FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Driving AWS DeepRacer Cars on Unknown Tracks
Student:	Vincent Jakl
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. This thesis aims to explore machine learning models and techniques to make a generic model to drive on an unfamiliar track. The aim is to drive on an unknown track smoothly, correctly (keep all wheels on a track) and with reasonable speed. For experiments and demonstration, use a simulated environment and as the last step, demonstrate the transfer of the created model into the physical car and autonomously drive it on different tracks.

Steps:
1. Explore and summarize different approaches to learning autonomous driving systems and technical details for the AWS Deepracer car.
2. Select appropriate techniques and demonstrate their capabilities in simple experiments.
3. Implement the selected model(s) and train the DeepRacer car to drive on virtual (simulated) tracks in a simulated environment. Focus on the correctness of driving and speed. Demonstrate the capabilities of the drive on simulated tracks that were not used in the training process.
4. Demonstrate the capabilities and accuracy of the trained model in a real-world environment on a physical car.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 23 February 2023 in Prague.

Bachelor's thesis

DRIVING AWS DEEPRACER CARS ON UNKNOWN TRACKS

Vincent Jakl

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

Citation of this thesis: Jakl Vincent. *Driving AWS DeepRacer Cars on Unknown Tracks* . 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 would like to thank first of all my supervisor, Miroslav Čepek for his time, and the knowledge he has given me throughout the process of writing this thesis. Without him, I would not have been able to overcome some of the obstacles this topic has thrown my way. I would also like to thank my parents and my friends for their support during the time of writing this thesis. Thanks also belong to Peter Kosorín and Adam Procházka for their patience and cooperation when setting up training. Lastly, a big thanks belong to the various members of the AWS Deepracer community, who have been very helpful and understanding, and who provided nudges and good suggestions especially when I was trying to get the training environment up and running.

Declaration

In Prague on May 11, 2023 . . . .

Abstract

AWS Deepracer is a popular platform for developing autonomous racing cars using reinforcement learning. In this thesis, the aim is to develop a model for AWS Deepracer that can navigate tracks it has not seen before. The method, that was used to achieve such a model involved using a combination of techniques including data augmentation, and hyperparameter tuning. The model was trained on a set of tracks that were not included in the evaluation dataset and evaluated its performance on a separate set of tracks.

The outcome of this thesis are two AWS Deepracer models, one slower but more careful one, and one faster but less accurate one. Both of these models, however are able to run fairly accurately on a wide variety of tracks, including ones, unseen in training. The outcomes of this thesis allow future AWS Deepracer developers that might want to build a general model, to start with some insights into the process or use the already trained models from this thesis as a base for their own models.

Keywords autonomous driving model, AWS Deepracer, deep reinforcement learning, computer vision

Abstrakt

AWS Deepracer je oblíbená platforma pro vývoj autonomních závodních vozů s využitím posilovacího učení. Cílem této práce je vyvinout model pro AWS Deepracer, který dokáže navigovat na tratích, které dosud neviděl. Metoda, která byla použita k dosažení takového modelu, zahrnovala použití kombinace technik včetně augmentace dat a ladění hyperparametrů. Model byl natrénován na sadě tratí, které nebyly zahrnuty do sady vyhodnocovacích dat, a jeho výkon byl vyhodnocen na samostatné sadě tratí.

Výsledkem této práce jsou dva modely AWS Deepracer, jeden pomalejší, ale pečlivější, a jeden rychlejší, ale méně přesný. Oba tyto modely jsou však schopny fungovat poměrně přesně na široké škále tratí. Výstupy této práce umožňují budoucím vývojářům AWS Deepracer, kteří by mohli chtít vytvořit obecný model, začít s některými poznatky o tomto procesu nebo použít již natrénované modely z této práce jako základ pro své vlastní modely.

Klíčová slova model autonomního řízení, AWS Deepracer, hluboké učení s posilováním, počítačové vidění

List of abbreviations

AWS	Amazon Web Services
CTU	Czech Technical University
FIT	Faculty of Information Technology
FoV	Field of View
GPU	Graphical Processing Unit
GUI	Graphical User Interface
RC	Radio Controlled
RL	Reinforcement Learning

Introduction

The technology of autonomous driving has the potential to transform the automobile business completely. Autonomous vehicles can function without the assistance of human drivers thanks to developments in artificial intelligence, machine learning, and sensor technology and potentially enhancing safety. Managing high amounts of uncertainty while navigating complicated and dynamic environments is one of the main challenges of autonomous vehicle driving. This demands the creation of resilient and adaptable algorithms and even entire systems that can take in knowledge from the past and apply it to new circumstances.

AWS Deepracer is an innovative platform for autonomous driving research that allows developers to train and evaluate deep reinforcement learning algorithms in a simulated racing environment. There are articles online about this topic, mainly regarding training on only one track and optimizing for one race. I focus on making a robust model that can perform on tracks that it has previously not been trained on. This is closer to real-life scenarios where training a car on every road is impossible. It also presents a significant challenge, as the model must generalize to new environments with different track layouts, lighting conditions, and distracting background objects.

My primary motivation for this topic was my recent interest in computer vision. The field of computer vision has been proliferating in recent years, with numerous applications in various domains, from autonomous vehicles to factory automation to healthcare. My main interests are the ways in which computer vision can be applied to solve real-world problems because I am convinced that these technologies can, in the future, make life easier for many. The goal is to explore this topic further in this thesis and to leave some knowledge and insight for the future developers of DeepRacer.

Goals

The main goal of this thesis is to implement and test a driving algorithm for the autonomous AWS DeepRacer vehicle, which will be able to drive on a virtual (simulated) track at a reasonably high speed and without leaving the track. Furthermore, the goal is to verify the model's capabilities on previously unknown tracks.

The sub-objective is first to describe the different autonomous driving architectures and parameters and then select the appropriate model for the vehicle mentioned above. Then select the right hyper-parameters and develop an efficient reward function for the reinforcement learning algorithm.

After training the implemented model, the next goal is to load the algorithm on the physical model of the rover and test the functionality in a real environment on a constructed track.

Chapter 1

Overview of Autonomous Driving and Learning Systems

In this chapter, I provide an overview of autonomous driving and learning systems. I start with a definition of autonomous driving and the levels of autonomy. I introduce shallow machine learning, deep learning, and reinforcement learning approaches for autonomous driving, along with their advantages and limitations. I also add a part about the hardware debate and safety.

1.1 Definition of autonomous driving

There is not one single universal definition for autonomous driving. [1] But in general, the term refers to a vehicle's ability to operate to a certain degree without human input, using sensors, software, and other technologies. It ranges from driver assistance to full automation, where the vehicle can operate without human intervention. [2]

1.1.1 Levels of autonomy

As I mentioned above, the term "autonomous driving" can be very broad, and previously it was not easily distinct what amount of autonomy a particular system has. In 2016 The Society of Automotive Engineers came up with a system to determine and differentiate between levels of autonomy. [2] It has these six levels:

Level 0 – No Automation The driver is in control all the time, with zero assistance from the system.

Level 1 – Driver Assistance The vehicle has some automated features, such as cruise control, but the driver is always responsible for steering and braking.

Level 2 – Partial Automation The vehicle can take over some driving tasks, such as steering and acceleration, but the driver must remain attentive and keep his hands on the steering wheel at all times.

Level 3 – Conditional Automation The vehicle can perform most of the driving autonomously under certain conditions, like highways or trained areas. The driver still needs to be prepared to take control.

Level 4 – High Automation The vehicle can operate autonomously in most situations but may still require human intervention in certain circumstances, such as severe weather conditions. This is the first level, where the driver usually becomes the passenger unless there is a special event.

Level 5 – Full Automation The vehicle can operate completely autonomously in all situations and does not require human attention. [2]

Today, probably the most popular Tesla Autopilot and Enhnaced Autopilot are at around Level 2, according to the Tesla driving manual. [3] The only cars with level 3 autonomous driving licenses produced today are Mercedes-Benz Drive Pilot. Which, for now, is approved by the German transport authorities for some German highways. It also recently got approved by the US state of Nevada. [4]

The scale cannot be accurately applied to the vehicle in this bachelor's thesis since it is only a model car not made for driving in real traffic. However, if I had to place it on the scale somewhere, it would be around Level 3 since it performs autonomously, but there always has to be someone who takes care of the vehicle if it comes off the designated circuit.

1.2 Hardware/sensors

There are numerous sensors that an automaker can use to build a self-driving or assisted-driving vehicle. However, three main ones see repeated usage: cameras, LiDARs, and radars.

1.2.1 Cameras

Cameras are the most widely adapted sensors for autonomous driving. A camera detects light emitted from the surrounding environment on a photosensitive surface behind a camera lens positioned in front of the sensor. This process results in the creation of clear images of the surrounding area. Cameras are not expensive, and with the right software, they can accurately detect objects in the surroundings. These capabilities provide the ability to recognize traffic signs, traffic lights, and other vehicles or pedestrians on the road. With the right setup, cameras can provide 360° vision around the vehicle.[5]

The most popular setup is to use two cameras with space between each other to provide the perception of depth, as with animal or human eyes. The software behind these cameras then uses epipolar geometry and triangulation methods to calculate the distances between the perceived objects and the cameras. [5]

Other setups may make use of fisheye cameras. Fisheye cameras have a higher FoV, so automakers need fewer fisheye cameras than normal cameras to achieve the desired 360° vision. They have some disadvantages, though. The ultra-wide lenses often cause image distortion, which may change straight lines to curves on the sensor. This then needs to be corrected with some software processing.[5] Fisheye cameras find use even in non-self-driving cars. Today, they are broadly used as parking cameras, where a higher FoV is often needed.

In conclusion, cameras are affordable sensors that can provide colored and high-resolution images, and they are irreplaceable for traffic sign and light detection. However, they lack proper 3D vision, and their effectiveness often drops with worsening weather, such as rain or snow.

1.2.2 LiDARs

LiDAR, or Light Detection And Ranging, is a fundamental technology in developing autonomous-driving systems. It functions as a remote sensing technology that emits pulses of infrared or laser light toward target objects, which then reflect toward the LiDAR. By measuring the time interval between the emission and reception of the reflected light, LiDAR can estimate the distance to the target object. As the LiDAR device scans its surroundings, it produces a 3D representation of the scene as a point cloud.[5]

There are two main categories of LiDAR scanners: mechanical and solid-state. Mechanical LiDARs are the most popular form. They use optics and a rotary motor to scan in a 360°FoV. On the other hand, solid-state LiDARs are stationary and do not use mechanical parts to move. This gives them the advantage of being more robust because there is a higher chance of breakage with moving parts. Because of the lack of moving parts, the cost of solid-state LiDARs is also lower. However, the lack of rotation also lowers the FoV to around 120°. [5]

Unlike cameras, LiDARs are way more costly. That is why some automakers, like Tesla, choose not to utilize them in their systems to provide more competing prices to the customers. LiDARs are also more prone to malfunction because of the mechanic parts. Affordable LiDARs also lack the range that radars can provide. [6]

LiDARs are an essential part of many autonomous vehicle sensor kits. They can provide an accurate 3D mapping of the surrounding area and, instead of cameras, provide night coverage. Unlike cameras, LiDARs do not perceive color, so the sole usage of LiDARs would not be sufficient for autonomous driving systems, and outputs from LiDARs are often fused with outputs from other sensors to utilize the advantage of both.

