DIPLOMARBEIT

# Advanced Driver Assistance System to Stabilise the Powerslide by Reinforcement Learning

Fahrerassistenzsystem zur Stabilisierung des Powerslides mittels Reinforcement Learning

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Diplom-Ingenieurs
eingereicht an der Technischen Universität Wien
Fakultät für Maschinenwesen und Betriebswissenschaften

von

**Tobias Schuster**
Matrikelnummer 01525504
[redacted]
[redacted]

unter der Leitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Johannes Edelmann
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Manfred Plöchl
Institut für Mechanik und Mechatronik

Wien, März 2024

Tobias Schuster

# Eidesstattliche Erklärung

Ich nehme zur Kenntnis, dass ich zur Drucklegung meiner Arbeit unter der Bezeichnung
**Diplomarbeit**
nur mit Bewilligung der Prüfungskommission berechtigt bin.

*Eidesstattiche Erklärung*
Ich erkläre an Eides statt, dass die vorliegende Arbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen von mir selbstständig erstellt wurde. Alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur, sind in dieser Arbeit genannt und aufgelistet. Die aus den Quellen wörtlich entnommenen Stellen, sind als solche kenntlich gemacht.

Das Thema dieser Arbeit wurde von mir bisher weder im In- noch Ausland einer Beurteilung oder Begutachtung in irgendeiner Form als Prüfungsarbeit vorgelegt. Diese Arbeit stimmt mit der von den Begutachtern beurteilten Arbeit überein.

Ich nehme zur Kenntnis, dass die vorgelegte Arbeit mit geeigneten und dem derzeitigen Stand der Technik entsprechenden Mitteln (Plagiat-Erkennungssoftware) elektronisch-technisch überprüft wird. Dies stellt einerseits sicher, dass bei der Erstellung der vorgelegten Arbeit die hohen Qualitätsvorgaben im Rahmen der geltenden Regeln zur Sicherung guter wissenschaftlicher Praxis „Code of Conduct“ an der TU Wien eingehalten wurden. Zum anderen werden durch einen Abgleich mit anderen studentischen Abschlussarbeiten Verletzungen meines persönlichen Urheberrechts vermieden.

Wien, März 2024

Tobias Schuster

# Acknowledgement

I would like to thank the Vienna Scientific Cluster for the access to their computational resources that contributed to the success of this diploma thesis.

Appreciation is extended to the members of CARIAD SE and Dr. Ing. h.c. F. Porsche AG for providing a test vehicle, giving the opportunity to evalute results of this work. In the same way I would like to thank the ÖAMTC Fahrtechnikzentrum Wachau for providing the track for testing and collecting measurements.

# Danksagung

Ich möchte mich bei all jenen herzlichst bedanken, die mich auf dem Wege der Fertigstellung meiner Diplomarbeit sowohl fachlich als auch menschlich unterstützt haben.

Zu besonderem Dank bin ich Herrn Florian Jaumann verpflichtet. In unzähligen Besprechungen haben wir Probleme und Lösungsansätze diskutiert. Dies hat nicht nur maßgeblich zum Erfolg dieses Projekts beigetragen, sondern auch die Motivation für meine Arbeit aufrechterhalten. Herzlichen Dank auch an Herrn Prof. Johannes Edelmann und Herrn Prof. Manfred Plöchl, die mich mit ihrer Erfahrung im Bereich der Fahzeugdynamik sehr unterstützt haben. Für den Freiraum, den sie mir vor allem in thematischer aber auch zeitlicher Hinsicht geschaffen haben, bin ich sehr dankbar.

Ich möchte mich bei allen Lehrenden, die im Laufe meiner schulischen Laufbahn mein naturwissenschaftliches Interesse geweckt und gefördert und mir das technische Rüstzeug mit auf den Weg gegeben hat, ganz herzlich bedanken. Vielen Dank auch an meine Mitstudenten, die durch gemeinsames Lernen einen großen Anteil am Studienerfolg haben und ebenso für Spaß und Motivation im Studium gesorgt haben.

