CHALMERS
UNIVERSITY OF TECHNOLOGY

Exploring Machine Learning for Autonomous Drive with the Help of Simulators

Using Deep Reinforcement Learning to Investigate the Extent of Learning Objectives for Autonomous Vehicles

Master's thesis in Master Program System, Control and Mechatronics

Mart Waldenstål, Sebastian Svanland

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

MASTER'S THESIS 2023

Exploring Machine Learning for Autonomous Drive with the Help of Simulators

Using Reinforcement Learning to Investigate the Extent of Learning Objectives for Autonomous Vehicles

Mart Waldenstål
Sebastian Svanland

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023

Exploring Machine Learning for Autonomous Drive with the Help of Simulators
Using Reinforcement Learning to Investigate the Extent of Learning Objectives for Autonomous Vehicles
Mart Waldenstål
Sebastian Svanland

© Mart Waldenstål, 2023.
© Sebastian Svanland, 2023.

Supervisor: Hamid Ebadi, Infotiv AB
Advisor: Jafar Banar, Department of Electrical Engineering
Examiner: Giuseppe Durisi, Department of Electrical Engineering

Master's Thesis 2023
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2023

Exploring Machine Learning for Automotive Drive with the Help of Simulators
Using Reinforcement Learning to Investigate the Extent of Learning Objectives for Autonomous Vehicles
Mart Waldenstål
Sebastian Svanland
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Autonomous vehicles are a popular topic with much progression each year. Already, there are advanced driver-assistance systems in commercial use to alleviate drivers from monotonous tasks. These systems often require extensive training with gathered data from real-world driving. This thesis aims to explore how simulators and Reinforcement Learning (RL) can be utilized to expand the training process further, as well as investigate their possibilities and limitations. A significant advantage of using simulators in combination with RL is the possibility of exploring the state-space and actions without the expense of real-world incidents. In this thesis, we studied several simulators and RL algorithms. However, we focus mainly on implementing an end-to-end autonomous driving system using the DonkeyCar simulator and the double deep Q-network algorithm. The training is executed with different configurations for the simulator and algorithm. The trained models are then evaluated in two regards, training progression and driving capability. Finally, a discussion followed by proposals for further investigations concludes the work.

Keywords: Autonomous Drive, Deep Learning, Driving Simulators, Reinforcement Learning, CNN, DDQN.

Acknowledgements

First off, we would like to express our sincere gratitude to Infotiv AB and our supervisors Hamid Ebadi and Martin Karsberg for the opportunity to utilize their equipment as well as their guidance throughout the entire project. We also want to thank our academic advisor, Jafar Banar, who helped us with academic questions. Lastly, we are grateful to Giuseppe Durisi, our examiner, for his feedback.

Mart Waldenstål, Gothenburg, June 2022
Sebastian Svanland, Gothenburg, June 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

A2C	Synchronous advantage actor critic
A3C	Asynchronous advantage actor critic
ACC	Adaptive cruise control
AD	Autonomous driving
ADAS	Advanced driver-assistance system
ADS	Autonomous driving system
AI	Artificial intelligence
ALC	Automated lane centering
AV	Autonomous vehicle
CNN	Convolutional neural network
CTE	Cross tracking error
DDPG	Deep deterministic policy gradient
DDT	Dynamic driving task
DDQN	Double deep Q-network
DM	Driver monitoring
DQN	Deep Q-network
DRL	Deep reinforcement learning
FCW	Forward collision warning
GNSS	Global navigation satellite systems
IMU	Inertial measurement unit
LDW	Lane departure warning
MDP	Markov decision process
ML	Machine learning
MPC	Model predictive control
PID	Proportional integral-derivative
ReLU	Rectified linear unit
RL	Reinforcement learning
SAE	Society of automotive engineers
SLAM	Simultaneous localization and mapping
V&V	Verification and validation

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices

k, t	Indices for time step

Sets

A	Set of actions
S	Set of states

Parameters

γ	Discount factor
α	Learning rate

Functions

Q	Action-value function
R	Reward function
T	Transition function
V	Value function
π	Policy

Variables

a_t	Action at time t
r_{k+1}	Reward at time $k+1$
s_t	State at time t

Contents

List of Acronyms viii

Nomenclature xi

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 Purpose . . . 2
- 1.2 Objective . . . 2
- 1.3 Scope . . . 2
- 1.4 Thesis Outline . . . 3

2 Autonomous Drive 5
- 2.1 Existing Systems . . . 7
  - 2.1.1 Openpilot . . . 8
  - 2.1.2 Apollo . . . 8
  - 2.1.3 Autoware . . . 9
- 2.2 Components in AD systems . . . 9
  - 2.2.1 Sensing . . . 10
  - 2.2.2 Computing . . . 10
    - 2.2.2.1 Perception . . . 10
    - 2.2.2.2 Decision . . . 11
    - 2.2.2.3 Planning and Driving policy . . . 12
  - 2.2.3 Actuation . . . 12

3 Machine Learning 13
- 3.1 Markov Decision Process . . . 14
- 3.2 On- and Off-Policy . . . 14
- 3.3 Model-based and Model-free Algorithms . . . 14
- 3.4 Value-based vs Policy-based Algorithms . . . 15
- 3.5 Deep Learning . . . 15
  - 3.5.1 DQN . . . 16
  - 3.5.2 DDQN . . . 17
  - 3.5.3 A3C . . . 18
  - 3.5.4 A2C . . . 19

3.5.5 DDPG 19

4 Simulator **21**
4.1 Criteria 22
4.2 Open-source simulators 22
4.2.1 DonkeyCar 22
4.2.2 CARLA: CAR Learning to Act 23
4.2.3 SVL 24
4.2.4 Environment Simulator Minimalistic (esmini) 25

5 Implementation **27**
5.1 End-to-end setup 27
5.1.1 DonkeyCar Simulator 28
5.1.2 Image Processing 30
5.1.3 CNN 31
5.1.3.1 Network 1 31
5.1.3.2 Network 2 31
5.2 Training 32
5.2.1 Reinforcement Learning algorithm 33
5.2.2 Reward function 33
5.3 Verification and Validation 34

6 Results **35**
6.1 Network architecture and batch size 35
6.2 Learning rate and episodes 37
6.3 Multiple tracks 39
6.4 Stricter restart condition (max CTE) 41
6.5 Visualization of convolutional filter outputs 42

7 Conclusion **45**
7.1 Future work 46

A Appendix **V**

List of Figures

2.1 Autoware ADS architecture overview 10

3.1 Structure of actor critic algorithms. 19

4.1 DonkeyCar simulator. 23
4.2 Example screenshots from CARLA simulator. 24
4.3 Example of scenario simulations in SVL. 25
4.4 Two examples of scenario simulations in esmini. 25

5.1 Target structure of the implemented system. 28
5.2 The tracks from the DonkeyCar simulator used in the project. 30
5.3 Image before and after image processing. 30
5.4 Visualization of network 1. 31
5.5 Visualization of network 2. 32

6.1 Training progression of different networks and batch sizes. 36
6.2 Training progression of same configuration. 37
6.3 Training progression of different learning rates and episodes. 39
6.4 Training progression of different networks trained on different tracks. 41
6.5 Comparison with stricter cte 42
6.6 Input image from the camera on the car. 42
6.7 Visual output of network 1. 43
6.8 Visual output of network 2. 44

A.1 Flowchart diagram of parameter and architecture of Network 2 VI
A.2 Flowchart diagram of parameter and architecture of Network 1 VII

List of Tables

6.1 Standard hyperparameters. 35
6.2 Average accumulated rewards of models during test 1. 36
6.3 Accumulated reward of same configuration over two different training sessions. 37
6.4 Average accumulated rewards of models during test 2. 38
6.5 Average accumulated rewards of models during test 3. 40
6.6 Average accumulated rewards of models trained with different max CTE. 41

1 Introduction

Autonomous Drive (AD) is a topic that gets a great deal of attention and has a high potential to increase safety and efficiency in the transportation and mobility industry. The major actors in the mobility industry are commercially using systems today referred to as Advanced Driver-Assistance Systems (ADAS). These are a central part of the next step toward deploying fully Autonomous Driving Systems (ADSs). Tools to train and test these safety-critical systems need further research to make the transition possible.

The history of AD goes back to 1980, when Ernst Dickmanns developed one of the first autonomous cars [1], and the project is said to have made way for the research into AD. Other early examples are the EUREKA Prometheus Project, an EU-financed project from 1987 to 1995 that aimed to develop a fully functional autonomous car, and CMU NAVLAB in 1995, which managed 6,000km of autonomous driving with 98% driven autonomously [2]. Another significant milestone in autonomous driving was the usage of Machine Learning (ML) and Artificial Intelligence (AI). It was used in the DARPA Grand Challenges in 2004 and 2005. The challenge's goal was to develop a driverless car that could navigate an off-road course as fast as possible. The winner of the 2005 race leveraged ML techniques for navigating the unstructured environment, which became a turning point in self-driving car development, acknowledging ML and AI as central components of autonomous driving.

In recent years Reinforcement Learning (RL) has shown the ability to achieve human-like or better results at games such as Go and chess. This shows RL's potential in simulated environments such as computer games [3]. RL is an ML subfield focused on solving Markov Decision Process (MDP) problems, where an agent learns to select actions in an environment attempting to maximize some reward function. Thanks to powerful 3D engines that can create lookalike models of the real world, RL is becoming a common methodology to use during the development of AD. Combined with deep neural networks, such as Convolutional Neural Network (CNN) and Recurrent Neural Network, it has the potential to achieve great results in building efficient and intelligent ADSs [2]. Some characteristics that make deep learning methods particularly well suited for ADS are detecting and recognizing objects in 2D images and 3D point clouds acquired from video cameras and LiDAR.

Many ADSs tend to have complicated schemes and split up the problem into several parts like environment perception, path planning, and motion control [4]. Thus, some developers have started using a simpler structure, referred to as end-to-end.

End-to-end autonomous driving uses raw sensor data as the input and outputs the low-level control command like steering angle and acceleration. This method avoids designing complex modules by having a straightforward structure. In addition, no loss in perception information or error propagation will occur since all sensor data are directly used for driving, as can happen in other architectures.

Exploring methods to develop agents in simulators is one way of increasing test coverage, training efficiency, and understanding the challenges of AD. A key component of developing an ADS is to shift some parts of the Verification and Validation (V&V) work from the vehicle to a simulation platform. By generating true-to-life replicas of traffic scenarios in simulation environments, ADSs can be safely tested, verified, and validated. Usage of ML components and other non-coded software functions in safety-critical systems is rapidly increasing and creates a need for testing and developing these in an efficient and safe way.

1.1 Purpose

Infotiv is currently building a framework that aims to replicate the automotive industry on a smaller scale. The framework includes tools to model and test AVs. By investigating how ADSs can be trained in a simulation environment, simulators' compatibility with ML algorithms, and simulators' capabilities for testing, this thesis can also contribute to their purpose.

1.2 Objective

The main task of the master's thesis is to investigate what simulator tools and ML methods can be used to train an AD agent. The specific goals are to:

- Decide on a simple traffic scenario for training and testing.
- Choose an ML method.
- Choose a simulator.
- Train an ADS with the help of ML.
- Evaluate the ADS.

1.3 Scope

This project intends to show the first steps of training an ADS and set up an environment with tools needed to develop and test ML agents. Rather than complex systems, the focus is on simplicity in ADS structure, ML, and functionality. The project will therefore only consider implementing an end-to-end ADS using simple algorithms to train such a system. The project also focuses on the learning process rather than solving a specific traffic situation and limits itself to open-source simulators. The trained ADSs should not be considered finished products but instead examples to compare the performance of different training parameters.

1.4 Thesis Outline

This report begins with theoretical background in Chapters 2 *Autonomous Drive* and 3 *Machine Learning*. They are used as a foundation for the decisions made later in the report. In Chapter 2 we present the taxonomy for autonomous cars, three open-source ADSs, and the architecture of an ADS. In Chapter 3, ML is introduced, and successful algorithms in the field are described. Chapter 4 *Simulator* presents the criteria requirements of the simulator and four open-source simulators used in AD research. Next is Chapter 5 *Implementation*, describing the thought process of decisions, as well as how the implementation, training, and tests were performed. Thereafter, Chapter 6 *Result* will compare different training parameters and present the tests' outcome. The report finishes with *Conclusion*, which concludes and discusses the results of the thesis as well as proposes further investigations to elaborate the work.

2

Autonomous Drive

Self-driving cars are autonomous decision-making systems that process streams of observations coming from different onboard sources, such as cameras, radars, LiDARs, ultrasonic sensors, GPS units, and inertial sensors [2]. The car's computers use these observations to make driving decisions. The main goal of an ADS is to create a driverless system that can intelligently navigate a vehicle with full automation [5]. First, an ADS must be able to automatically locate the autonomous driving vehicle and obtain relevant information about the road and surrounding environment. Second, an ADS needs to efficiently process the information received in real-time based on the current environmental condition. Third, an ADS needs to decide how the autonomous driving vehicle should act in the environment. Finally, an ADS sends the movement decision through control signals to maneuver a vehicle.

As the area of AD progresses, the lack of universal consensus on the terminology used within AD has become an issue [6]. Therefore, the Society of Automotive Engineers (SAE) [7] has provided a taxonomy for classifying ADSs, which has become a standard. It consists of six levels ranging from level 0 (fully manual) to level 5 (fully autonomous). These levels have been adopted by the U.S. Department of Transportation and are used in this report when discussing the complexity of existing ADSs. The ADSs are defined at a level based on how the responsibility for the Dynamic Driving Tasks (DDTs) is allocated between the driver and each AD feature. The lower two levels of driving automation (1 and 2) refer to cases where the user continues to perform part of the DDT while the driving automation system is engaged. These are therefore referred to as "driver support" features. The upper three levels of driving automation (3 to 5) refer to cases in which the ADS performs the entire DDT on a sustained basis while engaged. These are therefore referred to as AD features. In the report, SAE presents the taxonomy for autonomous driving as:

Level 0 - No Driving Automation

The driver is fully responsible for driving the vehicle, even if there are active safety systems to support the driver.

Level 1 - Driver Assistance

The AD executes either the lateral (braking, acceleration) or the longitudinal

(steering) vehicle motion control and expects that the driver performs the remainder of the DDT. A Level 1 ADS only supports limited object and event detection, meaning that there are events that the driving automation system is not capable of recognizing or responding to.

- **Example:** ADS features that fits under level 1 are Adaptive Cruise Control (ACC) where the engine and brake power maintain and vary speed and parking assistance where steering is automated while speed is under manual control.

Level 2 - Partial Driving Automation

The ADS executes both the lateral and longitudinal vehicle motion control. As Level 1, a Level 2 ADS is only capable of limited object and event detection, meaning that there are some events that the driving automation system cannot recognize or respond to.

- **Example:** A level 2 ADS can perform multiple features simultaneously, such as Automated Lane Centering (ALC) with simultaneously ACC or parking assistance where both steering and speed is controlled by the ADS.

Level 3 - Conditional Driving Automation

The ADS executes the entire DDT under normal operating conditions. The system expects the user to be ready to ADS-issued requests to intervene to preclude continued vehicle operation and achieve a minimal risk condition. Unlike Level 1 and 2, all Level 3 and 4 AD features are designed to monitor and enforce their operational design domain limitations while engaged. A Level 3 ADS also expects the user to be receptive to handling failures in the vehicle systems that do not necessarily trigger an ADS-issued request to intervene, such as a broken body or suspension component. Although a Level 3 feature may be capable of performing an automated DDT fallback, the system is not expected to achieve a minimal risk condition under certain limited conditions.

- **Example:** An AD feature capable of performing the entire DDT in low-speed, stop-and-go freeway traffic.