1.2.3 Radars

Radars release electromagnetic waves and then receive the bounced waves from the target objects. They then calculate the distance and motion of the target using the Doppler property.[5] (The explanation of the Doppler effect is beyond the scope of this thesis.)

Radars work at different ranges. From Short-Range Radars through Medium-Range to Long-Range. Just like cameras, radars are also implemented in many non-self-driving vehicles today. Short-Range Radars are commonly used for parking assistance, Medium-Range Radars are suitable for rearview blind-spot detection, and Long-Range Radars for adaptive cruise control.[5]

Radar sensors, in general, are one of the most well-known sensors in both autonomous and non-autonomous vehicles. Unlike LiDARs, they can function even in bad weather conditions, and they supplement cameras with their ability to work at night. In addition to the distance of objects, they can also provide information about the movement of perceived objects. Some of the disadvantages of radars are their reliability and the low resolution of detected objects. That is why they are also beneficial for self-driving applications, but just like LiDARs, they cannot be the only sensor. Their data is often fused with input from cameras, LiDARs, and other sensors. [5]

1.2.4 Other sensors

Automakers also make use of additional sensors, such as humidity sensors. [4] GPS sensors can be used to determine the car's location. [7]Microphones for sound detection, like sirens and honks, and even localization. [8] However, these additional sensors primarily assist the camera and LiDAR systems and vary from automaker to automaker.

1.3 Autonomous driving software

Autonomous driving algorithms are one of the main parts of self-driving cars, taking in data from sensors and outputting driving commands such as steering or speed adjustments. There are several approaches to developing these algorithms, with traditional algorithms relying on hand-coded rules and logic. In contrast, deep learning algorithms use neural networks to learn from data, and reinforcement learning algorithms rely on trial and error to develop optimal decision-making strategies. Each approach has its advantages and limitations, and researchers are constantly exploring new ways to improve the performance and safety of autonomous driving algorithms.

1.3.1 Machine learning

Machine learning is a field of study that involves training computer algorithms to automatically learn patterns in data and improve their performance with more data. Quoting from [9], “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.”

Most autonomous driving algorithms are machine learning algorithms, and there are two main approaches when developing an autonomous driving algorithm. Most popular today are some kinds of deep learning or, possibly, deep reinforcement learning algorithms. Another approach is to use “traditional” shallow learning algorithms. I will expand on this in the following sections.

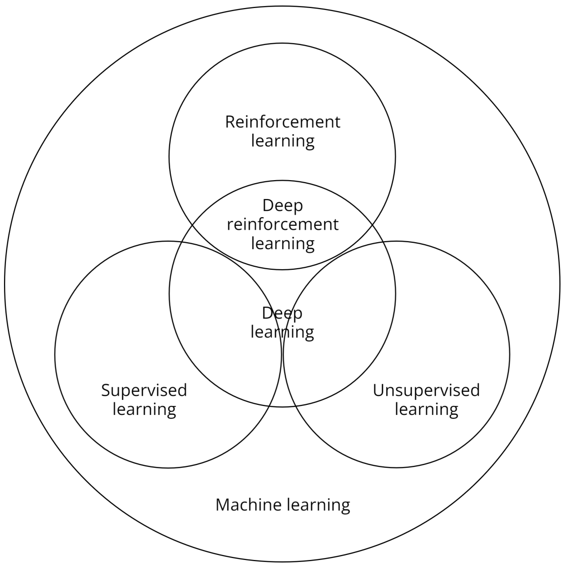


Figure 1.1 Machine learning relations [10]

1.3.1.1 Reinforcement learning

Reinforcement learning is based on an agent and an environment that interact with each other. The agent observes the state of the environment and, based on the state and its policy, takes action. Reinforcement learning is inspired by how animals and even humans learn by receiving positive or negative feedback from the environment around them. [11]

As I show in 1.2, reinforcement learning works by repeating several steps:

1. The agent observes the current state S_t
2. Based on the current state S_t and the policy trained by the agent so far, it will choose an action A_t to perform
3. Based on how relevant the action was, the environment rewards the agent with some positive or negative reward R_t
4. The agent adjusts its policy based on the reward R_t it received

[11] Because these steps are repeated, there is a need to distinguish the action, state, and reward for each iteration. That is why it is essential to use the subscript t.

This approach aims to create a policy that will maximize the reward over time. The problem this creates is balancing short-term rewards with actual long-term goals.

1.3.1.1.1 Exploration vs. exploitation Policy optimization loosely transitions into the problem of exploration vs. exploitation. The agent needs to explore the reward function to find the maximum reward it can get. It needs to explore. However, to learn from its experience, it must also exploit the knowledge it has already gotten from previous episodes. This is the problem of exploration vs. exploitation. [11]

1.3.1.1.2 Reward function The reward function is an essential part of a functional reinforcement learning model. It is a function specified by the developer. This function defines which behaviors are favorable and which are not. This function can alter the training process dramatically depending on how it is written. For some more straightforward tasks, for example, a pong game, the reward function can be as simple as only rewarding points for and penalizing points against. However, designing a reward function may not be as straightforward for more complex problems, with many inputs and possible outcomes. The "right outcome" can be very broad of a term, and designing a good reward function can prove challenging because a reward function that only rewards the right *outome* can be too broad and may lead to the model not converging. So with more complex tasks, rewarding actions that together lead to the desired outcome are often desirable. However, one of the advantages of the reinforcement learning model is that it can often find patterns and associations that a human cannot see, so setting a reward function too close to how a human would asses a specific situation might not be desirable. Finding the right balance can then prove to be a real challenge.

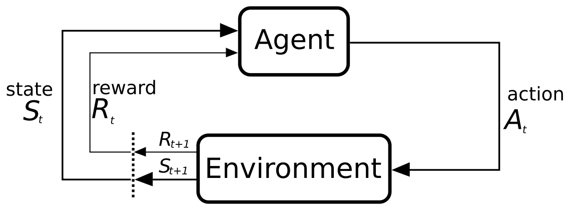


Figure 1.2 Reinforcement learning schema [12]

Unlike supervised and unsupervised learning, Reinforcement learning does not need a structured training dataset. It can just be put into an environment, and the training itself "generates" the dataset on the run. Also, unlike supervised or unsupervised learning, which can only be as good as the training dataset, reinforcement learning can find new approaches to tasks that humans would not. This is why reinforcement learning has been used, for example, for training chess models which beat humans afterward. With supervised learning, for example, it would be tough to achieve this since the model could, at best, learn the strategies at the same level as in the games it trained on. In comparison, reinforcement learning can find new approaches.

Table 1.1 Supervised vs. Reinforcement learning

	Supervised learning	**Reinforcement learning**
Learning process	By utilizing labeled data	By interacting with the environment
Dataset	Labeled dataset	No predefined dataset
How is it learning	Teach by examples	Teach by experience

1.3.1.2 Deep learning and neural networks

Deep learning is built on the idea of artificial neural networks. It entails processing a lot of data to teach a computer to spot patterns and reach decisions. Deep learning models use neural networks with linked layers of nodes to process information. [13] Just like reinforcement learning takes inspiration from nature, deep learning imitates how the human brain functions.

Deep learning's fundamental process includes using a large dataset to train a neural network. The network is built to find patterns in the data and learn how to forecast or categorize things correctly. A high quantity of data is fed into the network's input layer during training. After that, each layer of nodes runs calculations on the data and passes the findings to the following layer. This process is repeated until the output layer produces a final result. [13]

The capability to learn from data without explicit programming is one of the most important characteristics of deep learning. Instead of hand-tuning the weights, the network modifies its internal parameters during training to enhance performance on the given task. The learning of the network works through a process called backpropagation. Backpropagation is the process where the weights of the connections between the nodes are adjusted to minimize the error between the expected output and the actual output. [13]

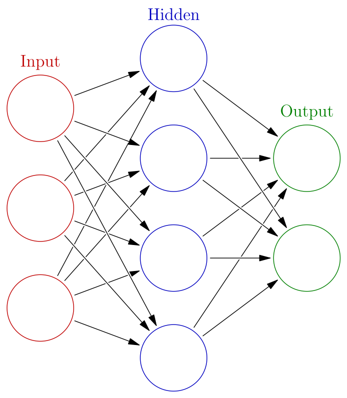


Figure 1.3 Basic neural network [14]

Some examples of the neural networks that are used today are convolutional neural networks (CNNs), recurrent neural networks (RNNs), and generative adversarial networks (GANs). Each network type is beneficiary for a different task. For example, CNNs are mostly used for image processing, while RNNs are frequently used for natural language processing. [15]

1.3.1.3 Deep reinforcement learning

Deep reinforcement learning puts together the concepts of both Deep and reinforcement learning. Unlike traditional reinforcement learning, where policy and value function is usually represented using decision trees or some linear function, deep reinforcement learning represents these by some neural network. This allows for more complex state spaces in which traditional approaches could struggle.[10] Deep reinforcement learning algorithms have a wide range of usages, like controlling robots, and this technique has seen huge success when implemented for playing games like chess or Go. For example, AlphaGo, which uses deep reinforcement learning, is the first program ever to beat a professional Go player. [16]

One of the significant advantages of deep reinforcement learning is its ability to train from raw sensor inputs and learn patterns just from those. This lowers the need to design complex feature extraction systems, which allows developers to create a system with less difficulty. [17]

Deep reinforcement learning has some issues as well. Unlike classical reinforcement learning, it is more complex and needs more computational power to train. This can translate to longer training time unless high-performance hardware is used. [18] Deep RL can also be prone to overfitting, which may not be a problem when used for playing games; however, this can pose a problem when used for more complex systems, for example, the autonomous driving ones. [19]

1.3.1.4 Other approaches

For many parts of autonomous driving, either “traditional” algorithms are more suited, or deep learning might be an overkill. These include path-finding algorithms and shallow machine-learning algorithms for classification and regression. For example, some version of the classic A* algorithm might be sufficient for route planning.[20] Another case could be to use k-means to cluster preprocessed input data from sensors into actions that the vehicle is supposed to do. [21]

1.4 Safety

An enormous part of autonomous driving are also safety precautions. I will not go into more detail about safety precautions in this thesis since the focus of this thesis, the model car, does not pose a threat with its size and speed. It is also run in a controlled environment. However, I still felt the need to include this section as it is an enormous part of autonomous driving today.

Chapter 2

AWS Deepracer

In this chapter, I discuss AWS Deepracer, the physical AWS DeepRacer car and its technical details, the virtual model, its parameters, and the possibilities for training the virtual model.

2.1 What is AWS Deepracer

AWS DeepRacer is a fully autonomous 1/18th scale race car driven by deep reinforcement learning. It consists of three parts: *physical vehicle virtual part* and *AWS DeepRacer league.* [22] The general approach for working with AWS Deepracer is to train a model in a virtual environment and then either upload it to the Deepracer virtual league or a physical model for further testing. This work is mainly focused on the virtual model and its training. Towards the end of the thesis, however, there is a section where I examine the capabilities of the virtually trained model after it has been uploaded to the physical model.

The AWS Deepracer League is a league in which you can compare your trained model with other users who also participate in the league. In case of success in the virtual rounds of the league, an individual or a team can take their model to the final round, which is held with physical models.[22] For this bachelor thesis, however, the league is irrelevant, and I will not deal with it further here.

