FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Collision avoiding model for autonomous driving
Student:	Peter Kosorín
Supervisor:	Ing. Miroslav Čepek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

AWS DeepRacer cars are a platform for experimenting with autonomous driving using machine learning. The aim is to explore machine learning models and techniques for obstacle avoidance in autonomous driving. The thesis focuses on static obstacles, i.e. the ones that do not move during the driving epoch. Focus on working with a single track. For experiments and demonstration, use a simulated environment. As a last step, demonstrate the transfer of the created model into the physical car and autonomously driving it on a track.

Steps:
1) Review literature on obstacle avoidance in autonomous driving and appropriate machine learning model architectures.
2) Review the capabilities with types and placing of obstacles in the simulation environment and the model learning tools prepared by the "AWS DeepRacer Community" project.
3) Within the simulation environment, experiment with different machine learning architectures and scenarios.
4) Demonstrate the trained models in the simulated environment and summarize the capabilities of each model created.
5) Transfer the created model to a physical vehicle and test it on a test track and different obstacle layouts.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 28 February 2023 in Prague.

Bachelor’s thesis

COLLISION AVOIDANCE MODEL FOR AUTONOMOUS DRIVING

Peter Kosorin

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Miroslav Čepek, Ph.D.
May 11, 2023

Citation of this thesis: Kosorin Peter. *Collision avoidance model for autonomous driving.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I thank my supervisor, Ing. Miroslav Cepek, Ph.D., for their insight and guidance throughout this thesis. I am also grateful to my family for their unwavering support and encouragement.

Declaration

Abstract

Within this thesis, a comprehensive literature survey of various autonomous driving methodologies and machine learning model architectures has been conducted, with a particular focus on object avoidance. The thesis goes on to explore the capabilities of the AWS DeepRacer autonomous racecar platform. This platform is utilized to investigate the feasibility of training end-to-end self-driving models focused on object avoidance using reinforcement learning.

Two self-driving architectures were compared, namely a three-layer convolutional neural network and a five-layer convolutional neural network architecture. Furthermore, the impact of sensor choice on the autonomous object avoidance task was compared. Experiments in the simulated environment showed, that the three-layer convolutional neural network architecture, equipped with a stereo camera and LiDAR sensors performed the best. The model was subsequently deployed to the DeepRacer vehicle and demonstrated in the real world. The thesis successfully demonstrated the feasibility of training end-to-end autonomous models using the AWS DeepRacer platform and simulated environment.

Keywords autonomous driving, reinforcement learning, deep learning, object avoidance, AWS DeepRacer

Abstrakt

V rámci této práce byl proveden literární průzkum různých metodik autonomního řízení a architektur modelů strojového učení se zaměřením na vyhýbání se objektům. Práce dále zkoumá možnosti platformy autonomního závodního vozu AWS DeepRacer. Tato platforma je využita ke zkoumání proveditelnosti trénování end-to-end modelů autonomního řízení zaměřených na vyhýbání se objektům pomocí posilovaného učení.

Byly porovnávány dvě architektury samořídicích modelů, a to třívrstvá konvoluční neuronová síť a pětivrstvá konvoluční neuronová sít. Dále byl porovnáván vliv volby senzorů na úlohu autonomního vyhýbání se objektům. Experimenty v simulovaném prostředí ukázaly, že nejlépe si vedla architektura třívrstvé konvoluční neuronové sítě, která byla vybavena stereokamerou a LiDAR senzorem. Model byl následně nasazen do vozidla DeepRacer a demonstrován v reálném světě. Práce úspěšně prokázala proveditelnost trénování end-to-end autonomních modelů s využitím AWS DeepRacer platformy a simulovaného prostředí.

Klíčová slova autonomní řízení, posilované učení, hluboké učení, vyhýbání překážkám, AWS DeepRacer

Abbreviations

ADS	Automated Driving System
ANN	Artificial Neural Network
AWS	Amazon Web Services
CNN	Convolutional Neural Network
DL	Deep Learning
GPS	Global Positioning System
LSTM	Long Short-Term Memory
LiDAR	Light Detection and Ranging
MLP	Multi-layer Perceptron
PPO	Proximal Policy Optimization
R-CNN	Region-based Convolutional Neural Network
RL	Reinforcement Learning
RNN	Recurrent Neural Network
ReLU	Rectified Linear Unit
SAC	Soft Actor Critic
SGD	Stochastic Gradient Descent
TOF	Time Of Flight

Introduction

The topic of self-driving cars has garnered much attention in recent years, with various technological advancements paving the way for its realization. Despite this progress, however, the road to achieving fully self-driving vehicles is not without challenges. These challenges include developing sophisticated object detection and avoidance algorithms and ensuring the security and reliability of these self-driving systems. Nonetheless, the breakthroughs in artificial intelligence and related fields have led to remarkable strides in full vehicle autonomy, providing a glimpse into a future with safer and more efficient transportation.

The end goal of autonomous driving research is to develop an Automated Driving System (ADS) that can operate without human input, while simultaneously increasing passenger safety and driver efficiency while decreasing accidents caused by human error. If widespread deployment can be realized, the annual social benefits of ADSs are projected to reach nearly $800 billion by 2050 in the US alone through congestion mitigation, road casualty reduction, decreased energy consumption, and increased productivity caused by the reallocation of driving time. [1]

Current automobile manufacturers see self-driving software as a key differentiator in the automotive industry, further adding legitimacy to the claim that vehicle autonomy is achievable with current or near-future technologies. While researchers and manufacturers may adopt different methodologies to tackle the self-driving problem, common practices have emerged in the field. Historically, ADSs have divided the task of autonomous driving into subcategories and attempted to solve each independently. More recently end-to-end driving emerged as an alternative to modular approaches where various artificial intelligence models have become a dominant part of many of the proposed solutions. [2]

The objective of this thesis is to conduct literature research and provide a comprehensive survey of various autonomous driving methodologies, with a specific emphasis on the domain of object detection and avoidance. Next, the capabilities of the AWS DeepRacer platform and simulation environment will be explored. The AWS DeepRacer platform will be utilized to explore and evaluate different self-driving architectures fit for the object avoidance task, further comparing their performance in different driving scenarios. Finally, the self-driving model will be demonstrated on a model vehicle.

Chapter 1

Theoretical background

The detection of objects, their localization, and subsequent avoidance is a crucial subdomain of the self-driving task. It is a complex problem that requires a deep understanding of the underlying principles of sensing, perception, and decision-making.

This chapter will provide a theoretical background that covers several key topics related to self-driving systems, including sensors such as cameras, radar, and LiDAR. Furthermore, deep learning techniques will be introduced, such as feedforward neural networks, convolutional neural networks, and recurrent neural networks, as these architectures serve as a foundation for many approaches discussed further. In addition, optimization algorithms such as gradient descent and other optimizers that are commonly used to train deep neural networks will be discussed. The chapter ends with an overview of reinforcement learning, as this framework will be used to train the object-avoidance model.

1.1 Sensors

If self-driving cars are to be widely adopted, they need to be equipped with an advanced perception of their surroundings to effectively navigate high-pressure situations and make appropriate decisions that prioritize safety at all times. [3] These vehicles are often equipped with a variety of sensors with the aim of building a realistic internal representation of their environment in all conditions. This section aims to establish how self-driving systems interface with their environment, and in what form raw data gets fed in for further processing.

Sensors are devices that map events or changes in the surroundings to a quantitative measurement for further processing. In general, sensors are classified into two classes based on their operational principle: **proprioceptive** and **exteroceptive**. Proprioceptive sensors, or internal state sensors, capture the internal of a system, e.g. force, angular rate, wheel load, battery voltage, etc. Whereas exteroceptive sensors, meaning external state sensors, detect information from outside of the system, such as distance to objects or light intensity. [4] This section will focus on exteroceptive sensors, as they are critical for object detection, localization, and path planning.

1.1.1 Cameras

Optical cameras are one of the more obvious sensor choices for autonomous driving, as they mimic the human optic system around which the road system has been designed and built. Camera systems detect visible light emitted and reflected from their surroundings, capturing it on a photosensitive sensor, after passing it through a lens. They are relatively inexpensive and provide high-resolution videos and images, including color and texture information of the perceived surroundings.

Autonomous vehicle systems can employ monocular cameras, binocular cameras, or a combination of both. Monocular camera systems utilize a single camera, whereas binocular camera systems use two cameras placed side by side. The disparity between the two images enables inherent depth perception. As the performance of a purely optical system is highly dependent on the environmental condition and illumination, image data is often complemented with other sensor data, to generate reliable environment representation. [5], [6]

1.1.2 LiDAR

Light detection and ranging, or LiDAR, first came into prominence in the 1960s, with areal terrain mapping being its primary application. [7] LiDAR is a remote sensing technology, used to measure distances. It works on the principle of time of flight (TOF); a pulse of laser light is sent out from the sensor and the time it takes for the reflected pulse to come back is measured. This creates a 3D representation of the environment surrounding the sensor in the form of a point cloud. [5]

The relatively high cost of LiDAR systems is offset by their performance - they offer high precision even at range and a variety of environmental conditions. This robustness along with accuracy offers great benefits, especially to autonomous vehicles. [8]

1.1.3 Radar

The origins of Radar (Radio Detection and Ranging) can be traced back to the period prior to the Second World War. The technology works by transmitting electromagnetic (EM) waves within a particular area of interest and then receiving the waves that have been scattered or reflected by objects within that area. The reflected waves are then processed to establish information about the range of the objects. Doppler shift, which is the variation in wave frequency that results from relative motion between a wave source and its targets, is also used by Radar to determine the relative speed and position of objects. [5], [9]

To summarize, radar sensors are one of the most utilized technologies in autonomous systems, and provide a reliable perception of obstacles regardless of illumination and weather conditions.

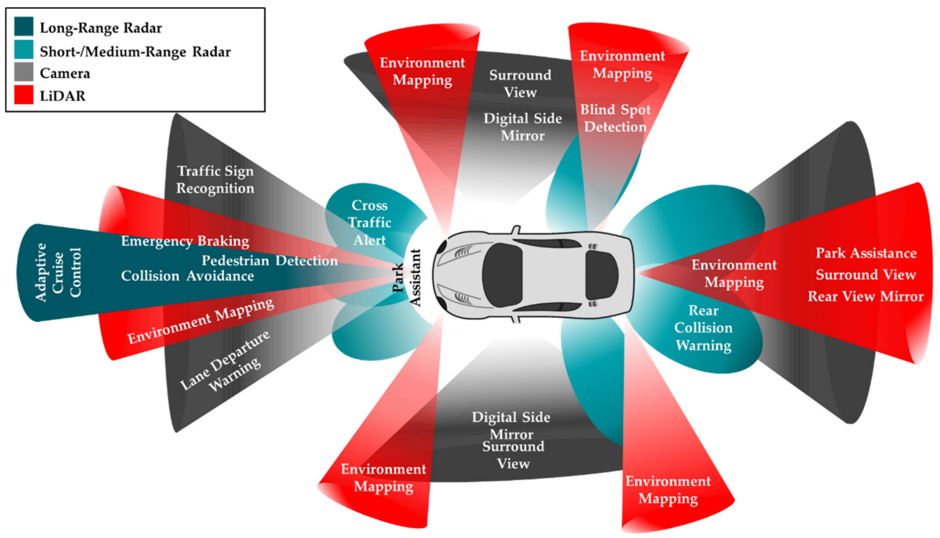

Figure 1.1 Example of the type and positioning of sensors in an autonomous vehicle. [4]

1.2 Deep learning

Broadly speaking, deep learning (DL) is a subfield of machine learning that involves building and training **artificial neural networks (ANNs)** with multiple layers. These models, which are made up of mathematical representations of interconnected processing units known as artificial neurons, are inspired by the organic brain's structure. Each connection between neurons produces signals that can be strengthened or weakened by a weight that is continuously changed throughout the learning process, similar to synapses in the brain. The key feature of DL is its inherent ability to automatically learn representations of data from raw inputs, without the need for manual feature engineering. [10], [11], [12]

1.2.1 Artificial neurons

The basic building block of all ANNs is the so-called **artificial neuron**, which mimics the function of an organic neuron. The output of this computational unit is calculated by applying an activation function to the **inner potential** ξ**:**

$$\xi = w_0 + \sum_{i=1}^{n} w_i x_i = \mathbf{w}^T \mathbf{x} + w_0 \tag{1.1}$$

where $\mathbf{x} = (x_1, \ldots, x_n)^T$ is a vector of the neuron inputs, $\mathbf{w} = (w_1, \ldots, w_n)^T$ is a vector of weights and w_0 is the bias. Finally, the activation function is applied to this sum. A model that consists of only a single neuron is also called a single-layer perceptron. Single-layer perceptrons are not particularly interesting, as they are only able to represent linear functions. [13], [14]

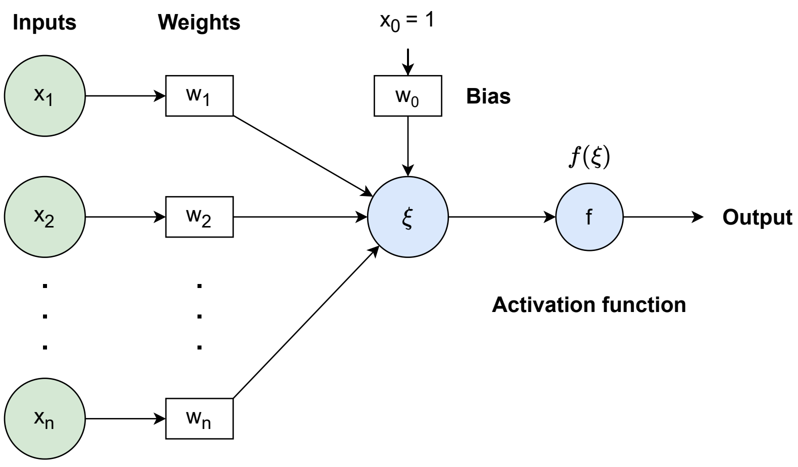

Figure 1.2 Artificial neuron diagram.