Level 4 - High Driving Automation

The AD feature executes the entire DDT and DDT fallback within the feature's operation domain. Thus, the user becomes a passenger and does not need to supervise a Level 4 AD feature while engaged. A Level 4 AD feature

should be capable of automatically performing DDT fallback and achieving a minimal risk condition if the user does not resume control of the DDT when the AD feature's operational domain ends. This automated DDT fallback and minimal risk condition capability is the primary difference between Level 3 and Level 4.

- **Example:** A vehicle with an AD feature responsible for motorway and highspeed freeway conditions will need the user to perform the DDT when the freeway ends and the feature reaches its operational domain limit. However, the feature will automatically perform the DDT fallback and achieve a minimal risk condition if the user fails to take over when the freeway ends (e.g. because she/he is sleeping).
- **Example:** A Level 4 AD feature can perform the entire DDT during valet parking (i.e., curb-to-door or vice versa) without driver supervision.

Level 5 - Full Driving Automation

The ADS executes the entire DDT and DDT fallback unconditionally of the operation domain. It means that the ADS can operate the vehicle on-road anywhere within its region of the world and under all road conditions in which a human driver can reasonably operate a conventional vehicle. In the event of a DDT performance-relevant system failure of the ADS or the vehicle, a Level 5 ADS automatically performs the DDT fallback and achieves a minimal risk condition.

- **Example:** The ADS has no design-based weather, time-of-day, or geographical restrictions on where or when it can operate the vehicle. However, there may be conditions not manageable by a driver in which the ADS would also be unable to complete a given trip (e.g., white-out snow storm, flooded roads, glare ice).
- **Example:** A vehicle with an ADS that, once programmed with a destination, is capable of operating the vehicle throughout complete trips on public roadways, regardless of the starting and end points or intervening road, traffic, and weather conditions.

2.1 Existing Systems

Many companies are working to achieve AD, and car manufacturers such as Volvo, Audi, and Tesla have commercially implemented ADAS features. This includes systems to provide warnings or momentary intervention, such as forward collision warning systems, lane-keeping assistance (LKA) systems, and automatic emergency braking (AEB) systems, as well as some convenience features that involve Level 1 and 2 driver support features, such as adaptive cruise control (ACC) and cer-

tain parking assistance features [7]. Tesla uses a system called AutoPilot which is deployed as an ADAS component, which customers can turn on or off at their convenience. The advantage of Tesla resides in the large training database, consisting of more than 1 billion driven miles. The database has been acquired by collecting data from customer-owned cars [2]. Waymo is another company developing Self-driving cars. They are building their vehicles directly as Level 5 systems and have currently more than 10 million miles driven autonomously. Thus, some companies are working with end-to-end implementations of ADS without a direct association with major car manufacturers but have achieved a high level of AD performance [8] [5] [9] [10]. Examples of such companies or associations are Baidu which develops Apollo, comma.ai which develops Openpilot, and The Autoware Foundation developing Autoware. In a survey performed by Consumer Reports 2020 on 17 existing ADASs used in major car manufacturers' cars and OpenPilot, OpenPilot's ADAS scored the highest rating in the overall results [10].

2.1.1 Openpilot

Comma.ai is an AI start-up founded by George Hotz in September 2015. The company is developing self-driving technology based on ML algorithms. In 2016 Holtz built a working self-driving Acura ILX and launched Openpilot [9]. Openpilot is an open-source, semi-automated driving assistance system that can be bought and installed by anyone who buys the development kit available in comma.ai online shop and flashes the Openpilot software available on the public GitHub repository. Openpilot reaches level 2 on the driving automation levels defined by SAE, meaning that the vehicle on which the device is installed can control steering, accelerating, and decelerating. At a level 2 automation, the human still monitors all the tasks and can take control at any time. Openpilot supports a growing variety of car brands, models, and model years. [11]. Openpilot offers the AD features:

- Adaptive Cruise Control (ACC)
- Automated Lane Centering (ALC)
- Forward Collision Warning (FCW)
- Lane Departure Warning (LDW)
- Driver Monitoring (DM)

2.1.2 Apollo

Baidu is one of the leading companies in the field of ADS. They have been putting great effort into building an open community with their open-source self-driving solution since 2017 [5]. Apollo contains many ML models responsible for various tasks such as traffic light recognition, lane detection, obstacle perception and detection, and trajectory prediction. Apollo also combines information from multiple types of sensors such as cameras, LiDAR, and radar to make predictions of the environment. Apollo release different, updated versions, along with the development of the framework. The first version is Apollo 1.0 released in 2017 and the latest version is Apollo 7.0 released in 2021. Apollo 7.0 has become an advanced system and is said

to have reached Level 4 on the SAE scale. The AD cars using Apollo 7.0 have driven over 12 million kilometers in road tests by June 2021. Baidu has also opened its Apollo Go Robotaxi service to the public in Guangzhou, Changsha, Cangzhou, and Beijing. The previous versions of Apollo are more directed to developers. Apollo 1.0, also referred to as the Automatic GPS Waypoint Following, was made available for only closed track driving [12]. The next version, Apollo 1.5, has features like object detection, decision-planning, cloud simulation, and HD map services, all with support for both day and night scenarios. Apollo 2.0 supports automated driving on urban roads and Apollo 3.0 is focused on providing a platform for developers and researchers to build upon in a closed venue low-speed environment. Using the platform, vehicles can maintain lane and cruise control and avoid collisions with vehicles ahead of them. In Figure 4.3, an example of scenario simulation in SVL for pedestrian detection using Apollo 6.0 is shown [13].

2.1.3 Autoware

The Autoware Foundation is a non-profit organization supporting open-source projects enabling self-driving mobility. [8]. The original version of Autoware was officially released in August 2015 by the research group of Nagoya University directed by Prof. Shinpei Kato but has since 2018 been owned by The Autoware Foundation [14]. It enables self-driving vehicles to be tested in private areas, urban roads, and highways. It provides a complete set of modules for an AD stack, including localization, detection, prediction, planning, and control in a modular architecture with defined interfaces and APIs. Autoware Foundation also works with Autoware.Auto, an ADS for commercial deployment for self-driving vehicles. Autoware uses 3D map and LiDAR sensor data for most of the main functional modules, such as localization, detection, tracking, prediction, and planning. However, using a camera enhances the recognition systems and enables features such as recognizing traffic lights. GNSS and IMU are also useful to complement the result of localization. The system can be used both in real-world cars through the Autoware interface and in simulation environments using Robotic Operating System (ROS) packages, a software framework for robot applications.

2.2 Components in AD systems

The following section will explain the most commonly used modules for an ADS. The literature study in Section 2.1 showed that the above mentioned ADSs use similar architectures to develop their system [2],[12],[8],[15]. Figure 2.1 illustrates the parts of an ADS and demonstrates the pipeline from the sensor stream to the control actuation of the vehicle [16]. ADSs are generally structured into three major modules: sensing, computing, and actuation, although different developers have their own way of dividing and representing the modules.

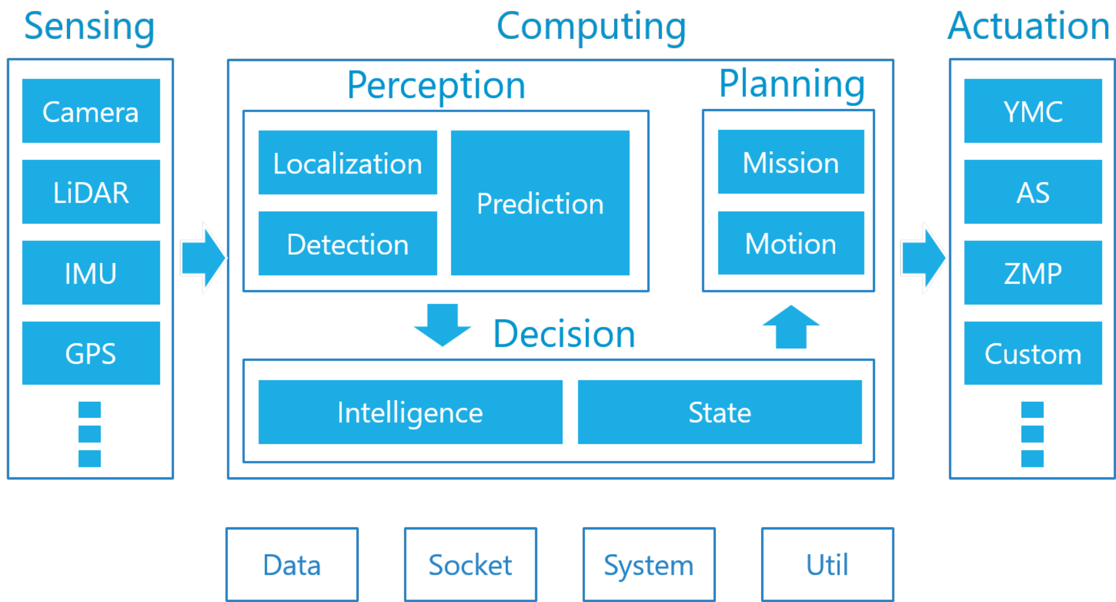


Figure 2.1: Autoware ADS architecture overview [16].

2.2.1 Sensing

The sensing module is responsible for the sensors used in an ADS. ADSs support sensors including camera, LiDAR, IMU, radars, ultrasonic sensors, and GPS [12],[3],[2]. The selection of sensors and computers highly depends on the requirements of the ADS features [8]. LiDAR, cameras, radar, and ultrasonic sensors are mainly used to recognize the surrounding environment, IMUs are used for inertial measurement of the ego-vehicle, and GPS to find the position of the ego-vehicle. Ego-vehicle is the vehicle on which the specific ADS is running.

2.2.2 Computing

The computing module contains all systems from receiving and processing the sensor data to deciding the action of the vehicle [8]. It consists of the sub-modules: perception, planning, and decision. The perception module is again divided into three tasks, localization, detection, and prediction, and the planning module into mission and motion planning.

2.2.2.1 Perception

The goal of the perception module is to create an intermediate-level representation of the environment state, i.e., a bird-eye view map of all obstacles and agents in the environment [3]. This state includes lane position, drivable zones, location of agents such as cars and pedestrians, state of traffic lights, and others. The representation is then utilized by a decision-making module to produce the driving policy. Uncertainties in the perception can propagate to the rest of the system; therefore, robust sensing is critical for safety. This can be achieved by using point

cloud data obtained from LiDAR scanners in combination with tasks like semantic segmentation, motion estimation, and depth estimation to create a multi-task model.

Localization and Mapping

The localization module finds the position of the ego-vehicle in the real world [12]. Localization is achieved by 3D maps and Simultaneous Localization and Mapping (SLAM) algorithms in combination with the sensor data from the sensing module. Different ADSs use different algorithms and sensors to manage this task. Map creation can be achieved using the Normal Distributions Transform (NDT) algorithm along with SLAM and LiDAR. It can also be achieved using Real Time Kinematic (RTK) and multi-sensor fusion-based methods together with GPS, IMUs, and LiDAR. ADSs also support deep learning and traditional machine learning approaches for object detection. Two examples of algorithms that use fully CNNs architectures for object detection are Single Shot MultiBox Detector (SSD) and You only look once (Yolo2) [8].

Detection

The detection module's task is to learn about the surrounding environment and can also be called the tracking module [12]. The detection module detects both stationary and moving objects such as traffic lights, traffic signs, pedestrians, and vehicles. Detection can utilize both LiDAR and camera for detectors. The detection module is often divided into smaller task modules with the assignment to track different items for different features, such as lane tracking or object detection.

Prediction

The prediction module predicts the future trajectories of surrounding objects of the ego-vehicle, such as cars and pedestrians, and assesses the risk of a potential collision with the ego-vehicle [12]. It receives the information and results from the localization and detection modules and generates the predicted waypoint trajectories of the objects by using probabilities based on headings, velocities, and accelerations of the objects and the ego-vehicle. Examples of methods used to perform the probability evaluation are Multi Layer Perceptron (MLP) evaluator and Recurrent neural network evaluator.

2.2.2.2 Decision

The decision module acts as a bridge for the perception and the planning module [12]. Depending on the result of perception, the decision module chooses a driving behavior so that an appropriate planning function can be selected. The current approach to decision-making is a rule-based system. In addition, the ADS allows the driver to control automation commands i.e., changing the state determined in critical cases.

2.2.2.3 Planning and Driving policy

The planning module predicts the ego-vehicles trajectory based on outputs of the decision module [12]. It computes a trajectory that is safe and comfortable for the vehicle's control module to execute. The module is divided into mission and motion planning.

Mission planning

Based on the driving states, such as lane changes, passings, and junctions, the mission planner uses a rule-based system to decide path trajectories, including lane selections as an array of waypoints to the destination. Global routing from source to destination is also counted as part of mission planning. The 3D map contains stationary road features such as buildings, which can be used for this mission planning function. The basic policy of the mission planner module is the ego-vehicle to follow the center lines of the lanes over the route generated by the map navigation system, involving the 3D map information.

Motion planning

The motion planning module is responsible for creating possible local trajectories by following the global trajectory given by the mission planning module and bearing in mind drivable areas given by 3D map, traffic rules, and perception results when planning for amorphous environments such as parking areas.

2.2.3 Actuation

The actuation module, also called the control module, executes the DDT of the vehicle [12]. It receives the car status and planned trajectory and generates control commands such as throttle, brake, and steering angle. The control module uses different control algorithms to make a comfortable driving experience. The velocity control is based on classical methods of closed-loop control such as Proportional Integral-Derivative (PID) controllers and Model Predictive Control (MPC) [2]. PIDs and MPC methods aim to stabilize the vehicle's behavior while tracking the specified path.

3 Machine Learning

We start this chapter by introducing ML as a whole, followed by some core concepts of RL. Further, we present some algorithms that have found success within AD previously, such as Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Deep Deterministic Policy Gradient (DDPG), Asynchronous Advantage Actor-Critic (A3C), and Advantage Actor-Critic (A2C).

ML is a broad field with many algorithms designed to teach a program to connect an input to the desired output [17]. These algorithms can be divided into three major categories: Supervised learning, Unsupervised learning and Reinforcement Learning (RL).

Supervised learning aims to generalize the input in order to classify it where the correct label is given during training [17]. The generalization results from a learned function approximation made by the structure of the model. The wide variety of structures includes decision trees, decision forests, logistic regression, support vector machines, neural networks, kernel machines, and Bayesian classifiers. The classification can also be different for different models. The most popular are binary (true or false), multiclass (one of multiple classes), or multilabel (a combination of multiple classes) classification. Examples of supervised learning are spam classifiers of e-mail, face recognizers over images, and medical diagnosis systems for patients.

Unsupervised learning aims to learn without knowing the label of the input [17]. Examples are principal components analysis, manifold learning, factor analysis, random projections, autoencoders, and clustering. The latter tries to divide the input data into multiple classes by learning their features without knowing their true labels, whereas the others try to reduce the input dimensions to a more comprehensive representation.

Reinforcement Learning learns through interacting with its environment and receives feedback, a reward, that indicates how good the action was instead of being given the correct label [17]. If the action does not receive the highest score, the model tries to find a better action. Although the reward can come directly after the action, most often, the reward is given after a sequence of actions letting the model learn a policy to optimize every action given a state. RL algorithms are typically used to "solve" MDPs [3], described further in Section 3.1.