2.2 Physical model

The AWS DeepRacer vehicle is a Wi-Fi-connected physical vehicle that uses a reinforcement learning model to drive itself around a physical track. [22] The physical construction of the vehicle is very similar to a conventional RC model. The vehicle has, like RC models, a chassis with an electric motor, tires, a servo to control the angle of the front wheels, etc. Unlike RC models, however, the control of the vehicle is autonomous. It does not require an external control unit in the form of a remote control (thanks to the Wi-Fi connection, the vehicle can be controlled remotely, but this is not its main functionality).

The vehicle's autonomy is provided by two main components: the front camera and the processor. The processor, with a trained model on board, controls the speed and angle of the wheels according to the visual input on the camera.

2.2.1 Lidar, two cameras

There are currently two versions of the Deepracer vehicle. The main one is the one described above. The other version has two other input devices apart from the one camera. A second

Figure 2.1 AWS DeepRacer physical vehicle [23]

camera and a LiDAR. This provides the vehicle with a stereoscopic view as well as a 3D representation of the space around it.

In this thesis, however, the assignment is to use the one-camera vehicle to demonstrate the possibility of navigating without LiDAR. I have explained the possible reason for excluding LiDAR in autonomous vehicles in section 1.2.2. Apart from these reasons, the more sensors used, the more data inputted into the neural network, and the harder it is for the training to converge.

2.3 AWS Deepracer simulation environment

The virtual environment consists of two main services:

Amazon SageMaker is generally a machine learning training service. Within AWS Deepracer, it encapsulates the trained machine-learning model and neural network. [22]

AWS RoboMaker is generally a service for developing, testing, and deploying robots. [22] In the context of AWS Deepracer, it is a service that simulates a virtual environment. That is the track, the track environment, and the rover.

2.4 Training a model of AWS Deepracer

A model's training goes as follows: The basic operation is called a *step*. The reward function is evaluated in each step, and the virtual vehicle performs an action based on the model trained in SageMaker so far. These steps are bundled into *episodes*. In one episode, the car starts at a predetermined position on the track, and the episode ends either after the vehicle has gone around the whole track or when the vehicle has gone off track. After each episode, the experience, defined as an ordered list of quads (state, action, reward, new state) associated with each step, is cached by RoboMaker. There is a certain number of episodes in one *training iteration*. After the training episodes of an iteration are finished, the RoboMaker sends the cached experience to SageMaker. From this experience, the SageMaker then trains the policy. There are several *training epochs* in each training iteration. In each one, the SageMaker selects the best episodes regarding reward, which it then sets as input for training the neural network. This process is then repeated until the training is complete. [22] This process is shown in Fig. 2.2

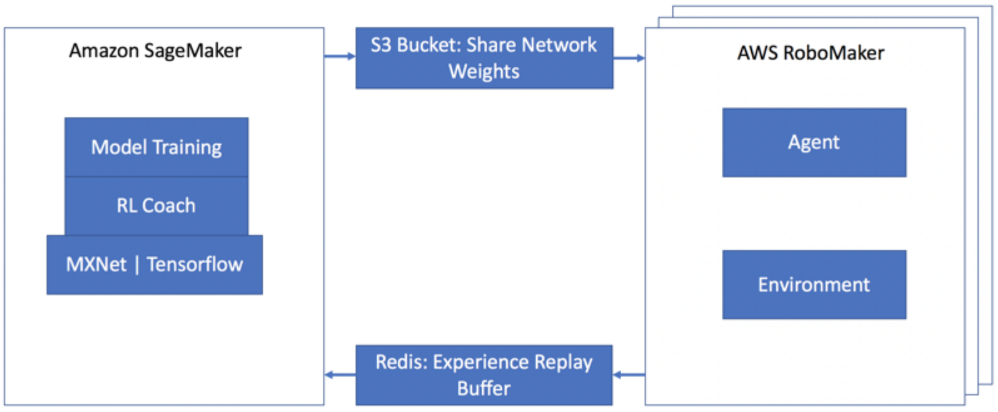


Figure 2.2 Deepracer architecture [22]

2.4.1 Training with the AWS DeepRacer console

Training on the AWS Deepracer site is the easiest way to start with AWS DeepRacer. All the parts like track specification, race type, or action space are available to change in the provided GUI. Furthermore, there are explanations for each element and sometimes even links to tutorials. Amazon also provides new users with ten free hours of their Web services for training and model evaluation. [24]

Together, all these features make the console a great starting point for beginners in DeepRacer and reinforcement learning in general. However, for more extensive training of a more robust model, more than 10 hours are usually needed, and you are charged for any additional hours. For this and other reasons, there is also local training.

2.4.2 Local training

Training the model on a local machine is more suitable for this thesis to save costs and time. Local training is enabled by "deepracer-for-cloud" developed by the AWS DeepRacer community. [25] The environment is based on the AWS RoboMaker and AWS SageMaker containers. However, these containers do not run on the AWS but rather inside a docker image on your computer or server. This also allows the training of multiple models at once, the usage of GPUs, or the analysis of the training logs. The only cost of this training is your machine's computing power and electricity. Local training also allows for more precise training customization and log analysis than console training.

Local training is not for all users, though. The setup of local training and tweaking of the model is not very straightforward. It is easy to run into issues depending on the system architecture or graphics card one uses. Perfect hardware is not strictly needed; however, the training could take significantly longer with weak hardware, which might not be preferable.

In the project for this thesis, I use local training since the amount of training and testing is over the 10 hours provided. I also use many of the customization options that are provided only in the local training. Since the university can also provide high-speed hardware, this was an easy choice.

2.4.2.1 How to setup local training

As I mentioned above, local training is the setup that I use in the process of this thesis. All of the training, experimentation and evaluation is done using *deepracer-for-cloud*.

The setup is accurately explained on the *deepracer-for-cloud* website. [25] By following the tutorial there, is how I set up local training. This would also be the setup to use with the models, I added to the attachment of this thesis.

2.5 Race types

There are three general race types in Deepracer: time trial, object avoidance, and head-to-head racing.

In the time trial, which I am building a model for in this thesis, the car goes around a predetermined track and is timed. There is a time penalty for every time the car goes off-track.

Object avoidance is similar to time trial, the only difference being boxes on the track that the car has to go around. In this mode, the car is penalized by a time penalty not only when it goes off the track but also when it hits an obstacle.

In head-to-head, there is not usually time; instead, the car races against another model. This type of race might be considered the hardest since it is similar to object avoidance with the added difficulty of the objects moving. The car should not hit the other car and should also be able to run faster than the opponents.

2.6 Customizable parameters

Many parameters can be tuned in Deepracer. The reward function is one of the most critical parameters that enter the training. Finding the proper reward function is essential for the model to work correctly and converge in real time. Another parameter that can be customized is the action space, which specifies what specific actions the vehicle can perform, like speed and turning angles. Other parameters that can be tinkered with are the hyperparameters of the deep RL algorithm.

2.6.1 Reward function

In Deepracer, the algorithm runs mainly in Python. The reward function should be written in Python as well. In this case, the reward function is evaluated at every training step. Many parameters can be used for each training episode, as seen in Fig. 2.1. The reward function is one of the main ways to alter the training outcome. With a poorly chosen reward function, not only does the training not have to converge, but even if it does, it may create an unstable or unusable model.

Table 2.1 Input parameters of the AWS Deepracer reward function [22]

Parameter name	Data type	What it describes
all_wheels_on_track	bool	indicates if the agent is on track
x	float	agent's x-coordinate in meters
y	float	agent's y-coordinate in meters
closest_waypoints	[int,int]	indices of the two nearest waypoints
distance_from_center	float	distance in meters from the track center
is_left_of_center	bool	indicates if the agent is on the left side of the track
is_offtrack	bool	indicates whether the agent has gone off track
is_reversed	bool	true if the agent is driving clockwise
heading	float	agent's yaw in degrees
progress	float	percentage of track completed
speed	float	agent's speed in m/s
steering_angle	float	agent's steering angle in degrees
steps	int	number of steps completed
track_length	float	track length in meters
track_width	float	width of the track
waypoints	[(float, float),]	list of (x,y) as milestones along the track
closest_objects	[int, int]	indices of the two closest objects to the agent's current position
objects_distance	[float,]	list of the objects' distances in meters from the starting line
objects_heading	[float,]	list of the objects' headings in degrees between -180 and 180
objects_left_of_center	[bool,]	list of flags indicating whether objects are left of the center
objects_location	[(float, float),]	list of object locations [(x,y), ...]
objects_speed	[float,]	list of objects' speed in m/s

For Deepracer, no reward function is more optimal than any other one. The reward function must reflect what the model is trying to achieve and be adapted based on the race type. For example, suppose a model is meant to run in an object-avoidance race, and the reward function does not use any object parameters. In that case, the model may not learn that it is unfavorable to hit obstacles when in reality, that is the primary goal of that race type.

Another example could be the topic of this thesis, which is to build a model that can run on previously unknown tracks. A possible approach to getting fast times on one track is creating an optimal route on a particular track first and then rewarding the agent based on how closely it follows this optimal line. I believe this approach would not work when trying to build a general model since it massively overtrains for that particular track, and as soon as it is put on a different one, it would most likely just drive off/overshoot the speed.

2.6.2 Action space

There are two possible action spaces that one can choose. Discrete and continuous. The physical abilities of the physical vehicle constrain the action space. It can turn its wheels up to 40° to each side and can reach maximum speeds of 4 m/s. These are also the maximum values that can be set in the action space of the training.[22] It is impossible to set higher values when training in the AWS console, and if these values are set higher in local training, the training will fail.

Discrete action space represents the possible action space of the agent as a finite set of actions.

A steering angle and speed represent one action in the set. When the model runs, the neural network chooses the action closest to the output node. [22]

Continuous action space represents the possible actions as a range instead of a finite set of actions. This means the user sets the minimum and maximum allowed speed and steering. The action that the neural network has at the output node will then be performed exactly unless it is outside of the scope of the action space, in which case, it will choose the min or max value, respectively. [22]

There are benefits and negatives for both. When using a continuous action space, the model does not have to compromise and chooses the values that come straight out of the network. This can lead to smoother transitions between speeds and turning angles and may perform better. A particular human bias is introduced in the discrete action space since the model may want to take an action that is not in the space. This may lead to poor performance if the discrete space is too small. However, it can also lead to some control apart from the reward function. For example, the user can set only lower speeds in sharper turns, which may prevent skidding. Another advantage a discrete action space may have over the continuous one is the training speed. Since there are way more actions in the continuous space, the model may take longer to converge, and a poorly designed reward function may not converge at all.

As I described above, there are advantages and disadvantages to using both action spaces, and the usage depends on the use case.

2.6.3 Hyperparameters

Hyperparameters are parameters that affect how the training progresses. The most often tradeoff is the training speed vs. how accurately the model will learn/explore the space, etc. The following are the hyperparameters for AWS Deepracer:

Batch size (32, 64, 128, 256, 512) Default: 64 The number of experiences taken from the most recent training iteration. The higher the number used, the smoother and more stable the model will learn; however, it may be at the expense of longer training times.

Beta entropy (float between 0 and 1) Default: 0.01 Specifies how often the agent explores random actions. The agent will explore the action space more with higher entropy, but the model may take significantly longer to converge. The model may not converge with too high entropy because it will not use its experience. This parameter exactly conveys the exploration vs. exploitation dilemma I discussed in section 1.3.1.1.