1.2.2 Feedforward neural networks

Feedforward neural networks, often also referred to as multi-layer perceptrons (MLPs) are the most prototypical deep learning models. In contrast to single-layer perceptrons, these models can approximate virtually any function. The networks are called "feedforward" since they lack feedback loops in their architecture, as opposed to networks later introduced in Subsection 1.2.4. They are composed of multiple layers of artificial neurons, where an output of a single layer serves as an input to the next layer. [10], [11]

1.2.3 Convolutional neural networks

Convolutional Neural Networks (CNNs) are analogous to traditional ANNs in that they are comprised of neurons that self-optimize through learning. They are specialized for processing data that has a grid-like structure, for example image data, which can be thought of as a 2D grid of pixels. [10], [15]

CNNs learn hierarchies of features in the data, from simple patterns in the beginning layers to continuously more complex features. An example could be detecting the edges of a shape and combining them to detect a traffic sign. One major feature that emerges from the architecture of a CNN is translation invariance – a feature can be detected regardless of where it is placed in the input image. Another benefit of CNNs over traditional fully connected ANNs is efficiency. In a traditional neural network, the output of a single neuron interacts with every other neuron in the next layer. Convolutional networks, however, typically have sparse interactions. In other words, neurons in the next layer only get inputs from the corresponding local part of the previous layer. This drastically reduces the number of connections, which in turn decreases computational complexity. [10], [16], [17]

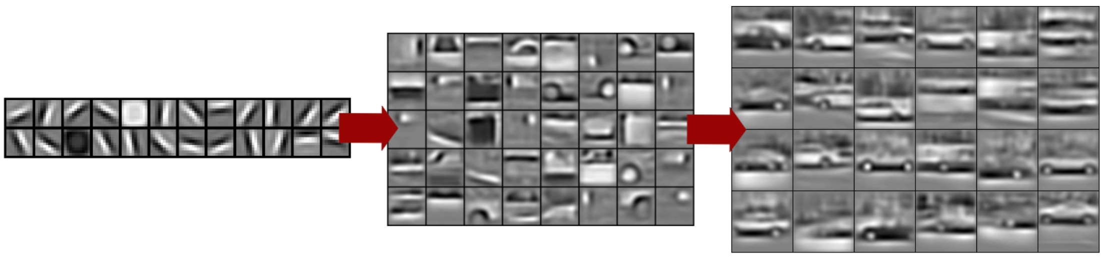

Figure 1.3 Visual representation of the hierarchy of features in a 3-layer CNN, from simple features to complex shapes. [17]

Typically, a single layer of a CNN consists of three stages (see Figure 1.4). The first stage performs convolutions in order to produce a set of linear activation outputs. In the second stage, sometimes referred to as the detector stage, outputs of the convolution are run through a nonlinear activation function, for example a rectified linear activation function. In the final stage, a pooling function is used to adjust the output further. [10], [16]

1.2.3.1 Convolution

As the name suggests, the convolution operation is the core building block in the network structure. During the forward pass, a set of learnable **kernels** ”slides” across the width and height of the input image and computes dot products between the entries of the kernel and the input at any given position. Kernels can be further specified with the **stride parameter**, which dictates how big the steps are when scanning across the input layer. As the kernel slides over the width and height of the input, the operation produces a two-dimensional activation map sometimes referred to as **feature map**. During the training process, the network will learn kernels that are active when a certain pattern is encountered, such as an edge or a color. [10], [19]

In a machine-learning context, convolution (often denoted with an asterisk) can be defined as follows:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1.2}$$

where I is the input image and K is the kernel. [10]

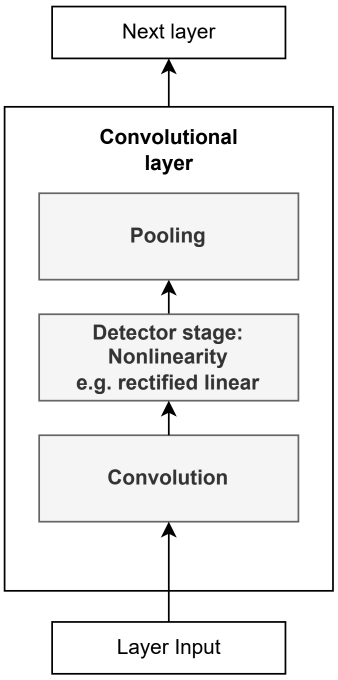

Figure 1.4 Stages of a convolutional layer. [10]

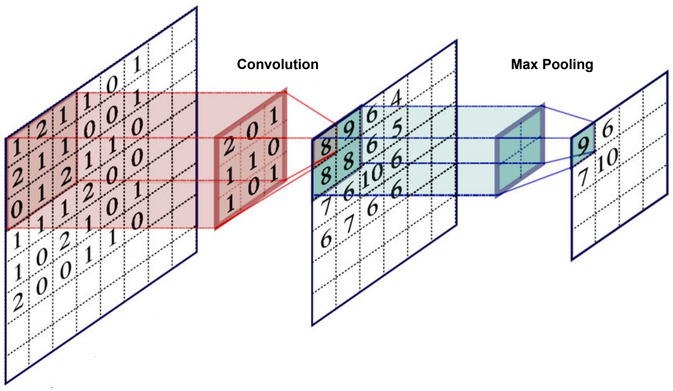

Figure 1.5 Operations carried out by the convolution and pooling stages. [18]

1.2.3.2 Nonlinearity

An element-wise, nonlinear activation function is included after the convolutional layer to cut off the output. The main idea behind introducing nonlinearity into the system is to enable the model to detect nonlinear features. Tanh and sigmoid activation functions were most common for a long time, however recently the Rectified Linear Unit (ReLU) has come into prominence because of its simple definition ($ReLU(x) = max(0, x)$), computational efficiency and its properties regarding the vanishing gradient problem. [20], [16]

1.2.3.3 Pooling

The main advantage of including the pooling operation is down-sampling. This achieves a reduction of input complexity for the following layers. For example **Max Pooling** takes a rectangular region and outputs only the maximum value in the region. An alternative to Max Pooling could be **Average Pooling**, where an average of a region is calculated. [16]

1.2.4 Recurrent neural networks

Whereas CNNs are designed to process data with a grid-like structure, recurrent neural networks (RRNs) are built to learn sequential data or data varying in time. A major limitation of traditional neural networks is their interface, they take a fixed-size vector as an input and output a fixed-size vector. RNNs operate with sequences as their input, output, or in the most general case both. RNNs have many applications in dynamic, time-dependent systems, from forecasting electric load to financial market prediction. [21], [22], [10],

Recurrent neural networks are fit for these tasks because they contain an internal state that can represent context information. This can be abstracted as including cycles in the graph of the network. These internal states allow the model to keep information about past inputs for a certain amount of time that is dependent on the weights of the model and on the input data. [23], [10]

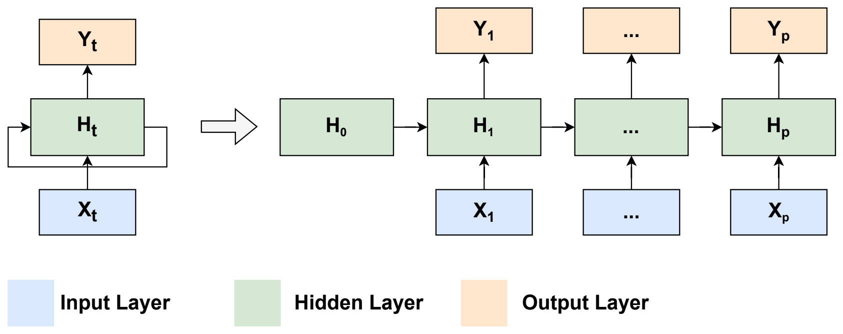

Figure 1.6 Basic RNN unfolded through time.

1.2.4.1 LSTM

LSTM (Long Short-Term Memory) networks are a type of recurrent neural network that is designed to deal with the problem of vanishing gradients in traditional RNNs. LSTM networks use a specialized memory cell, which can maintain information over a long period of time, and a set of gates to control the flow of information into and out of the cell. The key advantage of LSTM networks is their ability to handle long-term dependencies in sequential data. The memory cell allows the network to selectively store and retrieve information, while the gates control the flow of information based on the current input and the previous state of the network. [24], [25]

1.2.5 Training

Training deep learning models shares many similarities with training classical machine learning models. Typically, the average value of a loss function, which measures the prediction error given a training dataset, is minimized. The loss function L measures how much the prediction made by the model deviates from the real value. Let $(\mathbf{Y}_1, \mathbf{x}_1), \ldots, (\mathbf{Y}_N, \mathbf{x}_N)$ be the training data. We minimize:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} L(Y_i, g(\mathbf{x}_i; \mathbf{w})) \tag{1.3}$$

with respect to parameters $\mathbf{w}$ of a l-layer network, and $\mathbf{g} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_l}$ is a function which represents a forward pass of the network. Compared to many classical ML models, where a solution can oftentimes be found explicitly, nonlinearity introduced by neural network models causes the loss function to be non-convex. This necessitates the use of iterative optimizing algorithms based on gradient descent, discussed further. [10], [26]

1.2.5.1 Gradient descent

The algorithm for gradient descent can be summarized as follows:

- Let there be a neural network with parameters $\mathbf{w} = (w_1, \ldots, w_m)^T$ and α is the learning rate.
- Initialize the weights to small random values.
- Until the termination condition is met, do:

- Calculate the average prediction error for a given dataset using 1.3
- Calculate the gradient

$$\nabla_w J = (\frac{\partial J}{\partial w_1}, \ldots, \frac{\partial J}{\partial w_m}) \tag{1.4}$$

- Update the weights

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_w J \tag{1.5}$$

The learning rate hyperparameter controls the step size at which the algorithm moves toward the minimum. When a subset of the training data is used in each training episode, the algorithm is referred to as **Stochastic Gradient Descent (SGD)**. SGD is also often extended with a momentum term, which smooths out the updates and helps the algorithm avoid local minima.

1.2.5.2 AdaGrad

Adagrad (for adaptive gradient algorithm) adapts the learning rate for each parameter individually based on the history of the gradients. It assigns a larger learning rate to parameters with smaller gradients and a smaller learning rate to parameters with larger gradients. This can be especially useful for sparse data, however, it can also cause the learning rate to become too small and slow down the training. [26], [27]

1.2.5.3 RMSProp

RMSProp (for Root Mean Square Propagation) is similar to Adagrad in that it also adapts the learning rate individually, with the difference that it employs a moving average of the squared gradient instead of the sum of the squared gradient. This variation helps to circumvent the problem of the learning rate becoming overly small, which can occur with Adagrad and can lead to quicker convergence. [26]

1.2.5.4 Adam

Adam (Adaptive Moment Estimation) combines the best features of SGD with momentum and RMSProp. It adapts the learning rate for each parameter based on a running average of the gradient and the squared gradient and includes a bias correction term to account for the initialization of the running averages. Adam is a popular optimization algorithm due to its efficiency and effectiveness in many deep-learning applications. [26], [28]

1.3 Reinforcement learning

As with many modern artificial intelligence paradigms, reinforcement learning (RL) is a framework that is also inspired by nature. RL problems include an agent, sometimes also referred to as the learner, that explores a certain environment by trial and error, to achieve a given goal and maximize a reward signal in the process.

The main features that distinguish RL from other forms of learning are that RL systems are **close-loop**, meaning that the actions taken by the agent directly influence its later inputs, **not having explicit instructions** on which actions to take, and that the consequence of actions is not immediate, they can play out over an **extended time period**. [29], [30]

Most RL models consist of four key aspects [29], [31], [32]:

1. **A Reward** is a scalar value that represents the feedback from the environment to an agent's action. The reward is used to reinforce or discourage specific actions taken by the agent, therefore guiding the agent to a specific goal. The goal of the agent is to maximize the total reward it receives over a sequence of actions, often referred to as a single episode.

2. **Policy** is a function that maps states to actions taken by the agent. It is often denoted as π:

$$\pi(A_t = a | S_t = s) \tag{1.6}$$

where S_t is the state of the environment at timestep t and A_t is an action to be taken at timestep t. The policy function can be deterministic or stochastic. The main aim is to learn an optimal policy that maximizes the cumulative reward over time.

3. **A Value function** indicates which actions are preferable, taking into account further action context. It serves as an indicator to the agent how much reward it can expect to receive in the future, given its current state or the state-action pair it is in, and the policy it is following. In other words, even if a state provides a low immediate reward, it can still be preferable if followed by high-reward states. It calculates the expected cumulative rewards when starting from state s:

$$v_\pi(s) = E\{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots | S_t = s\} \tag{1.7}$$

where E represents the expected value, and γ is a discount factor, $\gamma \in [0, 1]$. If $\gamma = 0$, the agent is greedy, only taking into consideration the immediate reward, and if $\gamma = 1$ the same importance is placed on all of the future rewards.

4. **The Environment model**, although omitted in some RL frameworks, serves to predict the environment rewards and states. It is primarily used for future planning. Frameworks that utilize an environment model are often referred to as **model-based**, and ones that do not are labeled **model-free.**

The overarching idea of RL is that iterative updates are made to the policy, increasing the given value function. **Deep reinforcement learning** is obtained when deep neural networks are used to approximate the value function, policy, or environment model.

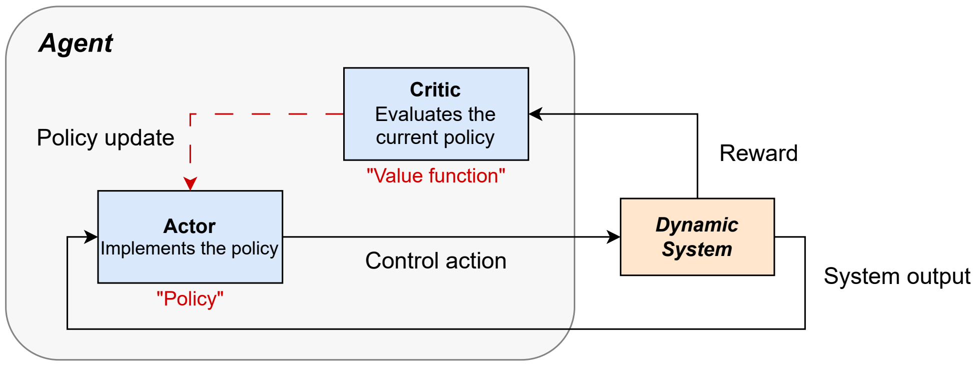

Figure 1.7 Visualization of a typical RL framework structure. Adapted from [30]

RL algorithms can be broadly divided into four classes [30], [32], [29]:

1. **Value-based methods** aim to estimate the value function directly and choose an action based on this estimate. Examples include Q-learning, Deep-Q-Networks, or SARSA.

2. **Policy-based methods** directly optimize the policy by adjusting its parameters using gradient descent or other optimization methods. Unlike value-based methods, which estimate the value function and then derive the policy from it, policy-based methods learn the policy directly. Notable examples are Proximal Policy Optimization (PPO), REINFORCE, or Trust Region Policy Optimization (TRPO)

3. **Actor-critic methods** combine value-based and policy-based methods, using a critic to estimate the value function and an actor to select actions based on the estimated values, such as Advantage Actor-Critic (A2C), Asynchronous Actor-Critic (A3C), and Soft Actor-Critic (SAC).

4. **Model-Based methods** involve building a model of the environment, which captures the dynamics of the system. This model is then used to plan and optimize the agent's behavior. Model-based methods can be used to learn optimal policies with fewer interactions with the environment than model-free methods. However, building an accurate model can be challenging, and the model itself can introduce bias and error into the learning process. Examples include Monte Carlo Tree Search or Dynamic Programming

This chapter will provide a more in-depth overview of **SAC** and **PPO**, as they are supported by the AWS DeepRacer platform later used to train the self-driving model.

1.3.1 Exploration vs exploitation

One of the biggest challenges in RL is the exploration-exploitation trade-off. **Exploration** refers to the agent's attempts to seek new actions to acquire a better understanding of the environment, in turn finding potentially more optimal actions. On the other hand, **exploitation** refers to choosing actions that are expected to yield the highest reward, using the current policy. Choosing to explore too much may lead to poor immediate rewards, while choosing to exploit excessively may lead to missing potentially better options in the future. The approach to this dilemma is one of the fundamental contrasts between RL algorithms and the choice of exploration strategy depends heavily on the environment and the specific problem domain. [29], [33]

1.3.2 Proximal policy optimization

Proximal Policy Optimization is a class of model-free, policy gradient, deep RL algorithms proposed by OpenAI in 2017. PPO aims to retain the data efficiency and reliability of TRPO-based algorithms while being simpler to implement and more fit for general use. PPO implements a novel objective with clipped probability ratios [34]:

$$L^{CLIP}(\theta) = \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}_t \right) \right] \tag{1.8}$$

where:

- θ denotes the policy parameters
- $\hat{E}_t$ is expectation over timesteps t
- r_t is the ratio of probability between the new and old policies $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
- $\hat{A}_t$ is the estimated advantage at timestep t (a measure of how much is a certain action a good or bad decision given a certain state)
- ϵ is a hyperparameter, most often 0,1 or 0,2
- the clip() function clips values from outside the interval to the interval edges

1.3.3 Soft actor critic

Algorithms such as PPO, TRPO, and other policy-based frameworks suffer from sample inefficiency – a completely new set of samples is needed after each policy update. Other off-policy methods solve this by introducing an experience replay buffer, aiming to learn from experiences acquired by older policies. However, these techniques often suffer from convergence brittleness – they are very sensitive to hyperparameters.