Ein großes Dankeschön auch an meine Eltern, die mir durch ihre Unterstützung die Möglichkeit eines Studiums überhaupt erst ermöglicht haben.

Mit herzlichem Dank,
Tobias

# Table of Contents

# Abstract

Controlling a vehicle's powerslide motion is a challenging control task, which is even more difficult in the presence of a human driver. While professional rally drivers make use of the powerslide to minimise stage time, it is rarely seen in normal road traffic for a good reason. The powerslide is an unstable motion, characterised by large side slip angles, large longitudinal forces on the rear axle and a steering angle pointing towards the outside of the turn. The aim of this work is to design a powerslide controller for an all-wheel drive battery electric vehicle with individually driven front and rear axles and a human driver in closed-loop using Reinforcement Learning.

Reinforcement Learning, a data-driven optimal control strategy, has gained increasing attention in recent years, proving to be a powerful tool for controlling dynamic systems, especially the field of Deep Reinforcement Learning, which involves the integration of Deep Neural Networks. During the learning process, the network's parameters are iteratively updated in order to best possibly satisfy an optimality criterion. The trained network is able to approximate complex, nonlinear mappings between input and output variables that may not be easily captured by traditional controllers.

The intention is to develop an Advanced Driver Assistance System to stabilise the powerslide by controlling front and rear axle drive torques, while the driver follows a given circular path solely by steering. This assistance system must account for the behaviour of a human driver, so the driver model used in simulation is designed accordingly.

# Kurzfassung

Die Stabilisierung eines Fahrzeugs im Powerslide ist eine anspruchsvolle regelungstechnische Aufgabe, speziell unter Berücksichtigung eines menschlichen Fahrers. Professionelle Rally-Fahrer bewegen ihre Fahrzeuge häufig im Powerslide durch Kurven, um diese möglichst schnell zu passieren. Im alltäglichen Straßenverkehr wird man diesem Fahrmanöver hingegen kaum begegnen. Grund dafür ist, dass es sich beim Powerslide um einen instabilen Fahrzustand handelt, bei dem große Schwimmwinkel, Schräglaufwinkel und Umfangskräfte an der Hinterachse sowie ein zum Kurvenäußeren zeigender Lenkwinkel auftreten. Ziel dieser Arbeit ist die Entwicklung eines Powerslide-Reglers, für ein allradgetriebenes Elektroauto mit einzeln angetriebenen Achsen und einem menschlichen Fahrer als Teil der Regelstrecke, mit Hilfe von Reinforcement Learning.

Bei Reinforcement Learning handelt es sich um einen Machine Learning Ansatz zur Regelung dynamischer Systeme, wobei das Regelverhalten mittels datengetriebener Methoden iterativ verbessert werden soll. Speziell der Bereich des Deep Reinforcement Learnings hat in den letzten Jahren an Bedeutung gewonnen. Hier wird der Regler durch ein künstliches neuronales Netzwerk repräsentiert, dessen Parameter während des Trainings schrittweise angepasst werden, um ein Optimalitätskriterium bestmöglich zu erfüllen. Das trainierte Netzwerk ist in der Lage komplexe, nichtlineare Zusammenhänge zwischen Eingangs- und Ausgangsgrößen zu approximieren, die klassische Regler nicht erfassen können.

Ziel ist es, ein Fahrerassistenzsystem zu entwickeln, um den Powerslide zu stabilisieren, indem die Antriebsmomente der Vorder- und Hinterachse geregelt werden, während der Fahrer durch entsprechendes Lenkverhalten einer vorgegebenen Kreisbahn folgt. Das Assistenzsystem muss dabei das Verhalten eines menschlichen Fahrers berücksichtigen, weshalb in der Simulation ein entsprechendes Fahrermodell verwendet wird.