Discount factor (float between 0 and 1) Default: 0.995 Discount factor contributes to the number of steps the agent considers when choosing an action to take. The higher the discount factor is, the more steps the agent considers. With a discount factor of 0.9, it considers ten steps ahead. With the discount factor of 0.999, the agent considers the following 1000 steps. Again the larger the discount factor, the higher possible accuracy, but longer training times.

Loss type (Huber loss, Mean squared error loss) Default: Huber loss This hyperparameter specifies the function that updates the weights in the neural network. The difference between these two functions is that the *Huber loss* changes less than the *Mean squared error* one in the bigger updates. This means that the *Huber loss* can train slower but will be more consistent with updating the weights. On the other hand, the *Mean squared error* function will make bigger updates and can converge faster; however, there is also a higher risk of overshooting and the model not converging.

Learning rate (float between 0.00000001 and 0.001) Default: 0.0003 Learning rate controls how much of a gradient-descend/ascent control the new weights in the neural network.

With a higher learning rate, the new experience will contribute more to the shifting of the neural network, and the network will learn faster overall; however, with it being too high, it may have a problem converging.

Number of episodes between training **(int between 5-100) Default: 20** This hyperparameter controls the number of episodes that the model performs between each training update. The higher the number, the more extensively the network can explore and, in turn, find the best possible outcomes to learn from. However, raising the number will make the network converge way slower since it must simulate more before each network update.

Number of epochs **(int between 3-10) Default: 3** number of passes through the training data to update the neural network.

Also, more hyperparameters, like *exploration_type*, are only available in the local training and are not mentioned in the AWS Manual. However, since these hyperparameters do not have documentation, I have decided not to include them in this description.

Chapter 3

Experiments

Finding the proper configuration of hyperparameters, reward function, and action space is the main challenge of this thesis. It is essential to find a combination of parameters that can converge in real-time while still being efficient enough to achieve the task of driving on unknown tracks. The process is what I discuss in this chapter.

Before explaining the experimentation process, I would like to define some terms I use later in this chapter.

Easy track is the Smile Speedway shown in Fig. 3.1. I consider this track as one of the easier ones because it has relatively shallow turns and is not very long. I use this term to show how well a model/reward function converges.

Hard track is the Expedition Super Loop track, shown in Fig. 3.2. I consider this track as one of the harder ones since it is long, has multiple S-shaped turns, and even features two parallel lines that can confuse the car. I use this track to evaluate the models' ability to learn on a harder track.

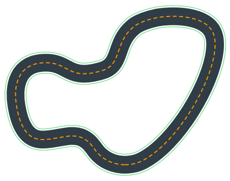


Figure 3.1 Smile Speedway track [26]

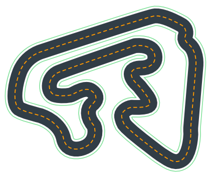


Figure 3.2 Expedition Super Loop track [26]

3.1 Reward functions

As I mentioned in section 1.3.1.1.2, the reward function is one of the essential parts of reinforcement learning. With improper reward function, everything else can be tuned to the best combination, but the model can still learn wrong behaviors. In my attempt to create a reward function that can generalize on multiple training tracks, I have developed three main ones for testing.

3.1.1 General reward function

Code listing 3.1 First reward

```
import math

def reward_function(params):
    dist_reward = ((math.cos(distance_from_center*(3/track_width)))/2)+0.5

    if not all_wheels_on_track:
        reward = 0
        return float(reward)
    ABS_STEERING_THRESHOLD = 15

    if abs_steering > ABS_STEERING_THRESHOLD:
        steering_reward = 0
    else:
        steering_reward = math.cos(0.104*abs_steering)

    speed_reward = math.sin(speed/1.8)
    speed_steering_reward = math.tan(((15-2*abs_steering)/66)*(speed-2))+0.5

    reward+=(steering_reward * 0.8)
    reward+=(speed_reward * 0.5)
    reward+=(speed_steering_reward * 0.7)
    reward+=(progress * 0.01)

    return float(reward)
```

The reward function that I started experimenting with is shown in code listing 3.1. I tried to make a reward function that rewards a car based on its closeness to the center line, speed, and steering. The idea was that just the weighted addition of these general metrics would produce a model that would quickly adapt to a new track and not overly focus on the track layout. The graphs of each reward part can be seen in Fig. 3.3. I also rewarded progress and penalized the model for not having all wheels on the track.

I mainly added the *speed_while_steering_reward* so that it rewards faster speeds while going straight and slower speeds while turning. This reward distribution can also be seen in Fig. 3.3.

Initially, I left the hyperparameters and action space at the default values and ran the training on several tracks. This eventually led me to realize the speeding/steering constraint. I also tweaked the reward function weights of the different factors.

After running several trials with this adjusted approach, the model was able to converge on more straightforward tracks.

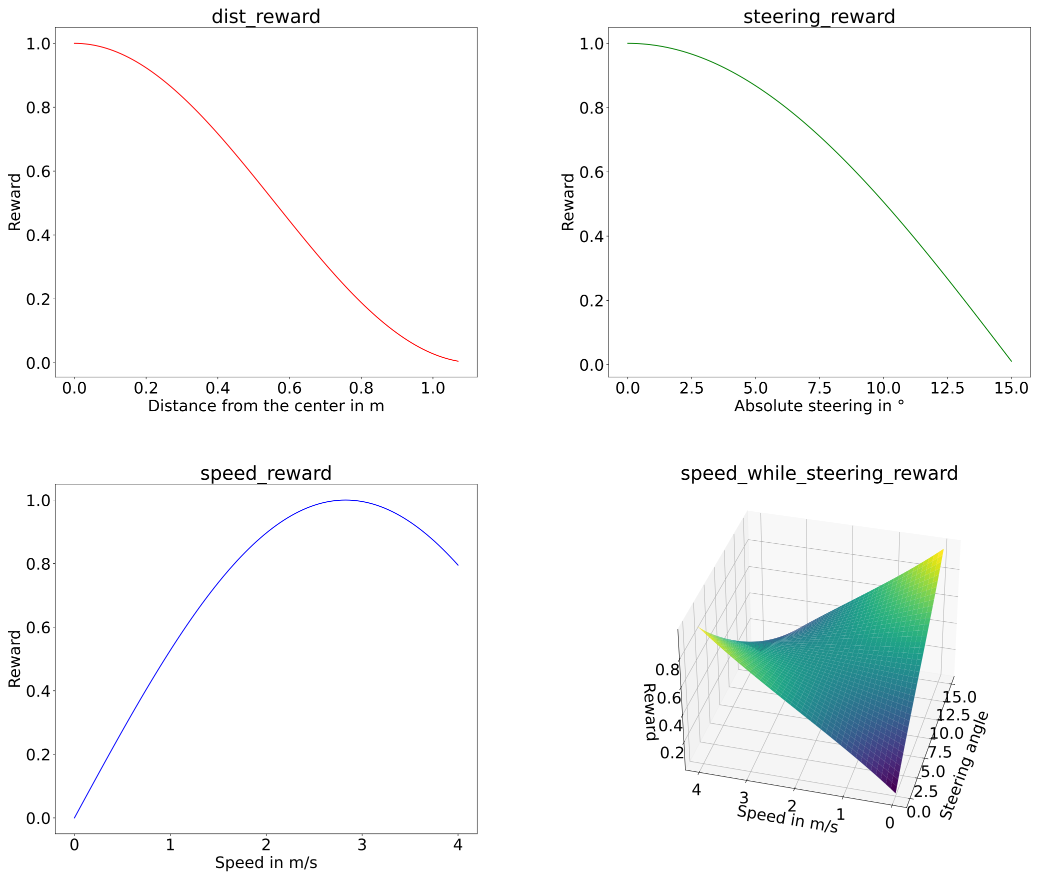


Figure 3.3 General reward parts

3.1.1.1 Penalizing wiggling

I tried adjusting this reward function by remembering the last turning angle and penalizing significant changes in steering between adjusting the action space could also create steps. However, the car did not want to turn after this, and after one experiment, I abandoned this idea. The result of the experiment can be seen in Fig. 3.4

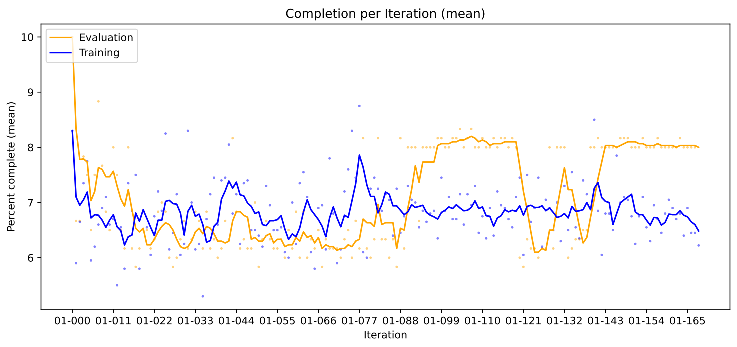


Figure 3.4 Training metrics for wiggling-penalizing reward function

3.1.1.2 Issues with this approach

Even though the model could converge on easier tracks, I was not satisfied with its performance. Not only was it unable to converge at longer and more complex tracks, but even at the easier ones, it took around one day to train. The difference in training between an easier and a more complicated track can be seen in Fig. 3.5. While both pieces of training were run for the same time (about 21 hours), the one trained on the more challenging track had quite a bit more iterations and, even so, could not get too close to converging. (The reason for the different number of iterations is that the further the car gets in one episode, the longer the whole episode will take. This means that the car was running off track way more on the more challenging track and could get more episodes in the time constraint.)

This was not a satisfying result since the objective was to create a robust model that could run on unknown and known tracks. For this, I hypothesized that training on many tracks would be needed, and with more than two days of training time per track, this approach would take too long.

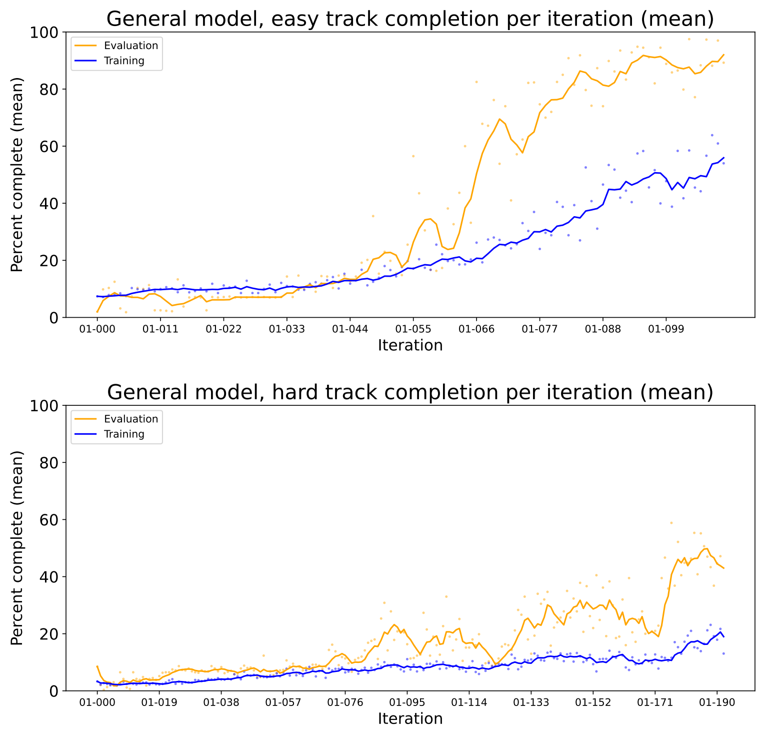


Figure 3.5 General reward model, training on easy and hard track visualization

The underlying issue with this approach was with the design of the reward function. In this case, I believe that hyperparameter adjusting could help but not solve this issue. The problems, in my view, were as follows:

- The reward function was punishing any steering angles larger than 15 degrees, which was meant to mitigate zig-zagging but would prevent the model from turning properly, often needed at harder tracks. This was also the case with penalizing hard steering in general.
- The number of smaller rewards that contributed to a reward for each step was too large, and it was hard to debug this approach and find what to change to improve things.