Soft actor critic is an actor-critic deep RL framework, which aims to solve convergence brittleness of other off-policy methods by introducing entropy. It uses a modified objective function which not only maximizes the reward received but also maximizes the entropy of the policy. This means that the actor is incentivized to succeed at the task while acting as randomly as possible. [35]

Chapter 2

Autonomous driving overview: Object avoidance

At their core, self-driving vehicles can be thought of as autonomous, decision-making agents, which use streams of observations from various onboard sensors to inform the decision-making process.

Upon reviewing the literature, two primary approaches for making the driving decisions have emerged: a **modular perception-planning-action pipeline**, which can involve the combination of artificial intelligence and deep learning techniques or classical non-learning methods, and the **end-to-end learning** strategy, where sensor data is directly mapped to control outputs. The modular pipeline is capable of supporting permutations of learning-based and non-learning-based components, such as using a deep learning-based object detector to provide input for a classical A-star path planning algorithm. [36], [37]

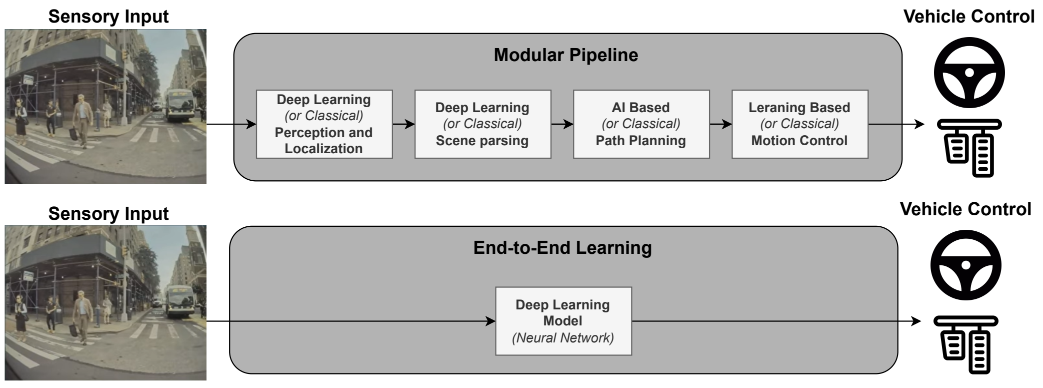
Sensory Input
Modular Pipeline
Deep Learning (or Classical) Perception and Localization
Deep Learning (or Classical) Scene parsing
AI Based (or Classical) Path Planning
Leraning Based (or Classical) Motion Control
Vehicle Control
Sensory Input
End-to-End Learning
Deep Learning Model (Neural Network)
Vehicle Control

Figure 2.1 Self-driving pipelines. Modular pipeline, comprised of various modules (top), end-to-end pipeline (bottom). Adapted from [37]

While classical perception, path planning, and motion control techniques can solve most driving scenarios, there are certain corner cases that cannot be addressed with traditional methods. These unsolved scenarios represent the limitations of classical approaches. This has prompted the use of deep learning models, which have proved to be able to solve not encountered edge cases. [36]

This chapter will provide an overview of deep learning techniques and their application in each of the subdomains of the perception-planning-action pipeline outlined in Figure 2.1. The chapter goes on to present state-of-the-art architectures fit for end-to-end driving.

2.1 Perception and scene parsing

The effective and secure operation of autonomous vehicle systems, especially in urban settings, necessitates the ability to detect a broad spectrum of objects, understand occlusions, as well as the presence of other traffic participants and drivable areas. To meet these demands, the field has increasingly turned to deep learning-based techniques, with convolutional neural networks emerging as the de facto standard for object detection and recognition. [38], [36]

2.1.1 Object Detectors

Two-dimensional object detectors can be broadly divided into **single-stage** and **double-stage** detectors.

Two-stage architectures divide the process into the region proposal stage and the classification stage. Firstly, several object candidates, also referred to as regions of interest (ROI) are proposed. Subsequently, the proposals are classified.

In contrast, **one-stage architectures** use a single feed-forward convolutional network that handles both the bounding boxes localization and object classification. [39], [40]

2.1.1.1 Two-stage Detectors

Region-based Convolutional Neural Networks (R-CNNs) [41] are at the core of most two-stage object detectors. The original R-CNN architecture proposed in 2014 uses selective search to extract regions of interest from the image, subsequently feeding each ROI through a neural network to extract features. Finally, support vector machine (SVM) based classifiers are used to classify the object. In 2015, **Fast R-CNN** [42] was proposed, improving the speed and accuracy of R-CNN by using the entire image as input to the CNN and sharing the convolutional features across the region proposals. Another performance boost was introduced later in 2015 with **Faster R-CNN** [43], which introduced integrated the ROI proposal into the CNN itself. Many later architectures were inspired by Faster R-CNN, for instance, **Feature Pyramid Networks (FPN)**. [39]

2.1.1.2 Single-stage Detectors

One of the first detectors to implement a single-stage architecture were the **The Single Shot MultiBox Detector (SSD)** [44] and **YOLO (You Only Look Once)** [45]. However, these early architectures suffered from relatively low accuracy, due to foreground-background imbalance (objects of interest are typically much smaller than the overall image, leading to bias towards the background, causing false positives). The **RetinaNet** [46] architecture tried to solve this problem by introducing a modified loss function used in SSD. Over the years, several incremental improvements were made to the YOLO family of architectures (YOLOv2 trough YOLOv5) [39] making it one of the most popular and widely used object detection algorithms in the field of computer vision.

2.1.2 Semantic Segmentation

Another approach to driving scene understanding is using **semantic segmentation**. In contrast to bounding-box-like object detection, semantic segmentation assigns a categorical label to every single pixel of the scene, forming a pixel-level segmentation map of continuous areas as can be

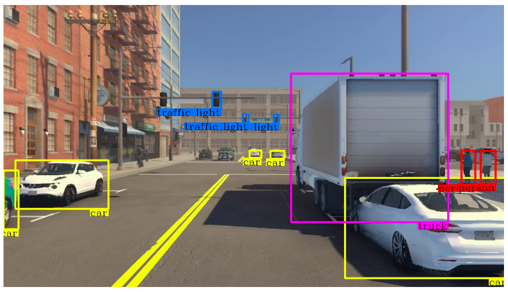

Figure 2.2 Bounding box object detection. [51]

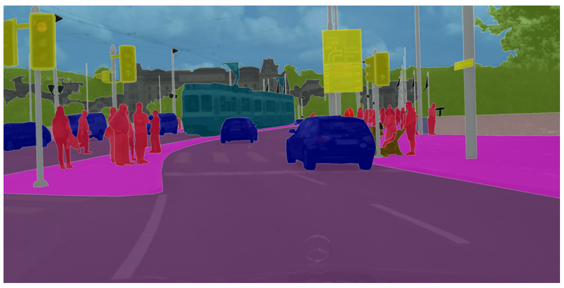

Figure 2.3 Driving scene semantic segmentation. [52]

seen in Figure 2.3. In a self-driving setting, these labels can include driveable areas, pedestrians, other cars, traffic cones, etc.

Many deep learning architectures such as **SegNet** [47], **ENet** [48], **ICNet** [49] or **Mask RCNN** [50], are based on the encoder-decoder model, with an added pixel-wise classification layer.

2.2 Path planning

The goal of path planning is to generate a feasible and collision-free trajectory between two points, taking into account all the present obstacles and the vehicle's physical constraints. Autonomous driving in a real-world setting can be seen as a multiagent problem, where the host vehicle must possess advanced negotiation abilities to ensure safe traffic flow. [53], [54]

2.2.1 Classical planning

Early self-driving vehicles demonstrated the feasibility of urban path planning in the 2007 DARPA Urban Challenge. All of the top three competitors used a three-level hierarchical planning framework composed of the mission planner, behavioral planner, and motion planner.

The **mission planner** looks at the strategic goal as a whole, making high-level decisions such as road selection. This is often achieved by classical graph search, utilizing algorithms such as A* or Djikstra's.

The main purpose of the **behavioral planner** is making real-time decisions to complete local objectives, such as lane changes or overtakes. Classical implementations often utilize finite-state machines or simple logic rules.

Finally, the **motion planner** outputs optimal paths and actions to fulfill local objectives, typically to avoid obstacles.

Variations of this hierarchical structure remain relevant in modern systems, utilizing classical path-finding algorithms and logic. However, a review of the literature reveals the recent increased interest in using deep learning in the path planning domain. [53], [54]

2.2.2 Learning-based planning

Two of the most prominent paradigms that have emerged in the learning-based local path planning domain are **Imitation Learning (IL)** and **Deep Reinforcement Learning (DRL)** based path local planners. These frameworks are also used in end-to-end driving systems further discussed in Section 2.3, however, the main difference between learning-based local planners and end-to-end driving is the output: end-to-end models map sensor input directly to vehicle control, whereas learning-based path planners output future trajectories, allowing them to be integrated into the modular self-driving pipeline presented in Figure 2.1 [55], [54], [36].

IL aims to learn optimal path planning from observed human behavior during driving. The main advantage of IL is that it can be used on real-world data, leading to organic and human-like driving trajectories. One of the disadvantages, however, is the lack of corner-cased in the training data, as human drivers rarely experience collisions or traffic rule violations. [56], [57]

DRL applied in the path planning domain pertains to finding optimal driving trajectories in a simulated environment, which is modeled after the real environment. In contrast to IL, DRL can explore various corner cases in the simulation. However, the transfer from the simulation to the real world can pose a challenge. [36]

2.3 End-to-end architectures

Modular self-driving systems, however, possess a number of disadvantages. The modular nature of the pipeline leads to information loss for subsequent components, for instance, the object detector compresses the driving scene into bounding boxes. As a result, information lost in this compression is not available for the ensuing modules. [58], [54]

In the autonomous driving context, **End-to-end architectures** abstract the entire pipeline to a single model, directly transforming the high dimensional sensor input (such as images or LiDAR point clouds) to driving commands. The predicted actions can either be the continuous operation of the steering and acceleration or chosen from a predefined set of discrete actions, for example slowing down and turning right. While the simplicity of End-to-end architectures can be conceptually appealing, the black-box nature of these models leads to problems with interpretability in case of unwanted driving decisions. [58], [54], [36]

Analogously to learning-based local planners discussed in subsection 2.2.2, two main approaches have emerged. The driving model is either trained using an RL framework or by supervised learning to mimic human behavior (via imitation learning).

2.3.1 Supervised deep learning models

The paradigm of training end-to-end driving models on human driver data was first introduced by the groundbreaking **ALVINN** [59] model proposed in 1989. This model predicted the steering wheel angle from a camera and a laser range finder using a 3-layer fully connected neural network. The next major breakthrough came in 2006, when the **Darpa Autonomous Vehicle (DAVE)** [60], utilizing a six-layer CNN and stereo cameras, demonstrated the ability to navigate through an obstacle course, after training on human-generated driving data.

However, the ever-increasing affordability of GPUs in recent years allowed for larger and more complex models to be trained. The **PilotNet** [61] architecture proposed by NVIDIA implemented a nine-layer CNN, which mapped the feed from a single front-facing camera directly to the steering control. The network trained on annotated data from a wide range of weather and lighting conditions, demonstrating the ability to follow lane markings.

Architectures taking into account the temporal context of self-driving were also studied. An FCN-LSTM model was proposed in [62], utilizing a fully convolutional encoder in conjunction with an LSTM network. Similarly, a novel C-LSTM (Convolutional Long Short-Term Memory) architecture was introduced in [63].

2.3.2 Deep reinforcement learning models

Training end-to-end models on annotated human driver data often suffer from an insufficient amount of corner cases in the training dataset. Using RL to train these models attempts to solve this by letting the agent freely explore its environment during the training phase, encountering varied edge case situations.

RL approaches are also known to have disadvantages. Compared to traditional learning methods, RL frameworks are known to be data inefficient, taking a relatively long time to converge. Another notable disadvantage is the so-called sim2real gap, describing the performance decrease when transferring the model from the simulated environment to the real world. [58], [36]

Various RL frameworks have been applied to train models. Namely, Deep Q-Learning, Proximal Policy Optimization, and asynchronous advantage ActorCritic have all been successfully used to train in simulation. [64], [65], [66] Training directly in the real world has also been explored in [67], [68]. These approaches used an independent steering controller or a safety driver, in case the RL policy deviated excessively, to prevent damage to the vehicle and surrounding property.

Chapter 3

AWS DeepRacer

The AWS DeepRacer is a platform designed by Amazon Web Services for developers to experiment with and evaluate end-to-end self-driving models. The service provides a virtual environment where a deep learning-based model can be trained on a simulated racetrack using different reinforcement learning algorithms, along with a 1:18 scale model autonomous car equipped with cameras and LiDAR, to which the trained model can be uploaded and demonstrated in the physical world.

However, mainly due to the monetary cost associated with training and evaluating models using the AWS DeepRacer service, an open-source solution for training the models on a local machine has emerged in the form of **DeepRacer-for-cloud**[1].

This chapter aims to provide an overview of the inner workings of DeepRacer, introduce the DeepRacer-for-cloud training environment, and summarize its capabilities and features, as it will be used to train and evaluate different object avoidance architectures.

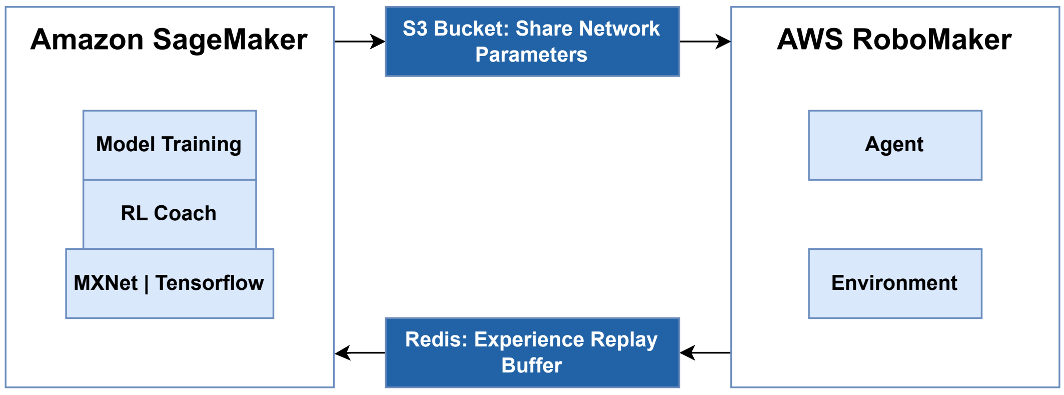

Figure 3.1 AWS DeepRacer service architecture diagram. [69]

3.1 Service architecture

The AWS DeepRacer platform is built upon various components provided by AWS, the main being **Amazon SageMaker** and **AWS RoboMaker**. **SageMaker** is a cloud-based machine

[1]https://aws-deepracer-community.github.io/deepracer-for-cloud/

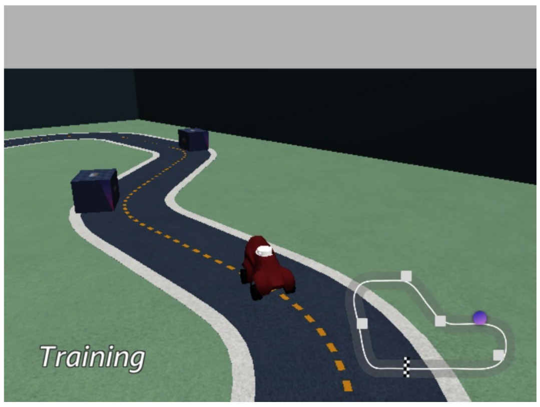

Figure 3.2 Racetrack with obstacles simulated in AWS RoboMaker.