# List of Abbreviations

| | |
|---|---|
| **ADAS** | Advanced Driver Assistance System |
| **AWD** | all-wheel drive |
| **BEV** | battery electric vehicle |
| **COM** | center of mass |
| **DNN** | Deep Neural Network |
| **DRL** | Deep Reinforcement Learning |
| **ELU** | Exponential Linear Unit |
| **LQR** | linear-quadratic regulator |
| **LSTM** | Long Short Term Memory |
| **MDP** | Markov Decision Process |
| **ML** | Machine Learning |
| **MLP** | Multilayer Perceptron |
| **MPC** | Model Predictive Control |
| **MSE** | mean square error |
| **ODE** | ordinary differential equation |
| **PD** | Proportional Derivative |
| **PPO** | Proximal Policy Optimization |
| **RL** | Reinforcement Learning |
| **RWD** | rear-wheel drive |
| **SB3** | Stable-Baselines3 |
| **SL** | Supervised Learning |
| **TRPO** | Trust Region Policy Optimization |
| **UL** | Unsupervised Learning |
| **VSC** | Vienna Scientific Cluster |

# 1 Introduction

The steady-state powerslide is defined as stationary cornering motion of a vehicle with large side slip angles, a steering angle pointing towards the outside of the turn and large longitudinal forces on the rear axle [1]. Stationary cornering is characterised by a constant trajectory curvature, a constant speed and a constant vehicle side slip angle. During steady-state regular cornering, the side slip angles remain small, the steering angle points towards the inside of the turn and primarily lateral tire forces are acting. This driving state is stable, in contrast to the powerslide. Due to its unstable nature, the powerslide has to be controlled either by the driver or an active system. Understanding and controlling the powerslide is of great interest to the research community, as it represents a nonlinear edge case of stability and controllability in vehicle dynamics.

Controllability characteristics of the powerslide for both rear-wheel drive (RWD) and all-wheel drive (AWD) vehicles are addressed by [2]. While steering inputs are primarily responsible for changing lateral motion variables at regular cornering, the powerslide can more easily be stabilised by throttle commands. Due to large tire side slip angles on the rear axle, horizontal tire forces are saturated. The coupling between longitudinal and lateral tire forces ensures the controllability of the lateral vehicle movement through the longitudinal tire forces via drive torque inputs [1].

The development of AWD battery electric vehicles (BEVs) offers the opportunity for new control strategies. While internal combustion engine (ICE) AWD concepts are characterised by drive trains mechanically linking front and rear axle, these new AWD BEVs typically comprise two electric motors, driving front and rear axles independently. Apart from steering inputs and total drive torque, the drive torque distribution between front and rear axle represents another possibility to influence the powerslide motion.

By addressing controllability and observability criteria of the unstable powerslide motion, [3] presents different options of stabilising the powerslide, including the drive torque distribution. In this work, the author also proposes a linear controller, assisting the driver to stabilise the powerslide by controlling front and rear axle drive torques. The controller design involves linearising the system around an operating point which includes defining a steady-state drive torque distribution.

## 1.1 Motivation

Data-driven control has become more and more popular in the past years, with Reinforcement Learning (RL) as one promising domain. Similar to a contribution of [3], the aim of this work is to design an Advanced Driver Assistance System (ADAS) to stabilise the powerslide in the presence of a human driver. While the author of [3] proposes a traditional control concept, this work contributes a novel controller based on RL.

The selection of RL as method for addressing this control problem is motivated by the question if it is feasible to stabilise the powerslide at an a priori unknown steady state. Traditional control concepts rely on the explicit knowledge of an operating point, whereas the data-driven approach may work without this information.

Additional aspects justifying RL as control concept include its end-to-end learning support. In Deep Reinforcement Learning (DRL) the controller is represented by a Deep Neural Network (DNN). These networks can efficiently extract relevant features from raw input data without the need for intermediate processing steps. As a consequence, the control engineer does not have to worry about the selection of relevant controller inputs, as the RL algorithm will eventually detect the relevant information on its own. In the end, the powerslide is characterised by nonlinear vehicle dynamics including nonlinear tire behaviour at an unstable driving state. Therefore, controlling the powerslide is supposed to be an appropriate use case to analyse if the proposed controller can demonstrate added value compared to traditional controllers.