Ultimately, I abandoned this approach to move on to a simpler one.

3.1.2 Simple reward function

Code listing 3.2 Simplified close to center reward function

```
def reward_function(params):
    reward = ((track_width - distance_from_center) / track_width) ** 2
    reward += ((progress / steps) * 1.5)
    if not all_wheels_on_track:
        reward = 0.01

    return float(reward)
```

Since the first reward function was not converging fast enough, I decided to try and simplify the reward for only staying in the center of the track, meaning it would get more immediate rewards. I abandoned the steering penalty and did not add any reward for speed, but instead decided to do this using an adjustment of the action space, which I analyze more in section 3.2. I also fixed an issue with rewarding wrong behaviors at completed laps, that could not even be seen in the first reward since with the *general reward function*, I never reached a part where the model correctly converged. I talk more about this issue in section 3.1.2.1.

I also slightly changed the distance from the center to a more straightforward function. The new simpler equation, when compared to the one in the *general reward function* (based on the sine function), rewards quadratically based on the distance and incentivizes staying in the center even more. This difference is visualized in Fig. 3.6. It is good to note that it is not a problem that the simple reward gives lower rewards overall since the distribution matters for training a model.

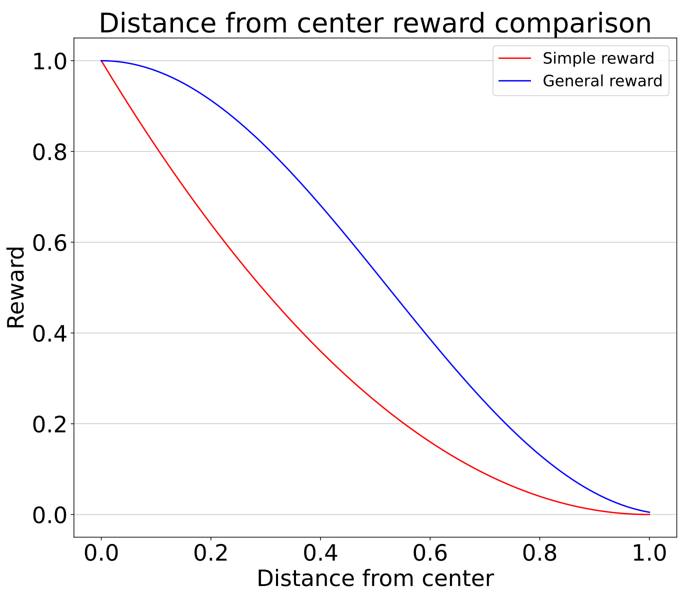


Figure 3.6 Distance from center reward distribution comparison between **general** and **simple** reward function

3.1.2.1 Wrongly rewarding slow model

The issue with both reward functions stated above is that since the model is rewarded at every step, the total reward is summed for each episode. With the designs I had so far, this would promote the behavior that gets the vehicle further on the track. However, when the vehicle starts completing laps and cannot get more steps by going further, it starts to compensate to get higher rewards.

Since, from my observation, the steps are calculated on a time basis rather than a distance basis, the best way to get a higher reward in a space that is constrained by the length (training track) is by either slowing down or zig-zagging on the track, to get more steps. Adding a disproportionately high reward would promote this reward when the car reached a track completion. Subtracting the number of steps at completion promotes a faster behavior, especially if raised to some power. This reward is shown in Code listing 3.3, and its distribution is shown in Fig. 3.7. I added this code to the *simple* reward function for the final model.

In this case, I use 700 steps as the maximum steps the car can achieve. I had to ensure that the number of steps never exceeded this. Since this is a quadratic function, if the number of steps got higher, the function would prefer a higher number of steps. This can be seen well in Fig. 3.7. The maximum number of steps can also be adjusted based on track, but in this case, I decided to leave it as one number to avoid possible errors when training on multiple tracks.

Code listing 3.3 Code rewarding faster behavior at completed tracks

```
max_steps = 700
if progress == 100:
    reward += ((max_steps-steps)**2)/300
```

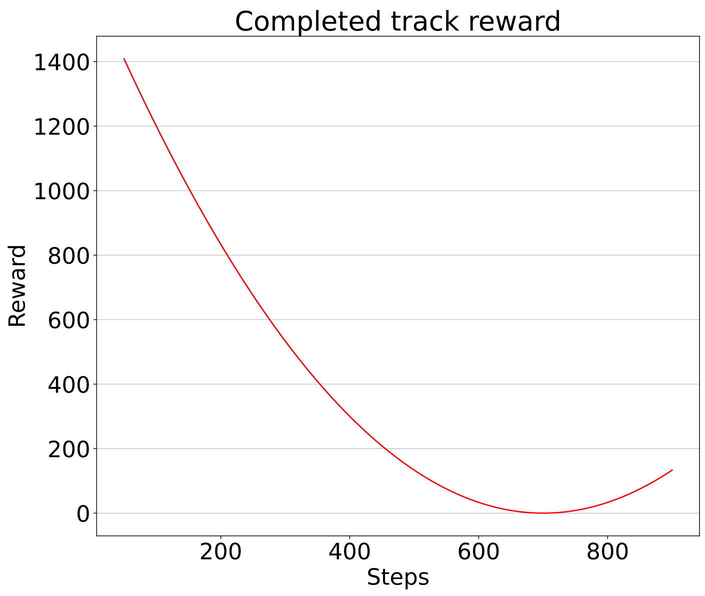


Figure 3.7 Reward distribution for complete laps

This more simple reward function worked better than the general reward function. When training, it was also easier to spot the wrong behaviors, making them easier to debug. I chose this reward function as one of the two for the final model.

3.1.3 Steer towards the center

Code listing 3.4 Steer to center reward function

```
def reward_function(params):
    next_waypoint_num = closest_waypoints[1]
    prev_waypoint_num = closest_waypoints[0]

    if prev_waypoint_num < next_waypoint_num:
        head_towards = (next_waypoint_num + 5) % (len(waypoints) - 1)
    else:
        head_towards = (next_waypoint_num - 5) % (len(waypoints) - 1)

    head_towards_x, head_towards_y = waypoints[head_towards]

    wanted_angle = math.degrees(math.atan2(head_towards_y - agent_y, \
                   head_towards_x - agent_x))

    wanted_steering_angle = wanted_angle - heading

    if wanted_steering_angle < -180:
        wanted_steering_angle += 360
    if wanted_steering_angle > 180:
        wanted_steering_angle -= 360

    difference = abs(wanted_steering_angle - steering_angle) / 60

    reward = 1 - difference
    if reward <= 0:
        reward = 0.01

    max_steps = 700
    if progress == 100:
        reward += ((max_steps-steps)**2)/300

    return float(reward)
```

To diversify, I decided to change my approach, and instead of the car's position, I realized it was possible to reward the steering direction of the vehicle.

The possible problem with the vehicle position approach is that the agent gets full points if it's in the center line, even though it can be turned to one side. However, this behavior should not be rewarded because it will reward the vehicle, even when just randomly passing the center line.

With the steering approach, I reward whether the vehicle is steering toward the center using checkpoints on the track. This approach still leads the vehicle towards the center line without rewarding steering outside. Another benefit is that a natural optimal racing line is created by not steering toward the closest checkpoint but instead one, that is, several checkpoints (in this case, 5) in front of the vehicle. This is shown visually in Fig. 3.8 where the yellow arrows represent the highest rewarded steering angle at each waypoint, and the blue line is the optimal racing line that this type of reward creates.

For this model, I used the constrained action space from section 3.2.2 and the adjusted hyperparameters from 3.3.

As shown in Fig. 3.9, this model converged quite well and in a meager amount of iterations for both the easier and harder track. As I stated previously, this would be important for training

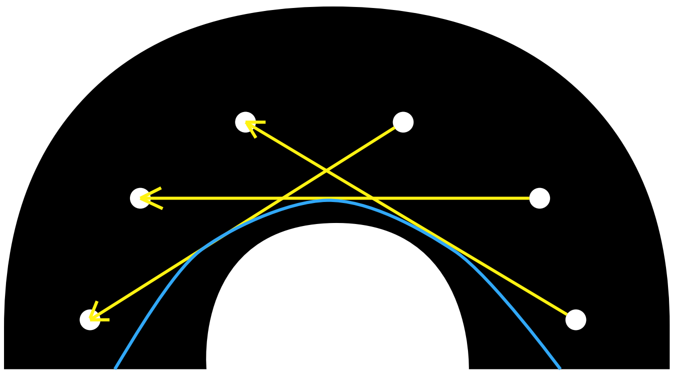


Figure 3.8 How turning towards the upcoming waypoints creates an optimal racing line.

the final model since the ability to converge is the leading property. Still, the training speed also plays a significant factor when training for a general model. So like with the simple reward function, I also decided to try and use this reward function for training the final model.

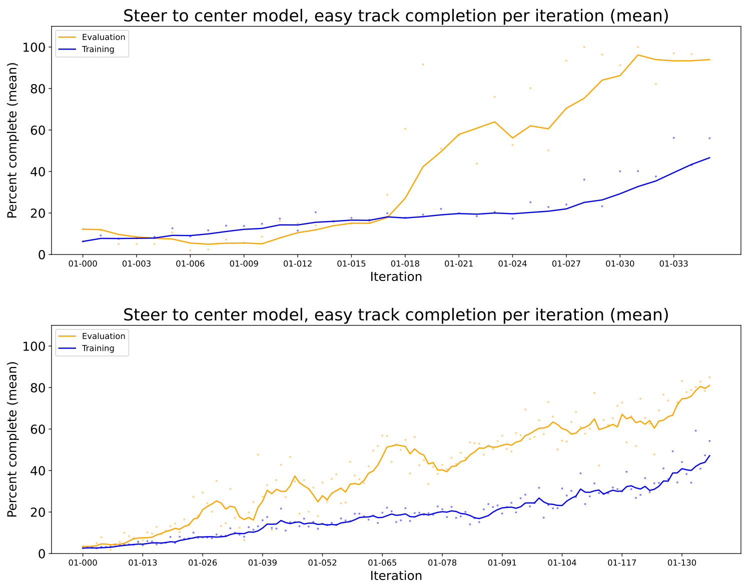


Figure 3.9 Steer to center model, training graphs

3.2 Action space

Action space constrains or allows the vehicle to perform certain actions. The deepracer car has a maximum speed of 4 m/s and a turning angle of 40° to each side. I have not experimented with the continuous action space for my model since it has higher volatility and can take longer to train. Also the discrete action space was better documented, so I decided to stick with that. I also needed to adjust the action space as a constraint in the training process. More about that in section 3.2.2

3.2.1 Free action space

At first, I created a free action space to give the model a bigger selection of speeds for all of its actions and let it decide by itself which actions to choose. This action space was only used to experiment with the *general* reward function. Later I moved on to constrained action space. I explain why in the next subsection.