3.1 Markov Decision Process

The MDP is used to describe a system or a process. It consist of a set of states (S), a set of actions (A), a transition function (T), and a reward function (R) [18]. The tuple $< S, A, T, R >$ says that for any state s there exists an action a that moves the environment and/or agent into a new state s' according to the transition function $T(s, a, s')$ and this transition gives a reward $R(s, a)$. A policy, π, is a mapping from states to probabilities of actions being chosen. In other words, an agent can use the policy to select an action for each state. RL algorithms can then improve the policy in order to find the optimal policy, π^*, that maximizes the return for each state [3]:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underbrace{\mathbb{E}_\pi\{\sum_{k=0}^{H-1} \gamma^k r_{k+1} | s_0 = s\}}_{:=V_\pi(s)}$$

The return is the sum of the immediate and future discounted rewards. Here, $\gamma \in [0, 1]$ is the discount factor of future rewards and r_{k+1} is the reward at time $k+1$. The expected return can also be denoted as a value function, $V_\pi(s)$, describing the value of the agent being in state s given that it follows the policy π. Closely related to the value function is the Q-function (action-value function), $Q_\pi(s, a)$, that tells the value of being in state s and taking action a given that following actions are selected according to policy π. One can find the Q-function related to the optimal policy by iterative updates according to

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha_t[r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)]$$

under certain conditions [19]. These conditions are typically not met in AD, mainly because the state space is not finite and the agent cannot be guaranteed to visit each state and action infinitely often. Instead, the Q-function is often estimated, typically by a neural network, allowing some generalization of the experience. However, this technique is prone to diverge due to approximation errors if utilized without care.

3.2 On- and Off-Policy

An on-policy algorithm learns the value of a policy by using the policy to take actions [3], [18]. On the other hand, an off-policy algorithm improves its target policy by using a behavior policy (separated from the target policy) to take actions. In this case, the behavior policy can let the agent explore the state-action space while the target policy is deterministic (greedy), which significantly benefits many applications. There are two main algorithms when talking about on- and off-policy, SARSA and Q-learning respectively.

3.3 Model-based and Model-free Algorithms

In a model-based approach, the agent represents the environment, its transition function, and its reward function in a model and therefore needs to store this in-

formation [3]. This approach is data-efficient [20] as it requires fewer interactions but is also limited by the estimated model since it is only an estimation based on model identification. A model-free approach on the other hand typically estimates the value function by sampling the MDP. This approach requires many samples to obtain a good estimation, unfit for real physical systems.

3.4 Value-based vs Policy-based Algorithms

Before describing the two types of algorithms, it is important to understand that both algorithms propose an action and evaluate this action [3]. However, a value-based algorithm will optimize the value function and have a policy follow the suggested action. The policy-based algorithm instead focuses on optimizing the policy, often estimated with a neural network, directly through gradient descent while the value function might not even be calculated. In general, value-based algorithms such as the DQN algorithm are commonly used for discrete problems like video games, while policy-based algorithms such as DDPG are frequently used for continuous problems like controlling robotics.

3.5 Deep Learning

In "shallow" learning, e.g. linear and logistic regression, there is often only an input layer connected to an output layer [20]. In deep learning, however, there are multiple layers in between, each connected with weighted sums of each unit of the previous layer with an activation function to follow (these structures are called deep neural networks, also commonly referred to as simply networks or neural networks). The activation functions are nonlinear transformations, e.g. logistic, tanh, and rectified linear unit (ReLU), to obtain a new representation in each layer. The idea here is then to use backpropagation to update the weights in order to optimize some loss function. Backpropagation is simply calculating the error derivatives backward, from output layer to input layer, and applying these gradients to each layer.

A special type of deep neural network is the convolutional neural network (CNN) which utilizes convolutional layers, pooling layers and the traditional fully connected layers [21]. The convolutional layers are designed to process data with grid patterns such as images, where they excel. These convolutional layers consist of filters that apply mathematical operations at each image position, and together with the pooling layers extract features in the images ranging from simple to complex throughout the network. The fully connected layers then map these features to the output, typically a classification.

Neural networks can often suffer from overfitting, which is a problem where the network has optimized its parameters to fit the training dataset but have difficulty generalizing to unseen data. A way to combat this problem is to introduce regularization by removing non-output units randomly from the layers during training

[20]. This is called dropout and helps the network optimize all units in each layer instead of strongly reacting to a selection.

It is also possible to use deep neural networks to estimate components in RL, such as policy, model, and Q- and value function [20]. Doing so leads to a subcategory within RL called deep reinforcement learning (DRL), which is a name for the subfield of RL algorithms that use this type of function approximator. While tabular representation often is the trivial representation, it can become intractable in real-world scenarios that are more complex due to, in Bellman's words, "curse of dimensionality" [3]. In these scenarios, deep neural networks are often a good approximator but when combined with off-policy and bootstrapping, it can lead to instability and divergence [20].

Continuing on DRL, when using estimated functions and values together with a maximization operation (popular set of operations among RL algorithm, i.e. choosing the action with the highest value), it can lead to maximization bias [18]. Maximization bias occurs when the true maximum value is not the same as the maximum of the estimated values. The example used in [18] considers a single state where all action values are zero and thus the true maximum is also zero. However, the estimated action values vary due to uncertainty where some values are above zero and some below. The maximum of the estimated action values is then positive leading to a maximization bias. To combat this bias, it is common to use double learning. Utilizing two estimators, one to estimate the maximizing action and one to estimate the value of the said action, the expected estimated action value is equal to the true action value of the same action and thus it is unbiased.

In [22], Smith emphasizes the difficulty in choosing hyperparameters for training neural networks. Hyperparameters often depend on the dataset, architecture of the network, and each other and it can be difficult to find a balance of these. Smith primarily discusses learning rate, batch size, momentum, and weight decay. The author found that the training benefits from a cyclical learning rate and that larger batch sizes are suitable for the learning rate cycle proposed in the report. However, for other learning rate schedules, it may not be the case. Smith also highlights the inconsistency in comparing batch sizes without changing other parameters such as the number of iterations or epochs. Larger batch size and learning rate gives better performance but drops off when too large (Smith argued it was because of too few iterations) but emphasizes the conditions of architecture, dataset, and hardware.

3.5.1 DQN

DQN uses Q-learning, which includes several concepts previously described such as off-policy, model-free and value-based, and a deep neural network for function approximation. Where combinations of these concepts often lead to instability, Mnih et al (2015) proposed the DQN algorithm that set the foundation for DRL [20]. The algorithm showed promising results at Atari games by connecting raw images to actions [23]. Using experience replay and target network, they managed

to stabilize the training and raise the performance significantly. Experience replay allows observations of state, action, reward, and transition to be used multiple times during training by storing them in a memory bank and sample them uniformly [24]. The target network is initially identical to the online network (the network producing the action at each timestep) but does not update its parameters except every τ timesteps when it copies the parameters of the online network. The target network is used to evaluate the Q-value of the optimal action in the next state. The algorithm also stacked 4 consecutive images to represent a single state in order to comprehend the temporal information [3]. A simple preprocessing step was applied to each image to reduce the input dimensionality before forwarding the state to the network [23]. In Algorithm 1 below follows the pseudocode for the DQN algorithm [23].

Algorithm 1 Deep Q-Network [23]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ε, select a random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j{+}1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi, a_j; \theta))^2$ with respect to the network parameters θ
    Every τ steps, reset $\hat{Q} = Q$
  end for
end for

3.5.2 DDQN

Double Deep Q-Network (DDQN) builds upon the DQN algorithm with the extension of double learning to tackle the maximization bias described in section 3.5. Van Hasselt et al. (2016) [24] suggested to use the online network to estimate the optimal action and the target network to estimate the value of that action. Doing so will avoid introducing additional networks and improve the stability and reliability of the training. Their results also show that the DDQN algorithm can find better policies than its predecessor. The DDQN algorithm can be seen in Algorithm 2 where only the calculation of the target y differs.

Algorithm 2 Double DQN [24]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ε, select a random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma\hat{Q}(\phi_{j+1}, \arg\max_{a'} Q(\phi_{j+1}, a'; \theta); \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi, a_j; \theta))^2$ with respect to the network parameters θ
    Every τ steps, reset $\hat{Q} = Q$
  end for
end for

3.5.3 A3C

Asynchronous Advantage Actor Critic (A3C) is a policy-based algorithm in the core utilizing the actor-critic structure, seen in Figure 3.1, common in many policy-based algorithms. However, actor-critic methods combine benefits from both policy-based and value-based algorithms by letting the "actor" choose the action (policy) and the "critic" evaluate the action by estimating the Q-function (value) [3]. Each network needs its own gradient to update and a baseline to calculate the gradient. In the case of A3C (and A2C as will be apparent in Section 3.5.4), the baseline used is called *advantage*, which intuitively is a measurement of how much better the chosen action is compared to the average. The *advantage* can be formulated as $A_\pi(a, s) = Q_\pi(s, a) - V_\pi(s)$. In addition, the A3C algorithm uses several instances of the environment in parallel and lets the agents perform actions according to different policies asynchronously. This method compared to using experience replay increases the stability of the learning and reduces the memory used. It is also shown to learn faster than other state-of-the-art algorithms at the time the algorithm was proposed.

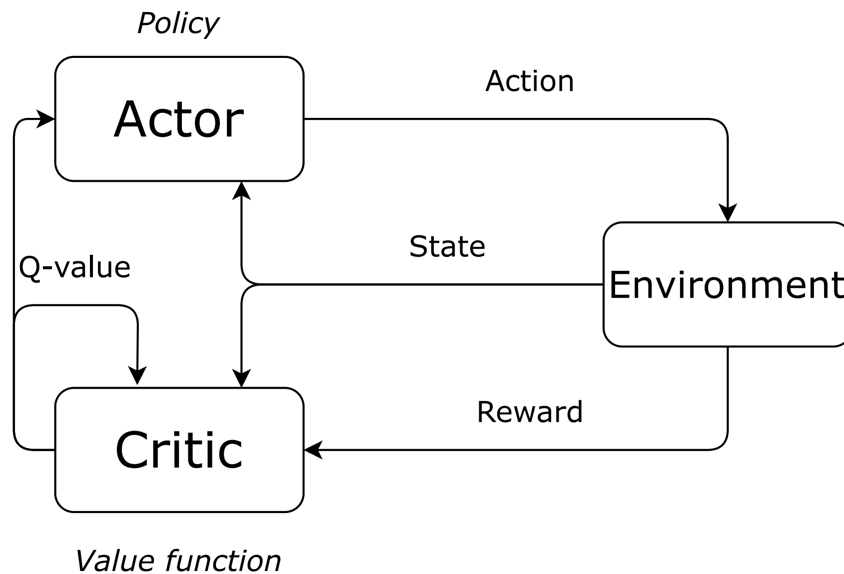


Figure 3.1: Structure of actor critic algorithms.

3.5.4 A2C

Advantage Actor Critic (A2C) is very similar to A3C with the difference that it is synchronous instead of asynchronous [3]. This means that the model waits for each agent to finish its work before the update and the next agent begins. This leads to a more cohesive training experience where each agent starts from the same policy at the start of each new iteration. The results of A3C and A2C are comparable.

3.5.5 DDPG

Deep Deterministic Policy Gradient (DDPG) is also an actor-critic algorithm. In contrast to A3C and A2C however, DDPG utilizes experience replay and a target network [25]. The target networks make "soft" updates toward the online networks (actor and critic separately) according to $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ where θ is the online network, θ' is the target network and $\tau \ll 1$ is a parameter that decides how much the target network should update toward the online network. The approach for incorporating exploration into the model is by adding sampled noise to the actions.

Algorithm 3 Deep Deterministic Policy Gradient [25]

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1,T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration
  noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

4 Simulator

In this project, we chose to utilize a simulator for training ADSs instead of using a real-world model car. Throughout this chapter, we will explain the reason why a simulator is used and present the key attributes we looked for while exploring simulators for our purpose. Finally, we will present each simulator we explored and its attributes.

An important part of training self-driving cars is to be able to model the variety of situations that they are expected to be exposed to on the roads [26]. The benefits of using a simulator are that the user has full control and knowledge of the environment. In the case of testing and validation of an ADS, this can prevent accidents and in development, this can increase project timelines and reduce costs. It also yields simplicity and adaptivity to the intended application. The simulators investigated in this project are 3D/game engine based and the simulator should be able to model situations that could be used to train a feature or an end-to-end ADS. The simulators considered are DonkeyCar-simulator, LG-SVL, UdaCity, esmini and CARLA. All simulators already have environments and tracks built that are ready to use. If, however, the project would be conducted in real-world, using physical models of the car and tracks, the tracks would need to be built or traveled to in order to perform experiments. This would not be appropriate in two aspects, test reproducibility, and travel dependency. It is also very inconvenient to manually observe every episode when training models with RL instead of letting the simulator reset the vehicle when it crashes. Two other advantages are the noise-free observations, which are suitable for an initial project, and the non-existent cost since the simulator should be open-source. However, simulators also have drawbacks. An ideal simulator should be as close to reality as possible. This means it must be highly detailed in terms of 3D virtual environment and very precise with vehicle calculations such as the physics of the car. With the graphical abilities of modern 3D engines combined with GPUs, scenarios can be realistically reproduced but they might come with processing/resource limitations. In the last aspect mentioned, it could be argued that a simulator needs special hardware to be run on which can be expensive. Therefore, it is a must to find a trade off between the realism of the 3D scene, the simplification of the vehicular dynamics, and the cost and resource requirements. Fortunately, the hardware for this thesis was provided by Infotiv and was well suited for the purpose of the project.

4.1 Criteria

To find a simulator suitable for the purpose of this thesis, extensive research into ADS and what ML methods are used for ADS training in simulators was made. This made it possible to set up a list of criteria to consider while investigating the different simulators available. The simulator should satisfy the broad criteria given below:

- Allow manual editing of the environment such as placing objects on the road and manipulate the layout of the road/track.
- Run in a decent update rate on the provided working computer, i.e., not lagging too much.
- Include tools to train and test an ADS.

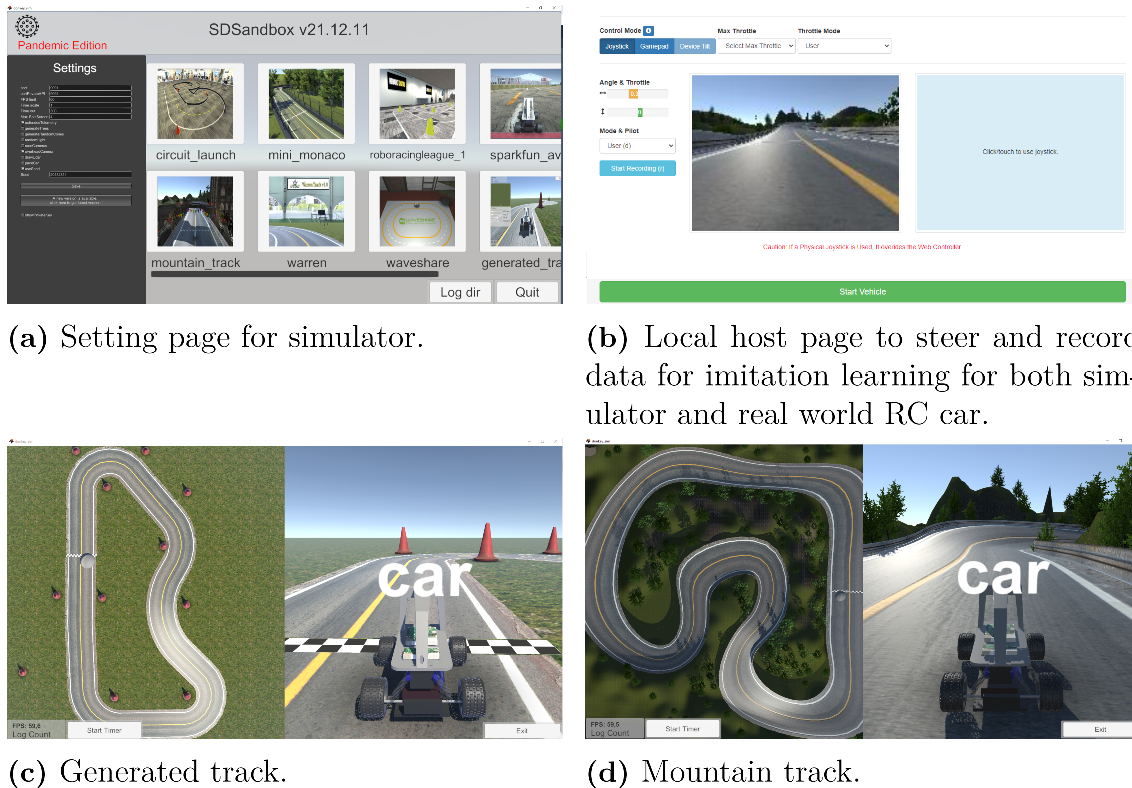
These criteria are not very strict and the choice of simulator will likely be subjective in the end but are presented here to give a rough estimation of the thought process. Graphics and other performance metrics will not be evaluated as criteria but could potentially affect the subjective decision.