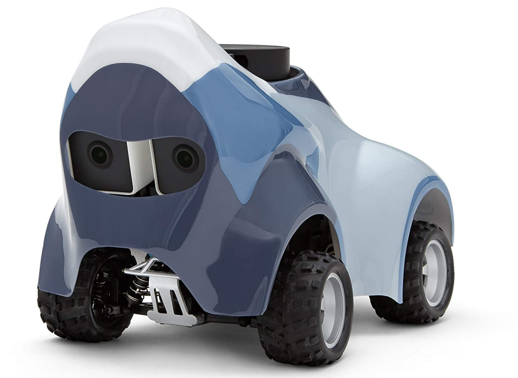

Figure 3.3 Physical DeepRacer car equipped with LiDAR and stereo cameras.

learning service, which provides an environment for building, training, and deploying machine learning models. **AWS RoboMaker** is a cloud service to develop and test robot agents in a virtual and interactive simulated environment. Other AWS cloud services, such as Amazon S3 object storage are utilized to support the DeepRacer workflow. The purpose of each of these services is further discussed in this section.

3.1.1 SageMaker

In the context of AWS DeepRacer architecture, **SageMaker** is used to train the **policy neural network** model according to which the RL agent executes driving decisions. The model is initialized with random weights, which are updated according to simulation data provided by the RoboMaker environment.

SageMaker supports established deep learning frameworks such as MXNet or TensorFlow, to enable the implementation and training of deep learning models. To enable reinforcement learning, SageMaker also supports the Intel RL Coach python framework, developed by Intel AI Lab, which contains the implementation of many state-of-the-art RL algorithms

3.1.2 AWS RoboMaker

Inside the DeepRacer architecture, **RoboMaker** is used to create a simulated environment for the agent to explore. The agent, which corresponds to the physical car, drives along a specified track, utilizing the policy network which has been trained in SageMaker up to a certain amount of time. Each simulation run in RoboMaker, known as an episode, ends with the agent in a terminal state, meaning either off the track, or crashed. Each episode is divided into time steps, where for each time step an **experience**, represented as a tuple of *(state, action, reward, new state)* is cached using the **Redis** in-memory database. This experience replay buffer is then randomly sampled by SageMaker, to update the parameters of the policy neural network. The retrained model is then stored in Amazon S3, for RoboMaker to produce more experiences. This cycle continues until a condition is met, either a set number of episodes has been run, or the average reward received reaches a certain threshold.

3.2 DeepRacer-for-cloud

DeepRacer-for-cloud (DRfC) is an open-source DeepRacer training environment, developed independently of AWS, which can be deployed using a cloud service (such as Amazon AWS or Microsoft Azure), or in the case of this thesis, on a local machine. DRfC has been developed by a community of enthusiasts that has emerged around the DeepRacer, which sought better affordability of training, along with easier customization and fine-tuning of the training algorithms and simulations. The DRfC service architecture is analogous to the original AWS DeepRacer architecture outlined in Section 3.1, however, the individual services have been replaced by dockerized versions maintained by the community. The dependence on the Amazon S3 bucket has been replaced by introducing **MinIO** – open-source object storage that can be run locally.

The following subsection aims to review the capabilities and features of DRfC, which will be utilized to train and compare object avoidance models.

3.2.1 Race types

The environment supports various racing tracks and three main race types: **Time trial**, **Object Avoidance**, and **Head-to-bot** racing.

Time trial mode is the simplest of the available training modes, the agent races on an unobstructed track, aiming to complete laps in the shortest amount of time.

The **object avoidance** mode adds static obstacles to the track in the form of boxes or other DeepRacer cars. Obstacles can either be static, meaning not changing positions for the duration of the training, or randomly placed during each training episode. The number of obstacles and the minimal distance of their spacing can also be specified.

Head-to-bot racing introduces one or more bot vehicles, which move along the track at a predefined constant speed. The bot vehicles can be enabled to change lanes at random time intervals or stay in a single lane.

3.2.2 Reward function

The reward function is a critical part of any RL-based framework – it is used to reinforce or discourage specific actions, in turn guiding the agent to a specific goal.

The DeepRacer environment implements the reward function in the form of a modifiable Python script, which is evaluated at each time step of the simulation, returning a floating point value. It takes a dictionary object `params` as an input, containing the current state of the simulation. For further specification of the `params` dictionary, see [69].

3.2.3 Supported algorithms

The training algorithms supported in Deepracer-for-cloud are identical to the original AWS DeepRacer service – **Proximal Policy Optimization** and **Soft Actor Critic**. The environment allows the specification and fine-tuning of relevant hyperparameters, presented in Table 3.1.

3.2.4 Action space

The **action space** defines the set of all valid actions the agent can take at any given time. In the case of the DeepRacer, the actions consist of **speed** and **steering angle** pairs The training environment allows the choice between **discrete** and **continuous** action spaces.

Discrete action space represents all of the possible actions explicitly as a **finite set**. The choice is limited to a set number of speed and steering action pairs, for example, speed of 0,8ms

Hyperparameter	Description
Gradient descent batch size	The number of samples taken from the experience buffer, used for updating the policy. Larger batch sizes support smoother updates and preserve underlying dependencies. **Values**: (32, 64, 128, 256, 512)
Number of epochs	The number of passes sampling the training data during gradient descent. **Values**: [3-10]
Learning rate	Controls the step size during gradient descent. **Values**: [10^{-8} - 10^{-3}]
Entropy	Determines how much randomness is added to the policy decision. Larger entropy encourages more exploration. **Values**: [0-1]
Discount factor	Specifies how much the future rewards contribute to the expected reward. **Values**: [0-1]
Loss type	Specifies the loss function used when updating the network weights. **Values**: Huber loss|Mean square error
Number of episodes between policy update	Specifies how many episodes to run between policy network updates. **Values**: [5 - 100]

Table 3.1 Tunable hyperparameters, their description, and valid values.

and steering angle of 30°. Training models with discrete action spaces oftentimes means quicker convergence time, as the model has fewer degrees of freedom.

Continuous action space allows the agent to choose actions from a predefined **range** – the entire action space is defined by the maximum and minimum steering angle and speed the agent can choose. This enables smooth changes between different speeds and steering values and generally leads to better real-world performance. Continuous actions however introduce much more choice, leading to longer model convergence time.

3.2.5 Supported Models

The DeepRacer agent is controlled by a neural network model, which selects a speed and direction based on the input from cameras or LiDAR sensors. This network also called a **policy network**, consists of three parts: **the input embedder**, **fully connected middleware**, and **action output**. The architecture is designed in a modular fashion and allows the user the modify each of the parts separately.

3.2.5.1 CNN input embedder

As outlined in Section 1.2.3, CNNs serve as feature extractors from 2D data, in this case, image input. This stage of the network is designed to convert the image input into a feature vector representation. Conceptually, this part of the model is what ”identifies” the track features, such as the lane markings or obstacles. The concrete shape of the input embedder depends on the input size, therefore on the choice of sensors.

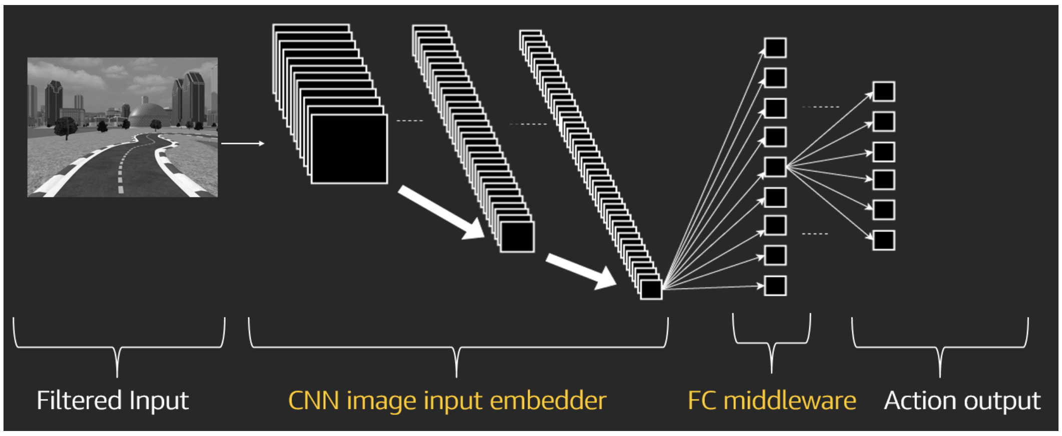

Figure 3.4 Policy network architecture. [69]

3.2.5.2 FC Middleware

The fully connected middleware layer, found after the embedder, helps to further optimize driving decisions. The training environment allows the number of layers and nodes to be specified, however, the depth of the model affects the amount of computation exponentially, in turn lengthening the training time. More depth can potentially also lead to overfitting. This part of the network is where LiDAR data comes into play, the sensor data is fed into the fully connected layers.

3.2.5.3 Action Output

The action output layer depends entirely on the action space chosen. In the case of discrete action spaces, the nodes in the output layer correspond to each available action. However, in continuous action spaces, the output layer consists of only two nodes, each corresponding to steering angle and speed.

3.3 Physical car

Once the model is trained, it can be deployed to a physical DeepRacer vehicle. The DeepRacer vehicle is a battery-powered 1:18 scale, four-wheel drive model car. It can drive autonomously by running inference based on the trained model, using the onboard compute module. For detailed hardware specifications refer to Table 3.2.

CPU	Intel Atom™ Processor
MEMORY	4GB RAM
STORAGE	32GB (expandable)
WI-FI	802.11ac
CAMERA	Stereo 4 MP cameras with MJPEG
LIDAR	360 Degree 12 Meters Scanning Radius LIDAR Sensor
SOFTWARE	Ubuntu OS 16.04.3 LTS, Intel® OpenVINO™ toolkit, ROS Kinetic
DRIVE BATTERY	7.4V/1100mAh lithium polymer
COMPUTE BATERRY	13600mAh USB-C PD
PORTS	4x USB-A, 1x USB-C, 1x Micro-USB, 1x HDMI

Table 3.2 DeepRacer vehicle hardware specification. [69]

Chapter 4
Experiments

The primary goal of this chapter is to present the experiments carried out in the development and evaluation of end-to-end self-driving models using the AWS Deepracer platform. The chapter explores two architectures, namely a 3-layer CNN and a 5-layer CNN, and investigates their performance in various scenarios, including object avoidance and head-to-head racing on a simulated racetrack. Additionally, this chapter examines the impact of different sensor combinations, including single camera, stereo cameras, and stereo cameras with LiDAR, on the training process and the driving performance of individual models.

4.1 Training setup

This section aims to outline the training conditions that were consistent across the proposed models. The training was executed using the open-source Deepracer-for-cloud environment described in Section 3.2, as it provides for more granular training customization compared to the original AWS DeepRacer service, which also carries a substantial cost burden.[1] The Deepracer-for-cloud environment was installed according to the instructions described in [70].

The Deepracer-for-cloud environment supports both Proximal policy optimization and Soft actor critic algorithms, however seeing as only PPO supports the chosen discrete action space outlined further, PPO was chosen for all the experiments conducted. Overall, most of the training runs took well over 48 hours, even with the training running on a dedicated GPU. Next, an optimal number of training iterations needed to be established for each experiment. Given the constraints of limited resources, a balance needed to be struck between allowing sufficient training time for different architectures and sensor choices to take effect, and reasonable computation times. A single training iteration is made up of many training episodes, where an episode describes a single complete driving run, where the agent starts at the simulated race track's starting line and ends with either a collision or an "off-track" event. After each iteration, the policy network was updated according to the collected experiences across the training episodes. The specific PPO hyperparameters chosen for each model are outlined in their respective sections along with the number of training iterations.

Next, the simulated environment had to be chosen. The Deepracer-for-cloud training environment allows for a wide variety of tracks to be chosen, however, the track used for training was the "A to Z speedway" shown in Figure 4.3, chosen for its relative simplicity and wide lanes, which facilitates more straightforward object avoidance. To gauge the performance of the trained models the "Smile speedway" track was chosen which can be seen in Figure 4.4.

[1]Training the models described in this thesis would cost in excess of 300 USD.

Finally, to reduce the degrees of freedom and in turn speed up convergence, a simple discrete action space was chosen. The action space consist of **two speed levels**: 0.3 m/s and 0.7 m/s and **five steering angles**: -30°, -15°, 0°, 15°, 30°, resulting in ten total available actions. This action space remained consistent across all models.

4.2 Reward function

In any RL-based framework, the reward function serves as the main incentive to guide agent behavior, as the agent aims to maximize the cumulative reward signal. The concrete reward function used to train all of the presented models, shown in Code listing 4.1, incentivizes the vehicle to stay in one of the two lanes of the road. When an obstacle is encountered in the same lane as the vehicle, the reward is continuously decreased depending on the distance of the obstacle in front of the vehicle. If the agent changes lanes, in turn avoiding the obstacle, it is rewarded. Finally, the reward for avoiding and lane keeping are added.

Code listing 4.1 Lane keeping reward function.

```
import math
def reward_function(params):
    all_wheels_on_track = params['all_wheels_on_track']
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']
    objects_location = params['objects_location']
    agent_x = params['x']
    agent_y = params['y']
    _, next_object_index = params['closest_objects']
    objects_left_of_center = params['objects_left_of_center']
    is_left_of_center = params['is_left_of_center']
    # Initialize reward with a small number but not zero
    reward = 1e-3
    # Reward if the agent stays inside one of the lanes
    if all_wheels_on_track and (0.5 * track_width - distance_from_center) >= 0.05:
        reward_lane = 1.0
    else:
        reward_lane = 1e-3
    # Penalize if the agent is too close to the next object
    reward_avoid = 1.0
    # Distance to the next object
    next_object_loc = objects_location[next_object_index]
    distance_closest_object = math.sqrt((agent_x - next_object_loc[0])**2 + (agent_y -
        next_object_loc[1])**2)
    # Decide if the agent and the next object is on the same lane
    is_same_lane = objects_left_of_center[next_object_index] == is_left_of_center
    if is_same_lane:
        if 0.5 <= distance_closest_object < 0.8:
            reward_avoid *= 0.5
        elif 0.3 <= distance_closest_object < 0.5:
            reward_avoid *= 0.2
        elif distance_closest_object < 0.3:
            reward_avoid = 1e-3 # Likely crashed
    # Calculate the reward by putting different weights on the two aspects above
    reward += reward_lane + 3.0 * reward_avoid
    return reward
```

4.3 Policy network architectures

Two policy neural network architectures were evaluated, including a 3-layer convolutional network and a more complex 5-layer convolutional neural network. Both architectures use convolutional layers as image feature extractors, followed by two fully connected layers of 512 nodes respectively, as shown in Figure 4.1.