Table 3.1 and Fig. 3.10 show the distribution of actions. Higher speed is only allowed for going straight; however, the distribution of turning angles is relatively stable at lower speeds.

Table 3.1 Free action space actions

Steering angle in °	Speed in m/s
-40	1.1
-30	1.0
-30	1.6
-30	2.2
-15	1.0
-15	2.0
-15	2.7
0	1.5
0	2.0
0	2.5
0	3.0
0	3.5
0	4.0
15	2.7
15	2.0
15	1.0
30	2.2
30	1.6
30	1.0
40	1.1

3.2.2 Constrained action space

After unsuccessful experiments with the general reward function, I decided to create a more constrained action space that only allows speeds proportionate to the turning angle. This allowed me to make simpler reward functions that focused more on careful driving and less on achieving good speeds etc. Not only does this not allow the car to go too fast in steep turns, but it also limits the minimum speed in straights. I have also removed the 40° angles since they were not used in the previous action space and might cause issues in the physical car. The action space is listed in table 3.2.

This new constrained action space also had fewer actions altogether than the previous one, making the model easier to train since fewer actions meant fewer neural network output nodes.

The distribution can be seen in Fig. 3.11

3.3 Hyperparameters

Hyperparameters are explained more in 2.6.3.

Figure 3.10 Free action space visualized

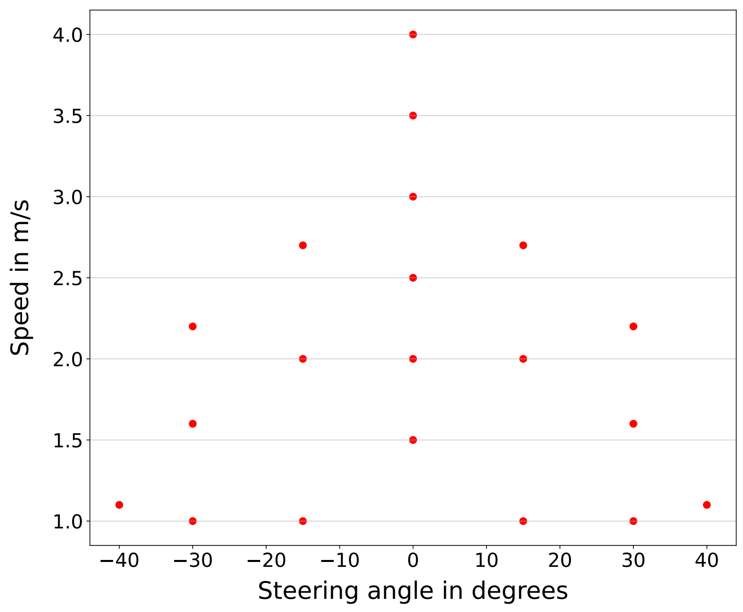
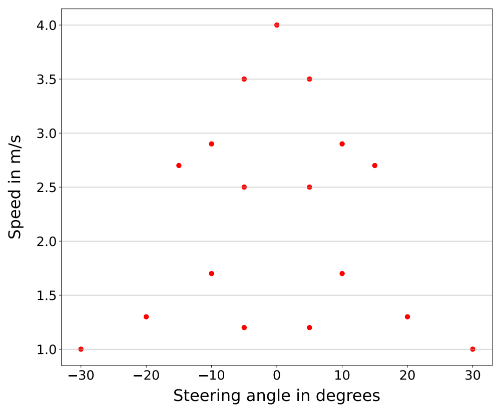


Table 3.2 Constrained action space actions

Steering angle in °	Speed in m/s
-30	1.0
-20	1.3
-15	2.7
-10	1.7
-10	2.9
-5	1.2
-5	2.5
-5	3.5
0	4.0
5	1.2
5	2.5
5	3.5
10	1.7
10	2.9
15	2.7
20	1.3
30	1.0

They are a part that, after some testing, I left almost unchanged from the default values. I chose only two significant changes from the default: The number of training epochs and the discount factor.

Figure 3.11 Constrained action space visualized



3.3.1 Used hyperparameters

The hyperparameters that I ended up with are shown in table 3.3

Table 3.3 Used hyperparameters

Hyperparameter	Value
Batch size	64
Beta entropy	0.01
Discount factor	0.98
Loss type	Huber
Learning rate	0.0003
Number of episodes between training	20
Number of training epochs	5

3.3.1.1 Discount factor

From the defaults, I significantly changed the discount factor from 0.998 to 0.98. This was due to multiple reasons. As described in section 2.6.3, the model considers a certain number of future steps to choose a proper action. To be more precise it uses the following equation:

$$r_0 * d^0 + r_1 * d^1 + r_2 * d^2 + \dots$$

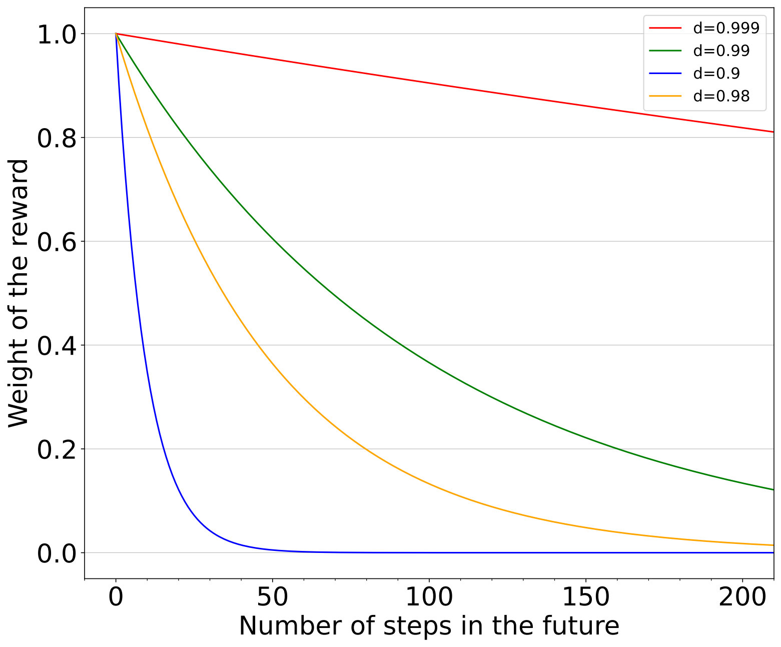
Where r_t is the reward for t being the number of steps in the feature, and d, is the discount factor. So essentially, the discount factor indicates the weights of future steps for the reward. The different weights of discount factors can be seen plotted in Fig. 3.12.

Suppose I consider the distribution from [22] that states that the discount factor of 0.99 considers about 100 steps. In that case, that means that future steps stop being relevant at about 0.35 weight. From my observation, the model took around 50 steps to go around the sharpest turns. For the model to take into account 50 steps, there is the equation

$$d^{50} = 0.3$$

from which the $d \approx 0.98$.

Figure 3.12 Discount factor visualized



This discount factor is high enough to consider long turns and low enough to train efficiently and not cause too much wiggling.

3.3.1.2 Number of training epochs

I do not have an explanation for the change in the number of training epochs, like for the discount factor. The main reason is that the training was on a fast GPU, and increasing the number of training epochs makes the training more stable.

3.3.2 Failed hyperparameter experiments

I also tried experimenting with some other hyperparameters, but that was mainly to see what would happen since I believed some defaults were set up quite well. I only show one experiment where I cranked the learning rate value to 0,05. The result was a failed training, not converging at all, as shown in Fig. 3.13. I believe finetuning hyperparameters more would be something that would be used more when training for one track; however, for this case of a general model, the hyperparameters had to be quite universal.

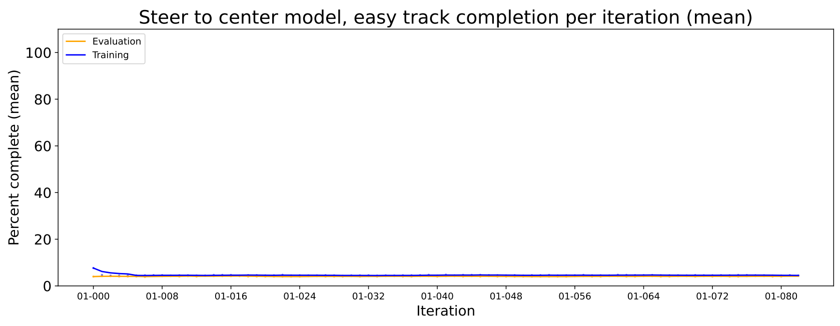


Figure 3.13 Training graph for a model with simple reward function and 0.05 learning rate

Chapter 4

Final model

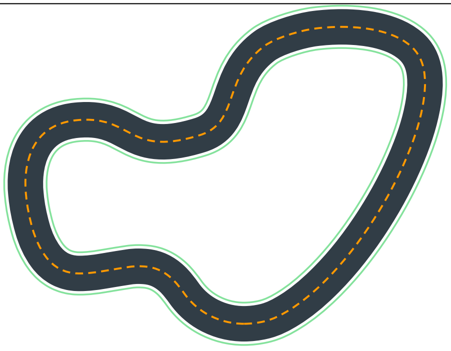
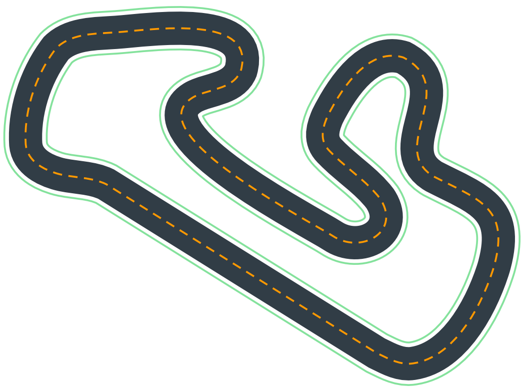
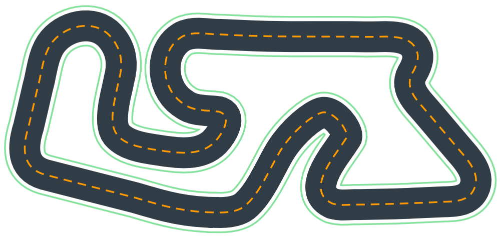
After experimentation with different hyperparameters, action spaces, and reward functions, I stuck with a selection of two reward functions, one action space, and one set of hyperparameters. Now the challenge of training a generalized model started.

As I mentioned several times in this thesis, I had a theory: to build a model that could drive on previously unknown tracks, it would need to be trained on multiple different ones so as not to overfit. For the training, I chose the adjusted hyperparameters, constrained action space, and both the simple and steer-to-center reward functions because both performed quite well in the experimentation.

4.1 Evaluation tracks

The first part of training for the final model was choosing tracks not used in training. Since evaluation is also performed on the track during training, I only had to choose testing tracks for evaluation of the performance of my model. The track dataset is not uniform, with variance in length, width, and, most importantly, curvature, but I tried to choose tracks representative of different combinations. They are shown in Table 4.1. Later, like in the attachment, I refer to these tracks as “short”, “long”, and “long-hard” evaluation tracks.

