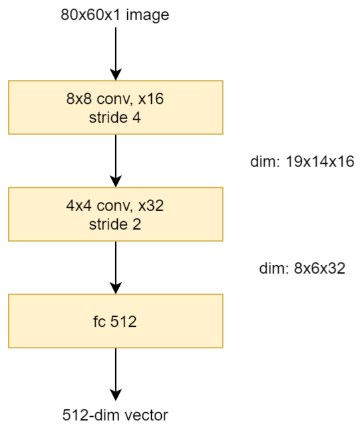


Figure 3.3: Illustration of the "simple" visual encoder, which is the default used by ML-Agents, and is the one used by our models that learn a visual encoder from scratch. Each layer has a LeakyReLU activation.

Value network head

The value network head consists of a single fully connected layer with one output value. No activation function is used since the value can be any real number.

3.1.2 Visual encoders

The visual encoder is the module that turns the visual observation (image) into a feature vector. A Convolutional Neural Network is used for this task as it is specialized in handling image processing tasks.

The ML-Agents framework has a few visual encoders to choose from, where the default one is called the "simple" encoder, and it is the encoder used in the our experiments that train a visual encoder from scratch. The encoder consists of two convolutional layers and a fully connected layer, each with a LeakyReLU activation function. An illustration of this network can be seen in figure 3.3. Implementation B uses the tanh activation function instead.

The other visual encoder used is the encoder from a Variational Autoencoder that is pre-trained on images collected in the environment.

3.1.3 Variational Autoencoder

Variational Autoencoders have been shown to improve training speed and performance of reinforcement learning models as shown in Kendall et al. [2018].

Autoencoders are a type of neural network that consists of an encoder and a decoder. The encoder encodes a high dimensional input such as an image into a low dimensional latent space vector with z_{dim} dimensions. The decoder takes the low dimensional vector as input and tries to reconstruct the original image. This network can be trained by minimizing the mean square error of the pixel values in the output compared to the input. This way the low dimensional latent space vector can be seen as high-level features that describe the image. This low dimensional vector is used as the encoded image in the backbone network for the reinforcement learning models and therefore the network does not need to learn a visual encoder from scratch. A Variational Autoencoder is a type of autoencoder introduced in Kingma and Welling [2013] that predicts a distribution of latent space vectors instead of a deterministic latent space vector. This is done by predicting mean and standard deviation vectors that are used to sample a vector that is passed back through the decoder. The loss function for VAEs includes a KL-divergence term to hinder this distribution from straying too far away from a normal distribution. Figure 3.4 shows the Variational Autoencoder used for the grayscale images in the Unity lane following environment.

3.1.4 Proximal Policy Optimization

The Proximal Policy Optimization algorithm was chosen to be used in this thesis. PPO was chosen because it is a good general purpose Deep Reinforcement Learning algorithm that has been shown to work well for the type of environment we are interested in. PPO also performed well in our Unity environment created in the specialization project. Recently the Phasic Policy Gradient algorithm was published which supposedly is better than PPO, so a decision had to be made on whether to use that algorithm instead.

PPO vs. PPG

Mohanty et al. [2021] run a competition with Atari-like environments from the ProcGen benchmark. Participants submit algorithms that are evaluated for generalization ability and sample efficiency on ProcGen benchmarks, as well as a few hold-out environments created specifically for the competition, which the participants did not have access to during the competition. The highest performing submission both in terms of generalization and sample efficiency was based on PPG, but PPO-based submissions were not far behind. There was also a PPG-based implementation that performed worse than multiple PPO-based submissions in terms of generalization ability. All submissions were highly tuned implementations and the results show that using PPG instead of PPO doesn't automatically improve performance. PPG also has a few additional hyperparameters compared to PPO. This makes hyperparameter optimization more difficult

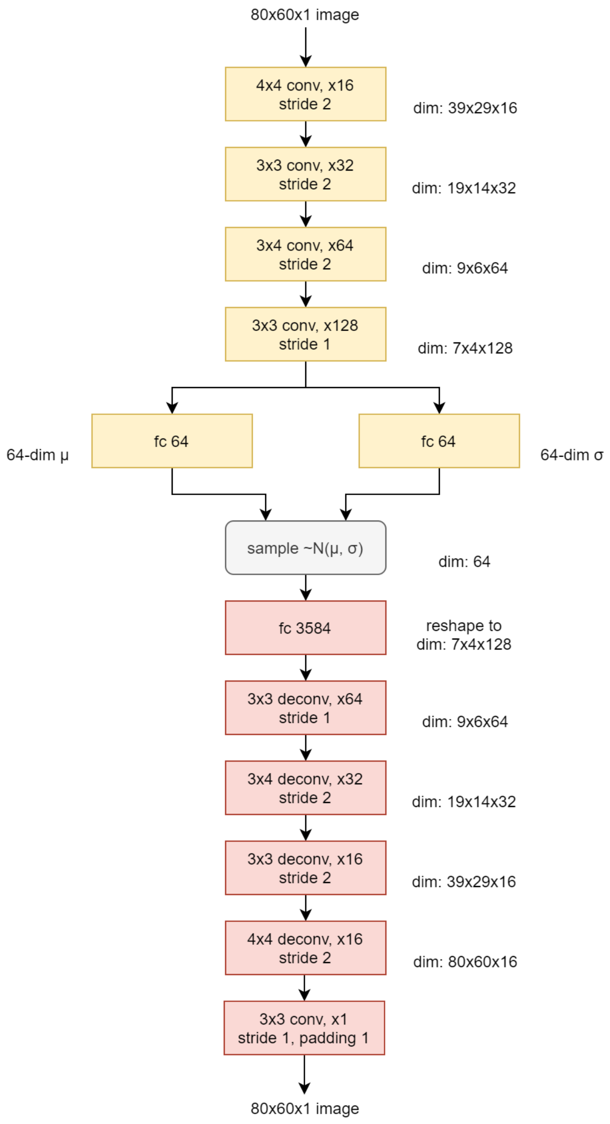


Figure 3.4: CNN archiecture of the Variational Autoencoder used for the Unity environment. Note that the *deconv* layers in the decoder are more accurately called transpose convolution layers. Each convolutional and transpose convolutional layer is followed by a batch normalization layer that normalizes each channel and a LeakyReLU activation function. The exception is the final convolutional layer which doens't have a normalization layer and uses the tanh activation.

and time-consuming, which is a big disadvantage since hyperparameter optimization for PPO is already challenging with our limited computational resources.

The main difference between PPG and PPO is the feature distillation phase which supposedly helps the policy learn better features. The original paper's authors conjecture that visual domains will have the most to gain from this since their high dimensional input space increase the importance of sharing parameters between the value and policy network. This gives reason to believe that the benefit will be less significant when a Variational Autoencoder is already used to extract a lower-dimensional set of features. Finally as shown in sections 2.4.1 and 2.4.2 the code level optimizations are sometimes more impactful on final performance than the actual underlying algorithm. Based on this reasoning, we chose to continue with PPO, which we already had a working implementation of at the time the competition paper showing PPGs superior performance was published.

PPO Implementation

The policy and value network weights are randomly initialized when training starts. The current policy is used by potentially multiple agents in parallel to collect experience into episodes. For each timestep the following is stored in the agent's buffer:

- Environment state
- Sampled action
- Reward received
- Log probability of the sampled action
- Value of the state as predicted by the value function
- Whether the state was a terminal state

Whenever an agent completes an episode by either failing or reaching the maximum length, the discounted reward for each time step is computed, and a list of state transitions is created. If the episode is terminated due to reaching the maximum length, the discounted reward at the final step is set to the prediction from the value function.

Completed episodes are added to the global buffer, which collects state transitions from episodes until it reaches a threshold number of state transitions that trigger a policy update. After the policy update, all agent environments are reset, and any unfinished episodes are discarded since they were collected using a now outdated policy. During the policy update, the buffer is split into batches, and the advantage estimate is computed. The loss is calculated with the clipped policy loss, value loss and entropy loss. The loss is backpropagated and an optimizer step is taken using the Adam optimizer. This is repeated until multiple passes (epochs) over the entire buffer are completed.

Implementation A computes discounted rewards once, creates minibatches by iterating over the buffer sequentially. Advantages are computed by subtracting the value as predicted by the value function from the discounted rewards (equivalent to GAE with $\lambda = 1$).

Implementation B computes Generalized Advantage Estimates for the advantage, which are recomputed for every pass over the data. This implementation also shuffles the dataset at the state-transition level when creating minibatches instead of sampling sequentially from the buffer like implementation A.

3.2 Lane following in a Unity environment

Unity was chosen as the platform to create a low fidelity environment due to its ease of use and the availability of the ML-Agents framework facilitating rapid deployment of RL agents.

3.2.1 Procedurally generated roads

Creating procedurally generated roads has many advantages. Each road is different due to random generation, which prevents overfitting to a specific set of roads which could happen with premade road maps. It also allows a high degree of control over the generated road shape by tuning parameters of the road generator such as road width and curvature.

The procedural road generator works by keeping track of the position and rotation of the current endpoint of the road and creating and attaching a newly generated segment whenever the endpoint's distance from the car agent falls below a set threshold distance. The new segment will have a fixed length and a curve that defines how much the road direction will rotate over the length of the segment. Road segments that the car has passed will be deleted as they are no longer needed. Figure 3.5 shows what such a segment looks like, while figure 3.6 shows what a road with many segments stitched together looks like. The road can be rotated about any of the three axes x, y and z, which will cause hills, turns and banks, respectively.

3.2.2 Creating a car in Unity

A car can be created in Unity using its included physics engine. This can be done by first creating four WheelColliders as the wheels and then attaching them to a

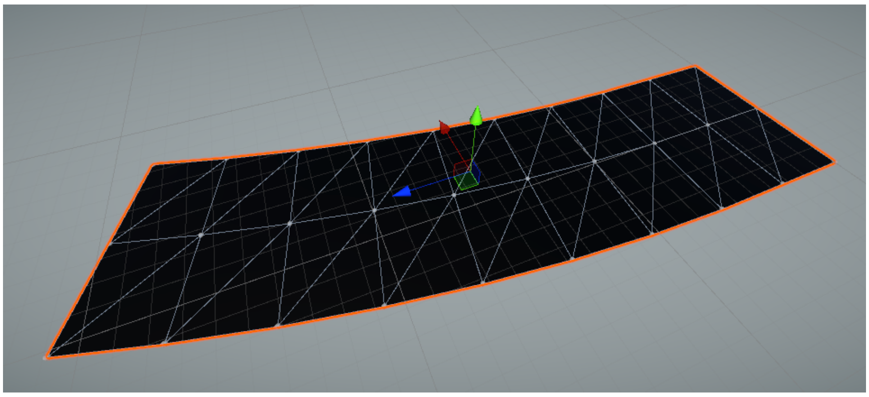


Figure 3.5: A 30 meter long and 8 meter wide segment of track with a 20-degree rotation about the y-axis. The triangle mesh that makes up the track is visible.

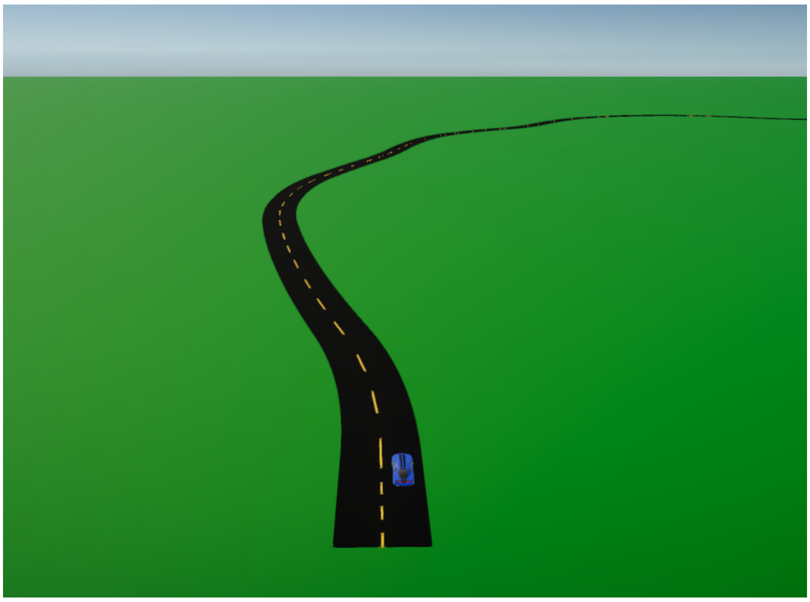


Figure 3.6: An example of what a randomly generated track can look like in the Unity lane environment. The track is 8 meters wide and is made up of 30 meter long segments each with rotation about the y-axis sampled uniformly in the $[-25°, 25°]$ range.

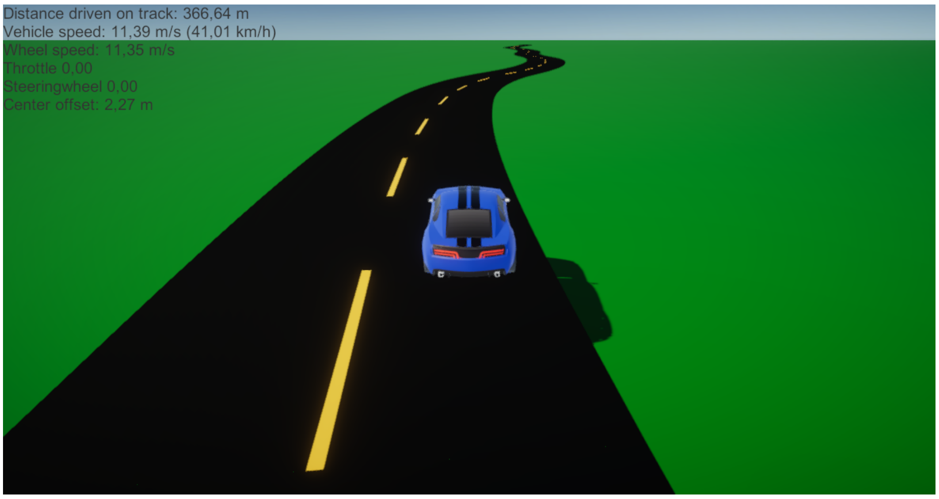


Figure 3.7: The car driving in the Unity environment.

RigidBody as the car's body. A torque is applied to the front wheels to accelerate the car, and a brake torque is applied to all wheels to brake. The front wheels are rotated to steer the car. A CameraSensor is placed on the windshield at the top of the car which will be used as the visual observation.

The model for the vehicle is purely cosmetic and doesn't affect the vehicle's physics in any way. The exception is that the hood of the car is visible in the visual observation. We do believe that the CNN will learn to ignore this, however. A free model from the Unity Asset Store was used for the car in this project. Figure 3.7 shows the car in the lane environment.

3.2.3 Integration with ML-Agents

The ML-Agents framework requires an Agent script to be created, which needs to implement specific methods.