## 1.2 Literature

Numerous potential applications for Machine Learning (ML) concepts exist within the domain of the powerslide. Apart from directly training a controller to stabilise the powerslide, other approaches focus on the possibility of augmenting traditional controllers with data-driven methods. The majority of literature in this field addresses powerslide control within the context of autonomous driving.

Using RL, the authors of [4] solved the autonomous powerslide problem. They introduced a framework that combines simulation models with real-world robotics to optimise control policies. Controlled powerslide manoeuvres are demonstrated on a robotic car.

In [5], a DNN for controlling the powerslide on arbitrary trajectories is trained using RL. The trained controller is transferred to a RWD model car to validate its performance. The authors consider the autonomous powerslide as the controller is also responsible for steering inputs.

In [6], the authors propose a hybrid control structure encompassing a Model Predictive Control (MPC) strategy and a road terrain classifier. Their motivation is to develop a system able to perform powerslide manoeuvres autonomously in a wide range of scenarios. They use real-world data generated with a test driver in the loop to learn tire friction properties and powerslide operating points represented as DNNs. These characteristics are required by the lower-level controller to adjust its behaviour accordingly.

The authors of [7] propose a new family of tire models based on neural ordinary differential equations [8]. These tire models replace an analytical tire model in an existing MPC framework to control the autonomous powerslide.

## 1.3 Structure of this Work

The remainder of this work is structured as follows. In Chapter 2, basics of RL are introduced, while in Chapter 3, theoretical background of model-free RL algorithms, and especially the mathematical fundamentals of the state-of-the-art model-free RL algorithm, is provided. In Chapter 4, the vehicle model and driver model are presented and in Chapter 5, the integration of these models into a framework providing a standardised RL interface is described. In Chapter 6, relevant parts of the RL training procedure are covered and in Chapter 7, the evaluation of a successfully trained controller is discussed. In Chapter 8, a summary, a conclusion and an outlook are given.

## 1.4 Notation

Chapter 2 and Chapter 3 cover theoretical background and algorithmic insights of RL. This sections outlines some relevant notation utilised in the RL context.

The conditional probability of event A given that event B has occurred, is denoted as $P(A|B)$. The tilde operator $\sim$ symbolises sampling from a probability distribution. Let $\mathbb{E}[X]$ denote the expected value of a random variable $X$. Variables with a circumflex like $\hat{x}$ indicate a sample mean to approximate the true value, e.g., $\hat{x} \approx \mathbb{E}[X]$.

# 2 Reinforcement Learning Basics

This chapter serves as a basic introduction to RL. First, a brief overview of different ML methods is presented, while Section 2.2 introduces the fundamental ideas and terminology associated with RL. A comparison of RL with optimal control theory is given in Section 2.3.

## 2.1 Machine Learning Overview

ML focuses on developing algorithms that enable computers to learn from data in order to make predictions or decisions. In this context, *learning* refers to the process of iteratively adjusting the parameters of some arbitrary function to best possibly satisfy an optimality criterion. ML algorithms are classified into three categories: Supervised Learning (SL), Unsupervised Learning (UL) and RL algorithms.

### Supervised Learning

The concept of SL is characterised by training on labeled datasets. The goal is to learn a function that provides a mapping between input data and corresponding output labels. Models trained with SL algorithms aim to identify relationships in the data, enabling them to classify new, unseen data. An application example of SL is visual road sensing to detect obstacles. In this scenario, the data consists of images with their respective pixel values (input) and corresponding descriptions of the content (label).

### Unsupervised Learning

UL algorithms focus on identifying patterns and uncovering hidden structures within datasets that have no labels. An example use case of UL could be anomaly detection in vehicle dynamics. The UL algorithm can identify deviations from normal behaviour by data from various sensors embedded in the vehicle. Detected anomalies could indicate potential failures or malfunctions, and therefore enhance vehicle safety.