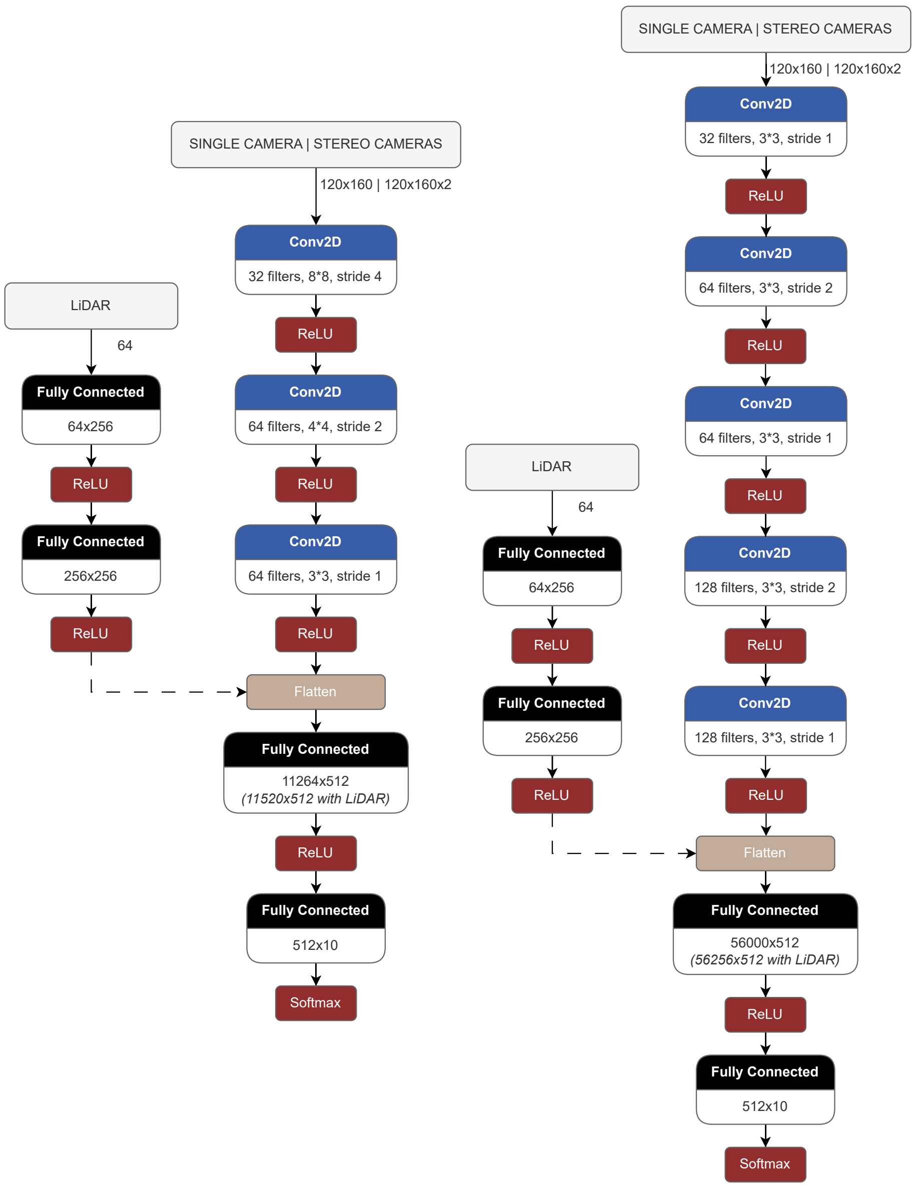

Figure 4.1 Proposed 3-layer CNN model architecture (left) and 5-layer architecture (right). If the LiDAR sensor is not present, the section embedding the LiDAR data is omitted.

The input image data is preprocessed by resizing it to 160 x 120 pixels and converting it to grayscale, before being fed into the convolutional layers. In the case of stereo cameras, the image feed is stacked into a 160 x 120 x 2 tensor to capture depth information.

If the LiDAR sensor is present, the network becomes multimodal, combining both image data with LiDAR distance measurements. LiDAR data is presented to the network in the form of a vector – the sensor partitions the space surrounding the vehicle into 8 sectors where each sector provides 8 distance measurements, resulting in a total of 64 values. This vector is then embedded using two fully connected layers before being concatenated with the flattened image data. All layers, including the dense layers, use the ReLU activation function, which helps to introduce non-linearity into the model and improve its ability to learn complex patterns in the data. The final layer uses a softmax activation function, which normalizes the output of the network to represent the distribution over the 10 possible actions the agent can take.

4.4 Sensor comparison for object avoidance

To compare the impact of sensor choice on model training and subsequently on the task of object avoidance, three training experiments were conducted using the 3-layer CNN model.

Three sensor combinations were explored: a single front-facing camera, stereo front-facing cameras, and stereo cameras with LiDAR. In each training episode, five randomly placed stationary box obstacles [2] were redistributed along the two lanes of the simulated track. Each sensor combination was trained for 100 iterations consisting of 25 episodes each, resulting in a total of 2500 episodes.

Hyperparameter	Value
Batch size	256
Number of epochs	10
Learning rate	0.00025
Loss type	Huber loss
Entropy	0.01
Discount factor	0.995
Episodes between policy update	25

Table 4.1 PPO hyperparameters used for training the 3-layer CNN.

The hyperparameter choice was mainly influenced by previous experiments, combined with experimental results provided by the AWS DeepRacer community.

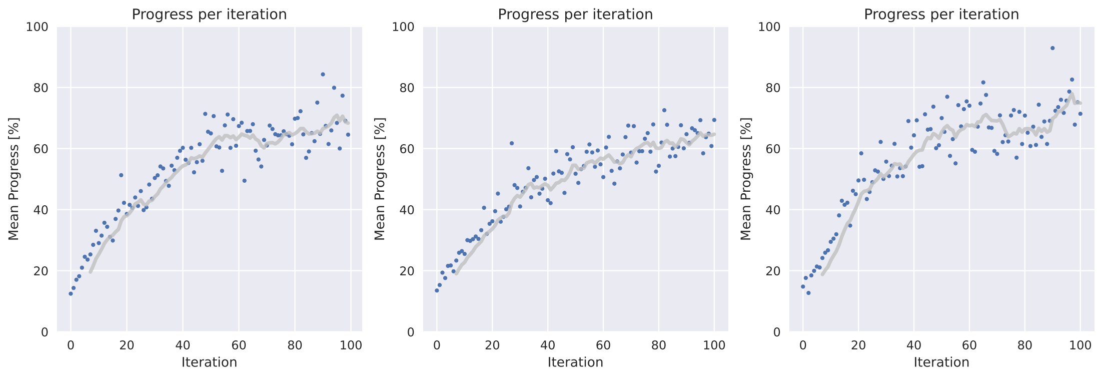

Figure 4.2 Mean iteration reward vs. iteration number for three sensor configurations: single camera (left), stereo cameras (center), and stereo cameras with LiDAR (right).

Figure 4.2 shows the mean percentage of track completion per iteration during the training progress for each sensor configuration. As training progresses, the mean reward increases for

[2]The obstacles are DeepRacer physical vehicle packaging boxes.

all sensor configurations, with the full suite of sensors showing improved performance over the single-camera and stereo-camera configurations. The stereo and single-camera setups show very similar performance. However, the rewards for the LiDAR configuration are more scattered, which could be explained by the increased model complexity when LiDAR is present.

This metric is a telling indicator of the model quality, as values near 100 indicate that almost every driving run in the iteration crossed the finish line. Stereo cameras with LiDAR also show improvement in other training metrics such as mean reward per iteration, shown in Figure A.3.

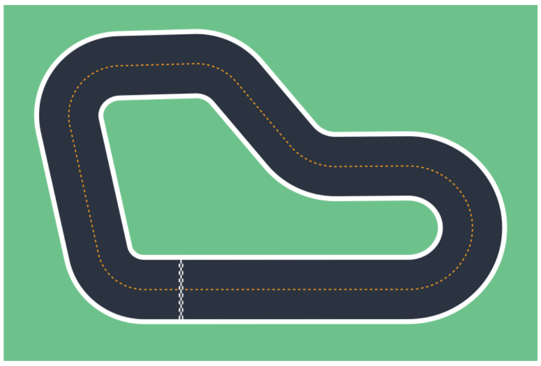

Figure 4.3 The ”A to Z speedway” track chosen for training.

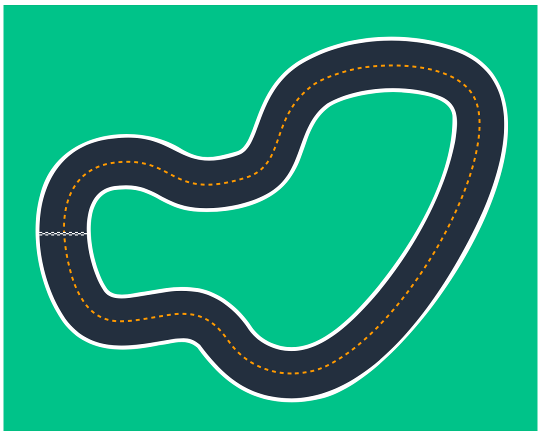

Figure 4.4 ”Smile Speedway” track used for model evaluation on an unknown track.

4.4.1 Evaluation

Following the training phase, each sensor combination was evaluated in simulation on the training track, along with a never before seen track. The evaluation consisted of three laps on both tracks, with consistent obstacle placement across all of the evaluation runs. In the event of a collision, or in case the agent left the track, the vehicle was reset to the position where the collision or an off-track event occurred. The maximum number of resets was set to ten, after which the agent was deemed unable to finish the track. The evaluation run was completed once three full laps were finished.

As is evident from the evaluation results in Table 4.2, only the configuration equipped with a LiDAR sensor completed three laps on the training track without crashing, with the single camera and stereo camera setups being reset once and twice respectively. On the evaluation track, the LiDAR-equipped model performed the best again, with the stereo camera configuration being a close second. However, the single-camera model was unable to finish the evaluation run, as the agent surpassed 10 resets.

A more comprehensive understanding of how the models make driving decisions and why the single camera model was unable to finish the evaluation can be obtained by applying the Grad-CAM technique, as proposed in [71] to analyze the image data. This method uses the gradients of the network to generate a localization map, which identifies the key regions responsible for driving decisions in the image, thereby practically producing a visualization of what the model ”looks at” at any given time.

As can be seen in 4.5, the model mainly focuses on the lane borders, as well as the dashed lane-dividing line and regions around the obstacles. However, background regions around the walls of the simulated environment are also highlighted, suggesting that the worse performance on the unknown evaluation track might be caused by the different backgrounds.

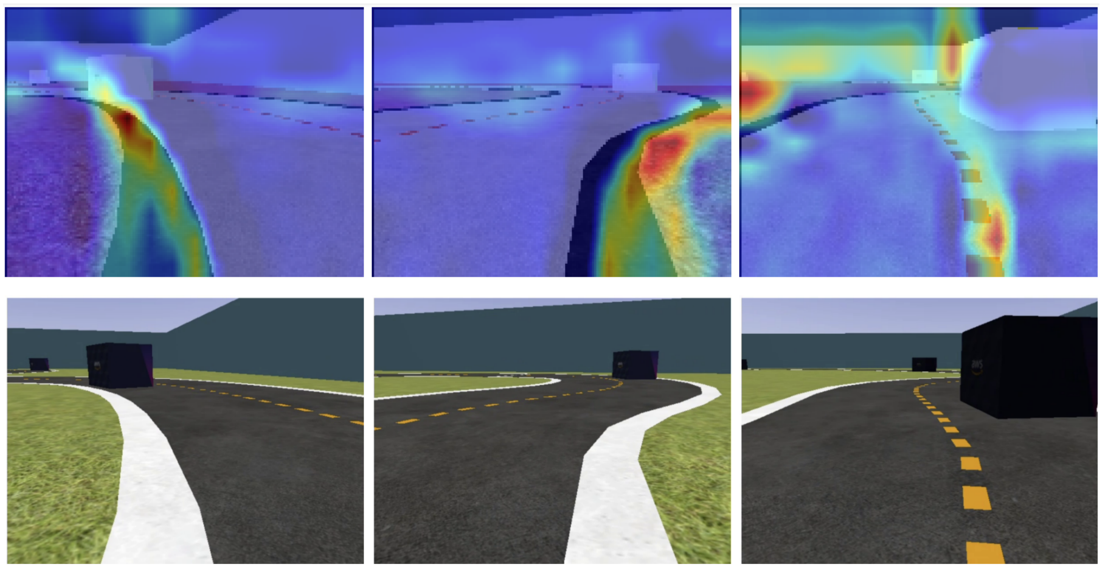

Figure 4.5 Localization maps of features considered important by the 3-layer CNN model with a single camera, obtained using the Grad-CAM method. [71]

4.5 Architecture comparison

After obtaining positive results with a 3-layer CNN architecture equipped with the full sensor suite, an experiment was conducted by training a 5-layer CNN with the same sensor combination. To make the training runs comparable, the hyperparameters were set up identically as with the 3-layer CNN, referenced in Table 4.1.

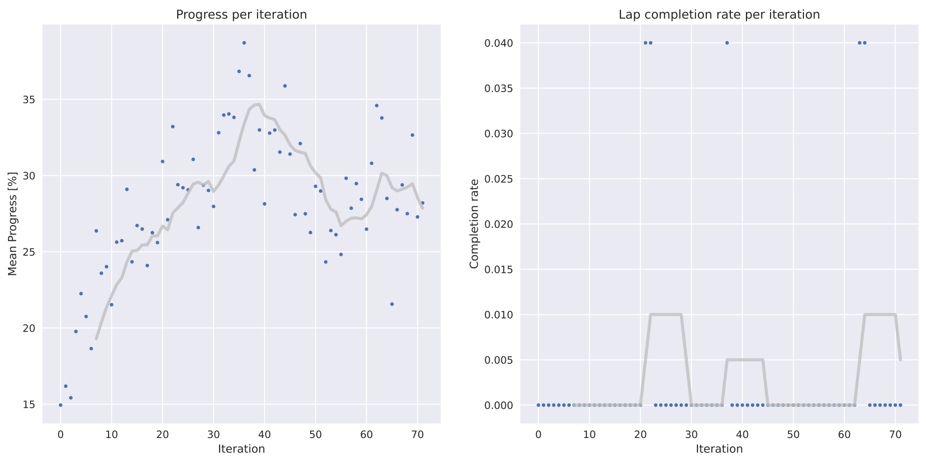

Figure 4.6 Mean iteration progress (left) and lap completion rate (right) for the 5-layer CNN model with stationary obstacles.

As evident from Figure 4.6, the training was stopped at around iteration 70, as the model showed little signs of improving since iteration 40. The mean progress at iteration 70 was only around 30%, whereas the 3-layer architecture displayed almost double the completion rate at the same iteration number. The lap completion rate for the 5-layer CNN, also shown in Figure 4.6, is almost zero throughout the training, suggesting that only a handful of episodes from 1800 conducted reached the finish line. To reach iteration 70, the model was trained on a dedicated GPU for over 50 hours.

In summary, it appears that the increased complexity of the model led to a significant decrease in performance and almost a doubling of training time compared to the simpler 3-layer architecture.

4.5.1 Evaluation

The trained 5-layer CNN model was again evaluated for three laps on the training track and a not seen before evaluation track. Unsurprisingly, the model completed the training track with 8 resets and failed to complete the evaluation track within the allotted 10 resets, indicating its incapacity to finish the track as outlined in Table 4.2.