Table 4.1 Final model evaluation tracks [26]

Name	Curvature
Smile Speedway	
Ace Super Speedway	
DBro Super Raceway	

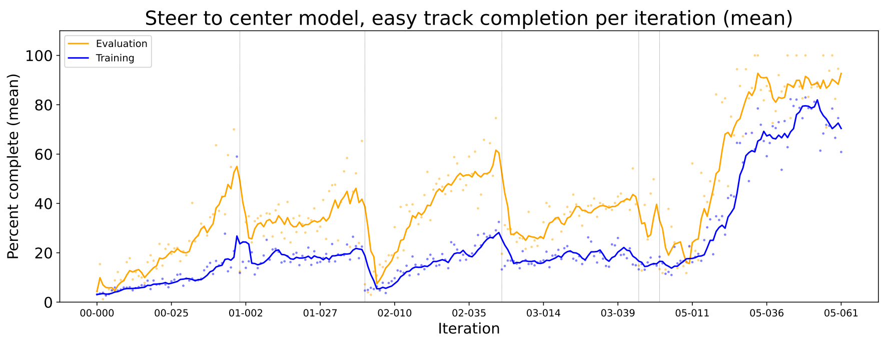


Figure 4.1 Final model with **simple** reward function training graph.

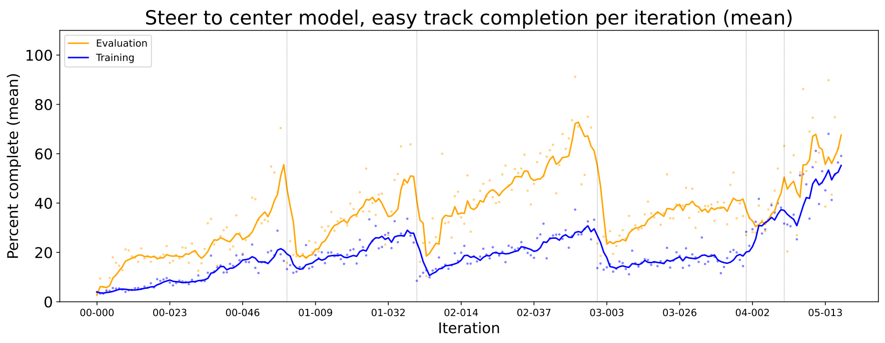


Figure 4.2 Final model with a **steer to center** reward function training graph.

4.2 Training strategy

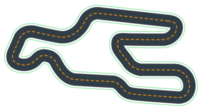
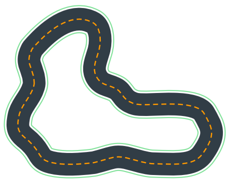
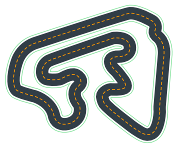
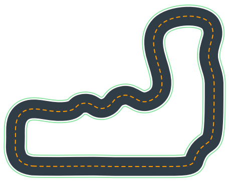
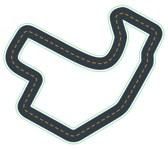
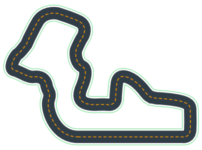
My training strategy was quite simple. I started training the model on a track. When it started to finish the track in evaluation episodes, I stopped the training and moved on to the next track. I continued the process until the model could run on a new track from the beginning of training. For both reward functions, six training tracks were sufficient to reach convergence.

The training process is visualized in Figures 4.1 and 4.2. The grey lines are the restarts of training and track changes. With the simple model, I was a bit afraid I overtrained on one track; however, in the evaluation, it performed pretty well.

4.2.1 Training tracks

Training tracks were essential for a general model. Like with the evaluation tracks, I tried to choose a good selection of different tracks but focused more on longer and more challenging tracks since I had a theory that if the models could learn on those, there would not be a problem with easier and shorter tracks. In Table 4.2, there is the order of tracks in training, their length, and curvature.

Table 4.2 Final model training tracks [26]

Training order	Name	Curvature
1	Hot Rod Super Speedway	
2	Cumulo Carrera Training	
3	Expedition Super Loop	
4	Kumo Torakku Training	
5	Roger Super Raceway	
6	Cosmic Circuit	

4.3 Model evaluation

For the evaluation, I ran the model on each evaluation track 10 times, setting the penalty for going off track to 2 seconds. There are also MP4 videos in the attached media folder for each evaluation run.

Table 4.3 Simple model evaluation on the **Smile Speedway**

episode	time	off_track
8	15.266	0
4	15.617	0
5	15.646	0
0	16.063	0
2	16.549	0
9	16.600	0
3	16.605	0
6	16.683	0
1	16.788	0
7	19.562	1

Table 4.4 Steer to center model evaluation on the **Smile Speedway**

episode	time	off_track
6	14.134	0
7	14.327	0
1	14.343	0
9	14.384	0
4	14.559	0
5	14.664	0
0	14.818	0
8	15.466	0
3	15.654	0
2	16.380	1

Table 4.5 Simple model evaluation on the **Ace Super Speedway**

episode	time	off_track
9	46.790	0
4	47.123	0
7	47.251	0
8	47.406	0
3	47.762	0
5	47.858	0
0	47.976	0
1	48.524	0
2	48.799	0
6	49.470	1

Table 4.6 Steer to center model evaluation on the **Ace Super Speedway**

episode	time	off_track
5	42.102	0
3	42.766	0
7	43.281	0
2	44.882	1
1	44.939	1
0	45.582	1
9	46.574	2
8	46.631	1
6	46.825	2
4	46.936	1

Table 4.7 Simple model evaluation on the **DBro Super Raceway**

episode	time	off_track
7	42.554	0
4	43.216	0
8	43.584	0
3	43.790	0
1	44.013	1
9	44.223	1
2	45.142	1
0	45.183	1
6	45.817	2
5	45.850	1

Table 4.8 Steer to center model evaluation on the **DBro Super Raceway**

episode	time	off_track
4	38.583	1
3	39.813	1
1	40.010	1
2	40.181	1
8	40.652	1
0	41.297	2
5	41.511	1
9	41.894	1
7	42.104	1
6	42.265	2

4.3.1 The importance of training on multiple tracks

I wanted to see if my hypothesis about training on multiple tracks was correct and if it really made a difference in the performance of the final model. So I trained another model with my simple reward function and the same parameters as my general models. This time only on one training track; I would then repeat the evaluation on the evaluation unknown tracks. That way, it will be evident whether training on multiple tracks was crucial for developing a general, not overfitted model. This model is called “overfitted_one_training_track” in the attachment.

Table 4.9 One track trained evaluation on the **Ace Super Raceway**

episode	time	off_track
4	64.852	8
6	66.908	8
0	67.609	9
1	69.182	10
7	70.444	10
8	72.809	11
5	72.860	10
9	76.930	12
2	78.202	13
3	78.690	13

The theory has proven true. I ran an evaluation for ten laps, like with the previous models, and the results can be seen in table 4.9. The lowest off-tracks this model could manage in one run was 8, with 9,6 off-tracks per run on average. The run times are, of course, affected by the off-track penalties, which in this case, are pretty low at 2s. This evaluation clearly shows the difference between training on one track and multiple tracks. Even though the training could converge on one track, it does not mean it will generalize on the others.

4.3.2 Comparison to an overtrained model

To have a time comparison, I also trained a model with the same hyperparameters, action space, and reward function as the *steer-to-center* model, however this time only on the *Ace Super Speedway*, which is one of the evaluation tracks. I ran the training until it was not getting any better. This model is called “overfitted_evaluation_track” in the attachment.

The goal of this model was to overfit and possibly get faster evaluation times. However, to my surprise, this did not happen. Even though in training, the overfitted model was able to have one run during training, which was 39.4 seconds, which is about 3 seconds faster than the fastest evaluation run of the general *steer-to-center* model. However, when I ran the evaluation on this overfitted model, the fastest run in the ten laps was 42.94, about 0.7s slower than the general model. It also had only two runs where the car did not go off-track, compared to the general model’s three.

The reason for this might be that the reward function that was designed for generalized models. Also, adjusting the action space to this track, such as setting up only faster actions, might have been beneficial for faster times.

Building a particular overfitted model would be a topic for itself, and I wanted only a quick comparison. In the future, however, a comparison between the fastest models from the AWS competitors and this general model might be interesting.

4.4 Final model discussion

The simple model was relatively stable. On the easier track, it only ran off once in the ten evaluation runs and, on average, ran off-track 0.1 times per evaluation episode. On the longer track, the car also ran off once, making it, on average, 0.1 times per episode. On the third evaluation track, the model struggled more, running off 0.7 times on average. I still believe this was a success, considering I chose one of the most complex tracks from the dataset, and the car still completed the track, running off, on average, less than once per run.

The *steer to center* model was a bit less stable than the *simple* model. The difference was not noticeable on the easy track, and in the ten runs, it only came off once, like the *simple* model, going off track on average 0.1 times. The lower accuracy was more noticeable on, the longer track, as the car went off track twice on two runs, with an average of 0.9 times off track per run. On the last track, the model could not do a run without going off track; the P-shaped turn was causing the most issues. On this track, it went off track 1.2 times per run. Even though this model was not as accurate as the *simple* one, I believe it has also met the goal of being quite a general model, with less precision but higher speed. I talk about this more in the next paragraph.

The times of runs are a bit harder to evaluate since I could not find a dataset with the best times on these tracks. The best I could do was the comparison in section 4.3.2. However, an essential correlation exists between finish times and average track completion. In both my models, I prioritized completion over any other metric. And even though a part of the reward function rewarded faster times when the track was completed, it was not utilized a lot since I always stopped the training when it started to complete tracks. However, the correlation is quite apparent when comparing the *steer towards center* model to the *simple* model. For example, on the second evaluation track, the *steer towards center* had a way higher percentage of going off track. Still, the laps where it stayed on were significantly faster than the ones with the *simple* model. (The penalty highly influences the final times for the runs where the car went off track.) In general, the times of the *steer to center* are notably lower than that of the *simple* model.

I believe that it is not possible to train a general model that will run on an unknown track as fast as one that is overtrained on an optimal racing line on that specific track. When building a general model, there will always be a tradeoff between speed and completion rate. Here I have been able to demonstrate both of these approaches quite well, with the *simple* model being the more careful one, going slower and staying at the center line and *steer towards center* being faster on completed laps, but running off the track more often since it tries to optimize the racing line and tries to cut corners. In the end, when building a model for a competition, for example, it is crucial to look for the criteria. If the model is evaluated by its best run, then a faster and less careful model might be the right choice. If the average run evaluates it, a more careful model might be the right choice.

Chapter 5

Testing on a physical vehicle

In this chapter, I briefly discuss the challenges of transferring a virtual model to a physical car and show my attempt to move the virtual model to a physical world.

The challenge in transfer between virtual and physical models lies between two main categories: the robustness of the trained model and the quality of the physical track.

As this thesis aimed to build a robust model, I believe this part of the virtual-to-physical change was entirely satisfactory. I even turned on domain randomization for training my models, which changes the lighting conditions for every training episode, to prepare for the physical model.

However, since the ordered physical printed track could not be delivered on time before finishing this thesis, we had to improvise and build a track ourselves. We used masking tape for the sides of the track and yellow tape for the middle sections. We then put up garbage bags as barriers around the track. The difference was quite significant, as seen in the comparison between Fig. 5.1 and 5.2. The most significant differences are the uniform background, the color distinction between the track and off-track space, and a more uniform center line. Of course, the track is also different, but that should not be the issue for a general model.