4.2 Open-source simulators

This section follows a summary of the different simulators found in the initial survey for the thesis. All simulators are developed for similar purposes and the intention here is to present their characteristics and uses to later motivate our choice.

4.2.1 DonkeyCar

The DonkeyCar simulator is a part of the Donkey self-driving-car platform for hobby RC cars [27]. It is developed by a community of enthusiasts, developers, and data scientists that enjoy racing, coding, and discussing the future of ML. DonkeyCar is made up of several components: a high-level self-driving library written in Python, Open-Source hardware designs for RC-cars constructed of actuation hardware, microcontroller, camera, and LiDAR setups. It was developed with a focus on enabling fast experimentation and easy contribution which makes it easy to build your own RC car or buy a complete model. Another interesting feature of the DonkeyCar concept is the simulator that enables the use of a simulated DonkeyCar without buying an RC car. It has a python API with some examples implemented utilizing OpenAI's gym wrapper environment and predefined tracks. The tracks are developed for the purpose of racing under simple conditions; thus no crosswalks, buildings, or traffic is implemented. DonkeyCar simulator is built on the Unity 3D/game engine which makes it easy to add and develop more tracks or add environmental conditions if wanted and the complete source code is provided through GitHub. The simulator shows great potential for an ML project aimed toward ADS by combining the internal physics and graphics of the Unity game engine and connecting it to a DonkeyCar Python process to train ML models.



(a) Setting page for simulator.

(b) Local host page to steer and record data for imitation learning for both simulator and real world RC car.

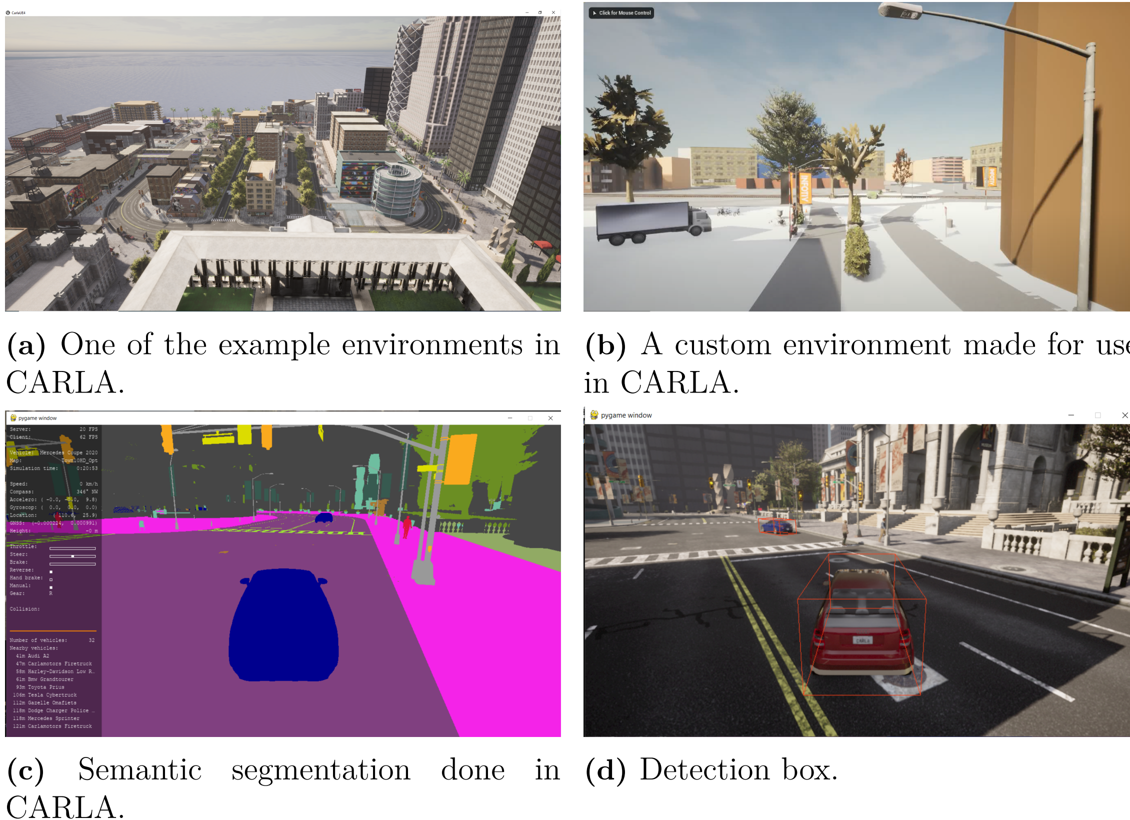
(c) Generated track.

(d) Mountain track.

Figure 4.1: DonkeyCar simulator.

4.2.2 CARLA: CAR Learning to Act

CARLA has been developed from the ground up to support development, training, and validation of autonomous driving systems [28]. CARLA is developed in the 3D engine Unreal Engine 4 and have a Python API that users can use to control the simulations [29]. The simulator is focused on urban layouts and can generate multiple agents such as vehicles and pedestrians and the maps are well detailed with buildings, signs and sidewalks to simulate urban traffic. It has a wide range of features including, but not limited to, multiple clients being run in the same environment, flexible sensor setups for sensors such as LiDAR, GPS & cameras and fast simulation without rendering. CARLAs software have high flexibility which allows and enables the user to adapt the environment to their purpose. One example of an AD project that has been performed in CARLA is presented in [29], where a study of three approaches to autonomous driving is explored. The approaches are a classic modular pipeline, an end-to-end model trained via imitation learning and an end-to-end model trained via reinforcement learning. CARLA also provides a platform to deploy existing ADS through ROS integration via ROS-bridge, e.g. the Autoware agent. With so many features, however, CARLA is perceived as a complex simulator which makes it difficult for inexperienced users to navigate through and make full use of its capabilities.



(a) One of the example environments in CARLA.

(b) A custom environment made for use in CARLA.

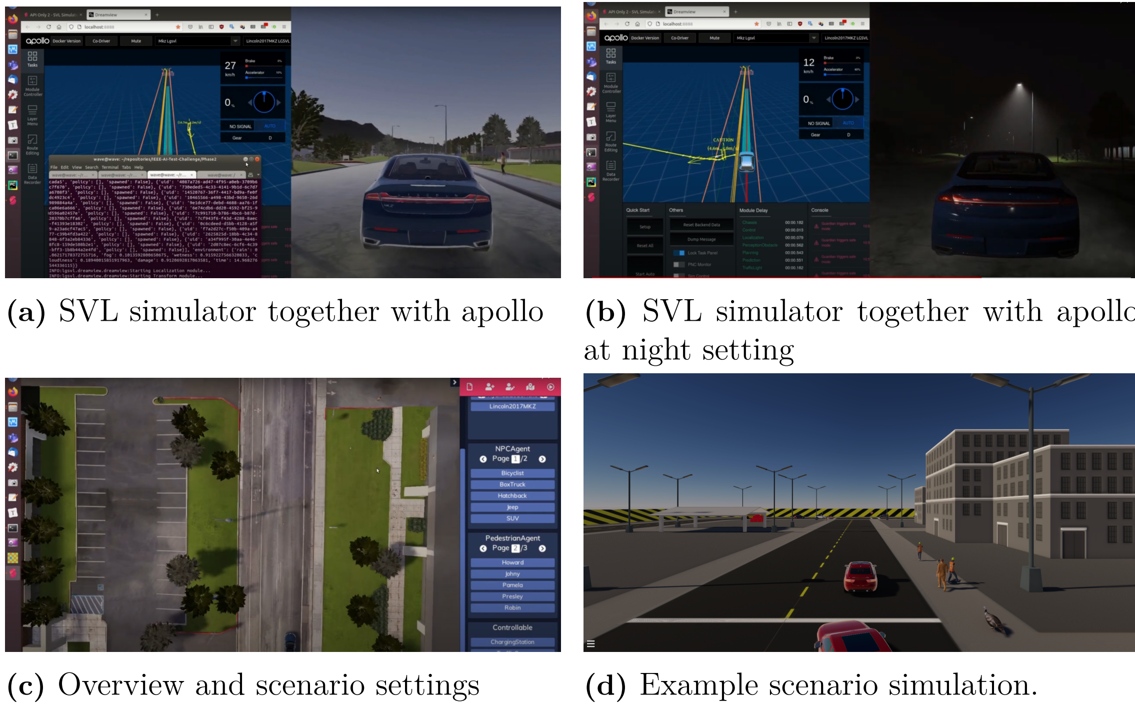
(c) Semantic segmentation done in CARLA.

(d) Detection box.

Figure 4.2: Example screenshots from CARLA simulator.

4.2.3 SVL

The SVL simulator is similar to Carla in many ways and focuses on providing a simulator for end-to-end, full-stack testing and training environment for ADS in urban traffic [30]. SVL provides a broad range of functions that can be broken down into three categories: environment simulation, sensor simulation, and vehicle dynamics and control simulation of an ego vehicle. The environment simulation includes traffic simulations as well as physical environment simulations like weather and time of day, sensor simulations such as camera and LiDAR, vehicle dynamics, and control simulations. All functionalities can be controlled via a Python API. As CARLA, the simulator has a communication bridge that enables passing messages between the simulator and ADSs as Autoware and Apollo by ROS bridge, ROS2, and Cyber RT. SVL is developed using the Unity game engine and the source code is available as open-source on GitHub. Unfortunately, since December 2021, the open-source version of the simulator is no longer updated and developed by LG.



(a) SVL simulator together with apollo

(b) SVL simulator together with apollo at night setting

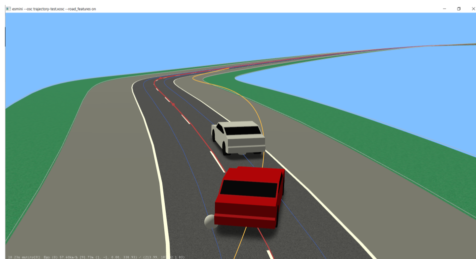
(c) Overview and scenario settings

(d) Example scenario simulation.

Figure 4.3: Example of scenario simulations in SVL for pedestrian detection using Apollo 6.0 [13]

4.2.4 Environment Simulator Minimalistic (esmini)

Esmini is a very minimalistic simulator with primitive graphics [31]. The simulator project had the purpose of implementing and exploring the emerging OpenSCENARIO data format. The simulator can simulate simpler scenarios such as highway passing and work great for this type of high-level scenario simulation. The simulator does have a variety of maps including urban and rural traffic but without support for first-person view, like a front camera. This makes it difficult to train a relevant ADS with RL. The project was not intended for standard production applications; therefore, the quality of the code is lacking when it comes to clarity, structure, comments, error handling, and coding guidelines. The Simulation Scenarios project is also closed and doesn't have any formal support from the initial contributors.



(a) Esmini scenario simulation: trajectory-test.

(b) Esmini scenario simulation: pedestrian-collision.

Figure 4.4: Two examples of scenario simulations in esmini.

5

Implementation

This chapter explains the methodology in which the project was carried out and provides more insight into the decision-making regarding the implementation of the end-to-end ADS, including algorithm, simulator, and configurations during training. Worth noting is that the selection of simulator and algorithm was performed simultaneously and to a degree of dependency on each other. DonkeyCar simulator was chosen in favor of other simulators and is presented with further reasoning in Section 5.1.1.

5.1 End-to-end setup

To demonstrate and test the findings of the literature studied, we implemented an end-to-end ADS. It should utilize a single camera to receive information about the environment and return a steering action to stay on a track while driving at a constant speed. In Chapter 2.2, an investigation was conducted into the architecture of ADSs. The findings demonstrated that an end-to-end system could provide a suitable ADS type for the project, thanks to the straightforward structure and the synergy with a simulator environment. Figure 5.1 shows the architecture of the intended end-to-end ADS. This can be seen as a simplified model of the end-to-end system described in Figure 2.1. The sensing module contains a single sensor, a synthetic camera in the simulator, to collect data about the environment. The images are then sent to the computing module which consists of two parts, one function to process the image input and a CNN to determine the appropriate action of the vehicle given the current state. RL will be used to train the CNN to predict appropriate actions, as further explained in Section 5.2. The determined action is then sent to the actuation module that controls the steering of the vehicle in the simulator. Further in this section, the preprocessing function, CNN architectures, and the setup of the DonkeyCar simulator are described.

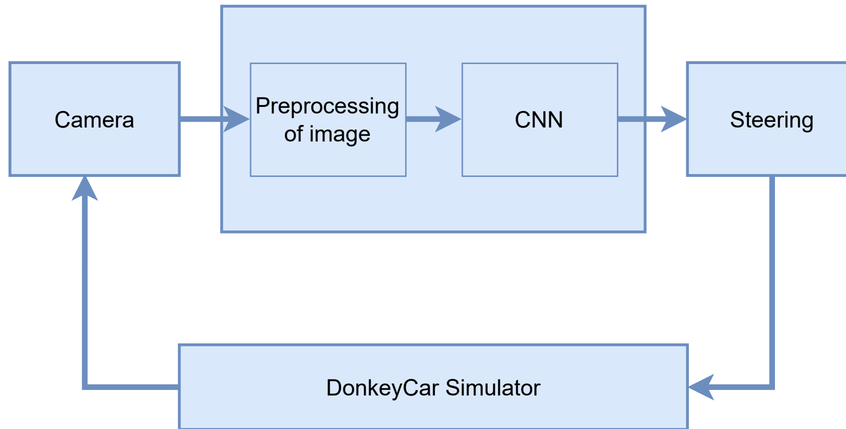


Figure 5.1: Target structure of the implemented system.

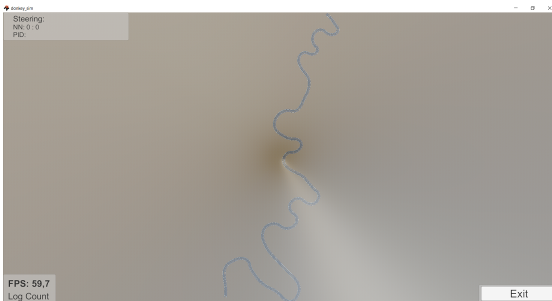
5.1.1 DonkeyCar Simulator

As described in Chapter 4, an investigation was conducted to explore different simulators and their potential for this project. The DonkeyCar simulator was decided to be used, primarily due to its intuitive Python API, the intended programming language. The other simulators also have Python API, however, they were not as easy to use. This made interacting with the simulator easier compared to the other simulators. Other reasons that motivated the choice were:

- **Esmini** has too minimalistic graphics and was developed to simulate scenarios rather than to train ADSs. This means that the simulator lacks features like synthetic sensors.
- **SVL** lost its support from the developers, which could lead to complications with further development of the project.
- **Carla** was in contention for a while but was ultimately not as beginner-friendly as DonkeyCar, while DonkeyCar still provided the potential for further development.