### Reinforcement Learning

While SL and UL algorithms learn from static, predetermined datasets, RL algorithms generate their training data dynamically throughout the training process itself. In an RL scenario, as shown in Figure 2.1, an agent interacts with an environment, sending actions and receiving observations and rewards. The primary objective of RL is to train the agent to successfully accomplish a task within the environment, with the reward serving as a quality measure of its actions. The agent aims to select actions that lead to positive outcomes and accumulate as much reward as possible. When the agent receives a high reward for a particular action or sequence of actions, the associated behaviour is reinforced, increasing the likelihood of its repetition in similar situations. Through an iterative process of trial and error, the agent learns to identify the actions generating the most cumulative reward, and adjusts its behaviour accordingly.

The concept of RL is inspired by the way humans (or animals) learn, including fundamental tasks such as walking. When humans learn to walk, they initially lack coordination and balance. Through trial and error, they take small steps, often stumbling and falling. However, every time they successfully take a step or maintain balance, they receive positive reinforcement in the form of encouragement from their environment.

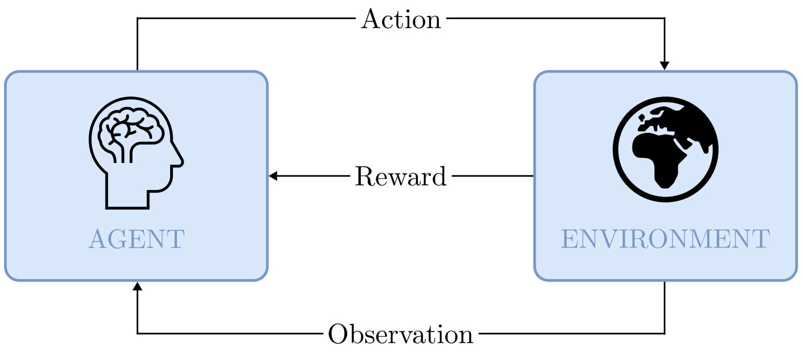


Figure 2.1: Basic RL concept.

## 2.2 Fundamental Ideas and Terminology

This section gives an overview of the fundamental concepts and terminology associated with RL. For a comprehensive introduction to RL, Sutton and Barto's book [9] is a valuable reference. A concise overview of DRL, which involves the integration of DNNs, is given in [10]. These techniques are especially useful when dealing with continuous problems. Some ideas of [9] and [10] are summarised in the remainder of this section.

RL involves the interaction between two characters: the learner and decision maker called the **agent** and everything outside the agent called the **environment**. Various forms of environments are possible, including simulated environments, e.g., computer programs or virtual worlds, where the agent interacts and learns without any direct, potentially harmful, consequences. Alternatively, the environment can be a real physical setup, such as a car on a road, where the agent experiences real-world consequences.

Figure 2.2 illustrates the interaction between the agent and the environment, occuring at discrete time steps. At each time step, the agent receives an **observation** of the current **state** of the environment, selects an **action** based on its decision-making **policy**, and returns it to the environment. In consequence, the environment moves to a new state and provides feedback to the agent in the form of a **reward** signal and the next observation. The agent's goal is to learn an optimal policy that maximises the cumulative reward over time. This learning process is driven by a **learning algorithm** that updates and improves the agent's policy based on the received reward.