4.6 Head-to-head racing

In head-to-head racing, the agent drives on the racetrack with a predefined number of other car agents, which drive along the track at a constant speed.

The goal of this experiment was to determine how well the proposed reward function and model translate to a more dynamic environment with moving obstacles such as moving cars.

The training environment was set up similarly to Section 4.1, however instead of randomly placed stationary obstacles, the simulation included four randomly distributed agents moving at a constant 0.3 meters per second. The model chosen for this experiment was the 3-layer CNN model with a full sensor suite, hoping that the additional LiDAR input aids the performance while overtaking other moving vehicles. The reward function and PPO hyperparameters used were identical to the previous experiments, however, the `batch_size` hyperparameter was halved to 128 with the aim of reducing computation time.

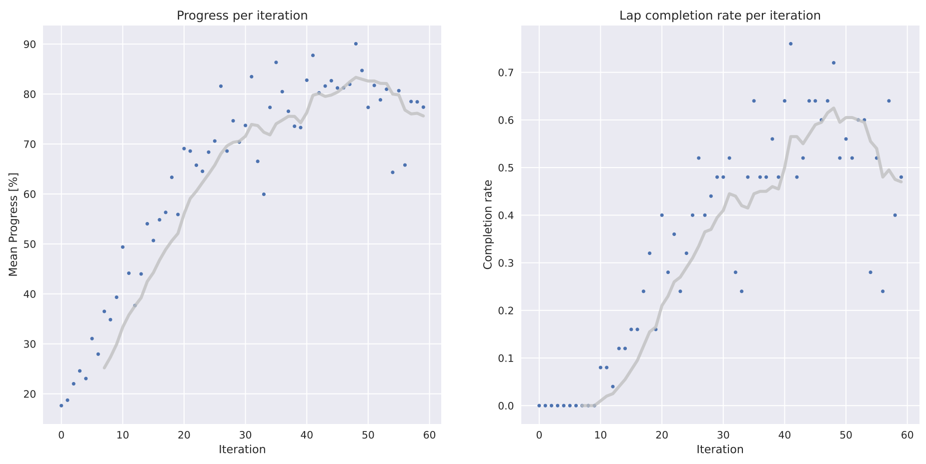

Figure 4.7 Mean iteration progress (left) and lap completion rate (right) for the 3-layer CNN model in an environment with moving obstacles.

The training was concluded at around iteration 80, as the mean reward and lap completion rate started to decrease, as is visible in Figure 4.7. This may be caused by the decreased batch size, causing the model to oscillate around a local minimum.

4.6.1 Evaluation

The evaluation was conducted in the same manner as in previous experiments – three laps were evaluated on both the training track (with four other vehicles) and a newer seen evaluation track (with seven other vehicles). As shown in Table 4.2 the model was able to successfully complete three evaluation laps on the training track without restarts, as well as the unseen track with just two. The main performance bottleneck of this model in head-to-head racing was its limited action space – the model would benefit from more action granularity and higher speed levels to better deal with the dynamic environment and execute overtaking maneuvers.

Model	Sensor choice	Race Type	Training track		Unknown track	
			Time - 3 Laps [mm:ss.s]	Resets	Time - 3 Laps [mm.ss.s]	Resets
3-Layer CNN	Single camera	Object avoidance	02:02.4	1	DNF	10
3-Layer CNN	Stereo cameras	Object avoidance	02:08.2	2	03:08.2	3
3-Layer CNN	Stereo cameras + LiDAR	Object avoidance	01:54.1	0	02:46.4	2
3-Layer CNN	Stereo cameras + LiDAR	Head-to-head	01:54.4	0	2:37.8	2
5-Layer CNN	Stereo cameras + LiDAR	Object avoidance	02:23.6	6	DNF	10

Table 4.2 Experiment evaluation metrics. DNF notes that the agent did not finish the laps within the 10 allotted restarts.

4.7 Real environment

To study the simulation-to-real gap, the trained 3-layer CNN model with a single front-facing camera was deployed to a physical DeepRacer vehicle. The real-world track was constructed out of masking tape, and foil barriers were placed around the track, in an effort to resemble the simulation as closely as possible and reduce background noise for the camera.

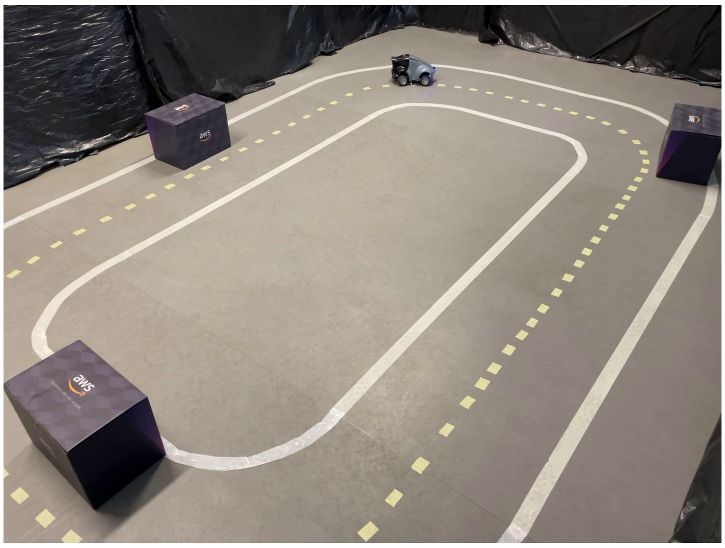

Figure 4.8 Real world demonstration track.

The model successfully navigated the track when the track was empty. However, when obstacles were introduced, the model's performance became heavily dependent on their specific placement. When obstacles were positioned along the straight sections of the track, the vehicle demonstrated the obstacle avoidance behavior learned during the simulation. When placed along the curves the vehicle struggled to avoid them in time. This may have been caused by a number of factors including the sharp turns of the track, the narrower lane width of only 60 centimeters instead of the meter in the simulation, and the relatively low contrast between the obstacles and the background barrier.

Additionally, it was observed that the model was highly sensitive to changes in the lighting conditions, and reflections from the foil barriers interfered with the driving inference. In hindsight, using non-reflective barrier material would be preferable.

Chapter 5
Conclusion

In conclusion, this thesis has achieved the set goals of conducting a comprehensive literature survey of various autonomous driving methodologies and model architectures, highlighting their limitations and use cases. Next, capabilities of the AWS DeepRacer platform and the open-source Deepracer-for-cloud training environment were explored, further utilizing these to compare and analyze different self-driving architectures for object detection and avoidance.

Through experimentation, it was discovered that a 3-layer convolutional neural network architecture combined with a full sensor suite supported by the DeepRacer vehicle performs best in a simulated object avoidance task, compared to other sensor combinations. The model was further successfully demonstrated in both static obstacle avoidance tasks and a dynamic environment with other moving vehicle agents. Training experiments were also conducted using a 5-layer convolutional network architecture, which was however found to be difficult to train due to the increased complexity. This leaves room for further experimentation with the 5-layer convolution neural network, such as systematic hyperparameter tuning and more training time. Finally, the trained model was deployed to the DeepRacer vehicle and demonstrated on a real-world racetrack, where the model displayed the capability for avoiding stationary obstacles. It was also found that the deployed model is sensitive to environmental conditions, such as background noise and light reflections. This observation highlights the need for future research to consider the transfer to the real world in the model training process and simulation design.

There are several avenues for further research and experimentation beyond the scope of this thesis. One such avenue is customizing the DeepRacer simulated environment to reproduce more complex driving tasks that were not explored in this work. Additionally, it would be beneficial to explore the impact of different reward functions on agent behavior, as this thesis relied on a single reward function for all proposed experiments. By varying the reward function, more insight could be gained into how the agent responds to different reward signals.

Appendix A
Training metrics

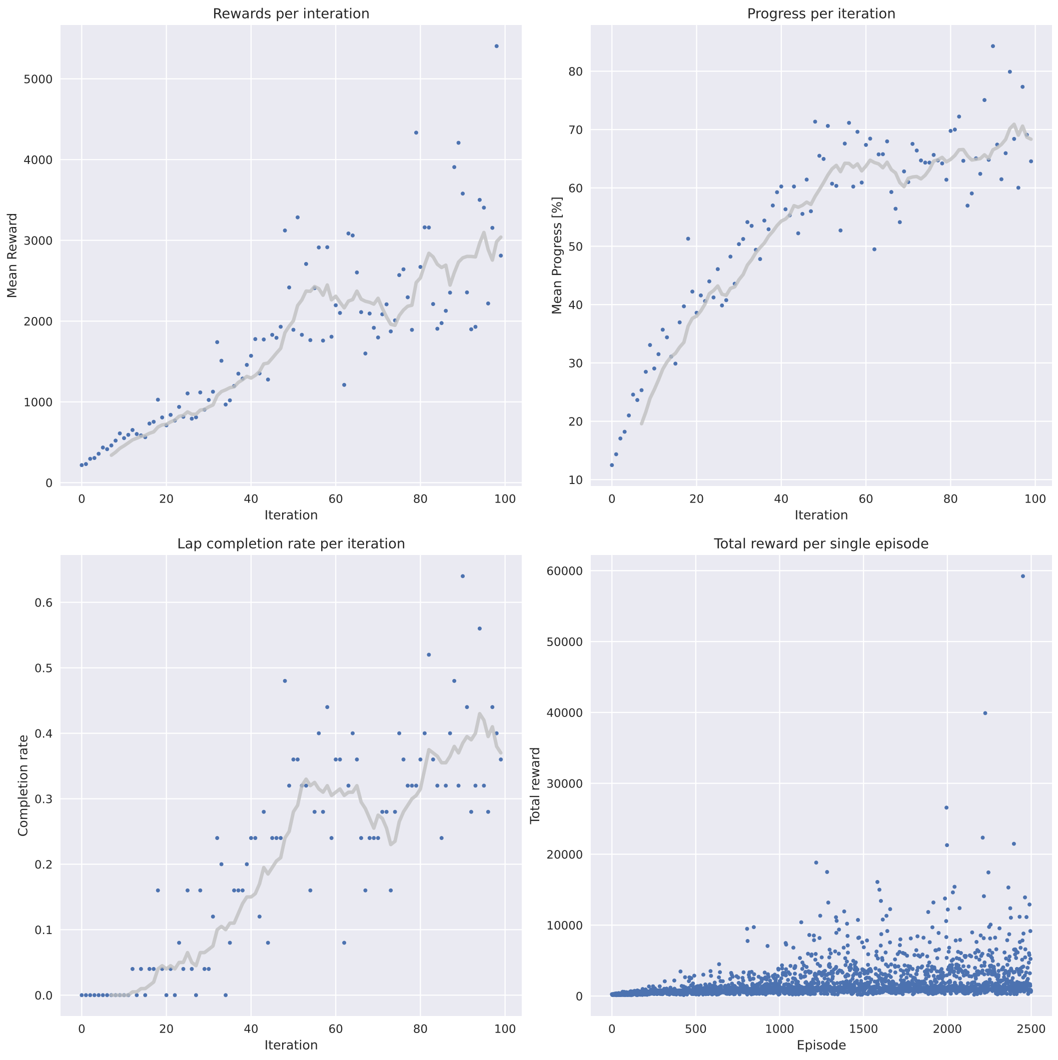

Figure A.1 Training metrics for the 3-layer CNN model with a single camera.

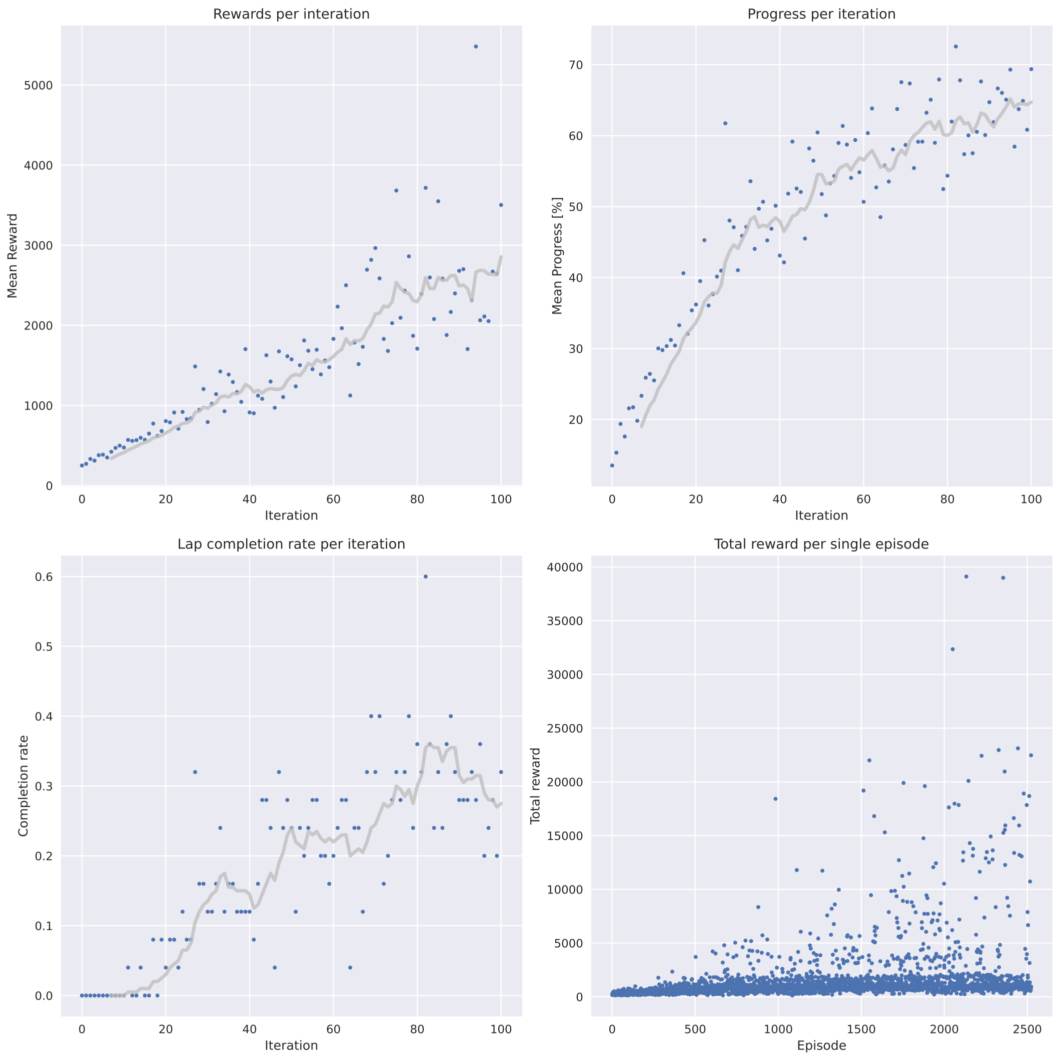

Figure A.2 Training metrics for the 3-layer CNN model stereo cameras.