DonkeyCar offers relatively simple tracks to train a basic end-to-end ADS and the environment wrappers and Python API offer all the necessary parts to develop the end-to-end system. Such parts are sensor support for camera and LiDAR, editable environments, and good control and information flow from the simulator and Python scripts. This made it possible to investigate features such as lane following and obstacle avoidance, two of the main tasks we sought to accomplish. Another advantage of DonkeyCar simulator is the RC car models provided by DonkeyCar, this enables a broad range of future projects and the opportunity of testing models and AD features in the real world. The DonkeyCar simulator has ten implemented tracks that are ready to use. Of the ten existing tracks mainly four tracks were used during the project: *Mountain track*, *Mini Monaco*, *Generated roads* and *Generated track*. These tracks have different environmental characteristics and features that could be utilized during the training process and testing of the AD models. The main features of the tracks are presented below followed by pictures in Figure 5.2 illustrating the tracks.

- **Mountain track**
  - Rails
  - Elevation
  - Surrounding environmental objects
- **Mini Monaco**
  - Rails
  - Trees around the track
- **Generated Roads**
  - Randomly generated path
  - Sand area off-road
  - Empty environment except the road
  - Its randomly generated road enables an infinite number of tracks with the same characteristics but different shapes
- **Generated Track**
  - Trees with randomly generated positions around the track
  - Cones, placed around the track
  - Grass area around the track



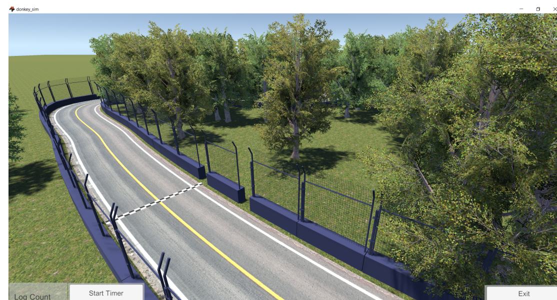
(a) *Generated roads.*

(b) Overview of *Generated Roads.*

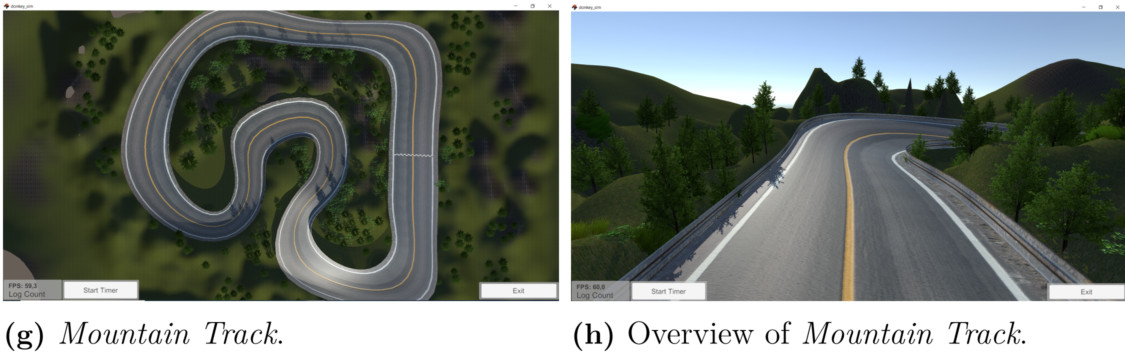
(c) *Generated Track.*

(d) Overview of *Generated Track.*

(e) *Mini Monaco.*



(f) Overview of *Mini Monaco.*

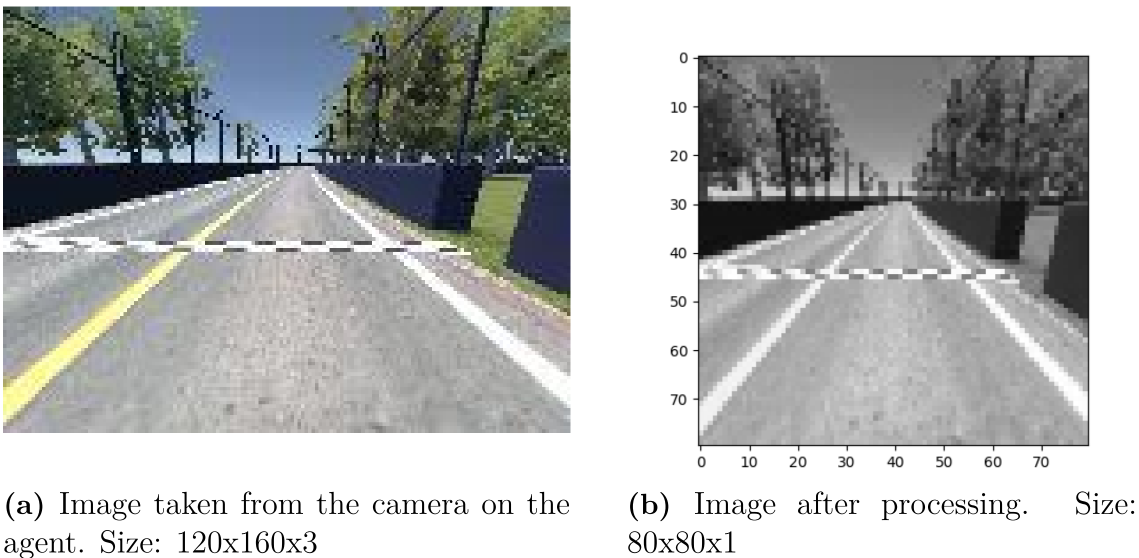


(g) *Mountain Track*. **(h)** Overview of *Mountain Track*.

Figure 5.2: The tracks from the DonkeyCar simulator used in the project.

5.1.2 Image Processing

The image processing consists of a color conversion and a down-sampling. The input image from the simulator has size 120x160x3, where 3 is the number of color channels, in this case RGB. Training on images of this scale is computationally heavy and occupies a lot of memory without contributing meaningfully to performance. Instead, a common choice is to downscale the input images by resizing them and converting them to grayscale. The output of the image processing is then 80x80x1 which can later be turned into an 80x80x4 tensor by stacking 4 consecutive images on top of each other, recalling the state input of the DQN algorithm presented in Section 3.5.1. The image processing can be further expanded by introducing more functions such as lane detection, object detection, and semantic segmentation. Although introducing these functions can lead to better performance, it increases the complexity of the end-to-end system and moves it closer to one of the examples of existing ADSs presented in Section 2.1 and away from the simplicity of the intended ADS architecture. Figure 5.3a shows the input image from the simulator and in Figure 5.3b is the image after processing.



(a) Image taken from the camera on the agent. Size: 120x160x3

(b) Image after processing. Size: 80x80x1

Figure 5.3: Image before and after image processing.

5.1.3 CNN

As mentioned in Section 3.5, CNNs are excellent at extracting features and concluding outputs from images, making them suitable for an end-to-end system that takes four frames as input and outputs a steering action. During research, two different CNN architectures were found in similar DonkeyCar projects, both using DDQN, see Section 5.2.1 for choice of RL algorithm (DDQN). Both networks take an 80x80x4 tensor as input and output the Q-value of 15 classes. Each class is associated with a discrete steering value between -1 and 1 and the highest Q-value selects the predicted action. This project incorporated both networks to compare their performance.

5.1.3.1 Network 1

Figure 5.4 shows a visualization of Network 1. The network has five convolutional layers, each followed by an activation layer, one flatten layer to map the extracted features from 3D-arrays to a 1D-array, and finally two dense layers, where the first dense layer is followed by another activation layer and the second produces the output classes. The size and number of filters of each layer can be seen in Figure 5.4 and the complete input and output of all layers can be seen in the Appendix A in Figure A.2. Network 1 consists of almost 1 million parameters and requires 3839kB memory.

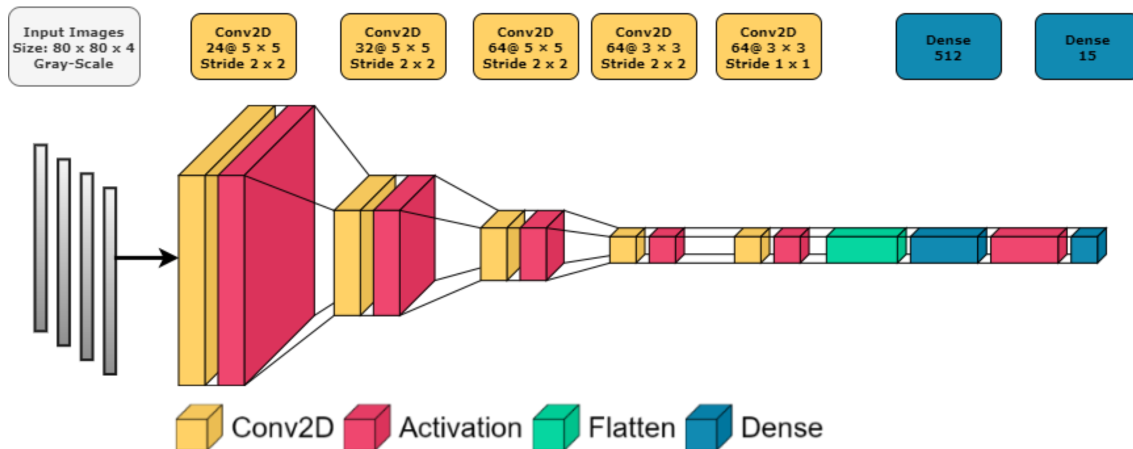


Figure 5.4: Visualization of network 1. All activation functions are RELU.

5.1.3.2 Network 2

Figure 5.5 shows a visualization of Network 2. The network is constructed by three convolutional layers, each followed by an activation layer, one flatten layer, and two dense layers, where the first layer is followed by another activation layer and the second produces the output classes. The size and number of each layer can be seen in Figure 5.5 and the complete input and output of all layers can be seen in the Appendix A in Figure A.1. Network 2 consists of about 3.4 million parameters and requires 13163kB memory.

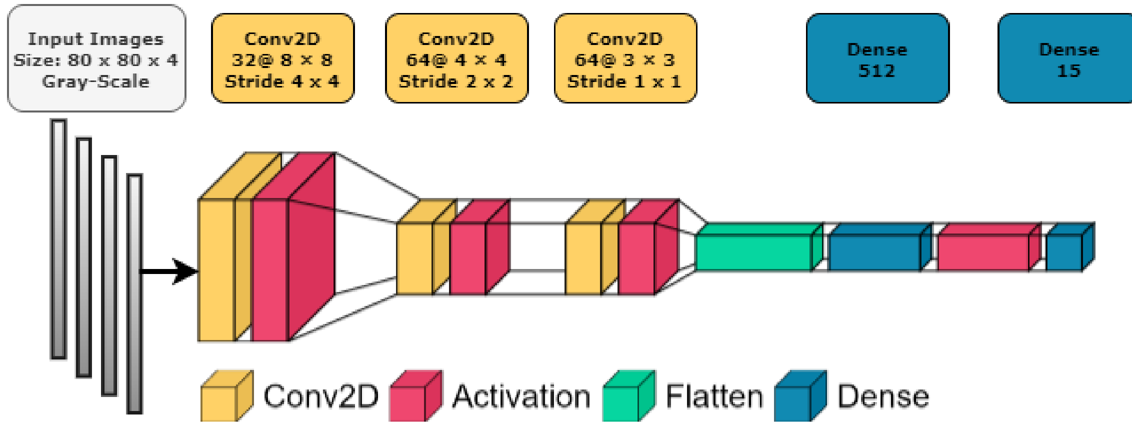


Figure 5.5: Visualization of Network 2. All activation functions are RELU.

5.2 Training

An overview of the training session can be seen in Algorithm 4. The session begins with starting and initializing the simulator and environment through the Python API, declaring the training track and maximum cross tracking error (explained further in Section 5.2.2). The ML agent also needs to be initialized with learning rate, batch size, network, discount factor, exploration parameters, action space, and replay memory. In this project, the three former parameters have been adjusted to create multiple configurations of trained models and the remaining parameters were kept fixed. The discount factor let the agent know how much it should value future rewards. The exploration parameters control the number of random actions taken during the training phase. The action space simply let the agent know what actions are valid, also used during exploration. The replay memory stores all past experiences in a buffer, which needs to be allocated memory before the training begins. Once initialized, the agent starts a new episode and preprocesses the first image and stacks it four times to form the first state. During each timestep in an episode, the agent will perform an action by either passing the latest state (four consecutive frames) to the online network, which will output a predicted action or select a random action. After the simulator has received and performed the action in the environment, it will return the next frame together with a reward and a flag indicating if the episode is over (done-flag). The episode ends if the agent has either hit an object or gone too far off the track (exceeded the max cross tracking error). The agent proceeds to preprocess the next frame and stacks it on top of the last state, discarding the first frame in the previous state and forms a new state. The experience with state, action, reward, and next state is saved in the replay memory. Thereafter, the agent will train the online network on a batch sampled from the replay memory, using the target network to estimate the Q-values (according to DDQN described in Section 3.5.2). Lastly, after each episode, the target network will be updated to have the same parameters as the online network. Worth noting is that the online network outputs the steering action, whereas the throttle value remains constant throughout the entire training. There is also a brake function in

the DonkeyCar simulator. However, we disregarded it.

Algorithm 4 Training session

```
Initialize simulator and environment.
Initialize the ML agent.
for each episode do
    Preprocess first image.
    for each timestep in episode do
        Predict action (CNN) or choose random action.
        Observe next image, reward, and done-flag.
        Preprocess next image.
        Store experience in replay memory.
        Train online network on batch from memory.
    end for
    Update target network.
end for
```

5.2.1 Reinforcement Learning algorithm

As mentioned in Chapter 5, the selection of the algorithm was partially dependent on the simulator. DDQN was one of the algorithms that seemed effective at AD tasks yet relatively simple according to the literature study. Coincidentally, DDQN was also already implemented in the Python API and thus only needed to be modified to work for our purpose. We, therefore, decided on DDQN as a starting point for our project to get familiar with the core concepts of training an AV. There was also an attempt to implement the DDPG algorithm to allow continuous actions and compare with the DDQN. However, this implementation did not perform very well and we believe the issue was an implementation bug rather than an issue with the algorithm itself.

5.2.2 Reward function

The reward function determines the feedback to the CNN during training and shapes its behavior. It can be made complex, but for a project in the DonkeyCar simulator, it makes sense to keep it relatively simple. Each track in the DonkeyCar simulator has a reference line throughout the track. On some tracks, it is in the middle of the road whereas on some tracks it is in the middle of the right lane. This inconsistency makes the models behave differently since the reward function is based on the distance between the agent and the reference line, also called the Cross Tracking Error (CTE). If the distance is too large, larger than the defined maximum CTE, the episode will end and a negative reward will be returned. The reward function will also return a negative reward if the agent collides with any objects, ending the episode. The speed of the agent also affects the reward, however, since the throttle is constant during training, this part is irrelevant. The reward function is summarized in (5.1).

$$\begin{cases} -2 & \text{if collision} \\ -1 & \text{if CTE} > \text{max_CTE} \\ (1.0 - \frac{|\text{CTE}|}{\text{max_CTE}}) \cdot \text{speed} & \text{otherwise} \end{cases} \tag{5.1}$$

5.3 Verification and Validation

We used both objective and subjective measurements to test and verify our models. We documented the accumulated reward given to the model for objective measurements during both training and testing. The accumulated reward for each episode during training indicates the learning curve of the model as well as when it saturates. In contrast, the average accumulated reward during testing is more suitable for comparing the performance of different trained models. The subjective measurements were observations of the overall performance by the authors during testing.