Observations The visual observation is an 80x60 image from the front-facing camera sensor and is handled automatically by the framework. Figure 3.8 shows an example image from this camera. Vector observations have to be added in a designated method in the Agent script. The vector observation consists of the car's velocity in each dimension relative to its own rotation (e.g., z-axis velocity is forward velocity). The previous action is concatenated, and the resulting 5-dimensional vector is stacked with the same vectors from the previous two steps.

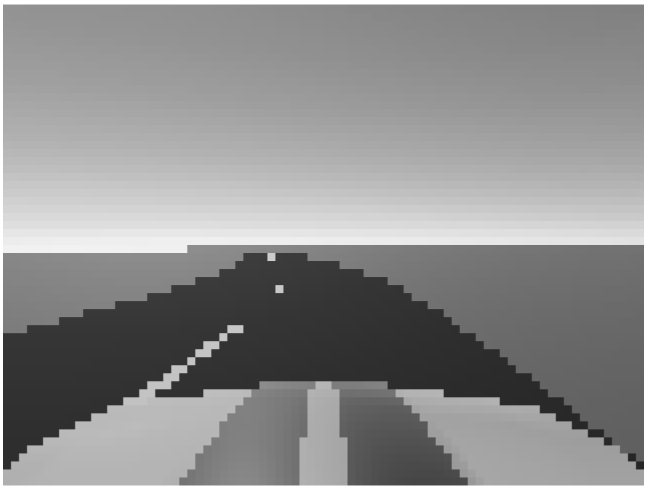


Figure 3.8: An 80x60 grayscale visual observation as seen by the agent and used as input to the RL model.

The final vector observation is thus a 15-dimensional vector.

Rewards The car's reward is proportional to the length of road covered. This is calculated as the exact length along the center of the road. This is almost equivalent to rewarding based on the speed of the vehicle. A side effect of this reward function is that it encourages inner turns as the distance traveled along the center is then slightly higher than the actual distance traveled by the car. If the car is in the wrong lane, the distance-based reward gets its sign flipped, and the episode terminates if the agent fails to return to the correct lane within 2 seconds. The agent acts in the environment for up to 3000 steps, which is equivalent to 5 minutes of simulated time. The episode terminates early with a large negative reward if any of the following fail conditions occur:

- The car drives off the road. This fail condition is triggered if a downwards raycast starting from the car's center of mass doesn't hit the road.
- One or more wheels are off the road. This fail condition is triggered if there is at least one wheel that isn't touching the track for two consecutive seconds.
- The car is driving on the wrong lane. This fail condition is triggered if the car is driving in the wrong lane for at least two consecutive seconds. The car is considered to be driving in the wrong lane if the car's center of mass

projected onto the road is less than 1 meter to the right of the center of the road.

- The car is moving too slowly. This fail condition is triggered if the average velocity of the car during the last ten seconds is less than 1 m/s.

Actions The actions predicted by the network are the steering and throttle signals which both are in the range of $[-1, 1]$. Another option is to predict a separate throttle and brake signal in the $[0, 1]$ range, but since one should never use both at the same time anyway, it seems appropriate to predict an action in the $[-1, 1]$ range and just interpret negative throttle signals as brake signals.

The throttle signal is applied to the Car by applying a torque proportional to the throttle signal to the front wheels. The steering signal is applied by rotating the front WheelColliders by an angle proportional to the steering signal.

Parallel environments ML-Agents supports parallel environments which allow multiple agents to train simultaneously in the same world. This was implemented by stacking 16 environments vertically. Each environment consists of the car, the generated road and the green background that is just a flat green plane below the road. Placing the environment far away from the origin of the Unity world causes the physics engine to behave differently due to larger errors in floating-point arithmetic. This is why the environments were stacked closely together with a vertical distance of 50 meters between each environment.

3.3 Lane following in Carla

The Carla environment was chosen as the high-fidelity simulated environment because it has photorealistic graphics, is easy to use and is widely used in the literature. There are many included maps in Carla, but most of them are urban environments with many intersections and terrain that is very different from the Unity environment. Each map comes with a list of spawnpoints which are all valid locations and orientations for cars to be spawned. This is useful wishing to place cars in random but valid locations. For a lane following environment in Carla the *Town07* map was chosen. The map is a part of the *AdditionalMaps* package and is downloaded separately. This map is a more rural area compared to other maps and contains sections with several hundred meters of rural road through forest. Figure 3.9 shows the car driving in this area. We limit the scope of the experiment to lane following since it is a core task in autonomous driving and a good starting point for a reinforcement learning approach. It is also the same task that was done in Unity, which allows us to reuse most of the methodology. A birds-eye view of the map used can be seen in 3.10.

Figure 3.9: The car driving in the outskirts of *Town07* which is where we defined our lane following environment.

3.3.1 Reinforcement Learning setup

Episodes Episodes are defined by pairs of spawnpoints. The first element in the pair is the starting point for the episode where the vehicle begins. The second element is the goal point of the episode, which is the point right before the road reaches an intersection. The episode is successfully completed if the distance to this point falls below a threshold distance. Figure 3.11 shows the stretches of road used. At the start of each episode, the vehicle is *primed* by enabling the autopilot for a random amount of time between 2 and 3 seconds of simulator time, before handing the control to the reinforcement learning system. The reinforcement learning system is oblivious to the priming phase and will only start collecting observations afterward. This has several advantages. First, it makes sure the car responds immediately to throttle signals. In Carla, a car standing still is a bit slow to react to throttle signals, taking about 2 seconds to react initially. After it starts moving, the car is much more responsive to control signals, meaning it is easier for the agent to see how its actions affect the environment. The second advantage is that it slightly randomizes the initial location of the car at the start of each episode, so that even episodes that start at the same spawn point will start in slightly different locations. We believe that these advantages are helpful to the learning process. An episode also terminates with a failure and a large negative reward if any of the following fail conditions are reached:

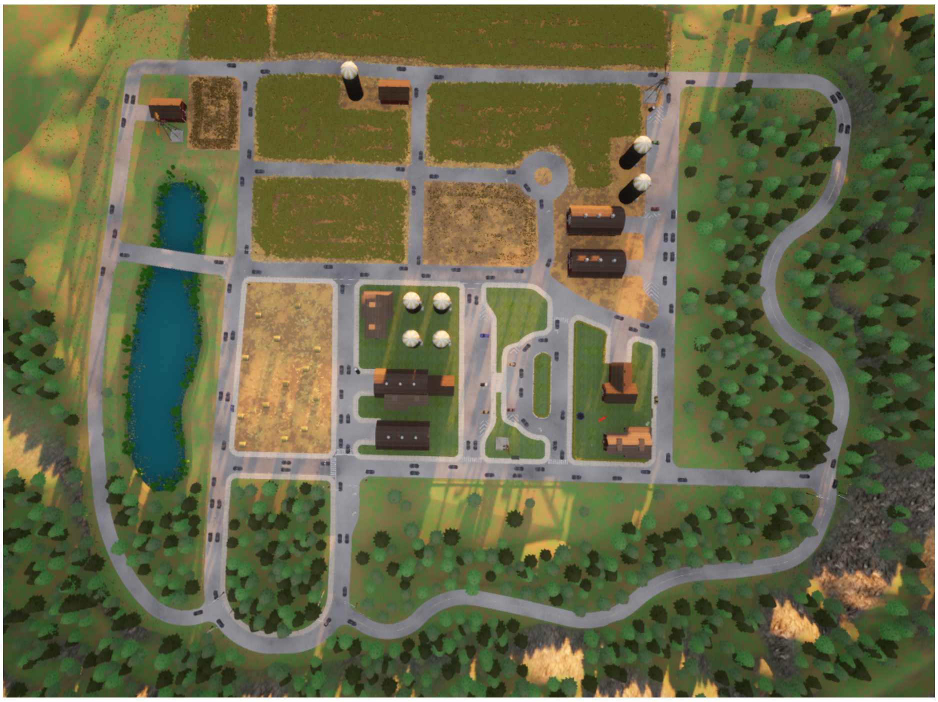


Figure 3.10: A birds-eye view of the *Town07* map in Carla. The spawnpoints have been visualized by spawning a black car at each one. The lane-following parts can be seen in the west, south and east outskirts of the map.

- The car's collision sensor is triggered.
- The car's lane invasion sensor is triggered. This sensor is triggered if the car gets too close to the other lane or drives off the road.
- The average car speed over the last 100 time steps is less than 1 m/s, the current speed is less than 0.5 m/s and the length of the current episode is more than 200 steps (20 seconds simulator time).

Lenient mode The lane invasion sensor is triggered very easily which terminates episodes that slightly touch the lane markings but otherwise would continue driving perfectly fine. Due to this, a *lenient mode* was added that replaces the lane invasion fail condition with one that is triggered if the lane center deviance is more than one meter. This is a more lenient fail condition and is used when testing some models that performed poorly under the stricter fail conditions.

Rewards The reward is given based on the car moving forward on the road. An easy way to do this that was initially implemented, is to give a reward proportional to the speed of the car. A few early models were trained with this approach. One problem with this reward scheme is that it is agnostic to the direction the car is traveling. The car can potentially receive more reward per meter of road by driving in a zig-zag pattern. Eventually, an analogous approach to the one used Unity was implemented: giving a reward proportional to the distance of road traveled. The Carla maps include detailed lane information making it possible to generate waypoints along the center of any lane on any road in the map. The PythonAPI includes a function that takes a location such as the location of the ego vehicle as input and returns that location projected onto the center of the nearest lane. Calculating the distance between these waypoints between steps thus gives an accurate distance measure for the length of road. Attempting to train with this distance measure quickly revealed that this function in the Carla API is very slow, with a single function call taking close to 100 milliseconds in our case. This resulted in the training time increasing by about of factor 4 as the function was called every step, so this distance measure was quickly abandoned.

A different approach was to use the Carla API to generate a list of waypoints containing evenly spaced waypoints for every lane in the map. This only needs to be done once at the beginning of the training session. Generating this list with waypoints spaced 0.5 meters apart results in a list of roughly 14000 waypoints. One could iterate through this list in pure Python code to find the nearest point which does result in a speedup, but this is still too slow. A better approach is to load all the waypoint coordinates into a single NumPy array and use NumPys efficient array operations to find the nearest point. The execution time for this

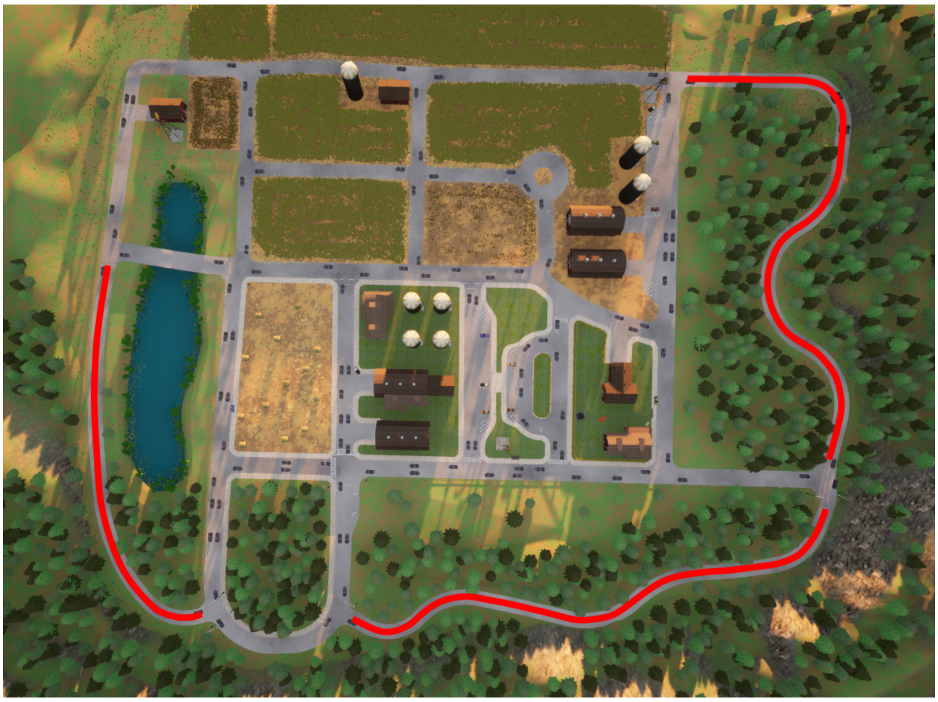


Figure 3.11: A birds-eye view of the *Town07* map in Carla with three routes marked in red. The marked routes are the routes used in our Carla lane following environment. An episode starts at a randomly chosen spawn point out of the ten that are inside the desired sections and ends right before it reaches an intersection.

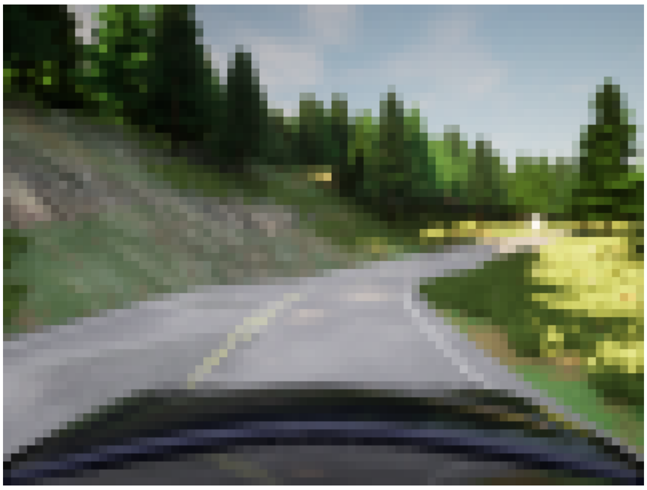


Figure 3.12: An 80x60 color visual observation as seen by the agent in the Carla lane following environment and used as input to the RL model.

approach is in the sub-millisecond territory, and the impact on training time is negligible.

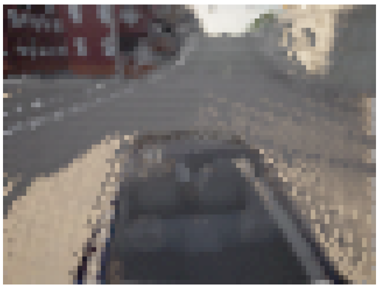
In order to calculate the precise distance the car traveled between two timesteps, a few extra steps are needed since the nearest waypoint only provides a granularity of 0.5 m. This is solved by also calculating how much in front of the waypoint the car is. This distance is calculated by defining a plane using the waypoints location and forward vector, and calculating the distance from the plane to the car's origin. This distance is essentially the car's distance along the road relative to the waypoint.