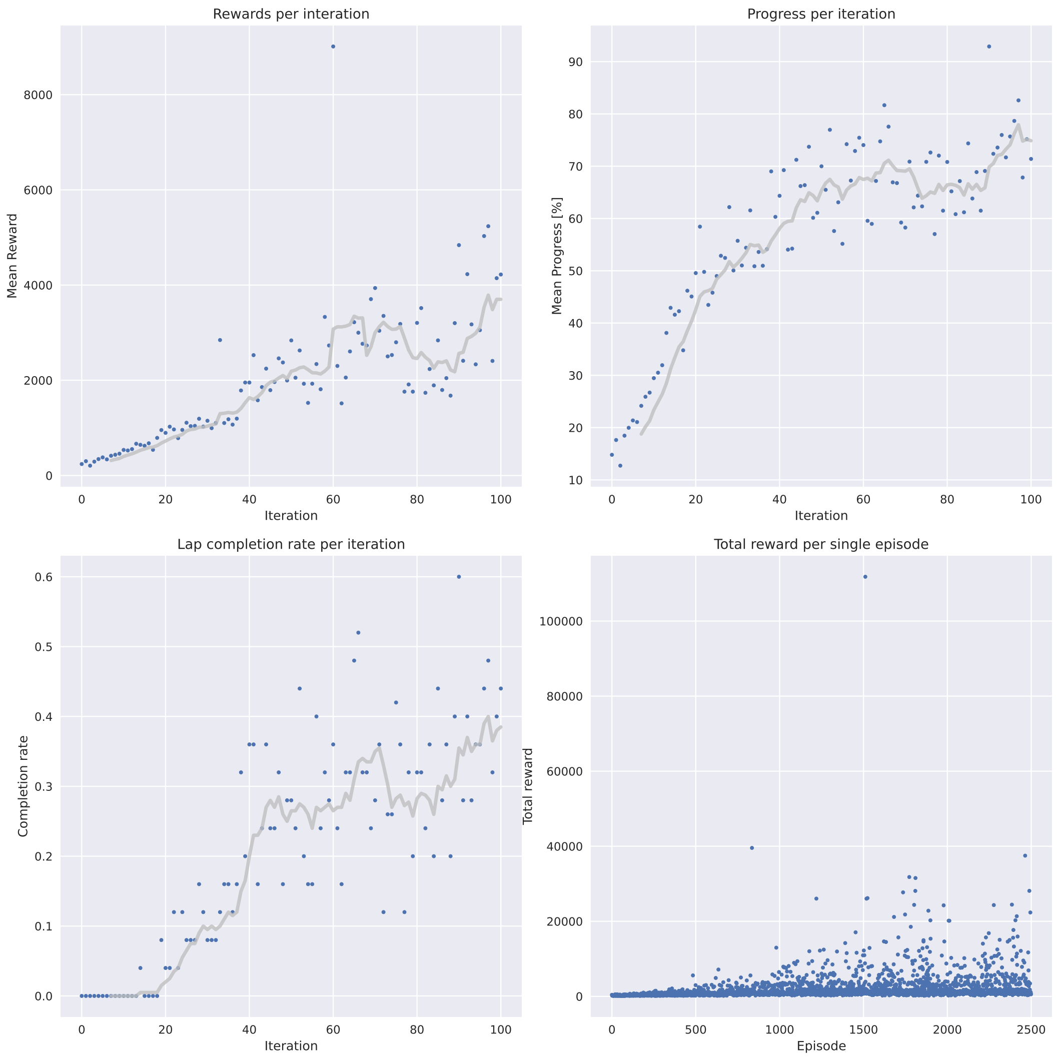

Figure A.3 Training metrics for the 3-layer CNN model with a full suite of sensors – stereo cameras and LiDAR.

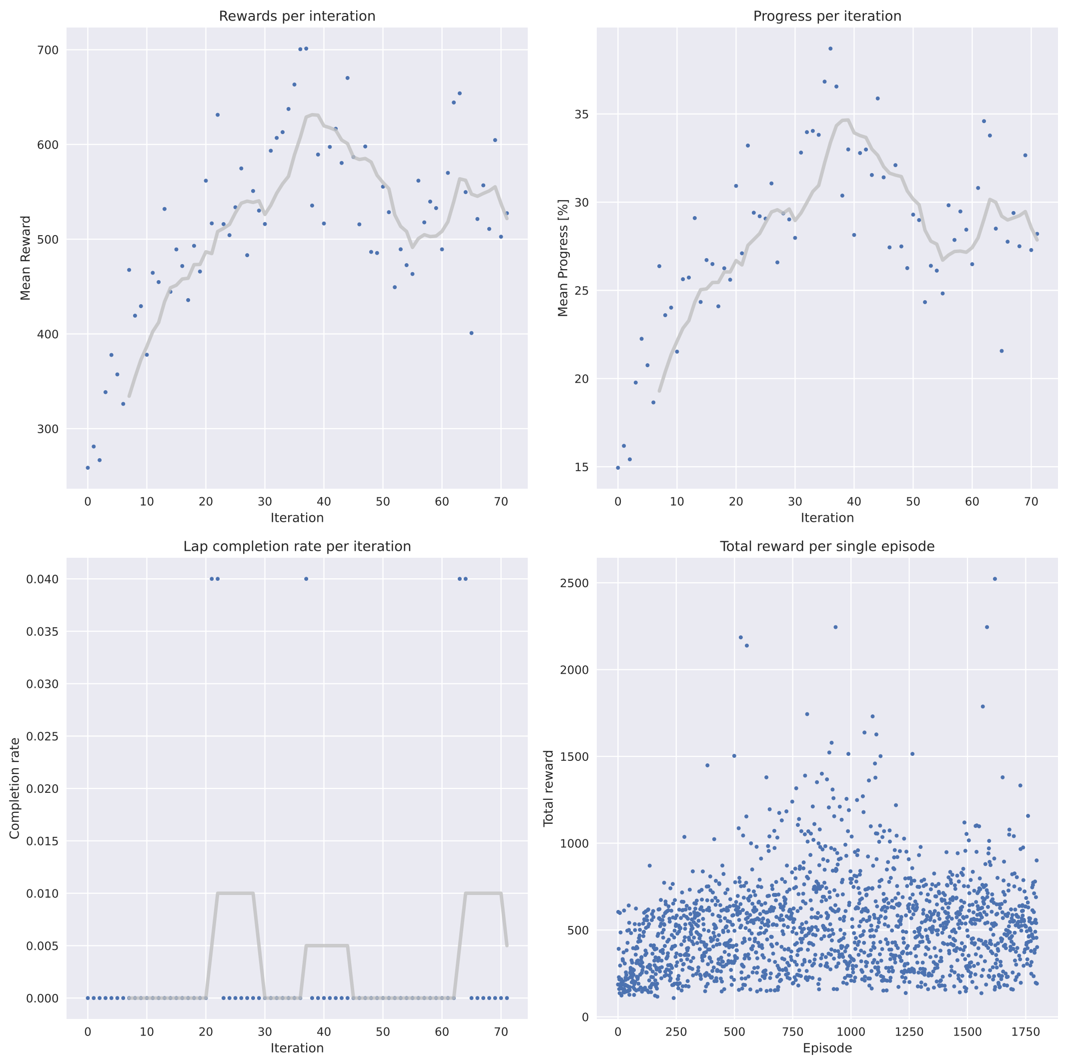

Figure A.4 Training metrics for 5-layer CNN model with stereo cameras and LiDAR.

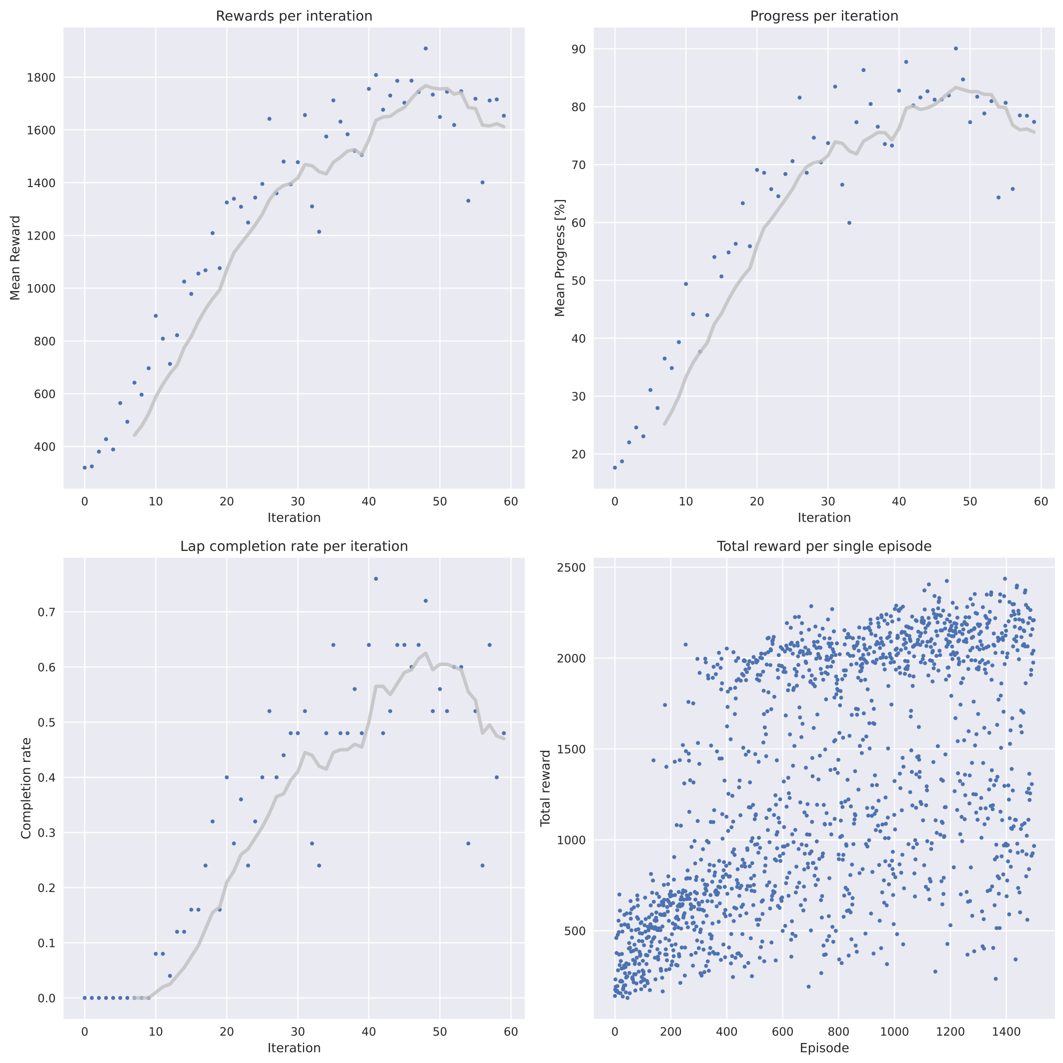

Figure A.5 Training metrics for the head-to-head racing model – 3-layer CNN with stereo cameras and LiDAR.

Bibliography

1. MONTGOMERY, W David; MUDGE, Richard; GROSHEN, Erica L; HELPER, Susan; MACDUFFIE, John Paul; CARSON, Charles. America's workforce and the self-driving future: Realizing productivity gains and spurring economic growth. 2018. Available also from: `https://trid.trb.org/view/1516782`.
2. YURTSEVER, Ekim; LAMBERT, Jacob; CARBALLO, Alexander; TAKEDA, Kazuya. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*. 2020, vol. 8, pp. 58443–58469. Available from DOI: `10.1109/ACCESS.2020.2983149`.
3. CALVERT, S. C.; SCHAKEL, W. J.; LINT, J. W. C. van. Will Automated Vehicles Negatively Impact Traffic Flow? *Journal of Advanced Transportation*. 2017, vol. 2017. ISSN 0197-6729. Available from DOI: `10.1155/2017/3082781`.
4. YEONG, De Jong; VELASCO-HERNANDEZ, Gustavo; BARRY, John; WALSH, Joseph. Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review. *Sensors*. 2021, vol. 21, no. 6. ISSN 1424-8220. Available from DOI: `10.3390/s21062140`.
5. CAMPBELL, Sean; O'MAHONY, Niall; KRPALCOVA, Lenka; RIORDAN, Daniel; WALSH, Joseph; MURPHY, Aidan; RYAN, Conor. Sensor Technology in Autonomous Vehicles : A review. In: *2018 29th Irish Signals and Systems Conference (ISSC)*. 2018, pp. 1–4. Available from DOI: `10.1109/ISSC.2018.8585340`.
6. SHAHIAN JAHROMI, Babak; TULABANDHULA, Theja; CETIN, Sabri. Real-Time Hybrid Multi-Sensor Fusion Framework for Perception in Autonomous Vehicles. *Sensors*. 2019, vol. 19, no. 20. ISSN 1424-8220. Available from DOI: `10.3390/s19204357`.
7. PETIT, F. *The Beginnings of LiDAR—A Time Travel Back in History*. 2020. Available also from: `https://www.blickfeld.com/blog/the-beginnings-of-lidar/`.
8. HECHT, Jeff. Lidar for self-driving cars. *Optics and Photonics News*. 2018, vol. 29, no. 1, pp. 26–33.
9. GAZIS, Alexandros; IOANNOU, Evangelos; KATSIRI, Elefteria. Examining the Sensors That Enable Self-Driving Vehicles. *IEEE Potentials*. 2020, vol. 39, no. 1, pp. 46–51. Available from DOI: `10.1109/MPOT.2019.2941243`.
10. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep learning*. MIT press, 2016. ISBN 9780262035613.
11. LECUN, Yann; BENGIO, Yoshua; HINTON, Geoffrey. Deep learning. *Nature*. 2015, vol. 521, no. 7553, pp. 436–444. Available from DOI: `10.1038/nature14539`.
12. SCHMIDHUBER, Jürgen. Deep learning in neural networks: An overview. *Neural Networks*. 2015, vol. 61, pp. 85–117. ISSN 0893-6080. Available from DOI: `https://doi.org/10.1016/j.neunet.2014.09.003`.

13. KRENKER, Andrej; BEŠTER, Janez; KOS, Andrej. Introduction to the artificial neural networks. *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech.* 2011, pp. 1–18.

14. JAIN, A.K.; MAO, Jianchang; MOHIUDDIN, K.M. Artificial neural networks: a tutorial. *Computer.* 1996, vol. 29, no. 3, pp. 31–44. Available from DOI: `10.1109/2.485891`.

15. O'SHEA, Keiron; NASH, Ryan. *An Introduction to Convolutional Neural Networks.* 2015. Available from arXiv: `1511.08458 [cs.NE]`.

16. ALBAWI, Saad; MOHAMMED, Tareq Abed; AL-ZAWI, Saad. Understanding of a convolutional neural network. In: *2017 International Conference on Engineering and Technology (ICET).* 2017, pp. 1–6. Available from DOI: `10.1109/ICEngTechnol.2017.8308186`.

17. LEE, Honglak; GROSSE, Roger; RANGANATH, Rajesh; NG, Andrew Y. Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations. In: Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 609–616. ISBN 9781605585161. Available from DOI: `10.1145/1553374.1553453`.

18. TEO, Yong Siah; SHIN, Seongwook; JEONG, Hyunseok; KIM, Yosep; KIM, Yoon-Ho; STRUCHALIN, Gleb I; KOVLAKOV, Egor V; STRAUPE, Stanislav S; KULIK, Sergei P; LEUCHS, Gerd, et al. Benchmarking quantum tomography completeness and fidelity with machine learning. *New Journal of Physics.* 2021, vol. 23, no. 10, p. 103021.

19. KARPATHY, Andrej et al. Convolutional neural networks for visual recognition. *Notes accompany the Stanford CS class CS231.* 2017. Available also from: `https://cs231n.github.io/`. Accessed on April 23, 2023.