6

Results

In this chapter, we present the results from the training as well as the tests made on a selection of different configurations. During the training, it is interesting to note the progression of the model. To do this, it is common to record the accumulated reward for each episode to see how fast the model improves and manages to get further along the track before crashing or going outside the track. Next, the trained models are tested on multiple tracks to investigate how well they have generalized the training. Some of the benchmarks are subjective, here we look for jitter in steering and maneuvering when approaching turns. Other benchmarks are objective, these include average accumulated reward and the fastest lap time.
Unless specifically stated otherwise, every model was trained with the hyperparameters presented in Table 6.1.

Hyperparameter	Value
Network Architecture	Network 1
Batch size	64
Learning rate	1e-4
Episodes	1000
Track	*Generated track*

Table 6.1: Standard hyperparameters.

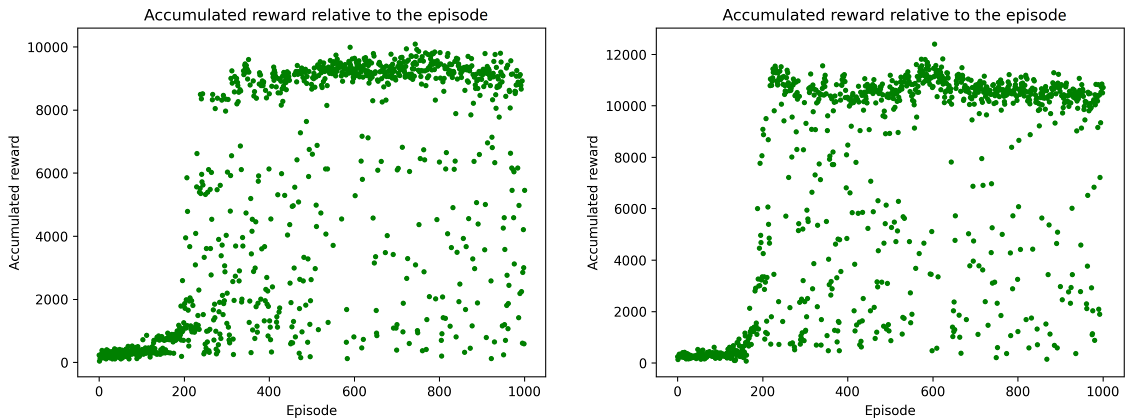
6.1 Network architecture and batch size

The first test compares two different network architectures and two different batch sizes. The graphs in Figure 6.1 show the training progression for the different networks and batch sizes. Both networks seem to learn faster with batch size 32, which is unexpected considering that the network needs more iterations to train on the same number of samples. Note that Figure 6.1b has a slightly different y-axis, indicating that it achieved slightly higher rewards during training. It can also be seen in Table 6.2 that even though the configuration with network 2 and batch size 32 seemed to have learned faster, the configuration with the same network but batch size 64 managed to outperform it, even if just slightly, highlighting the difficulty of choosing training parameters. The same table also supports the visual inspection which perceived the behavior of all models to be similar during testing. The models kept the vehicle on the road most of the time but could drive off-road during some parts. All other parameters were kept constant, with an exception for the *Generated roads* track which is randomly generated each time the environment is

initialized (i.e., each new training session and test run) without an option to fix the seed.

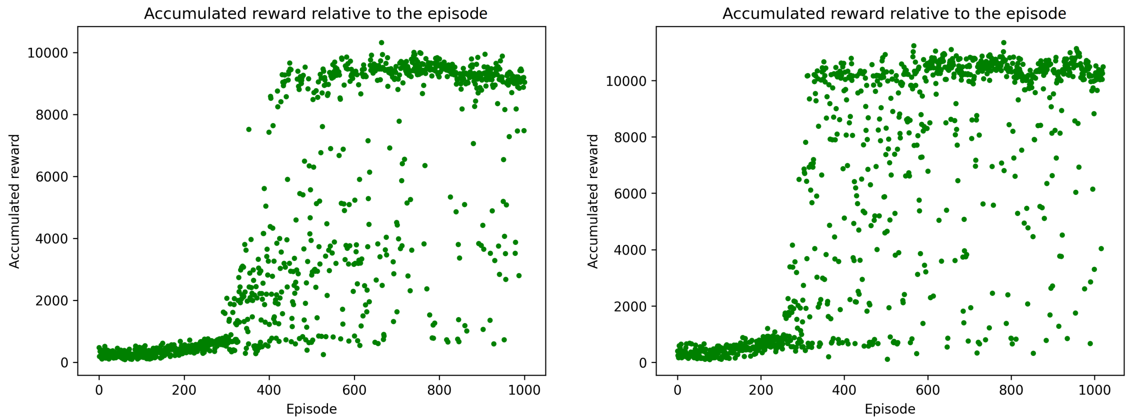
Model	Average accumulated reward
ddqn_net1_64_genRoads	27 331
ddqn_net1_32_genRoads	34 945
ddqn_net2_64_genRoads	35 627
ddqn_net2_32_genRoads	33 383

Table 6.2: Average accumulated rewards of models during test 1.



(a) Trained for 1000 episodes on *Generated roads* with Network 1 with batch size 64.

(b) Trained for 1000 episodes on *Generated roads* with Network 1 with batch size 32.



(c) Trained for 1000 episodes on *Generated roads* with Network 2 with batch size 64.

(d) Trained for 1000 episodes on *Generated roads* with Network 2 with batch size 32.

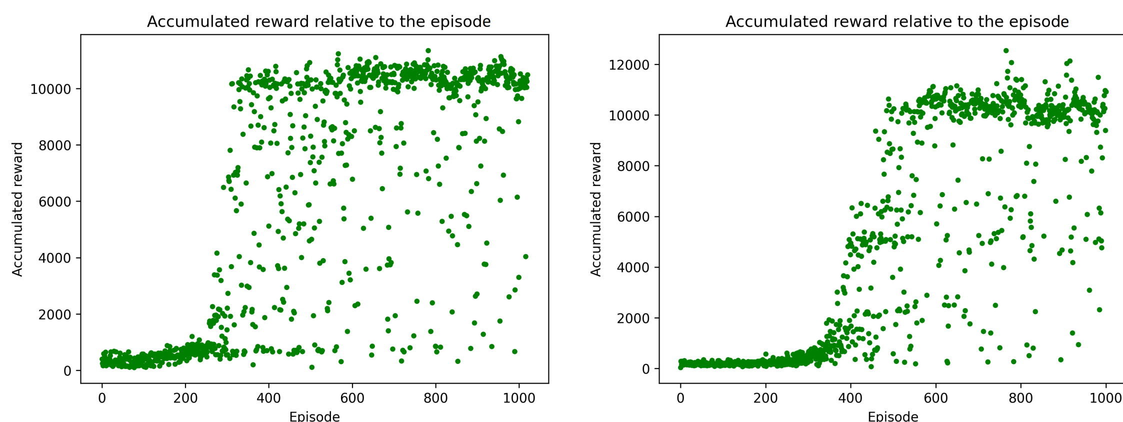
Figure 6.1: Training progression of different networks and batch sizes.

Another aspect included in the first set of tests is the randomness of the training sessions. Due to a mishap, the configuration with network 2 and batch size 32 was trained twice. Figure 6.2 displays that multiple training sessions with the same

configuration yield different learning curves. This is believed to depend on the randomness during exploration and initialization, but could also depend on *Generated roads* which is randomly generated at each initialization. However, even though the training progression is different, both models perform similarly as can be seen in Table 6.3.

Model	Average accumulated reward
ddqn_net2_32_genRoads_first	33 383
ddqn_net2_32_genRoads_second	35 460

Table 6.3: Accumulated reward of same configuration over two different training sessions.



(a) Trained for 1000 episodes on *Generated roads* with Network 2 with batch size 32. First training session.

(b) Trained for 1000 episodes on *Generated roads* with Network 2 with batch size 32. Second training session.

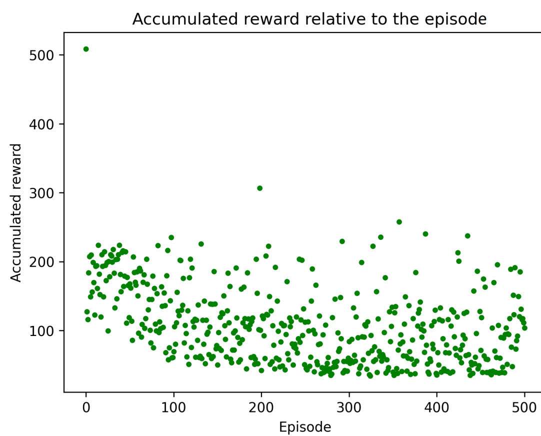
Figure 6.2: Training progression of same configuration.

6.2 Learning rate and episodes

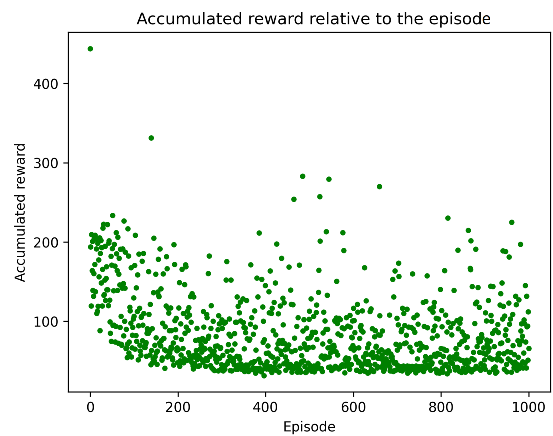
The next set of tests aims to compare different learning rates. It could also be seen in the previous graphs that each configuration starts learning the track pre 500 episodes during training on *Generated roads*. Therefore, these tests also compare training on 500 episodes vs 1000 episodes but this time on *Generated track* (no trees) with hyperparameters network 1 and batch size 64. As can be seen in Figure 6.3, not all configurations improve. In fact, only learning rate 1e-3 improves meaningfully according to the training progression, note that all graphs have different y-axis such that the shape of each graph is not enough to compare them but the scaling needs to also be considered. The average accumulated reward during the test runs is shown in Table 6.4 where the configuration with 1000 episodes and 1e-3 learning rate clearly outperforms the others. By visual inspection during testing of each configuration, the best configuration managed several laps whereas the rest did not make it past the first curve.

Model	Average accumulated reward
ddqn_net1_64_genTrack_1e2_500e	133
ddqn_net1_64_genTrack_1e2_1000e	280
ddqn_net1_64_genTrack_1e3_500e	182
ddqn_net1_64_genTrack_1e3_1000e	35 261
ddqn_net1_64_genTrack_1e4_500e	720
ddqn_net1_64_genTrack_1e4_1000e	802
ddqn_net1_64_genTrack_1e5_500e	519
ddqn_net1_64_genTrack_1e5_1000e	643

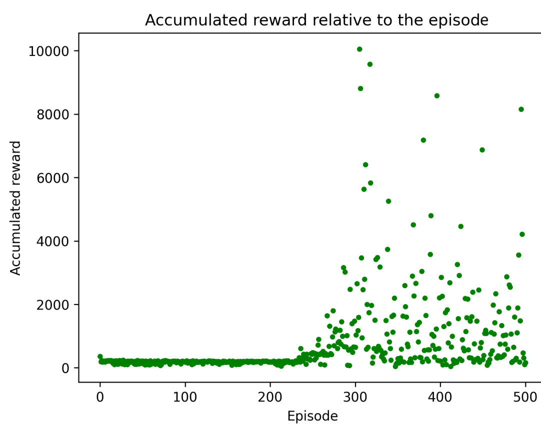
Table 6.4: Average accumulated rewards of models during test 2.



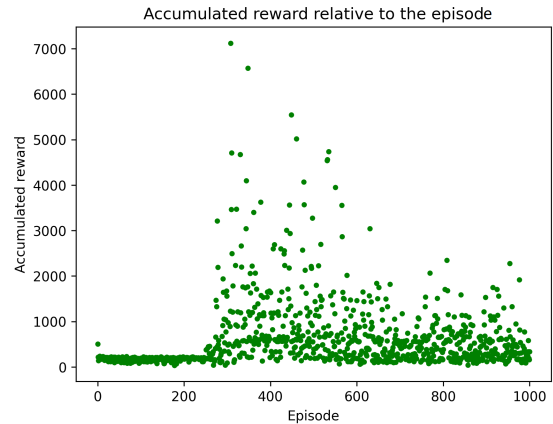
(a) Trained for 500 episodes on *Generated track* with learning rate 1e-2.



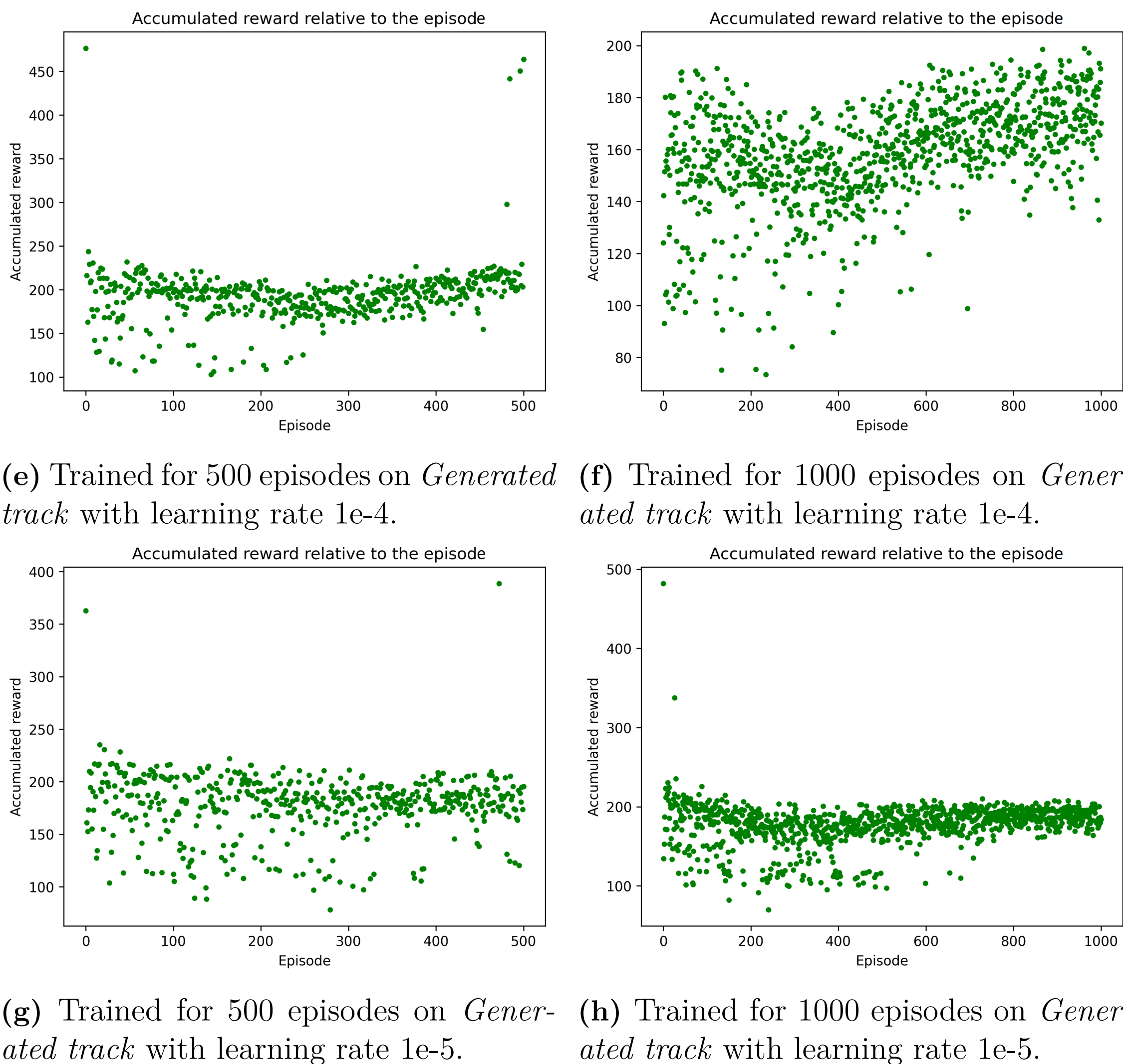
(b) Trained for 1000 episodes on *Generated track* with learning rate 1e-2.