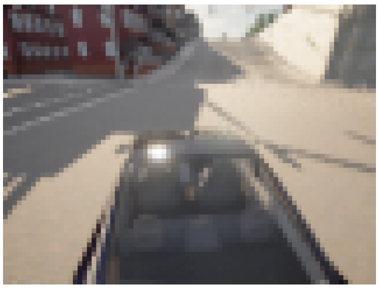
Observations A forward-facing camera is placed on the top of the windshield of the car as in the Unity environment. The camera provides 80x60 images with three color channels (red, green and blue). An example image observation taken from the environment is shown in figure 3.12. Color images are used since the graphics are more photorealistic and varied compared to the Unity environment, and the extra information in a color image is helpful in handling this additional complexity. A vector observation containing the actions and vehicle speed of the three previous steps is used. This resulted in a 9-dimensional vector.

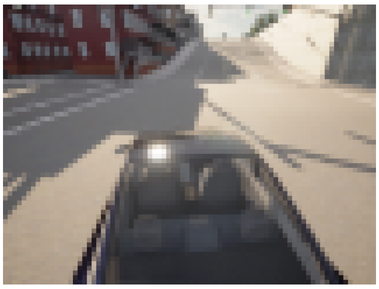
Supersampling of image observations It was observed that using a low rendering resolution such as 80x60 in Carla caused graphical glitches in the image when the car was moving and especially if the car was moving and turning at high speed. We suspect this is a form of *ghosting* caused by *temporal anti-aliasing* (TAA): an anti-aliasing technique used by many modern rendering engines including Unreal Engine which Carla is using. *Aliasing* is (in the context of computer graphics) an effect that causes edges in the image to appear jagged. TAA attempts to reduce aliasing in images by combining the frame with previous frames in order to smooth out the jagged edges. This can cause "ghosts" of previous frames to be visible especially if the scene contains fast-moving objects. The effect is more noticeable at low rendering resolutions. Another anti-aliasing technique called supersampling renders the image at a higher resolution internally before downscaling it to the desired resolution. The internal rendering resolution is controlled by the *render scale*. Supersampling with $render\_scale = 2$ would render an image with double the width and height, and each pixel in the downsampled image would correspond to the average of a 2x2 region in the original image. This achieves a high-quality anti-aliasing effect but comes with the downside of being computationally expensive since even a render scale of two quadruples the number of pixels needing to be rendered. An example of the ghosting effect and the benefit of higher render scales can be in figure 3.13.

It was decided to use a render scale of 3 as this mitigated the ghosting effect and resulted in a high-quality image without having too high of a computational cost. The render scale is implemented by setting the resolution of the camera in Carla to three times the desired width and height and then downscaling the image in Python before passing it to the RL system.

Actions The action definitions and interpretations were initially the same as the ones in Carla, with steering and throttle signals. Although some decent results were achieved with this setup, it caused some difficulties which will be explained in more detail in section 4.2. The throttle signal was therefore replaced with a target speed, and a simple low-level controller was created to convert the target speed into throttle or brake signals. This change would also be useful for the third experiment of transferring Unity policy to Carla. The low-level controller works by looking at the difference between the target speed and the current speed of the car. It then applies a brake or throttle signal depending on whether the car is driving faster or slower than the target speed. These signals are proportional to the speed difference and both max out at a speed difference of 5 m/s. The throttle signal is then capped at 1, and the brake signal is capped at 0.25. Figure 3.14 shows the control signals from the low-level controller as a function of the difference between the target speed and current speed. The brake signal has a lower cap because applying a brake signal of 1 has a greater impact and slows down the car faster than applying throttle of 1 accelerates it. This is problematic for learning. To see why this is problematic, consider a policy that samples target


(a) $render\_scale = 1$ (80x60)


(b) $render\_scale = 2$ (160x120)


(c) $render\_scale = 3$ (240x180)

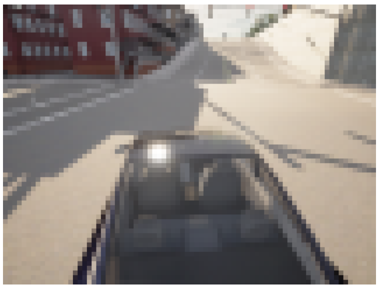

(d) $render\_scale = 4$ (320x240)

Figure 3.13: A comparison of 80x60 images from the Carla simulator at different render scales showing the benefit of rendering at a higher resolution to create a supersampled image. The render scale and the corresponding internal rendering resolution can be seen below each image. At a render scale of 1, some visual bugs can be seen. These disappear when increasing the render scale to 2. The difference between render scale 2 and 3 is less obvious but is noticeable when looking at details in the distance, such as the lane markings in the hill in front of the car. The lane markings are invisible in render scale 2, but can be seen in render scale 3 if looking closely. Increasing the render scale to 4 doesn't improve the quality in a meaningful way, so a render scale of 3 was used in experiments.

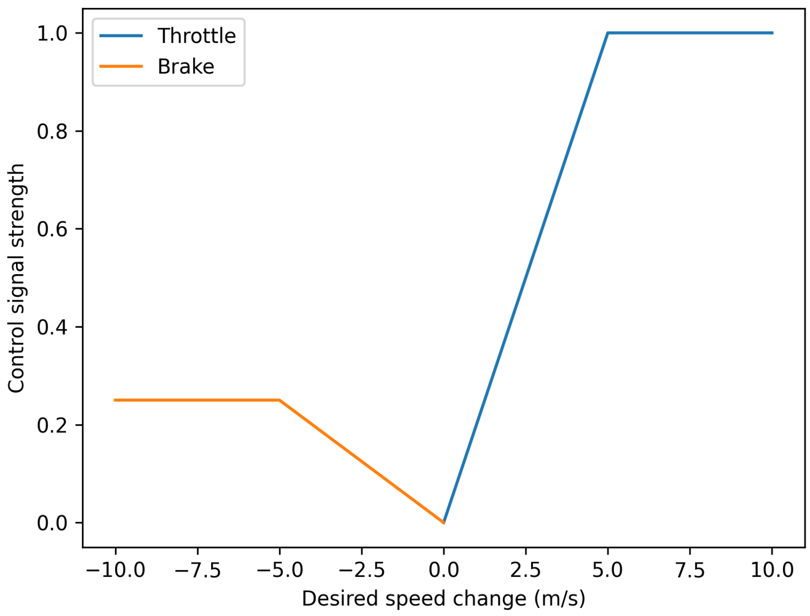


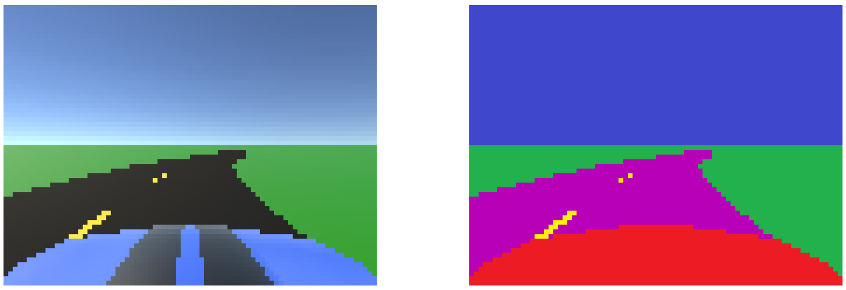
Figure 3.14: A plot showing the behaviour of the low level controller for the Carla environment converting a target speed to throttle and brake signals. The x-axis shows the desired speed change, which is defined as the difference between the target speed and the current speed.

speeds from a normal distribution $v \sim N(0, 0.3)$, which is what an initial policy will typically look like. Half the sampled target speeds will then result in throttle signals and other half will result in braking signals, but since the brakes have a greater impact on the speed they will more than negate any throttle signals causing the car to always stand still.

3.4 Transferring policy learned in Unity to Carla

3.4.1 Segmentation map in Unity

Due to the primitive graphics of our Unity lane following environment, it was assumed that creating segmentation maps based on the camera image would be easy since the different parts of the image to be labeled already have distinct



(a) Input image (b) Target label

Figure 3.15: The single training example used to train the semantic segmentation network. The semantic segmentation map has five classes: $\{Sky, RoadLines, Roads, Vegetation, Vehicles\}$. The Sky and $Vegetation$ classes were later merged into a single class.

colors.

Since the color of the pixel alone should be able to determine the class label, a Convolutional Neural Network with only 1x1 kernels was created. In practice this network acts as a fully connected network processing each pixel independently from each other.

The training data was created manually by creating color-coded segmentation maps in an image editor. Two examples pairs from grayscale images to segmenetation maps were created, but this was unable to provide satisfactory results. A modified version of the Unity lane environment that provided RGB image observations instead of grayscale was created, and a single segmentation example was manually created. This time the network converged to pixel-perfect segmentation maps on both the training example and several unseen test images. The network was trained with 10000 iterations over the training image using square error loss weighted by the inverse class frequency. The training example used to train the network can be seen in figure 3.15.

3.4.2 Segmentation map in Carla

Carla includes a segmentation map camera. This semantic segmentation image contains more classes than the one created in Unity, so some classes are merged together such that the number of classes match. This was done by mapping each class in the Carla image to the correct class as we defined. This was done by mapping the Carla $RoadLines$, $Road$ and $Vehicles$ to the same class in our image. Every other class including sky was mapped to $Vegetation$.

3.4.3 A stricter Unity environment

The Unity environment was modified before being used to train policies to be transferred to Carla. These modifications are intended to make the Unity environment behave more like the Carla environment. The width of the road was reduced from 8 meters to 7 meters meaning that each lane was half a meter narrower. The fail conditions were also made much more strict. Previously the episode would fail if the car's offset from the center of the road was less than 1 meter for 20 consecutive timesteps. This threshold was changed to 1.3 meters and with instant episode termination. A fail condition was also added to the outer edge of the road where termination happens if the offset from the edge of the road is less than 1 meter. Previously the episode would fail if at least one wheel was off the track for 20 consecutive timesteps.

These changes make the environment more like the Carla environment that terminates episodes upon triggering lane invasion sensors. The hood of the car which is visible in the camera is slightly bigger in the Carla environment. To ensure this didn't affect the policy, an overlay of this shape was added to the Unity semantic segmentation observation. This made the hood of the car have the same shape in both environments.

Chapter 4

Experiments and Results

This chapter will present the experiments and results and seeks to answer the research questions. Each experiment is organized into a setup, results and discussion part, and there are three experiments total.

The first experiment takes place in the Unity lane following environment. The purpose of this experiment is to use the low fidelity environment to build an autonomous vehicle system that performs well, and to investigate the effects of using Variational Autoencoders to encode the visual observations. This provides insight into *Research question 1* on Variational Autoencoders, and forms a starting point for experiment 2.

The second experiment takes place in the Carla lane following environment. Experiment 2 is based on the findings from experiment 1 and attempts to deploy an RL agent in Carla based on the best performing one in Unity. The findings from this experiment help answer *Research question 2*.

The third and final experiment takes place in both environments. The Unity lane following environment is used to train the RL agent before it is transferred to the Carla lane following environment and evaluated. This answers *Research question 3* on policy transfer.

A video compilation that shows the car agents driving in the various environments is available at `https://youtu.be/aqDVVh0rHzQ`. The compilation contains video samples of many driving policies driving in various environments. The video is not intended to be watched from beginning to end, but instead to serve as a library of videos of the various driving policies. Video recordings of the policies allow us to qualitatively evaluate them. The video contains samples of cars driving in the environment where it shows the car from a third-person view and the agent's visual observation that is used as input to the neural network. A graph shows the steering and throttle/speed actions over the previous three seconds. There are also video segments that compare the Variational Autoencoders of differing z_{dim} by showing the original input image and the reconstructed image after being passed through VAEs of different sizes.

4.1 Experiment 1: Lane following in Unity

Experiment 1 takes place in the Unity environment for lane following described in section 3.2. The main purpose of experiment 1 is to test the RL algorithms and model architectures such that the best-performing ones can be launched in Carla. This allows faster experimentation since simulation in the Unity environment is computationally cheaper than in Carla. An experiment was done using the same lane following environment in the specialization project, where ML-Agents' built-in RL algorithm was used.

The first goal of experiment 1 is to match the performance of the ML-Agents implementation of PPO to verify that the algorithm is implemented correctly. The next step is to use a Variational Autoencoder to encode the visual observations and see if it improves the speed of training and performance of the agent.

4.1.1 Setup

Models were initially trained on a desktop PC with an Nvidia GTX 1070 GPU, 16 GB of RAM and a 6-core Ryzen 5 1600X CPU. Some models were trained on a remote desktop virtual machine with a shared RTX 8000 GPU. Later on, most models were trained on the Idun GPU cluster described in Själander et al. [2019]. The cluster doesn't support graphics, so the Unity environment was run on the desktop PC and communicated with the trainer running on the cluster via sockets. This allowed faster testing of models since multiple models could be trained simultaneously. The Variational Autoencoders were all trained on the Idun cluster.

Each model is trained for up to a maximum of 15 million steps, which is equivalent to about 415 hours of simulated time. A checkpoint is saved during training every 100 000 timesteps in the simulator, with each timestep representing 100 milliseconds of simulated time. The model is evaluated at each checkpoint by collecting 100 episodes using the model and looking at the mean distance traveled as well as the number of successful episodes. An episode is successful if it reaches the maximum episode length without failing. When evaluating a model, the mean action is used instead of sampling from the action distribution.

Models are also evaluated for passenger comfort. This is done by measuring the jerk (rate of change of acceleration) experienced by the passenger. A higher average jerk implies a more uncomfortable ride. The change in throttle and steering angle actions between consecutive timesteps can be used as a surrogate metric for the longitudinal and lateral jerk. The jerk metric for an episode is then calculated as the mean absolute jerk of every pair of consecutive timesteps in the episode. This is averaged over 10 episodes for each checkpoint. We create a *lateral jerk score* based on this mean absolute lateral jerk metric. Both lateral and longitudinal jerk were highly correlated, so lateral jerk was chosen as the metric as it was also easier to see as an observer. The jerk surrogate metric has no defined unit, but it can still be used to compare policies against each other. The lateral jerk score has an upper bound value of 2, which is reached if the policy alternates between a steering signal of -1 and 1 at each timestep.

The experiments are done using two different PPO implementations: A and B, and both implementations are tested with and without a Variational Autoencoder. This gives a total of four classes of models tested. Different settings for the VAE are also tested with PPO implementation B.

Variational Autoencoder

A dataset is needed to train the Variational Autoencoder. This dataset was collected by training a model from scratch without the VAE, and saving the observed image every 100 steps. This ensures a diverse set of images including many near-failure states that might not be seen in the late stages of training but are seen in the early stages of its training. It is likely that a different model trained from scratch will observe a similar distribution of images during the course of training. Another advantage of collecting the dataset this way is less manual labor as an alternative would be to drive manually in the environment to collect the dataset. A third option would be to drive using autopilot to collect the dataset, but this will collect a different distribution of images as it will not contain near-failure states.

The resulting dataset of roughly 100 000 images were used to train the VAEs with different latent space dimensionalities z_{dim}. Ten images were chosen randomly and taken out of the training set to become to test set. The VAE models were qualitatively evaluated by manually checking if the reconstructed test images kept the shape of the road and lane markings roughly intact, as those are believed to be the most important features for the vehicle.