Even with such a difference between training space and the real-world track, the car could have a couple of runs around the track. The video of the car making it around the track is attached to this thesis. However, I have to say that there were many times when the car went off track. Incrementally higher than in the virtual model. The most probable cause for this was the reflections of light on the trash bags. These reflections can be seen as white by the car's camera. And since I believe the car mainly navigates by the white lines on the sides of the track, this cannot be very clear and get the car off the track. Further experiments with better conditions might bring better results, but this is the best that could have been done within the circumstances.

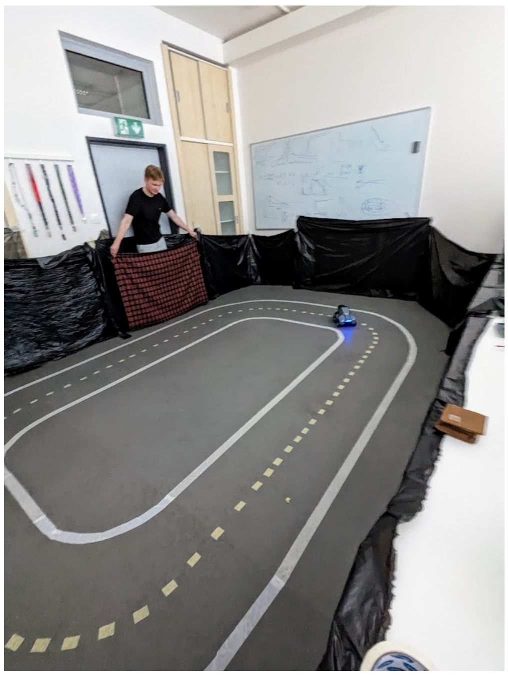


Figure 5.1 Real environment testing track

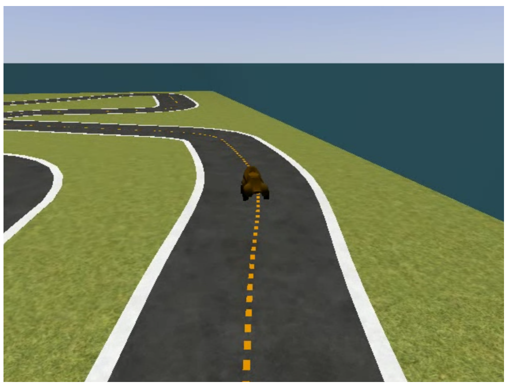


Figure 5.2 Virtual track

Conclusion

The main objective of this thesis was to create a robust AWS deep racer model that would be able to drive correctly and quickly on previously unseen tracks.

Based on my knowledge from exploring the possible configurations and learning techniques, I built two models, *simple* and *steer-to-center*, as an outcome of this thesis, both in the form of a convolutional neural network. These models can both navigate the Deepracer car around various tracks based only on a 2D image input from a camera onboard. Most importantly, I trained them, so they both could run on tracks they had not seen in training. On easier tracks, both of these models can run with high accuracy of only 0.1 off-track runs on average, with the *steer-to-center* following close to an ideal racing line. On longer and more complex tracks, the accuracy of both models drops slightly; however, both models are still reasonably accurate, with a maximum of 2 off-track runs and relatively low times on the completed laps.

I attached these models and evaluation run logs, including videos, to this thesis so that they are available to any future developers of AWS Deepracer. This allows future developers to expand on this idea of a general model, as I created this thesis mainly as proof-of-concept work to see if general models can even exist in the AWS Deepracer. The simple model can serve as a good base model for further improvements, with the potential to achieve 100% completion of tracks with no off-track runs. The *steer-to-center* model has shown promise in competing in fast times, possibly against overfitted models with further tweaking to its parameters. I also tested both these models on a physical track that we built with limited resources, where they performed worse that on a virtual track but still were able to navigate around.

In conclusion, I created a thesis that presents a proof-of-concept for the existence of general models in AWS Deepracer. The models and findings presented in this work can have practical applications for autonomous racing and related fields that require machine learning-based decision-making in dynamic environments.

Bibliography

1. MAURER, Markus. Introduction. In: *Autonomous Driving: Technical, Legal and Social Aspects*. Ed. by MAURER, Markus; GERDES, J. Christian; LENZ, Barbara; WINNER, Hermann. Springer Berlin Heidelberg, 2016, pp. 1–7. ISBN 978-3-662-48847-8. Available from DOI: `10.1007/978-3-662-48847-8_1`.
2. ON-ROAD AUTOMATED DRIVING (ORAD) COMMITTEE. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. 2016. Available from DOI: `https://doi.org/10.4271/J3016_201609`.
3. TESLA, INC. *Model S Owner's Manual* [online]. 2021. [visited on March 22, 2023]. Available from: `https://www.tesla.com/ownersmanual/models/en_us/GUID-EDA77281-42DC-4618-98A9-CC62378E0EC2.html`.
4. MERCEDES-BENZ. *Mercedes-Benz Drive Pilot: Autonomous driving for series production vehicles* [online]. 2022. [visited on March 22, 2023]. Available from: `https://group.mercedes-benz.com/innovation/case/autonomous/drive-pilot-2.html`.
5. 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`.
6. BRANDT, Eric. *LiDAR vs. Radar: Pros and Cons of Different Autonomous Driving Technologies* [Online article]. 2017. [visited on April 12, 2023]. Available from: `https://www.thedrive.com/article/16916/lidar-vs-radar-pros-and-cons-of-different-autonomous-driving-technologies`.
7. RAHIMAN, Wan; ZAINAL, Zafariq. An overview of development GPS navigation for autonomous car. In: 2013, pp. 1112–1118. ISBN 978-1-4673-6320-4. Available from DOI: `10.1109/ICIEA.2013.6566533`.
8. SIWEK, Patryk. Analysis of microphone use for perception of autonomous vehicles. In: *2021 25th International Conference on Methods and Models in Automation and Robotics (MMAR)*. 2021, pp. 173–178. Available from DOI: `10.1109/MMAR49549.2021.9528431`.
9. MITCHELL, T.M. *Machine Learning*. McGraw-Hill, 1997. McGraw-Hill International Editions. ISBN 9780071154673. Available also from: `https://books.google.cz/books?id=EoYBngEACAAJ`.
10. LI, Yuxi. Deep Reinforcement Learning. *CoRR*. 2018, vol. abs/1810.06339. Available from arXiv: `1810.06339`.
11. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249.

12. EBATLLEP. *Deep Learning Neural Network Diagram* [Own work, CC BY-SA 4.0]. 2020. [visited on April 2, 2023]. Available from: https://commons.wikimedia.org/w/index.php?curid=86248030.
13. LECUN, Yann; BENGIO, Yoshua; HINTON, Geoffrey. Deep learning. *Nature*. 2015, vol. 521, no. 7553. Available from DOI: 10.1038/nature14539.
14. GLOSSER.CA. *Artificial neural network with layer coloring* [online]. 2023. [visited on April 2, 2023]. Available from: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg.
15. VADAPALLI, Pavan. *Deep Learning Algorithm [Comprehensive Guide With Examples]* [online]. 2020. [visited on April 12, 2023]. Available from: https://www.upgrad.com/blog/deep-learning-algorithm/.
16. DEEPMIND. *AlphaGo: Mastering the ancient game of Go with machine learning* [online]. unknown. [visited on April 8, 2023]. Available from: https://www.deepmind.com/research/highlighted-research/alphago.
17. WALTHER, Thomas; DIEKMANN, Niklas; VIJAYABASKARAN, Saravanapriyan; DONOSO, José-Rodrigo; MANAHAN-VAUGHAN, Denise; WISKOTT, Laurenz; CHENG, Sen. Context-dependent extinction learning emerging from raw sensory inputs: a reinforcement learning approach. *Scientific Reports*. 2021, vol. 11, no. 1, p. 2713. Available from DOI: 10.1038/s41598-021-81157-z.
18. DING, Zihan; DONG, Hao. Challenges of Reinforcement Learning. In: *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Ed. by DONG, Hao; DING, Zihan; ZHANG, Shanghang. Singapore: Springer Singapore, 2020. ISBN 978-981-15-4095-0. Available from DOI: 10.1007/978-981-15-4095-0_7.
19. NIKISHIN, Evgenii; SCHWARZER, Max; D'ORO, Pierluca; BACON, Pierre-Luc; COURVILLE, Aaron. *The Primacy Bias in Deep Reinforcement Learning*. 2022. Available from arXiv: 2205.07802 [cs.LG].
20. BAST, Hannah; DELLING, Daniel; GOLDBERG, Andrew; MÜLLER-HANNEMANN, Matthias; PAJOR, Thomas; SANDERS, Peter; WAGNER, Dorothea; WERNECK, Renato F. Route Planning in Transportation Networks. In: *Algorithm Engineering: Selected Results and Surveys*. Ed. by KLIEMANN, Lasse; SANDERS, Peter. Cham: Springer International Publishing, 2016, pp. 19–80. ISBN 978-3-319-49487-6. Available from DOI: 10.1007/978-3-319-49487-6_2.
21. RYAN, Cian; MURPHY, Finbarr; MULLINS, Martin. Spatial risk modelling of behavioural hotspots: Risk-aware path planning for autonomous vehicles. *Transportation Research Part A: Policy and Practice*. 2020, vol. 134, pp. 152–163. ISSN 0965-8564. Available from DOI: https://doi.org/10.1016/j.tra.2020.01.024.
22. *AWS DeepRacer: Developer Guide* [online]. Amazon Web Services, Inc., 2023. [visited on March 22, 2023]. Available from: https://docs.aws.amazon.com/pdfs/deepracer/latest/developerguide/awsracerdg.pdf#glossary.
23. AMAZON WEB SERVICES, INC. *AWS Deepracer - the fastest way to get started with machine learning* [online]. 2023. [visited on March 22, 2023]. Available from: https://aws.amazon.com/deepracer/.
24. AMAZON WEB SERVICES, INC. *AWS Deepracer pricing* [online]. 2023. [visited on May 2, 2023]. Available from: https://aws.amazon.com/deepracer/pricing/.
25. AWS DEEPRACER COMMUNITY. *deepracer-for-cloud* [online]. 2023. [visited on March 2, 2023]. Available from: https://aws-deepracer-community.github.io/deepracer-for-cloud/.

26. AWS DEEPRACER COMMUNITY. *The AWS DeepRacer Community Race Data Repository* [online]. 2023. [visited on May 1, 2023]. Available from: `https://github.com/aws-deepracer-community/deepracer-race-data`.

Contents of the attached media

```
thesis
├── text ........................................ texts and bibliography
├── img ......................................... images used in the thesis
│   └── tracks .................................. track images
├── thesis.tex .................................. main LaTeX thesis source code
├── thesis.pdf .................................. thesis in pdf format
├── jaklvinc-assignment.pdf ..................... thesis assignment
└── ctufit-thesis.cls ........................... support file for the LaTeX thesis source code
models
└── MODEL_NAME .................................. folder with relevant files for each model
    ├── MODEL_NAME.tar.gz ....................... trained model
    ├── model.zip ............................... model files for possible more training
    └── custom_files ............................ folder with other relevant files for the model
evaluation
├── video ....................................... folder with MP4 files for each model evaluation
└── logs ........................................ folder with .log evaluation files
real_world_testing .............................. folder with real-world testing videos.
```