20. BHATT, Dulari; PATEL, Chirag; TALSANIA, Hardik; PATEL, Jigar; VAGHELA, Rasmika; PANDYA, Sharnil; MODI, Kirit; GHAYVAT, Hemant. CNN Variants for Computer Vision: History, Architecture, Application, Challenges and Future Scope. *Electronics.* 2021, vol. 10, no. 20. ISSN 2079-9292. Available from DOI: `10.3390/electronics10202470`.

21. MEDSKER, L.; JAIN, L.C. *Recurrent Neural Networks: Design and Applications.* CRC Press, 1999. International Series on Computational Intelligence. ISBN 9781420049176. Available also from: `https://books.google.cz/books?id=ME1SAkN0PyMC`.

22. KARPATHY, Andrej. *The Unreasonable Effectiveness of Recurrent Neural Networks.* 2015. Available also from: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/dated%20May`. Accessed on April 23, 2023.

23. BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks.* 1994, vol. 5, no. 2, pp. 157–166. Available from DOI: `10.1109/72.279181`.

24. YU, Yong; SI, Xiaosheng; HU, Changhua; ZHANG, Jianxun. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. *Neural Computation.* 2019, vol. 31, no. 7, pp. 1235–1270. ISSN 0899-7667. Available from DOI: `10.1162/neco_a_01199`.

25. STAUDEMEYER, Ralf C.; MORRIS, Eric Rothstein. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks.* 2019. Available from arXiv: `1909.09586 [cs.NE]`.

26. SUN, Ruo-Yu. Optimization for deep learning: An overview. *Journal of the Operations Research Society of China.* 2020, vol. 8, no. 2, pp. 249–294. Available from DOI: `https://doi.org/10.1007/s40305-020-00309-6`.

27. DUCHI, John; HAZAN, Elad; SINGER, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research.* 2011, vol. 12, no. 7. Available also from: `https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf`.

28. KINGMA, Diederik P.; BA, Jimmy. *Adam: A Method for Stochastic Optimization.* 2017. Available from arXiv: 1412.6980 [cs.LG].

29. SUTTON, Richard S; BARTO, Andrew G. *Reinforcement learning: An introduction.* MIT press, 2018. Available also from: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

30. SHIN, Joohyun; BADGWELL, Thomas A.; LIU, Kuang-Hung; LEE, Jay H. Reinforcement Learning – Overview of recent progress and implications for process control. *Computers & Chemical Engineering.* 2019, vol. 127, pp. 282–294. ISSN 0098-1354. Available from DOI: https://doi.org/10.1016/j.compchemeng.2019.05.029.

31. KAELBLING, Leslie Pack; LITTMAN, Michael L; MOORE, Andrew W. Reinforcement learning: A survey. *Journal of artificial intelligence research.* 1996, vol. 4, pp. 237–285. Available from DOI: https://doi.org/10.1613/jair.301.

32. LI, Yuxi. *Deep Reinforcement Learning: An Overview.* 2018. Available from arXiv: 1701.07274 [cs.LG].

33. AUDIBERT, Jean-Yves; MUNOS, Rémi; SZEPESVÁRI, Csaba. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science.* 2009, vol. 410, no. 19, pp. 1876–1902. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/j.tcs.2009.01.016. Algorithmic Learning Theory.

34. SCHULMAN, John; WOLSKI, Filip; DHARIWAL, Prafulla; RADFORD, Alec; KLIMOV, Oleg. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347.* 2017. Available from DOI: https://doi.org/10.48550/arXiv.1707.06347.

35. HAARNOJA, Tuomas; ZHOU, Aurick; HARTIKAINEN, Kristian; TUCKER, George; HA, Sehoon; TAN, Jie; KUMAR, Vikash; ZHU, Henry; GUPTA, Abhishek; ABBEEL, Pieter, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905.* 2018. Available from DOI: https://doi.org/10.48550/arXiv.1812.05905.

36. GRIGORESCU, Sorin; TRASNEA, Bogdan; COCIAS, Tiberiu; MACESANU, Gigel. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics.* 2020, vol. 37, no. 3, pp. 362–386. Available from DOI: https://doi.org/10.1002/rob.21918.

37. JANAI, Joel; GÜNEY, Fatma; BEHL, Aseem; GEIGER, Andreas. Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art. *Foundations and Trends® in Computer Graphics and Vision.* 2020, vol. 12, no. 1–3, pp. 1–308. ISSN 1572-2740. Available from DOI: 10.1561/0600000079.

38. ZHU, Hao; YUEN, Ka-Veng; MIHAYLOVA, Lyudmila; LEUNG, Henry. Overview of Environment Perception for Intelligent Vehicles. *IEEE Transactions on Intelligent Transportation Systems.* 2017, vol. 18, no. 10, pp. 2584–2601. Available from DOI: 10.1109/TITS.2017.2658662.

39. CARRANZA-GARCIA, Manuel; TORRES-MATEO, Jesús; LARA-BENITEZ, Pedro; GARCIA-GUTIÉRREZ, Jorge. On the performance of one-stage and two-stage object detectors in autonomous vehicles using camera data. *Remote Sensing.* 2020, vol. 13, no. 1, p. 89. Available from DOI: https://doi.org/10.3390/rs13010089.

40. ZOU, Zhengxia; CHEN, Keyan; SHI, Zhenwei; GUO, Yuhong; YE, Jieping. Object Detection in 20 Years: A Survey. *Proceedings of the IEEE.* 2023, vol. 111, no. 3, pp. 257–276. Available from DOI: 10.1109/JPROC.2023.3238524.

41. GIRSHICK, Ross; DONAHUE, Jeff; DARRELL, Trevor; MALIK, Jitendra. Rich feature hierarchies for accurate object detection and semantic segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2014, pp. 580–587. Available also from: https://openaccess.thecvf.com/content_cvpr_2014/papers/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.pdf.

42. GIRSHICK, Ross. Fast r-cnn. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448. Available also from: https://openaccess.thecvf.com/content_iccv_2015/papers/Girshick_Fast_R-CNN_ICCV_2015_paper.pdf.

43. REN, Shaoqing; HE, Kaiming; GIRSHICK, Ross; SUN, Jian. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In: CORTES, C.; LAWRENCE, N.; LEE, D.; SUGIYAMA, M.; GARNETT, R. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2015, vol. 28. Available also from: https://proceedings.neurips.cc/paper_files/paper/2015/file/%5C%5C14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

44. LIU, Wei; ANGUELOV, Dragomir; ERHAN, Dumitru; SZEGEDY, Christian; REED, Scott; FU, Cheng-Yang; BERG, Alexander C. Ssd: Single shot multibox detector. In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37. Available from DOI: https://doi.org/10.1007/978-3-319-46448-0_2.

45. REDMON, Joseph; DIVVALA, Santosh; GIRSHICK, Ross; FARHADI, Ali. You only look once: Unified, real-time object detection. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788. Available also from: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf.

46. LIN, Tsung-Yi; GOYAL, Priya; GIRSHICK, Ross; HE, Kaiming; DOLLÁR, Piotr. *Focal Loss for Dense Object Detection*. 2018. Available from arXiv: 1708.02002 [cs.CV].

47. BADRINARAYANAN, Vijay; KENDALL, Alex; CIPOLLA, Roberto. *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. 2016. Available from arXiv: 1511.00561 [cs.CV].

48. PASZKE, Adam; CHAURASIA, Abhishek; KIM, Sangpil; CULURCIELLO, Eugenio. *ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation*. 2016. Available from arXiv: 1606.02147 [cs.CV].

49. ZHAO, Hengshuang; QI, Xiaojuan; SHEN, Xiaoyong; SHI, Jianping; JIA, Jiaya. *ICNet for Real-Time Semantic Segmentation on High-Resolution Images*. 2018. Available from arXiv: 1704.08545 [cs.CV].

50. HE, Kaiming; GKIOXARI, Georgia; DOLLÁR, Piotr; GIRSHICK, Ross. *Mask R-CNN*. 2018. Available from arXiv: 1703.06870 [cs.CV].

51. *Deploying A scalable object detection pipeline: The inferencing process, part 2*. 2022. Available also from: https://developer.nvidia.com/blog/deploying-a-scalable-object-detection-pipeline-the-inferencing-process-part-2/?fbclid=IwAR2FRM6JBA-R8DFXdq02reBbWJpbVWg3J3KxFI3hN4DsE_Le9SViUwAR0v4. Accessed on April 23, 2023.

52. ANDY CHEN, Chaitanya Asawa. *Going beyond the bounding box with semantic segmentation*. The Gradient, 2021. Available also from: https://thegradient.pub/semantic-segmentation/. Accessed on April 23, 2023.

53. PENDLETON, Scott Drew; ANDERSEN, Hans; DU, Xinxin; SHEN, Xiaotong; MEGHJANI, Malika; ENG, You Hong; RUS, Daniela; ANG, Marcelo H. Perception, Planning, Control, and Coordination for Autonomous Vehicles. *Machines*. 2017, vol. 5, no. 1. ISSN 2075-1702. Available from DOI: 10.3390/machines5010006.

54. YURTSEVER, Ekim; LAMBERT, Jacob; CARBALLO, Alexander; TAKEDA, Kazuya. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*. 2020, vol. 8, pp. 58443–58469. Available from DOI: 10.1109/ACCESS.2020.2983149.

55. CALTAGIRONE, Luca; BELLONE, Mauro; SVENSSON, Lennart; WAHDE, Mattias. LIDAR-based driving path generation using fully convolutional neural networks. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 1–6. Available from DOI: 10.1109/ITSC.2017.8317618.

56. SUN, Liting; PENG, Cheng; ZHAN, Wei; TOMIZUKA, Masayoshi. *A Fast Integrated Planning and Control Framework for Autonomous Driving via Imitation Learning*. 2018. Available from DOI: 10.1115/DSCC2018-9249. V003T37A012.

57. GRIGORESCU, Sorin Mihai; TRASNEA, Bogdan; MARINA, Liviu; VASILCOI, Andrei; COCIAS, Tiberiu. NeuroTrajectory: A Neuroevolutionary Approach to Local State Trajectory Learning for Autonomous Vehicles. *IEEE Robotics and Automation Letters*. 2019, vol. 4, no. 4, pp. 3441–3448. Available from DOI: 10.1109/LRA.2019.2926224.

58. TAMPUU, Ardi; MATIISEN, Tambet; SEMIKIN, Maksym; FISHMAN, Dmytro; MUHAMMAD, Naveed. A Survey of End-to-End Driving: Architectures and Training Methods. *IEEE Transactions on Neural Networks and Learning Systems*. 2022, vol. 33, no. 4, pp. 1364–1384. Available from DOI: 10.1109/TNNLS.2020.3043505.

59. POMERLEAU, Dean A. ALVINN: An Autonomous Land Vehicle in a Neural Network. In: TOURETZKY, D. (ed.). *Advances in Neural Information Processing Systems*. Morgan-Kaufmann, 1988, vol. 1. Available also from: https://proceedings.neurips.cc/paper_files/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

60. MULLER, Urs; BEN, Jan; COSATTO, Eric; FLEPP, Beat; CUN, Yann. Off-Road Obstacle Avoidance through End-to-End Learning. In: WEISS, Y.; SCHOLKOPF, B.; PLATT, J. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 2005, vol. 18. Available also from: https://proceedings.neurips.cc/paper_files/paper/2005/file/fdf1bc5669e8ff5ba45d02fded729feb-Paper.pdf.

61. BOJARSKI, Mariusz; TESTA, Davide Del; DWORAKOWSKI, Daniel; FIRNER, Bernhard; FLEPP, Beat; GOYAL, Prasoon; JACKEL, Lawrence D.; MONFORT, Mathew; MULLER, Urs; ZHANG, Jiakai; ZHANG, Xin; ZHAO, Jake; ZIEBA, Karol. *End to End Learning for Self-Driving Cars*. 2016. Available from arXiv: 1604.07316 [cs.CV].

62. XU, Huazhe; GAO, Yang; YU, Fisher; DARRELL, Trevor. End-To-End Learning of Driving Models From Large-Scale Video Datasets. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

63. ERAQI, Hesham M.; MOUSTAFA, Mohamed N.; HONER, Jens. *End-to-End Deep Learning for Steering Autonomous Vehicles Considering Temporal Dependencies*. 2017. Available from arXiv: 1710.03804 [cs.LG].

64. WOLF, Peter; HUBSCHNEIDER, Christian; WEBER, Michael; BAUER, André; HÄRTL, Jonathan; DÜRR, Fabian; ZÖLLNER, J. Marius. Learning how to drive in a real world simulation with deep Q-Networks. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 244–250. Available from DOI: 10.1109/IVS.2017.7995727.

65. OSIŃSKI, Błażej; JAKUBOWSKI, Adam; ZIECINA, Paweł; MIŁOŚ, Piotr; GALIAS, Christopher; HOMOCEANU, Silviu; MICHALEWSKI, Henryk. Simulation-Based Reinforcement Learning for Real-World Autonomous Driving. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 6411–6418. Available from DOI: 10.1109/ICRA40945.2020.9196730.

66. PEROT, Etienne; JARITZ, Maximilian; TOROMANOFF, Marin; DE CHARETTE, Raoul. End-to-End Driving in a Realistic Racing Game with Deep Reinforcement Learning. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2017, pp. 474–475. Available from DOI: 10.1109/CVPRW.2017.64.

67. RIEDMILLER, Martin; MONTEMERLO, Mike; DAHLKAMP, Hendrik. Learning to Drive a Real Car in 20 Minutes. In: *2007 Frontiers in the Convergence of Bioscience and Information Technologies*. 2007, pp. 645–650. Available from DOI: `10.1109/FBIT.2007.37`.

68. KENDALL, Alex; HAWKE, Jeffrey; JANZ, David; MAZUR, Przemyslaw; REDA, Daniele; ALLEN, John-Mark; LAM, Vinh-Dieu; BEWLEY, Alex; SHAH, Amar. Learning to Drive in a Day. In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8248–8254. Available from DOI: `10.1109/ICRA.2019.8793742`.

69. *AWS DeepRacer Developer Guide*. 2023. Available also from: `https://docs.aws.amazon.com/pdfs/deepracer/latest/developerguide/awsracerdg.pdf`. Last accessed: 5-2-2023.

70. JAKL, Vincent; KOSORIN, Peter; PROCHAZKA, Adam. *AWS DeepRacer local training configuration*. 2023. Available also from: `https://apps.datalab.fit.cvut.cz/static/deepracer/deepracer_setup_whitepaper.pdf`. Last accessed: 5-11-2023.

71. SELVARAJU, Ramprasaath R.; DAS, Abhishek; VEDANTAM, Ramakrishna; COGSWELL, Michael; PARIKH, Devi; BATRA, Dhruv. Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization. *CoRR*. 2016, vol. abs/1610.02391. Available from arXiv: `1610.02391`.

Contents of the attached media

```
thesis
 ├── thesis.pdf .................................... The thesis in a pdf format
 ├── pic ........................................... Figures used in the thesis
 ├── text
 │    ├── appendix.tex ............................ Training metrics appendix file
 │    ├── bib-database.bib ........................ Bibliography database
 │    └── text.tex ................................ Thesis text file
 ├── assignment-include.pdf ........................ Thesis assignment in a pdf format
 ├── ctu-thesis.cls ................................ LaTeXsupport file
 └── ctu-thesis.cls ................................ Main thesis LaTeXfile
├── models ......................................... Folder with trained model files and hyperparameters
├── evaluation. .................................... Simulation evaluation videos
└── physical_track_testing. ........................ Videos from experiments on a physical track
```