Each instance of the Unity environment contains 16 agents with their own roads. During training all 16 agents are collecting experience and storing the experience from each time step in their own buffer.

4.1.2 Results

Implementation A

The first goal of implementation A was to match the performance of the ML-Agents implementation of PPO. The ML-Agents implementation was used with the same environment in the specialization project where it achieved a mean distance of 5020 m and an episode success rate of 90 %.

Figure 4.1 and 4.2 shows the results of two separate training runs with implementation A with a visual encoder trained from scratch. Figure 4.3 shows both runs plotted together. The first run was trained for about 12 hours wall-clock time before it reached 15 million training steps. The second policy collapsed after about 8 hours of wall-clock time and was kept running in the collapsed state for 15 more hours before it was canceled.

Figure 4.4 shows the results from implementation A where the visual encoder is replaced by the encoder from a Variational Autoencoder with $z_{dim} = 64$. This model was trained for 16 hours of wall-clock time.

Implementation B

Figure 4.5 shows the performance of implementation B with a visual encoder trained from scratch. The policy can be seen to collapse at the end after training for eight and a half hours of wall-clock time. The training was canceled after 13 million steps or 21 hours of wall-clock time as the policy had not recovered. This network uses the tanh activation instead of LeakyReLU and Swish as explained in section 3.1, with the exception of the value network, which still uses LeakyReLU and Swish as in implementation A. The reason for this was that the initial experiments using tanh in the value function resulted in saturated tanh outputs and the value function collapsing to a single value regardless of state.

The next experiment shows models trained using a Variational Autoencoder. This time latent space dimensionalities z_{dim} from 8 to 128 were tried and compared. The results can be seen in figure 4.6, and the best performing checkpoint from each run with respect to mean distance can be seen in table 4.1. Figure 4.7 shows a scatter plot showing the mean distance and episode success rate of every checkpoint of all the VAE based models.

Model comparison

Figure 4.8 shows the mean distance of all four classes of models tested. Figure 4.9 shows a scatter plot of mean distance and episode success rate of all checkpoints of the same four models. The comfort of each class of model throughout the training process can be seen in figure 4.10. A scatter plot showing the correlation between mean distance and comfort can be seen in figure 4.11. Table 4.2 shows the checkpoints with the best mean distance of each model class.

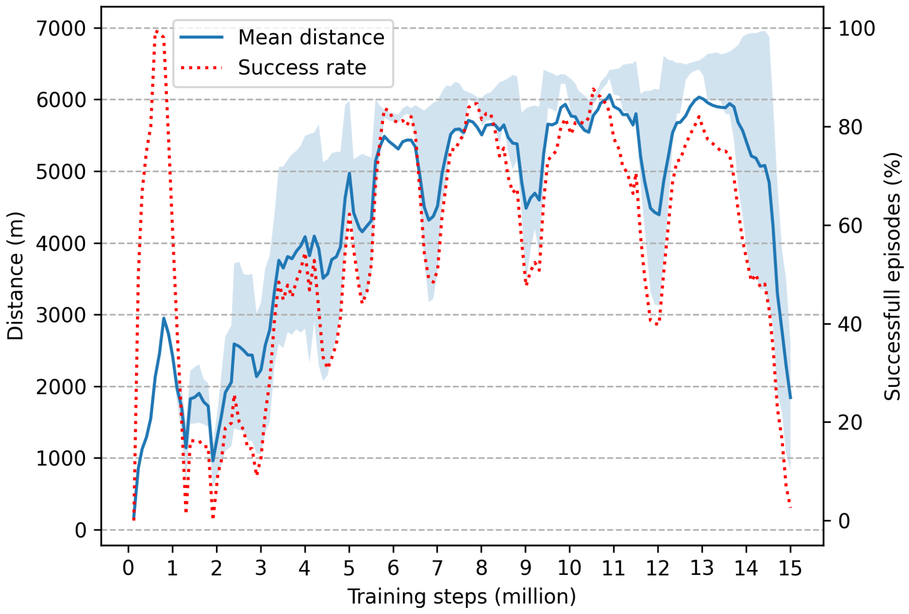


Figure 4.1: A run of implementation A with a visual encoder trained from scratch. The line shows the mean distance and the shaded area shows the interquartile range of distances traveled. The dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

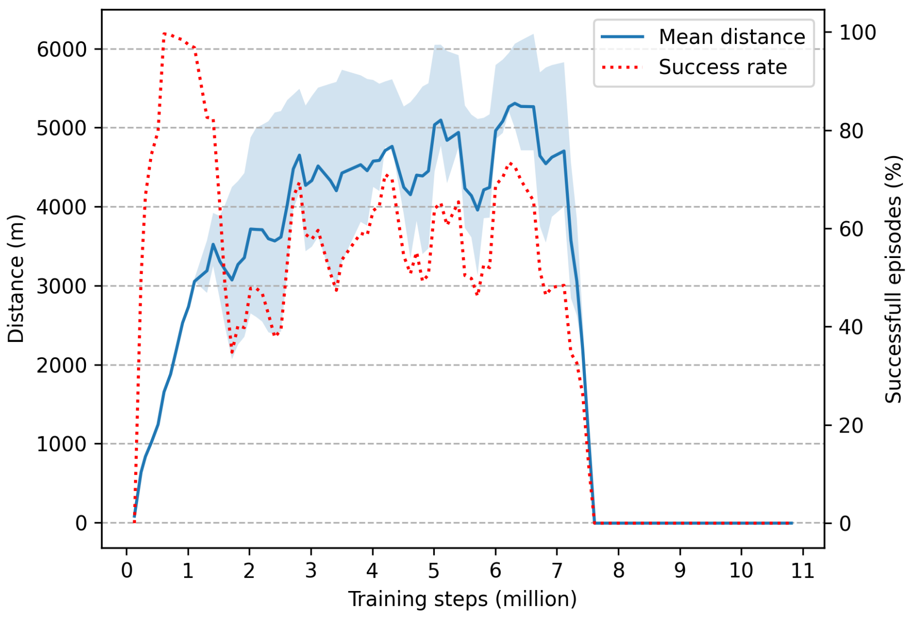


Figure 4.2: A second run of implementation A with a visual encoder trained from scratch. This run experienced a policy collapse after roughly 7 million steps of training and the policy never recovered. The line shows the mean distance and the shaded area shows the interquartile range of distances traveled. The dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

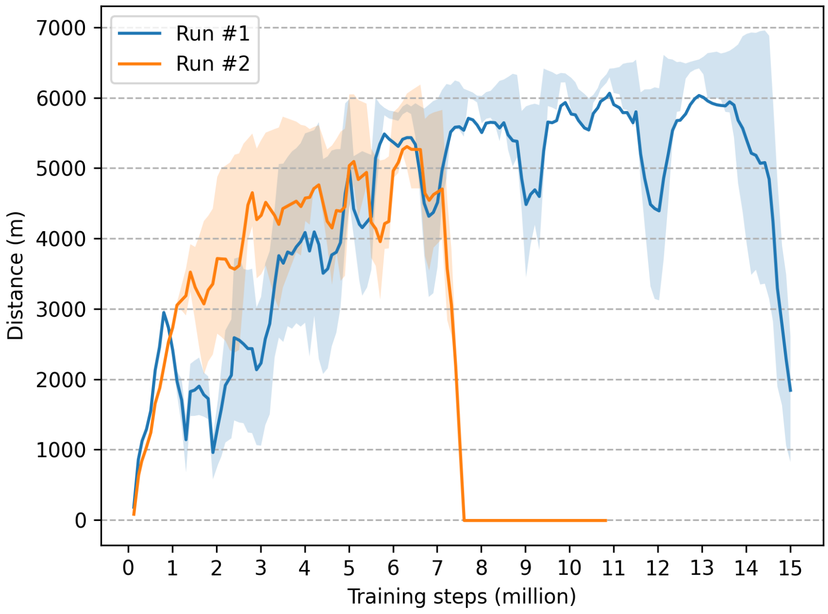


Figure 4.3: The two runs of implementation A with visual encoder trained from scratch shown together. The lines show the mean distance and the shaded areas show the interquartile range of distances traveled. A running average of the 5 preceding values is used to smooth out the graph.

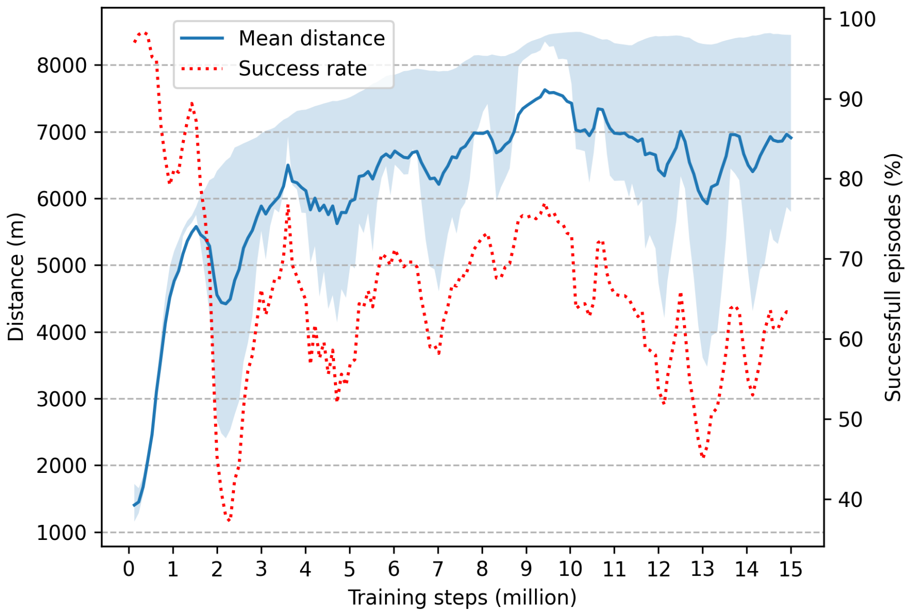


Figure 4.4: Evaluation of implementation A using a pre-trained Variational Autoencoder to encode the visual observation to a 64-dimensional vector. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. The dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

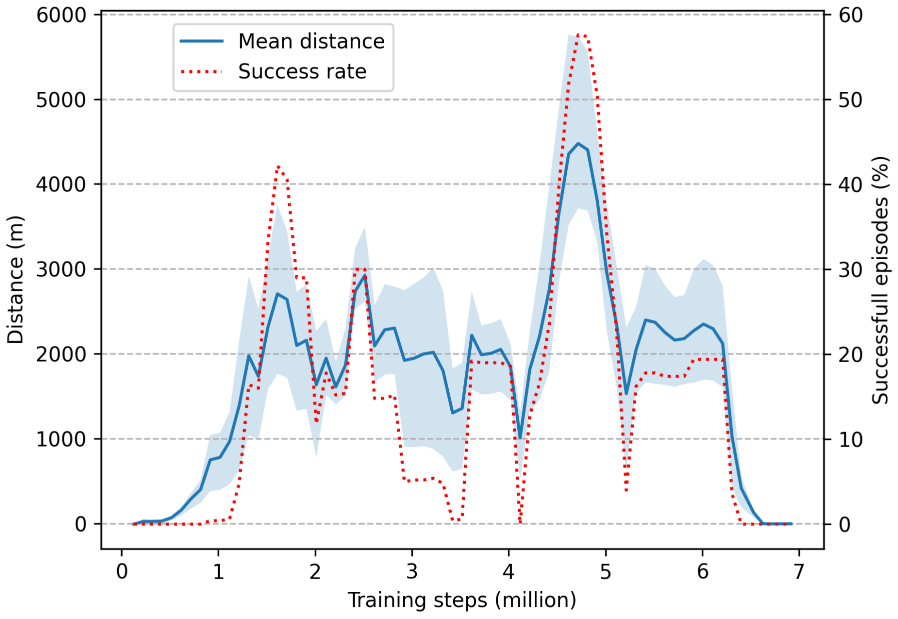


Figure 4.5: Evaluation of implementation B with a visual encoder trained from scratch. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. The policy collapses slightly before 7 million steps. A running average of the 5 preceding values is used to smooth out the graph.

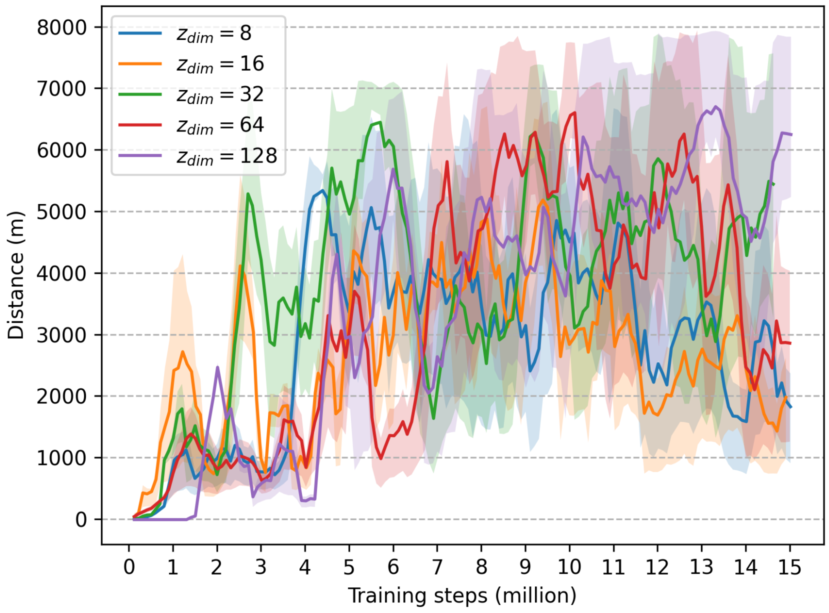


Figure 4.6: Training runs of implementation B using Variatational Autoencoders with differing latent space dimensionalities z_{dim}. The lines show the mean distance traveled by the agent, and the shaded areas shows the interquartile range of distances traveled. A running average of the 5 preceding values is used to smooth out the graph.

z_{dim}	**Step number** $(\cdot 10^6)$	**Mean distance**	**Success rate**
8	10.4	6725 m	78 %
16	9.3	6583 m	84 %
32	8.9	6835 m	80 %
64	9.8	**7526 m**	**98 %**
128	13.3	7045 m	69 %

Table 4.1: The model checkpoints with the best mean distance for models trained with varying z_{dim} in the Variational Autoencoder. The best results are marked in bold.