(c) Trained for 500 episodes on *Generated track* with learning rate 1e-3.



(d) Trained for 1000 episodes on *Generated track* with learning rate 1e-3.



(e) Trained for 500 episodes on *Generated track* with learning rate 1e-4.

(f) Trained for 1000 episodes on *Generated track* with learning rate 1e-4.

(g) Trained for 500 episodes on *Generated track* with learning rate 1e-5.

(h) Trained for 1000 episodes on *Generated track* with learning rate 1e-5.

Figure 6.3: Training progression of different learning rates and episodes.

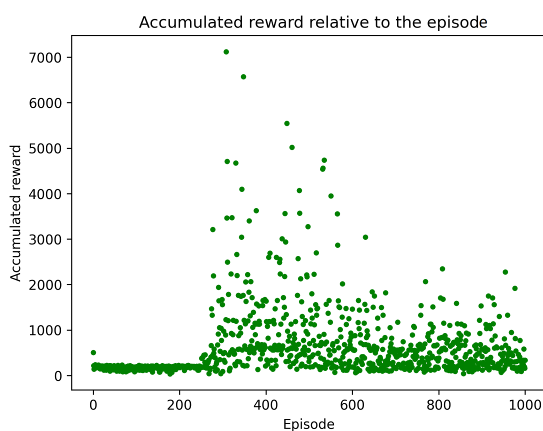
6.3 Multiple tracks

Test set number 3 aims to examine whether training on multiple tracks contributes to the generalization of input features. The generalization was believed to be different between network 1 and network 2, thus both networks were tested. The training was performed on *Generated track* followed by *Mini Monaco*, 1000 episodes on each with learning rate 1e-3. The models were then tested on both tracks and compared to models only trained *Generated track* with the same remaining configuration. Figure 6.4 shows the training progression of the models where the exploration parameters are reset in between the training sessions on different tracks. Again, the scaling on the y-axis is different in each graph and needs to be taken into consideration when comparing the models. Table 6.5 shows the test results confirming the suspicion risen when looking at the training progression, being that the models trained on both tracks struggle with learning a new track after having already been trained. Neither model trained on both tracks manage to drive past the first turn on either

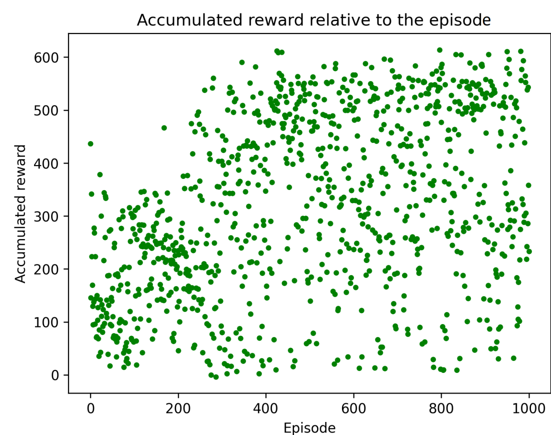
Generated track or *Mini Monaco*. Rather than improving, the models seem to perform similar to an untrained model.

Model	Test track	Avg acc reward
ddqn_net1_64_genTrack_1e3_1000e	*Generated track*	35 261
ddqn_net1_64_genTrack_1e3_1000e	*Mini Monaco*	565
ddqn_net1_64_genTrackMonaco_1e3_1000e	*Generated track*	472
ddqn_net1_64_genTrackMonaco_1e3_1000e	*Mini Monaco*	676
ddqn_net2_64_genTrack_1e3_1000e	*Generated track*	1515
ddqn_net2_64_genTrack_1e3_1000e	*Mini Monaco*	76
ddqn_net2_64_genTrackMonaco_1e3_1000e	*Generated track*	550
ddqn_net2_64_genTrackMonaco_1e3_1000e	*Mini Monaco*	1191

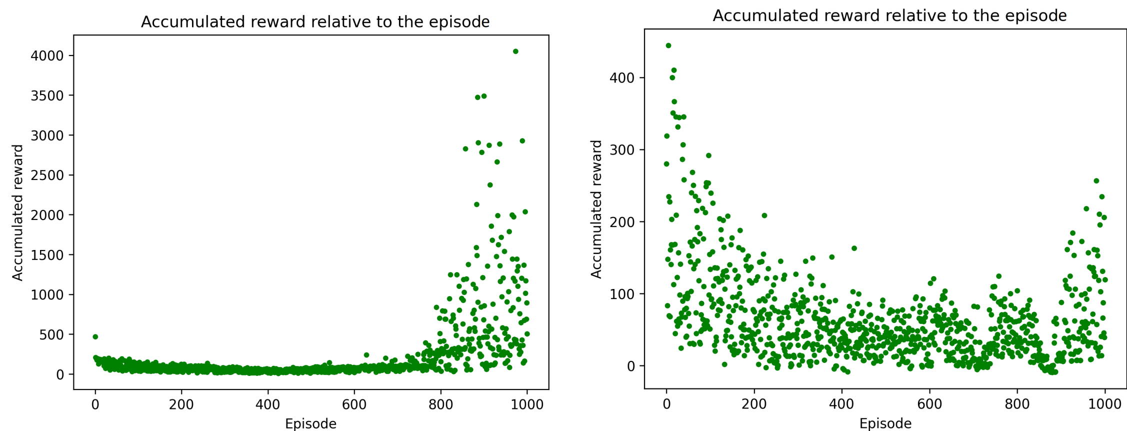
Table 6.5: Average accumulated rewards of models during test 3.



(a) Network 1 trained for 1000 episodes on *Generated track* with learning rate 1e-3. (Same as 6.3d)



(b) Network 1 trained for 1000 episodes on *Mini Monaco* by continuing on the model trained on *Generated track*.



(c) Network 2 trained for 1000 episodes on *Generated track* with learning rate 1e-3.

(d) Network 2 trained for 1000 episodes on *Mini Monaco* by continuing on the model trained on *Generated track*.

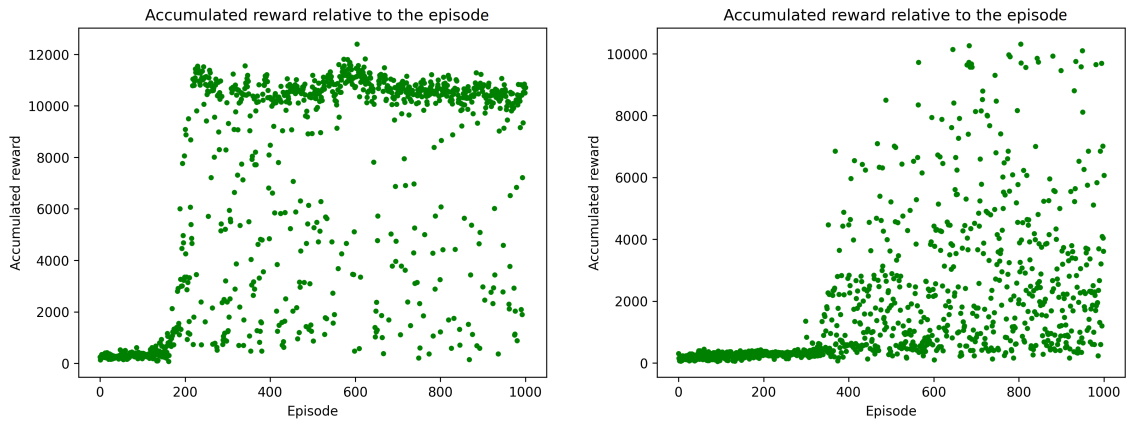
Figure 6.4: Training progression of different networks trained on different tracks.

6.4 Stricter restart condition (max CTE)

During the project's course, we noticed that many of the agents drove slightly outside the track at times. Looking into the python API, we discovered that the max CTE was the same for all tracks even though some of them had wider roads. We chose to rerun the training session of one of the better performing configurations but this time defining a stricter max CTE, more compatible with the specific track, and compare the two models to each other. The result in Table 6.6 shows that the model trained with stricter max CTE performs slightly better than its counterpart when tested on the track with both a stricter max CTE and a looser max CTE. The learning curve looks dramatically different, however, where the new model learned much slower and struggled to reach the end of the track even towards the end of its training session, see Figure 6.5.

Model	Max CTE (test)	Average accumulated reward
ddqn_net1_32_genRoads_cte8	8	34 945
ddqn_net1_32_genRoads_cte3	8	36 709
ddqn_net1_32_genRoads_cte8	3	35 991
ddqn_net1_32_genRoads_cte3	3	36 756

Table 6.6: Average accumulated rewards of models trained with different max CTE.



(a) Trained for 1000 episodes on *Generated roads* with Network 1 with batch size 32. CTE = 8.

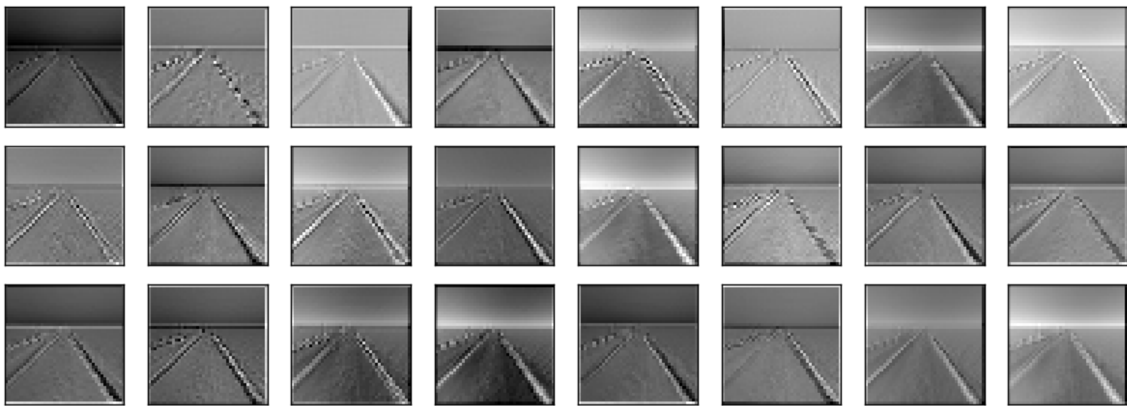
(b) Trained for 1000 episodes on *Generated roads* with Network 1 with batch size 32. CTE = 3.

Figure 6.5: Models with same hyperparameters but trained with different max CTE.

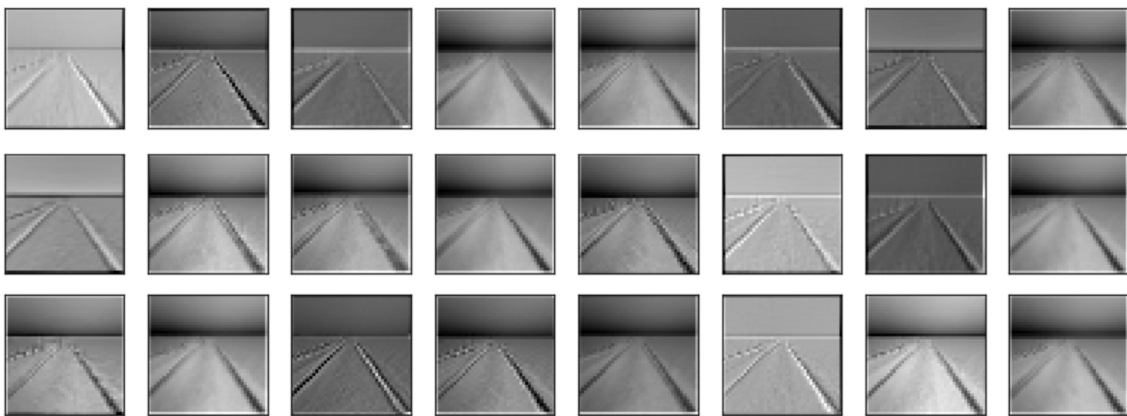
6.5 Visualization of convolutional filter outputs

Figure 6.6 shows the initial image from the car during a simulation. Figures 6.7 and 6.8 show the output after the initial image has been propagated through the first layer of both of the used networks, first before the network's training and then after training. It can be seen that both trained networks react stronger to some features in the image, in this case, the lanes.

Figure 6.6: Input image from the camera on the car.

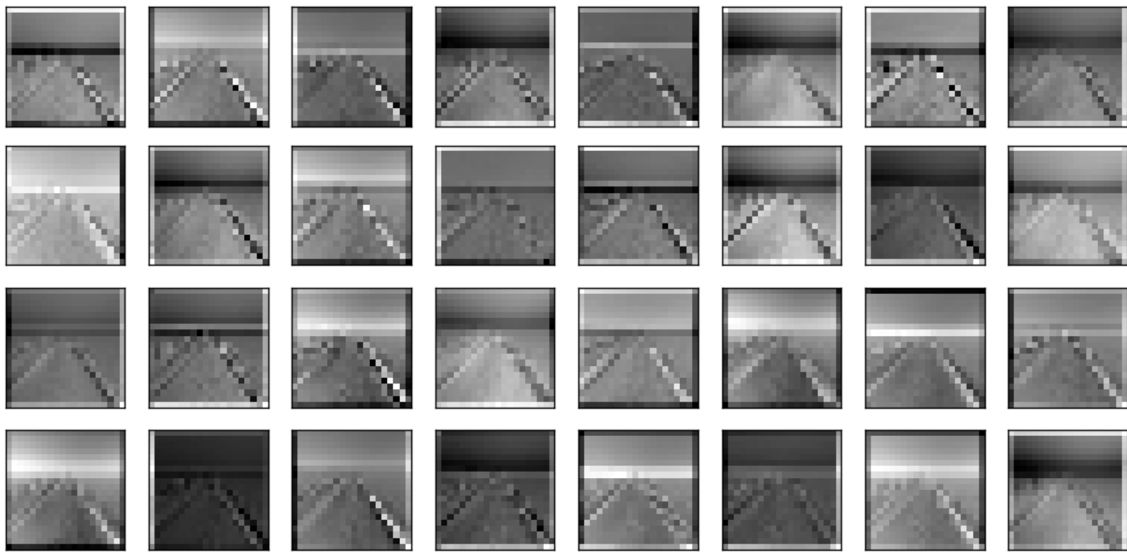


(a) Output of first convolutional layer from an untrained network 1.