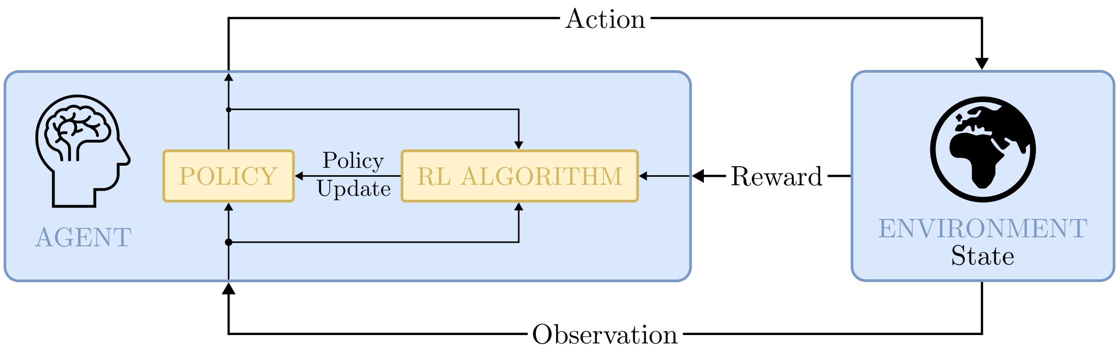


Figure 2.2: RL interaction loop during training.

## Environment and State

The state contains all available information of the environment at any time step $t$. When the agent takes an action $a_t$ in a specific state $s_t$, the environment transitions to a new state $s_{t+1}$. This transition can either be stochastic or deterministic, depending on the nature of the environment. In a stochastic environment, the identical combination of states $s_t$ and actions $a_t$ can result in different possible next states $s_{t+1}$. This variability can correspond to disturbances acting on the environment.

The RL problem is commonly modelled as a Markov Decision Process (MDP). In an MDP the probability of transitioning from one state $s_t$ to the next state $s_{t+1}$ solely depends on the current state $s_t$ and action $a_t$, irrespective of the history of states or actions that led to $s_t$. By assuming the Markov property, modelling RL problems becomes simpler, as it eliminates the need to consider the entire history of states and actions, enabling more efficient learning and decision-making in RL algorithms. The Markov property is formulated as

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|\tau)\,, \tag{2.1}$$

where $P(s_{t+1}|s_t, a_t)$ represents the probability of the environment transitioning to a specific next state $s_{t+1}$, given the current state $s_t$ and action $a_t$. $P(s_{t+1}|\tau)$ represents the probability of transitioning to state $s_{t+1}$ given the entire trajectory $\tau = \{s_0, a_0, ..., s_t, a_t\}$.

### Episode and Epoch, Termination and Truncation

Environments are distinguished based on whether or not they can transition to a terminal state. In an Atari game, a natural ending is reached, e.g., when the agent either wins or loses, resulting in the **termination** of the game. Such tasks are referred to as *episodic* or *finite-horizon* tasks, and the complete sequence from the initial state $s_0$, sampled from an initial state distribution $\rho_0$, to the terminal state is called **episode**. In the case of *continuing* or *infinite-horizon* tasks on the other hand, the interaction between agent and environment has no predefined ending, e.g., process- or robotic control tasks, which do not have a terminal state. An additional time limit is set to artificially end the episode. In literature, this time limit is referred to as **truncation** limit. By preventing the agent from getting stuck in a never-ending episode, this approach allows to gain more valuable experience, as the interaction of several episodes is considered in the update process. It is crucial to distinguish between termination and truncation, as it affects the learning process. If the episode has truncated, the agent would expect further state transitions and potential rewards, whereas if the episode has terminated by reaching the terminal state, no further rewards can be received.

During one training iteration, training data is usually collected by running multiple episodes. The number of environment transitions used by the agent to learn is determined by the size of the rollout buffer. One iteration of data collection and learning update is referred to as an **epoch**.

### Observation Space

The observation is the part of the environment accessible to the agent. It can represent either the complete state, fully observed, or a subset of it, partially observed, respectively. The observation space can be discrete or continuous depending on the nature of the environment. In DRL, states, and therefore observations, are represented as real-valued vectors or matrices.

### Action Space

The action space describes the set of all valid actions in a given environment, and it can either be discrete or continuous. In environments with discrete action spaces, only a finite number of actions are available to the agent. Examples include Atari games, where the agent can choose from a limited set of possible moves. Environments with continuous action spaces on the other hand, allow a wide range of possible actions, often represented as real-valued vectors, like in robotic control tasks.