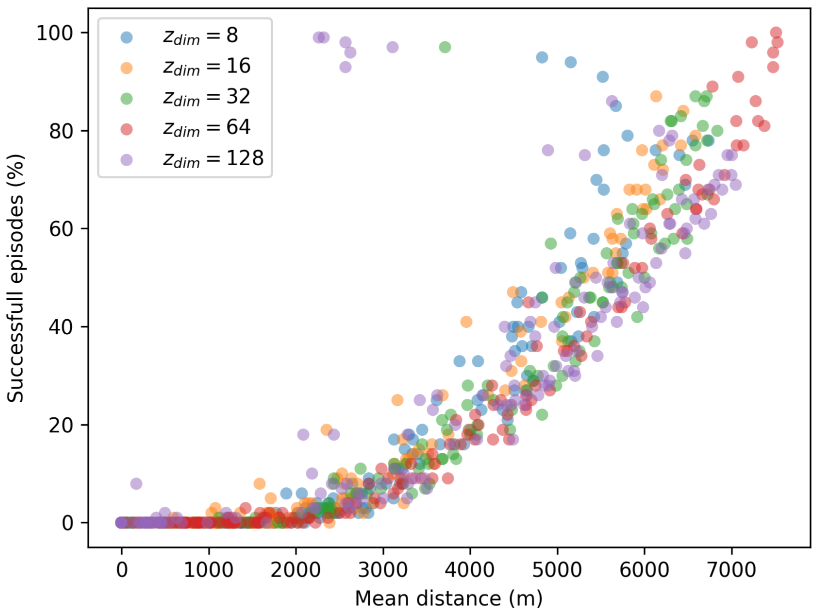


Figure 4.7: A scatter plot showing the mean distance and episode success rates of runs of implementation B with Variational Autoencoders with differing z_{dim}. Each point represents a checkpoint that was evaluated, with one checkpoint every 100 000 steps.

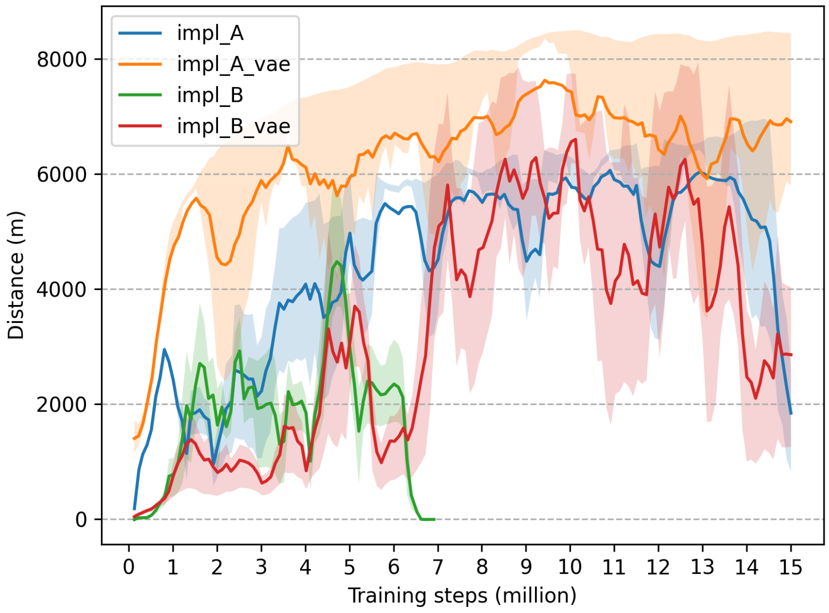


Figure 4.8: A comparison of implementations A and B both with and without a VAE with $z_{dim} = 64$. The lines show the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. A running average of the 5 preceding values is used to smooth out the graph.

Model class	**Step number** $(\cdot 10^6)$	**Mean distance**	**Success rate**
impl_A	10.9	6264 m	85 %
impl_A_vae	9.4	7895 m	80 %
impl_B	5.8	5451 m	79 %
impl_B_vae	9.8	**7526 m**	**98 %**

Table 4.2: The model checkpoints with the best mean distance for each class of model trained. The best results are highlighted in bold.

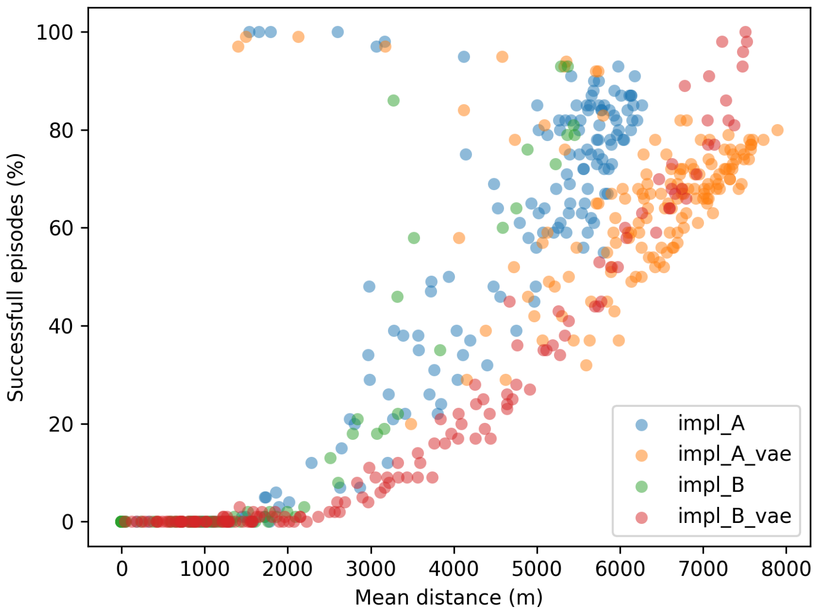


Figure 4.9: A scatter plot comparing implementation A and B both with and without a VAE with $z_{dim} = 64$. Each point represents a checkpoint that was evaluated, with one checkpoint every 100 000 steps.

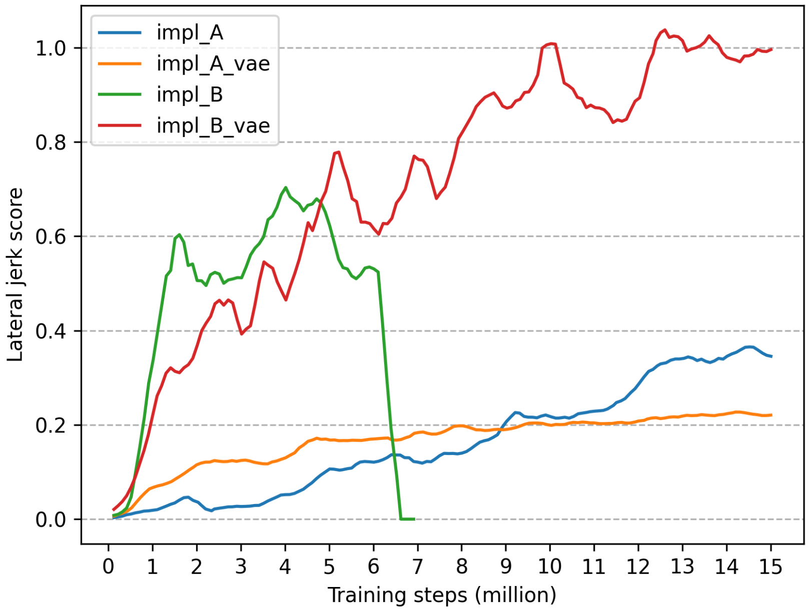


Figure 4.10: Evolution of comfort throughout the training process of each class of models. A lower jerk score corresponds to a more comfortable driving policy. A running average of the 5 preceding values is used to smooth out the graph.

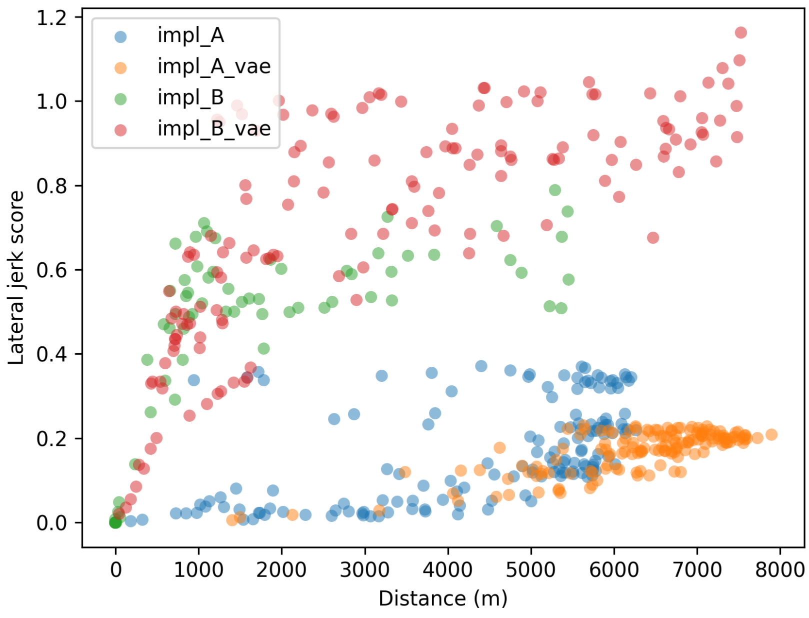


Figure 4.11: A scatter plot showing the lateral jerk score and mean distance of every checkpoint of each class of models. A lower jerk score corresponds to a more comfortable driving policy. Each point represents a checkpoint that was evaluated, with one checkpoint every 100 000 steps.

4.1.3 Discussion

Implementation A

The results from implementation A using a visual encoder trained from scratch show that performance varies wildly between runs. The training also appears to be unstable with performance drops happening often. Further demonstrating instability the second run shows the policy collapsing after 7 million steps and being unable to recover. Using checkpoints and evaluating every checkpoint allows us to be able to select good policies despite the unstable training. Both runs outperformed the ML-Agents reference model's performance of 5020 meters mean distance. However, the reference model still had a higher success rate of 90 %. It should be noted that the ML-Agents model had some disadvantages, such as only training for 5 million steps instead of 15 million and only evaluating the final model instead of every checkpoint. This means that the comparison is not entirely fair.

The difference between the runs highlights the issue in Reinforcement Learning that performance can vary wildly between runs due to the stochastic nature of the training process and randomization of initial parameters. Ideally, multiple runs should have been done with each experiment and set of hyperparameters in order to present an average of runs. Although the IDUN cluster at NTNU provided access to plenty of computational resources, the Unity environment had to run on hardware supporting graphics in order to provide visual observations. The availability of this type of hardware was more limited, and therefore only a single run was done for each experiment.

A run of implementation A with a buffer size of 64 000 instead of 32 000 was done which showed noticeably improved results. These results are not included here since we were unable to reproduce them and it was possibly just the results of a "lucky" run. The result of this run can be seen in the appendix.

Swapping the visual encoder with the encoder from a Variational Autoencoder caused significant improvements in the results. The model trained faster, was more stable and achieved longer mean distances. It outperformed the reference ML-Agents model both in terms of mean distance and success rate after fewer than 2 million training steps.

Implementation B

Implementation B with a visual encoder trained from scratch achieved a worse performance than implementation A by every presented metric. The mean distance is worse, the success rate is worse, the sample efficiency was lower, the jerk score was higher and the training was more unstable.

The experiments using pre-trained encoders showed much more intriguing results. The training is still unstable, but using checkpoints allows us to select the best policies. $z_{dim} = 16$ and $z_{dim} = 32$ appears to outperform early in the training while $z_{dim} = 64$ and $z_{dim} = 128$ takes the lead at later stages. One explanation for this that could explain training being easier in the beginning is that there is less information to work with meaning less noise. The lower noise makes it easier for the model to learn the important correlations. The models using a higher z_{dim} might be able to squeeze out extra performance later in the training process by exploiting the extra information available to it that is lost in the lower dimensional encoders. Compared to the VAE experiment in implementation A, the mean distance was lower, but the success rate is consistently higher, especially with the 64-dimensional encoder. The scatterplot in figure 4.7 shows $z_{dim} = 64$ significantly outperforming the other values of z_{dim}.

It is hard to know exactly why the 64-dimensional autoencoder performs so well. Since only one experiment was done, we cannot exclude the possibility that it was just a lucky run. The quality of the feature vector encoding learned by the VAE itself is also a variable that could be subject to "luck" and vary between runs, as its initialization and training process is also stochastic. This means that even if an average of multiple experiments suggested that $z_{dim} = 64$ was optimal, it would be hard to know if it was because 64 actually is the optimal z_{dim}, or if the VAE itself had learned an exceptionally good feature representation due to chance. Performing experiments to gain a deeper understanding of these issues would be too time-consuming given the resources at our disposal and are left for future work.

Model comparison

Of the four classes of models tested, all but the class using implementation B without a VAE have some advantages over the others. Implementation A with VAE seen at 2:00 achieves the highest mean distance of all models and is more comfortable than both models using implementation B. It is the most aggressive driving policy and the upper quartile distance is significantly higher than all other models throughout the entirety of the training process. It is also the fastest learner of all the models. A disadvantage is that it has a lower success rate presumably because of its aggressive driving. Implementation A without a VAE results in the most comfortable driving policies while still achieving a reasonable mean distance and success rate. The driving policy is more comfortable early in the training process and an early policy can be seen at 0:00 in the video. Implementation B with a VAE achieves a mean distance that is almost as good as the best model in that regard, but at a near 100 % success rate. The downside is that it is about as uncomfortable as a driving policy can possibly get with a very erratic steering behavior. The policy sometimes alternates between steering strongly to the left and strongly to the right every timestep. This behavior can be seen at 6:12 in the video.

Variational Autoencoder reconstruction

A policy from implementation B without VAE was chosen for the VAE reconstruction comparison video due to it being the worst class of policies. A checkpoint with a good mean distance but also a high failure rate was selected to increase the frequency of near failure and failure states. These states, such as the one seen in the video at 8:35 show that the VAE reconstruction breaks down when images that significantly differ from those seen during training are encountered. Otherwise, the VAEs do a good job at recovering the image even with as little as an 8-dimensional z_{dim}.

4.2 Experiment 2: Lane following in Carla

Experiment 2 is done in the Carla simulator where a lane following task is implemented as described in section 3.3. This experiment aims the deploy the best performing models from experiment 1 in the higher fidelity environment that is Carla. Simulation in Carla is more time-consuming, which is why fewer runs are done in Carla. The model with a 64-dimensional Variational Autoencoder using implementation B was chosen to be used in Carla.

4.2.1 Setup

Final models in this experiment were trained on the desktop PC with the Nvidia GTX 1070 GPU. Some initial experiments not shown here were done in the remote desktop machine described in section 4.1.1. The Variational Autoencoders were trained on the Idun GPU cluster as in the previous experiment.

The car is spawned in a random spawnpoint from a selection of the included spawnpoints. Autopilot is then enabled for a random amount of timesteps between 20 and 30 as the priming phase. This ensures the car doesn't always start in the same locations and that the car starts off with some forward speed. The car is not primed in evaluation mode.

Models are still trained for up to a maximum of 15 million steps which is also equivalent to about 415 hours of simulated time since the timestep duration is the same as in the Unity environment. A checkpoint is saved every 50 000 steps, but only the first checkpoint in each 100 000 step interval is evaluated to stay consistent with previous experiments. Each timesteps in the simulator represent 100 milliseconds of simulated time. Evaluation consists of collecting 100 episodes using the policy of each checkpoint, choosing the mean action instead of sampling, and collecting data such as the distance of each episode and number of successful episodes. A comfort evaluation is also performed that computes a lateral jerk score by computing the mean absolute lateral jerk over 10 episodes with the same method as the previous experiment.