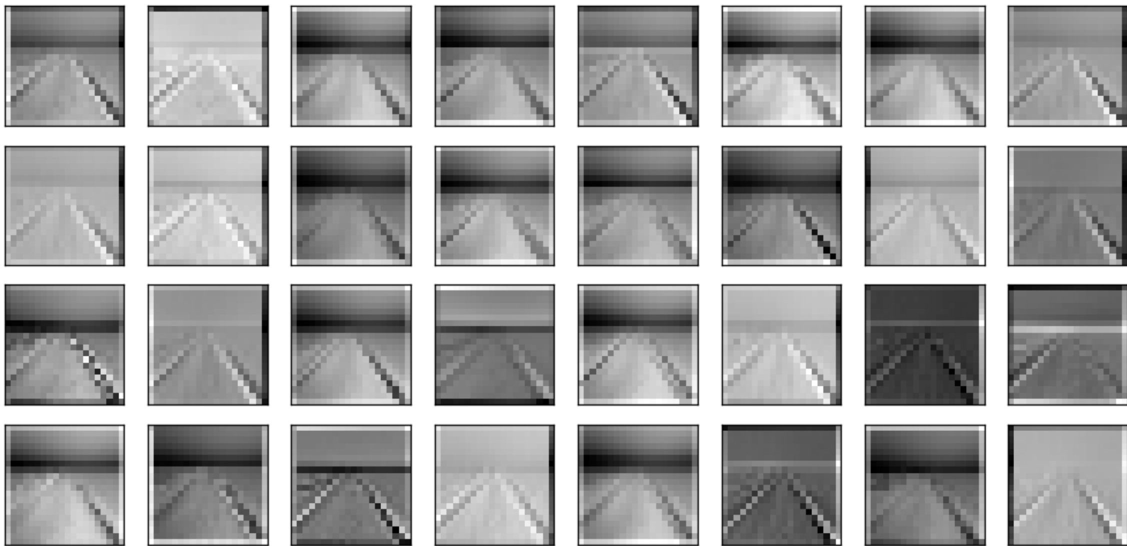


(b) Output of first convolutional layer from the model *ddqn_net1_32_genTrack*. The CNN was trained on Generated roads

Figure 6.7: Visual output of first convolutional layer from trained vs. untrained network 1.



(a) Output of first convolutional layer in an trained network 2. The CNN was trained on Generated roads



(b) Output of first convolutional layer from the model *ddqn_net2_64_genTrack*. The CNN was trained on Generated roads

Figure 6.8: Visual output of first convolutional layer from trained vs. untrained network 2.

7 Conclusion

The purpose of this project was to investigate how ML and open-source simulation tools could be used to train and test an ADS. This was made possible by first understanding what an ADS is, what existing systems there are, and how they are constructed. During this process, the concept of end-to-end systems was discovered. The end-to-end system was decided as an appropriate choice due to its simple structure and few modules to implement. The ML sub-field chosen to focus on was RL because it is a relatively new approach in the automotive industry that has shown potential and also achieved great results in video game alike environments that can be modeled as MDPs, which an AV could be. Further investigation toward ML used for ADS resulted in the choice to train CNNs and to implement the RL algorithms DDQN and DDPG. The DDQN found success when tuned properly whereas the DDPG did not. The poor performance is likely due to an error in the code rather than a problem with the algorithm itself. The investigation of simulators was conducted along with ML. The high-end but open-source simulators CARLA and SVL were initially considered. They were demonstrated to be graphically pleasing, and have great functionalities for modeling complex urban scenarios such as crossroad scenarios with multiple agents, such as pedestrians and cars, and different weather and time conditions etc., all of which are great if utilized properly. The chosen simulator to use was instead the DonkeyCar simulator. As described in Section 5.1.1, the motivation was based on a good balance of ease of use and potential for ML development. Another benefit of the DonkeyCar platform is the hardware compatibility, which increases the value and domain for further work as well as the potential for testing how agents trained in a simulator behaves in the real world.

Network 1 tended to learn faster than network 2 in the first comparison. However, the performance of both networks seemed to vary and did not yield a clear result whether either network performed better than the other after training. More tests on the network architectures are recommended to draw definitive conclusions. A higher batch size is often preferable as long as the computer hardware allows it.

In the second series of tests, on *Generated track*, it became evident that the model with learning rate 1e-3 clearly performed better than any other learning rate but needs more than 500 episodes to learn in contrast to the models trained on *Generated roads*. It is unclear exactly why that is, considering that both tracks are relatively similar with flat monochromatic background and no walls, but one possibility is that *Generated track* has sharper turns, making it more difficult to maneuver. The tests on learning rate and number of episodes also show the inconsistency of the graphs

illustrating the learning progression since the best performing model actually has a lower average accumulated reward compared to the model with the same learning rate, 1e-3, but 500 episodes.

In the third set of tests, the models seemed to overwrite or unlearn during the second training phase instead of generalizing the feature extraction to multiple tracks. Rather than performing better on both tracks, the models drove similar to an untrained model.

The fourth test set only made a comparison between two models, so conclusions should not be definitive. However, a small trend can be seen that stricter max CTE during training seem to incentivize the model to drive closer to the middle of the road and yield more reward. This is expected since having a looser restriction on the CTE allows the model to drive off-road without receiving a negative reward which affects the network's backpropagation.

The fifth test intended to visualize the feature extraction between a train and untrained CNN. Some difference could be seen between network 1 and 2 but no conclusion could be drawn by only a visual examination of the first layer. The initial thought was to show progress in the feature extraction when fine tuning the network parameters. However, the output images do not yield a clear result without deeper investigation into CNNs and how they can be optimized.

As expected, it is difficult to find the right balance of hyperparameters. Many configurations do not learn to drive properly and perform very poorly on the tests. However, some trained models manage to achieve decent driving capabilities.

7.1 Future work

During the project, many interesting ideas for improvement or investigation came up. Unfortunately, there is limited time, and training and testing ADS are an extremely time consuming process. This section is dedicated to introducing some of the ideas for future work building upon the purpose of this thesis.

Carla was for a long time a topic of interest due to its urban environments and traffic simulation. It would be interesting to see if it would be possible to teach a model more complex traffic situations such as crossing intersections with traffic lights and stopping for pedestrians at crosswalks.

Another topic of interest is other types of ML. Imitation learning is still considered a more reliable training approach for training specific traffic situations and is the primary approach by DonkeyCar users and the automotive industry. Combining imitation learning with RL by first training the model on the former and then exploring more options with RL seems like an interesting approach to expand the functionality of ADSs, also called transfer learning. They could also be compared against each other as two separate approaches.

A third idea was to design a custom environment, replicating a track in the real world, to train a model on and then test it in real world. This could also be combined with transfer learning by letting a model train on data collected from a real world car on the track and then extend the training with RL in simulator.

Further, network architectures and algorithm that enable continuous output, making it possible to train both steering and throttle, could be further investigated. Not only would it allow the agent to slow down before turns, but it would also allow the agent to reverse if it, for example, hits a wall just slightly or drives too close to the wall. To enable such an event, however, the python API would need to be modified to permit the agent to hit a wall under certain circumstances.

Bibliography

[1] E. D. Dickmanns and V. Graefe, "Dynamic monocular machine vision," *Mach. Vision Appl.*, vol. 1, no. 4, p. 223–240, oct 1988. [Online]. Available: https://doi.org/10.1007/BF01212361

[2] S. M. Grigorescu, B. Trasnea, T. T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *CoRR*, vol. abs/1910.07738, 2019. [Online]. Available: http://arxiv.org/abs/1910.07738

[3] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[4] B. Peng, Q. Sun, S. E. Li, D. Kum, Y. Yin, J. Wei, and T. Gu, "End-to-end autonomous driving through dueling double deep q-network," *Automotive Innovation*, vol. 4, no. 3, pp. 328–337, 2021.

[5] Z. Peng, J. Yang, T.-H. P. Chen, and L. Ma, *A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1240–1250. [Online]. Available: https://doi.org/10.1145/3368089.3417063

[6] M. Maurer, J. C. Gerdes, B. Lenz, and H. Winner, *Autonomous Driving: Technical, Legal and Social Aspects*, 1st ed. Springer Publishing Company, Incorporated, 2016.

[7] On-Road Automated Driving (ORAD) committee. (2021, apr) Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. [Online]. Available: https://doi.org/10.4271/J3016_202104

[8] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '18. IEEE Press, 2018, p. 287–296. [Online]. Available: https://doi.org/10.1109/ICCPS.2018.00035

[9] F. Fontana, "Self-driving cars and openpilot: A complete overview of the framework," Ph.D. dissertation, School of Industrial and Information Engineering, 2021.

[10] Consumer reports. (2020) Active driving assistance systems: Test results and design recommendations. Consumer reports, Data Intelligence. [Online]. Available: https://data.consumerreports.org/wp-content/uploads/2020/11/consumer-reports-active-driving-assistance-systems-november-16-2020.pdf

[11] Commaai, "Commaai/openpilot: Openpilot is an open source driver assistance system. openpilot performs the functions of automated lane centering and adaptive cruise control for over 150 supported car makes and models." [Online]. Available: https://github.com/commaai/openpilot

[12] V. M. Raju, V. Gupta, and S. Lomate, "Performance of open autonomous vehicle platforms: Autoware and apollo," in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, 2019, pp. 1–5.

[13] H. Ebadi, M. H. Moghadam, M. Borg, G. Gay, A. Fontes, and K. Socha, "Efficient and effective generation of test cases for pedestrian detection – search-based software testing of baidu apollo in svl," in *IEEE AITest 2021*, 2021.

[14] "Welcome to the autoware.ai wiki." [Online]. Available: https://github.com/Autoware-AI/autoware.ai/wiki

[15] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong, "Baidu apollo EM motion planner," *arXiv preprint arXiv:1807.08048*, 07 2018.

[16] K. Funaoka, "Autoware.ai (version 1.11.0)," 2019. [Online]. Available: https://github.com/Autoware-AI/autoware.ai

[17] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[19] M. Roderick, J. MacGlashan, and S. Tellex, "Implementing the deep q-network," *arXiv preprint arXiv:1711.07478*, 2017.

[20] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[21] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into imaging*, vol. 9, no. 4, pp. 611–629, 2018.

[22] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay," *arXiv preprint arXiv:1803.09820*, 2018.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[24] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[25] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[26] P. Kaur, S. Taghavi, Z. Tian, and W. Shi, "A survey on simulators for testing self-driving cars," in *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD).* IEEE, 2021, pp. 62–70.

[27] "Donkey simulator." [Online]. Available: https://docs.donkeycar.com/

[28] "Carla simulator." [Online]. Available: https://carla.org/

[29] D. Niranjan, B. VinayKarthik, and Mohana, "Deep learning based object detection model for autonomous driving research using carla simulator," in *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*. IEEE, 2021, pp. 1251–1258.

[30] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "Lgsvl simulator: A high fidelity simulator for autonomous driving," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, 2020, pp. 1–6.

[31] "Environment simulator minimalistic (esmini)." [Online]. Available: https://github.com/esmini/esmini

A
Appendix

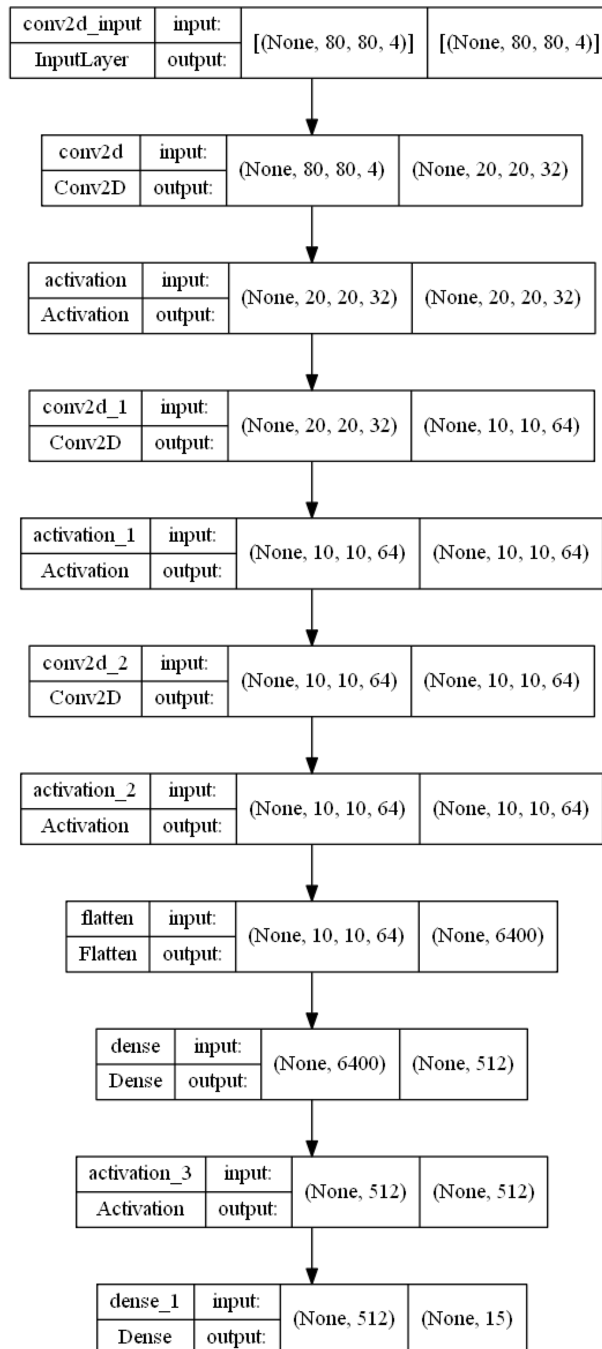


Figure A.1: Flowchart diagram of parameter and architecture of Network 2

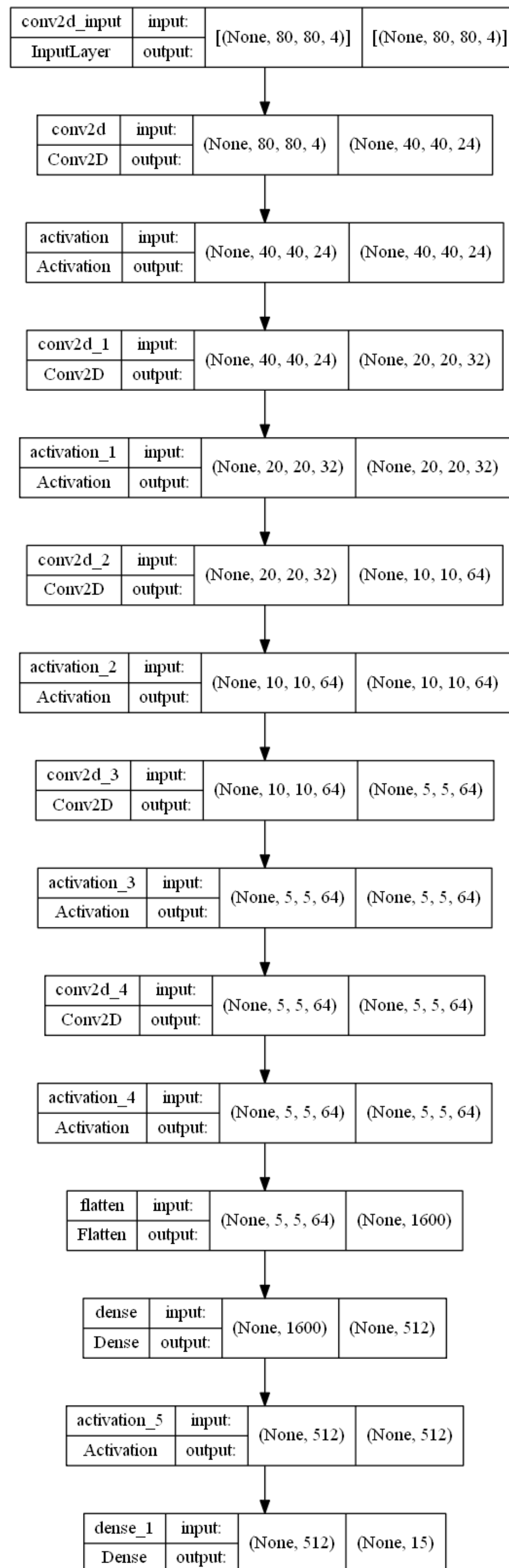


Figure A.2: Flowchart diagram of parameter and architecture of Network 1