Action space The action output was changed to predict a target speed instead of a throttle signal. This was after initial experiments using a throttle signal failed to yield satisfactory results. Another design choice with this failed model was to disable brakes since they made learning difficult. The resulting policy learned a low mean and a high standard deviation for the throttle action. This meant that network would output throttle values close to either -1 or 1 after the tanh activation with the values below 0 has no effect since the brakes were disabled. This meant it could drive in training due to the sampled action sometimes resulting in a high throttle value, but in testing the car would stand still. The car would even roll backwards if it started in a hill since it has no information in its observation that tells it it's on an incline, and gravity outweighed the throttle signal.

Variational Autoencoder

As in experiment 1, a dataset for the Variational Autoencoder was collected by training a model without a VAE and saving images during the training process. The autopilot was enabled for a longer period of time before each episode started during the data collection period to prevent the majority of the images from being from the same spawnpoint areas. Specifically, the number of autopilot steps was randomly chosen between 25 and 125 instead of 20 and 30. This was especially important as the models without a VAE did not perform well at all in Carla.

A dataset of more than 100 000 images were collected, and VAEs of several different latent space dimensionalities z_{dim} were trained, but a 64-dimensional latent space was used for experiments since it was the best performing one in the Unity experiment. Increasing the dimensionality further did not meaningfully reduce the training loss of the VAE, nor did it appear to improve the reconstructions on the ten test images.

Testing in a mirrored world

The Carla environment does not have procedurally generated roads, and the selection of roads that are fitting for the desired lane following environment is limited. This makes models trained here prone to overfitting. In order to test the trained models on unseen roads, a mirror world environment is created. When mirror world mode is enabled, the visual observation is flipped horizontally and the car's start position is moved over to the left lane so that it appears to be in the right lane in the mirrored image. The steering commands are then modified by flipping the sign before sending it to Carla.

Testing without episode termination

The car agent will also be tested in a modified environment where all termination conditions are disabled with the exception of a car crash. This essentially means that the car drives until it crashes since the episode success condition is also disabled. A video clip of the car driving in this mode will be shown. The intent of this test is to see what happens when the car encounters traffic situations it is not trained to handle, such as traffic intersections.

4.2.2 Results

Figure 4.12 shows the evaluation of the model trained with a Variational Autoencoder on the training environment with its success rates, while figure 4.13 shows the evaluation in the mirror world environment. An evaluation in the mirror world environment with *lenient mode* is shown in figure 4.16. The model was trained over the course of just above 100 hours of wall-clock time and over 415 hours of simulator time. Both train and test are seen together in figure 4.14. The average speed of train and test are shown in figure 4.15.

A few attempts were made to train a policy without a Variational Autoencoder using implementation A as this resulted in the most comfortable policies in the previous experiment, but none achieved satisfactory results.

The video shows the policy with the best performance in the test environment in the training environment at 10:00 and in the mirror world environment at 12:00. A later policy that drives much faster can be seen in the training environment at 14:00. Because this policy does not perform well in the mirror world, the video instead shows it in the lenient environment which can be seen at 16:00. A comparison between different choices of the VAE z_{dim} is shown at 18:00. At 19:00 the same comparison is made but with observations from the mirror world which the VAE has never seen during training. The video compilation shows the car launched in the environment with termination conditions disabled in the training and mirror world at timestamps 20:00 and 22:00 respectively.

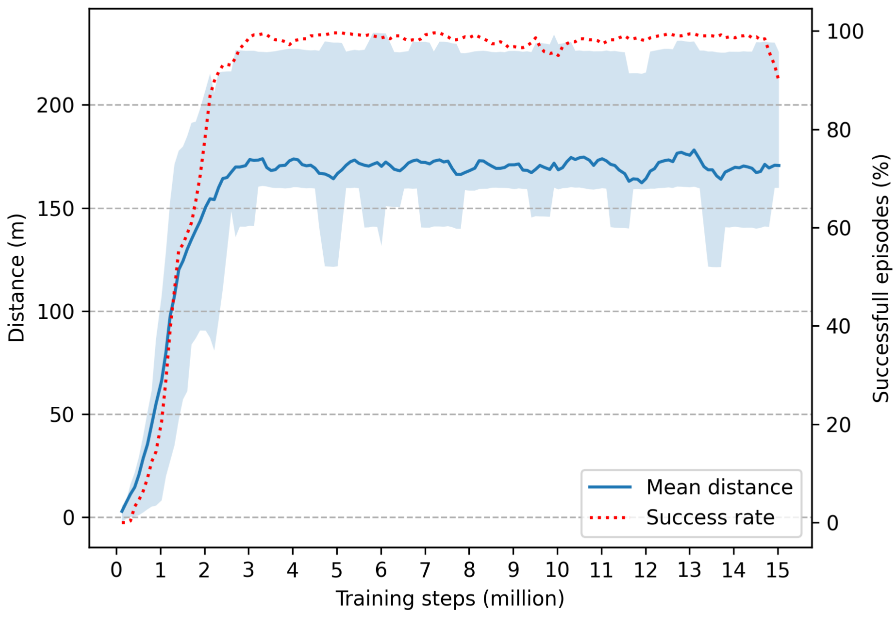


Figure 4.12: Evaluation of the Carla model using a Variational Autoencoder with $z_{dim} = 64$ in the Carla lane following environment. The evaluation is done in the same environment it was trained in. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. A running average of the 5 preceding values is used to smooth out the graph.

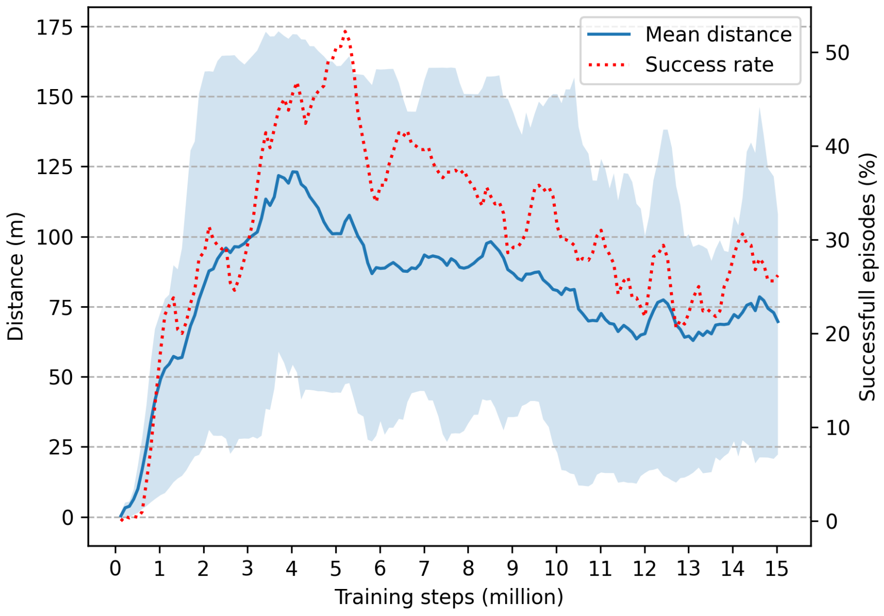


Figure 4.13: Evaluation of the Carla model using a Variational Autoencoder with $z_{dim} = 64$ in the Carla lane following environment. The evaluation is done in the mirror world environment. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. A running average of the 5 preceding values is used to smooth out the graph.

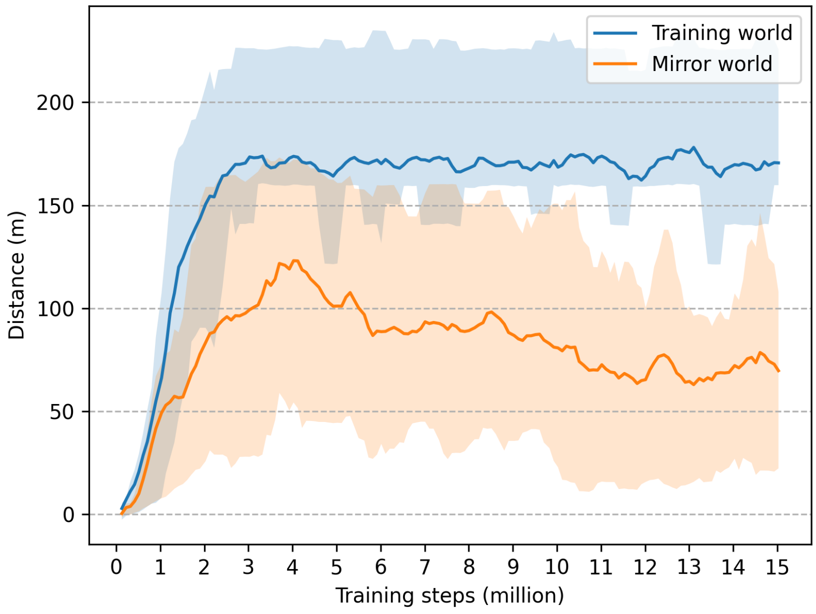


Figure 4.14: Evaluation of the Carla model using a Variational Autoencoder with $z_{dim} = 64$ in the Carla lane following environment. The plot shows both the evaluation in the training world and in the mirror world. The lines show the mean distance traveled by the agent, and the shaded areas show the interquartile range of distances traveled. A running average of the 5 preceding values is used to smooth out the graph.

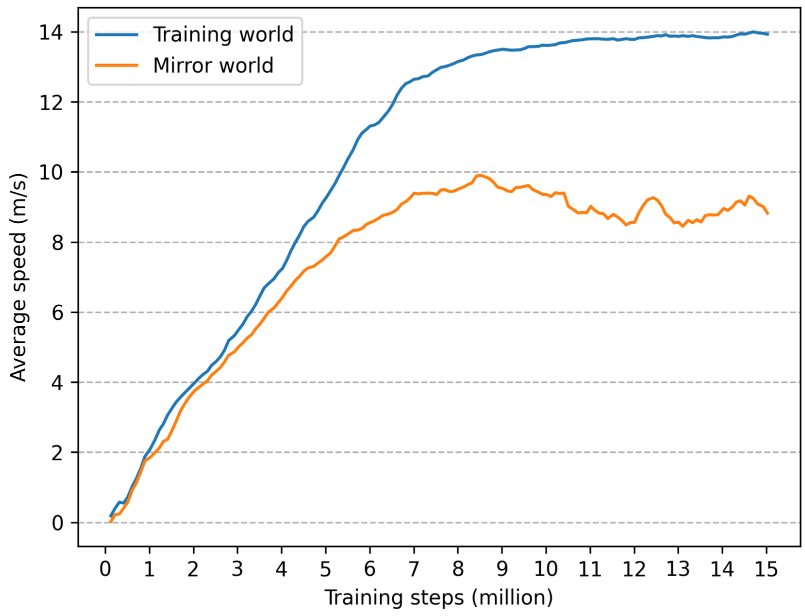


Figure 4.15: The average speed of the Carla model in the training environment and mirror world environment. The graph shows that the average speed keeps increasing throughout the training process, which means the model performance is increasing even if the mean distance plateaued. The model uses a Variational Autoencoder with $z_{dim} = 64$. A running average of the 5 preceding values is used to smooth out the graph.

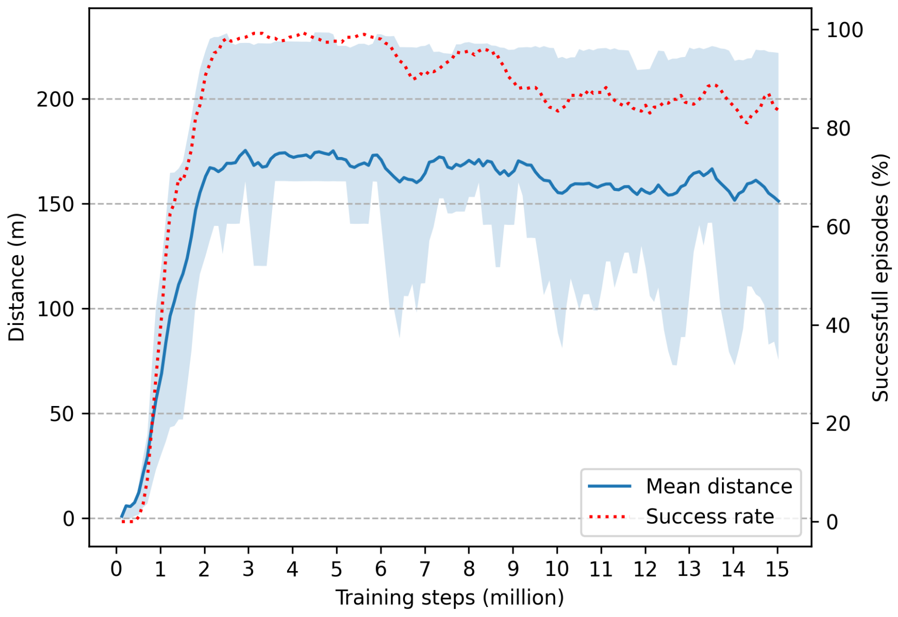


Figure 4.16: Evaluation of the Carla model with Variational Autoencoder with $z_{dim} = 64$ in the Carla lane following environment in *lenient mode*. It is the same environment and model as in figure 4.13 but with a more lenient fail condition. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. The red dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

4.2.3 Discussion

The model achieves near 100 % success rate in the training environment after about 3 million training steps. The mean distance plateaus after this since, unlike the Unity environment, the roads have a finite length. When the model reaches 100 % success rate it must also have the maximum possible mean distance. The interquartile range of distance loses its meaning after the plateau since it becomes entirely determined by the lengths of the different predetermined routes in the Carla lane following environment. The different interquartile ranges at different checkpoints are just the result of which routes were randomly selected when evaluating that checkpoint. The performance on test increases until about 5 million steps where the success rate peaks at 50 %. The performance degrades after this point which shows the model overfitting to the training environment. The figure showing the average speed shows an increasing trend for the training environment throughout the training process, albeit with diminishing returns. This shows that the policy performance keeps increasing with respect to discounted reward, as completing the lap faster gives a higher discounted reward. The average speed being lower in the test environment is due to the higher failure rate causing the car to spend more time at low speeds, such as at the beginning of episodes. When the model is evaluated in the lenient mirror world, the performance is much better and doesn't seem to suffer as much from overfitting. The video shows that the car often touches the lane markings which would fail the episode in the stricter mode, but continues on to complete the episode successfully. As an example, the episode starting at 16:00 shows the agent drive over the outer lane markings which would have failed the episode in the strict training environment, but it does not drive off the road and does successfully complete the episode. It would be interesting to see if a model trained in lenient mode would generalize as well in the mirror world as the one trained in strict mode. Perhaps training in a stricter environment than the target environment can improve performance in the target environment. This was not attempted due to time constraints.

The no-termination video show that the car sometimes can get through intersections even though it was not trained to, such as at 20:40. The intersection is more difficult if it is not straight, as seen at 20:18 where the car crashes into the guardrail.

The VAE reconstruction video shows that the quality of the reconstruction increases with higher z_{dim}, which is expected. Although it was not measured quantitatively, the video seems to show that VAE reconstructions of the images from the mirror world are worse. This is also expected behavior, seeing as the VAE was only trained on images from the non-mirrored world. Another interesting behavior of the VAE is seen at 21:02 where the cornfields which are unseen during training are reconstructed as forest.

Deploying the well-performing model from Unity in Carla was not straightforward. Many iterations and numerous tweaks had to be made to the model before any meaningful results were achieved. All attempts at training a visual encoder from scratch also failed. Training often showed promising progress until as much as 12 hours into training or 2 million steps before performance peaked and degraded from thereon. This made model iterations slow and highlighted the importance of testing in a computationally cheaper environment to get a good idea of where to begin in the higher fidelity environment. All models with visual encoders trained from scratch failed to achieve good performance. Switching the visual encoder trained from scratch from 2 convolutional layers to 4 layers with the same shape as the VAE encoder resulted in improved performance, but it was not enough to converge to a good policy. It might be helpful to use a visual encoder with more layers such as ResNet, especially in the Carla environment. We did not attempt to use ResNet due to it having many more trainable parameters making it harder to train.

4.3 Experiment 3: Unity to Carla policy transfer

The goal of experiment 3 is to train a model in the Unity environment and transfer the learned policy to the Carla environment. This is done by training the model on a semantic segmentation image instead of a grayscale or color image. This semantic segmentation image looks similar in both environments closing the visual "simulator gap".

4.3.1 Setup

The sky and vegetation channels were merged into a single channel in order to make both environments look more similar. The resulting semantic segmentation image consisted of the four classes $\{Road, RoadLines, Vehicle, Other\}$. The model was trained on the remote desktop machine. Evaluation of the model in the Unity lane environment was also done on the remote desktop machine. The Nvidia GTX 1070 desktop was used when evaluating the model in Carla. Both machines are described in more detail in section 4.1.1.

The strict Unity lane environment described in section 3.4.3 is used when training the model. This stricter environment aims to better mimic the Carla environment by having a much lower episode termination threshold. This environment also uses a target speed action instead of a throttle signal action, as this is what ended up being used in the Carla experiment. The vehicle physics in the Unity environment was altered by making the wheels use Ackermann steering geometry which makes each wheel trace out circles of different radii when turning. This is a better and more realistic steering behavior compared to rotating each wheel by the same angle as was done earlier. This makes the steering behave more like the steering in Carla.

The lenient Carla lane following environment was used when evaluating in Carla. This was chosen because the agent following the transferred policy struggled to complete even a single episode under the strict lane invasion sensor. The lenient environment allows the agent to successfully complete episodes while still mostly staying in its lane.

The model is trained for almost 9 million steps which is about 250 hours of simulator time. A checkpoint is saved every 100 000 steps and evaluated in the Unity environment and both the mirrored and non-mirrored Carla lane environment.

4.3.2 Results

The model is trained for 8.8 million steps over the course of 16 hours of wall-clock time. The model evaluated in the Unity environment where it was trained is shown in figure 4.17. The policy is then evaluated in the Carla lane following environment which can be seen in figure 4.18. Figure 4.19 shows the same evaluation in the mirrored Carla environment. Figure 4.20 shows the policy evaluated in both Unity and Carla with the distance normalized.

The car driving in the Unity environment where it was trained can be seen at 24:00 in the video. The same policy can then be seen driving in Carla at 26:00. The agent then drives with no termination in the Carla normal world at 28:00 and in the mirror world at 30:00.

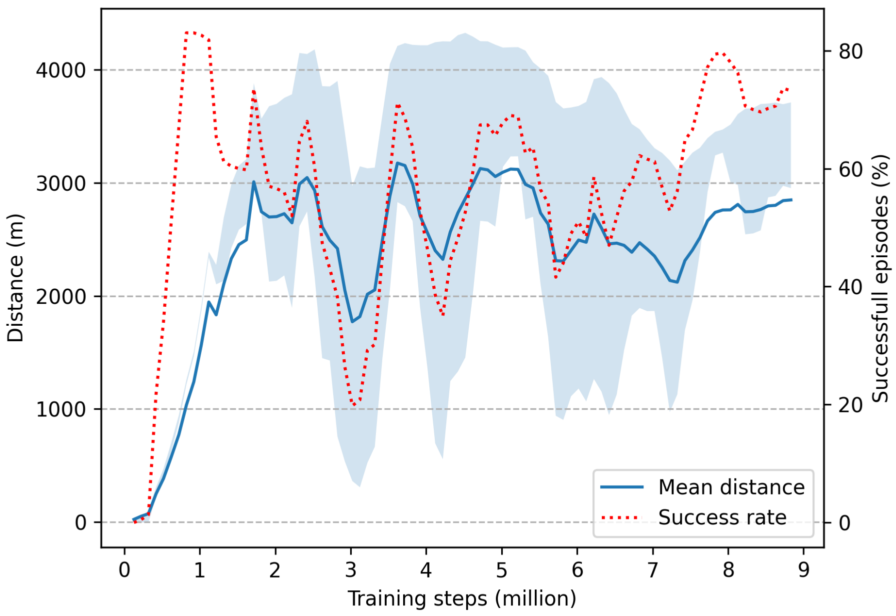


Figure 4.17: Evaluation of the model in the Unity lane following training environment with semantic segmentation map observations. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. The red dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

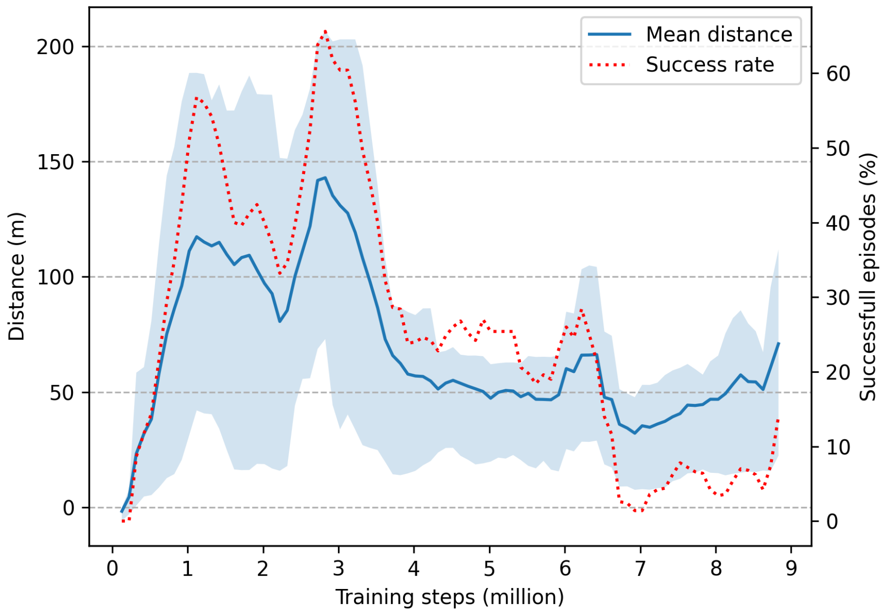


Figure 4.18: Evaluation of the model transferred to the Carla lane following environment with semantic segmentation map observations. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. The red dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

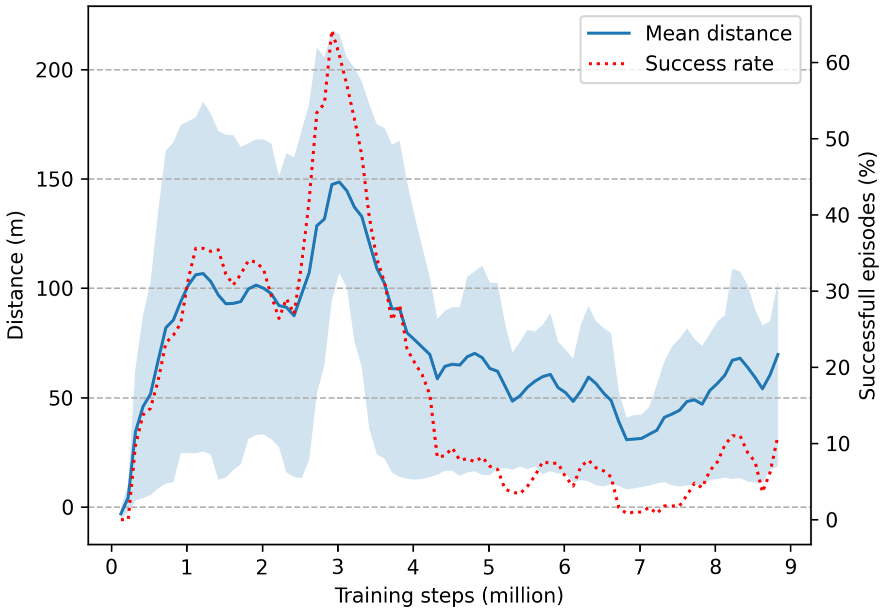


Figure 4.19: Evaluation of the model transferred to the Carla lane following environment with semantic segmentation map observations. This figure shows the evaluation in the mirror world Carla environment. The line shows the mean distance traveled by the agent, and the shaded area shows the interquartile range of distances traveled. The red dotted line shows the episode success rate. A running average of the 5 preceding values is used to smooth out the graph.

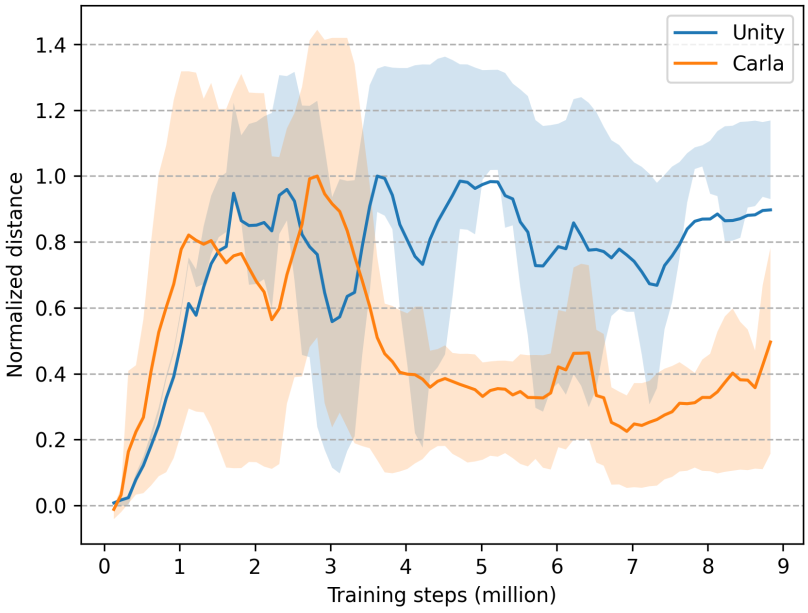


Figure 4.20: A comparison of how the model performed in Unity lane environment and in the Carla lane environment. The mean distance is normalized to fit in the same scale.

4.3.3 Discussion

The model's performance evaluated in Unity is lower than that of implementation A without VAE from experiment 1 which it is based on. This is unsurprising since the environment has stricter fail conditions and is therefore a more difficult environment. Another difference is that the environment in experiment 1 used a three-dimensional velocity vector in the vector observation while this environment uses the velocity magnitude.

The performance evaluations from Carla show that performance is mostly the same regardless of whether the world is mirrored or not. There are some differences such as the peak in mean distance and success rate right after 1 million steps that is present in the regular world but not in the mirror world. Although each checkpoint is evaluated for 100 episodes, there are only ten different routes defined in the Carla lane following environment. If a certain policy fails a particular route one time, it is likely that it will fail it every time since the policy and environment is almost deterministic. The policy is deterministic in evaluation mode, but the environment can introduce uncertainty such as when animated vegetation overlaps the road on slightly different pixels in the image depending on the elapsed simulator time. This low number of routes can cause significant differences in success rate and mean distance, even if only one or two routes are slightly harder. There is some overlap between routes so it can even be a single difficult turn causing failures for multiple routes.

The success rate of over 60 % in Carla shows that the model trained only in Unity can transfer reasonably well to Carla. The Carla and Unity environments are vastly different in terms of visuals and in terms of vehicle physics. The semantic segmentation map used here in place of the image observations addresses the visual gap between the environments, but the physics gap is still unaddressed. Measurements found the turning radius when turning maximally to be 4.4 meters in Unity and 6.4 meters in Carla. Informal attempts to close this gap by decreasing the turning radius in Carla were unsuccessful in improving performance, so the results presented here are with the original 6.4-meter turning radius. The visual observations were also a bit different since the style of road lines was different, and Carla also has road lines in the edges of the road instead of just the center. This is likely the cause of confusion at 31:40 where the car drives off the road but keeps the outer road lines to its left as if they were the center road lines. Another difference is that the Carla environment has some turns that are sharper than what was encountered during training. The video shows the agent failing a sharp turn at 31:20 which might be because it hasn't encountered such a sharp turn during training. Another difference that might cause trouble is the fact that Carla has inclines while the Unity environment only generated perfectly flat roads.

The performance decrease in Carla as training goes on is likely the result of overfitting which was also seen in experiment 2 when mirror world performance decreased towards the end of the training.

Chapter 5

Discussion

In this section, the findings from the three experiments are compiled together and discussed. The discussion starts with an overall discussion about the experiments' results. Next, the research questions are listed, with a discussion centered around each question. The experiment results are then compared to findings from related work. Finally, a reflection is done to identify things that could have been done better in this thesis.

5.1 Discussion

Setting up a simple Reinforcement Learning environment in Unity: ML-Agents was relatively easy and quick to do. Using the built-in trainer made testing different observation setups easy as the trainer automatically creates the neural network architecture based on the attached sensors. This also provided a useful baseline result to compare our custom implementation against.

Experimenting with different implementation details such as the neural network architecture, code level optimizations, and hyperparameters was important as it had a huge impact on policy performance as discussed in earlier literature such as Andrychowicz et al. [2020]. It is therefore important to be able to test different architectures rapidly. The use of a simple and computationally cheap environment that still stayed somewhat faithful to the target environment, allowed more iterations of design decisions and hyperparameters to be made. This let us make more informed decisions about these choices when deploying the model in the Carla environment. While a few changes were required to get good performance in Carla, we believe the lessons learned from the Unity experiments made the process easier.

In our results, none of the final policy checkpoints had the best performance, and the performance sometimes dropped significantly during the training process. This seemed to happen more often in the Unity environment. The performance drops have a simple explanation: the agent takes on more risk in an attempt to get a higher reward. Increasing the speed of the car means it can collect more reward, but it becomes harder to steer the car and stay on the road. An example of a scenario that can cause a destructive policy update is if the replay buffer only contains successful episodes at the time of the update. The agent will then see that every time it sampled a higher throttle than usual, it received more reward than expected (advantage was positive). The policy is then updated to output a higher throttle which then results in the car driving too fast and ends up driving off the road.

This instability justifies our decision to save policy checkpoints for evaluation often. Evaluating all the policy checkpoints used more computational resources than the actual training of the policy in some cases, but it allowed us to gain a deeper understanding of the training process and to select the best policies. This method of selecting policies was even more important in cases where the target environment was different from the training environment. This could be seen when evaluating in mirror world in experiment 2 and in Carla in experiment 3. In both cases, the best-performing policy checkpoint was from the first half of the training process after which the policy started to overfit to the training environment.

The Carla experiment demonstrates that modeling in a simpler environment is beneficial. Training models in Carla is very time-consuming, but fewer training attempts are required since the same setup used in the Unity environment can be used in Carla with relatively few changes.

5.1.1 Evaluating the Research Questions

Research question 1 How does using a pre-trained Variational Autoencoder to encode visual features influence both the training process and the resulting policy?

Using a pre-trained Variational Autoencoder leads to better policy performance in every tested scenario. Using a VAE seemed to increase sample efficiency, especially in implementation A, and it was essential in enabling the learning of a meaningful driving policy at all in Carla. The Carla mirror world experiment showed that the VAE works on unseen images from the mirror world and the agent can still drive well. The video also showed the agent driving through cornfields which the VAE encoded as forest. Encoding images using a Variational Autoencoder also prevented the unrecoverable policy collapse phenomenon that was observed in every experiment where the visual encoder was trained from scratch. The policy collapse might be because the non-VAE networks need to backpropagate the gradient through more layers. A higher number of layers can be problematic because no measures were taken to combat vanishing/exploding gradients. The VAE combats vanishing/exploding gradients by using a batch normalization layer after each convolutional layer.

In the Unity environment using a VAE leads to more uncomfortable policies through more noisy actions, although it might be possible to counteract this by penalizing jerk in the reward function as done in Zhu et al. [2020].

Research question 2 How can a low fidelity simulator be used to accelerate the process of building and deploying a reinforcement learning based autonomous vehicle in a more realistic environment?

Modeling in a low fidelity simulator allowed us to more quickly build and tune a reinforcement learning based autonomous vehicle agent. The system was then deployed in the high fidelity simulator Carla with some minor modifications to the system. Testing an architecture in Unity allowed us to see if a model would perform well in a couple of hours tops. Testing for the same number of steps in Carla would take at least six times as long using our setup.

Research question 3 To what extent can a driving policy learned in a low fidelity environment be deployed and drive successfully in an unseen high fidelity environment?

Our policy transfer experiments showed that a driving policy learned in a low fidelity environment could be transferred to a higher fidelity environment. We are happy with an episode success rate of 60 % considering there is still a slight visual difference in the segmentation maps and a significant physics difference between the environments. We believe the success rate can be further increased by improving the Unity environment by adding inclines and outer road markings to close the visual gap further.

Achieving such results despite the significant physics gap, including a significant difference in turning radius and tire traction at higher speeds, make us optimistic about the potential of policy transfer. The difference in turning radius/steering strength doesn't seem to matter that much. We believe that any under- or oversteering will just be corrected at later timesteps keeping the car in the lane. This might however cause a more erratic steering behavior. It would be interesting to investigate how tolerant driving policies are to steering signal strength. An interesting experiment for future work could be to add a steering strength multiplier to the environment and analyze how different steering strength multipliers affect the driving behavior.

5.1.2 Comparison to Related Work

Our findings are consistent with the findings of Kendall et al. [2018] that Variational Autoencoders significantly improve the training process. We find this to have a much bigger impact on Carla compared to Unity due to its higher graphical fidelity. It makes sense that a more varied distribution of images is harder to learn, which explained why VAEs are more important when dealing with photorealistic graphics.

The policy transfer experiment leveraging semantic segmentation maps uses a similar approach as Mûller et al. [2018]. We demonstrated that this approach also works for sim-to-sim policy transfer. The authors used semantic segmentation maps predicted by a CNN in both the simulator and in the real world. They did this despite ground truth segmentation maps being available in the simulator. The reason behind this was to show imperfect segmentation maps during training as this made it robust to noise in the real world. We used ground truth segmentation maps in our experiments since we had access to them in both simulators.

5.1.3 Reflection

A disproportionate amount of time was spent testing models in the Unity lane environment. This meant less time was available to run experiments in Carla. Ideally more time should have been spent running Carla experiments in order to have more than one successful training run. Obtaining a successful policy with a visual encoder trained from scratch would provide more insight into how a Variational Autoencoder influences the learned policy. Then we could compare the generalization ability measured by mirror world performance with and without a VAE.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have demonstrated a Deep Reinforcement Learning approach to driving autonomous cars using only 80x60 images, speed and previous action features as inputs to a neural network. Our results show that using a simple simulator is helpful in designing and prototyping reinforcement learning based autonomous vehicle systems. These design choices can then be reused when creating an autonomous vehicle system in the more complex simulator. We show that a simple simulator can even be used as a training environment for sim-to-sim policy transfer. While also highlighting potential downsides, we confirm findings from previous work that Variational Autoencoders speed up the training process.

Using Deep Reinforcement Learning to create autonomous vehicles in an end-to-end manner is an exciting field of research that we believe is still in its infancy. The recent release of photorealistic driving simulators has stimulated autonomous driving research. Even more photorealistic simulators such as Nvidia DRIVE, which uses real-time raytracing to create highly photorealistic visuals, are likely to be released soon. This means that while safely deploying fully autonomous (level 5) vehicles that can drive anywhere in the real world is still quite far from reality, the development of these systems is becoming easier every year as technology advances and more tools are developed.

6.2 Future Work

6.2.1 Policy transfer with RL-CycleGAN

We used semantic segmentation maps to close the visual gap between the two simulators. When working with simulators, one has the privilege of having access to perfect ground truth semantic segmentation maps. This is not the case in real life. CNN-based approaches are used to generate these semantic segmentation maps from RGB images, but they are not perfect and relying on them introduces an additional possible point of failure. Another way to close the visual gap is to use RL-CycleGAN which converts visual images from one domain to another in an RL-aware manner [Rao et al., 2020]. The paper demonstrates the method on robotics grasping tasks, so it would be interesting to see if the same method could work in an autonomous vehicle setting. The method could be tested between the two simulators and compared against the semantic segmentation approach. If the results are promising the next step would to launch a model trained in a simulator with RL-CycleGAN in the real world.

6.2.2 Variational Autoencoder's effects on generalization

We were unable to train a well-performing model in Carla without using a Variational Autoencoder. This means we couldn't compare how using a VAE affected generalization ability to the mirror world. Since it was observed that VAE reconstructions were worse in the mirror world environment, it would be interesting to investigate whether using a VAE to encode visual observations can lead to worse generalization ability due to for instance the VAE overfitting to images from the training environment.

Bibliography

Almahairi, A., Rajeswar, S., Sordoni, A., Bachman, P., and Courville, A. (2018). Augmented cyclegan: Learning many-to-many mappings from unpaired data.

Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., and Bachem, O. (2020). What matters in on-policy reinforcement learning? a large-scale empirical study.

Cobbe, K., Christopher, Hilton, J., and Schulman, J. (2019). Leveraging procedural generation to benchmark reinforcement learning.

Cobbe, K., Hilton, J., Klimov, O., and Schulman, J. (2020). Phasic policy gradient.

Codevilla, F., Santana, E., López, A. M., and Gaidon, A. (2019). Exploring the limitations of behavior cloning for autonomous driving.

de Haan, P., Jayaraman, D., and Levine, S. (2019). Causal confusion in imitation learning.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). Carla: An open urban driving simulator.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., and Madry, A. (2020). Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*.

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 Welling, M. (2013). Auto-encoding variational bayes.

Liang, X., Wang, T., Yang, L., and Xing, E. (2018). Cirl: Controllable imitative reinforcement learning for vision-based self-driving.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human level control through deep reinforcement learning. *Nature*, 518:529–533.

Mohanty, S., Poonganam, J., Gaidon, A., Kolobov, A., Wulfe, B., Chakraborty, D., Šemetulskis, G., Schapke, J., Kubilius, J., Pašukonis, J., Klimas, L., Hausknecht, M., MacAlpine, P., Tran, Q. N., Tumiel, T., Tang, X., Chen, X., Hesse, C., Hilton, J., Guss, W. H., Genc, S., Schulman, J., and Cobbe, K. (2021). Measuring sample efficiency and generalization in reinforcement learning benchmarks: Neurips 2020 procgen benchmark.

Mûller, M., Dosovitskiy, A., Ghanem, B., and Koltun, V. (2018). Driving policy transfer via modularity and abstraction.

Rao, K., Harris, C., Irpan, A., Levine, S., Ibarz, J., and Khansari, M. (2020). Rl-cyclegan: Reinforcement learning aware simulation-to-real.

Rong, G., Shin, B. H., Tabatabaee, H., Lu, Q., Lemke, S., Možeiko, M., Boise, E., Uhm, G., Gerow, M., Mehta, S., Agafonov, E., Kim, T. H., Sterner, E., Ushiroda, K., Reyes, M., Zelenkovsky, D., and Kim, S. (2020). Lgsvl simulator: A high fidelity simulator for autonomous driving.

SAE (2021). Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. SAE International.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. (2019). Mastering atari, go, chess and shogi by planning with a learned model.

Schulman, J., Levine, S., Mortiz, P., Jordan, M., and Abbeel, P. (2015a). Trust region policy optimization.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation.

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

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

Själander, M., Jahre, M., Tufte, G., and Reissmann, N. (2019). EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure.

SSB (2021). Trafikkulykker med personskade. Statistisk Sentralbyrå.

WHO (2020). Road traffic injuries. World Health Organization.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning.

Wymann, B., Dimitrakakis, C., Sumner, A., Espié, E., and Guionneau, C. (2015). Torcs: The open racing car simulator.

Ye, C., Ma, H., Zhang, X., Zhang, K., and You, S. (2017). Survival-oriented reinforcement learning model: An effcient and robust deep reinforcement learning algorithm for autonomous driving problem. pages 417–429.

Zhu, M., Wang, Y., Pu, Z., Hu, J., Wang, X., and Ke, R. (2020). Safe, efficient, and comfortable velocity control based on reinforcement learning for autonomous driving. *Transportation Research Part C: Emerging Technologies*, 117:102662.

Appendices

Appendix A

This appendix shows the result of a couple of training runs in the Unity lane following environment that weren't included in the main part of the thesis. Both training runs use implementation A with a visual encoder trained from scratch.

Figure 6.1 shows a training run where the resolution of the visual observation is increased from 80x60 to 120x90. There is no significant improvement in performance, and the increased resolution caused significantly slower training due to the higher computational cost of processing higher resolution images.

Figure 6.2 shows the results of training with a buffer size of 64 000 instead of 32 000. The performance is significantly higher, but subsequent training runs attempting to reproduce these results failed. This suggests that the results were from a lucky run and we therefore continued to use a buffer size of 32 000 for the remainder of the thesis.

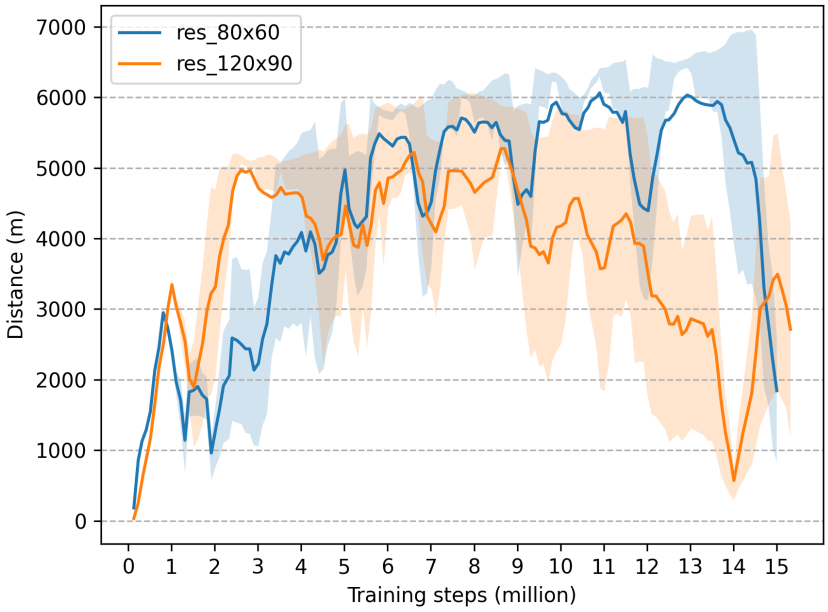


Figure 6.1: Evaluation of implementation A with a visual encoder trained from scratch comparing the results when increasing the image observation resolution to 120x90.

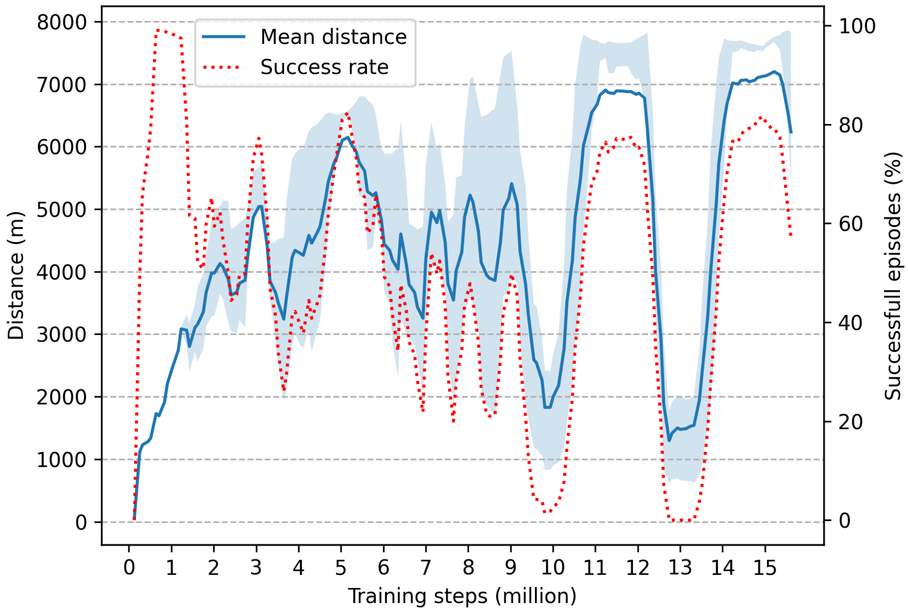


Figure 6.2: A run of implementation A using a visual encoder trained from scratch. A buffer size of 64 000 is used instead of 32 000. These results are from a lucky run that we were unable to reproduce. The best mean distance of 7261 meters happened at timestep 15.2 million with an episode success rate of 80 %.

Isak Grande Bjørnstad

Deep Reinforcement Learning for Autonomous Vehicles in Simulated Environments