The distinction between discrete and continuous action spaces has a significant impact on the choice of the RL algorithm. Some algorithms rely on discretising the action space. Applying such algorithms directly to environments with continuous action spaces requires modifications and adaptations to handle the continuous nature of the actions. There are also algorithms designed to address continuous action spaces, such as policy gradient methods, presented in Section 3.2.

### Reward Function, Return and Value Function

The reward $r_t$ is a single scalar value, the agent receives at each time step $t > 0$. It is determined by the reward function

$$r_{t+1} = R(s_t, a_t, s_{t+1})\,, \tag{2.2}$$

i.e., a function of the previous state, the action taken and the state that the environment arrived at, and serves as feedback to the agent, indicating the quality of its behaviour. The agent aims to maximise the cumulative reward, known as the return $G_t$, which is the total sum of the rewards received from a time step $t$ onwards.

Considering continuing tasks, the summation of rewards over an infinite number of time steps is problematic, as the sum may not converge to a finite value. To overcome this issue, an additional discount factor $\gamma \in (0, 1)$ is added to the formulation of the cumulative reward

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\,, \tag{2.3}$$

to ensure that the infinite sum converges. The choice of $\gamma$ is not only justified from a mathematical perspective, it also relates the importance of rewards in the near future relative to those received later. A smaller $\gamma$ encourages the agent to focus more on immediate rewards, while a larger $\gamma$ causes the agent to consider rewards over a longer time horizon.

Having a reliable estimate of the expected return under the current policy is crucial for the agent. This is referred to as the value, and the mapping from a state to the value is called value function

$$V^{\pi}(s) = \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[ R(\tau) | s_0 = s \right] . \tag{2.4}$$

This function represents the expected return if the environment is in state $s$ and the agents acts forever according to policy $\pi$. The action-value function

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[ R(\tau) | s_0 = s, a_0 = a \right] \tag{2.5}$$

represents the expected return if the environment is in state $s$, the agent takes an arbitrary action $a$, and then acts forever according to policy $\pi_\theta$.

## Policy and Exploration vs Exploitation

The policy $\pi$ can be described as the agent's behaviour and represents the mapping from observations to actions. When dealing with policy-gradient algorithms in DRL, this policy is a computable function depending on a set of parameters $\theta$, i.e., the weights and bias of a DNN. These parameters $\theta$ are randomly initialised at the beginning and adjusted during the training via some DRL algorithm to change the agent's behaviour. The theoretical background of the policy update process is presented in Section 3.2.

In the case of policy gradient algorithms, the policy is typically stochastic, implying that actions are selected based on a probability distribution. Assuming a fully observable environment, i.e., observations are equal to states, and without loss of generality, policy

$$\pi_\theta(a|s) = P(a|s,\theta) \tag{2.6}$$

represents the probability of choosing action $a$ given that the environment is in state $s$, subject to the policy parameters $\theta$. In DRL, two kinds of stochastic policies are typically used, categorical policies for discrete action spaces and diagonal Gaussian policies for continuous ones. In terms of Gaussian policies, observations are mapped to mean values and standard deviations of the corresponding actions. The actual action, which is sent back to the environment, is sampled from this distribution $a_t \sim \pi_\theta(\cdot|s_t)$. This kind of randomness in selecting actions helps the agent to find the right balance between exploring and exploiting, which is a critical aspect in RL. If the agent only sticks to what it already knows, it might end up trapped in a situation that seems good but is not the best.

## 2.3 Reinforcement Learning and Control Engineering

This section gives a brief comparison of RL to optimal control and summarises some ideas of [11]. RL is an optimisation problem and has similarities with the principle of optimal control theory. While the objective of both RL and optimal control theory is similar, namely determining the correct inputs into a system to achieve the desired system behaviour, they use different methodologies and terminology to represent analogous concepts. Figure 2.3 illustrates a standard control loop comparing optimal control theory an RL.

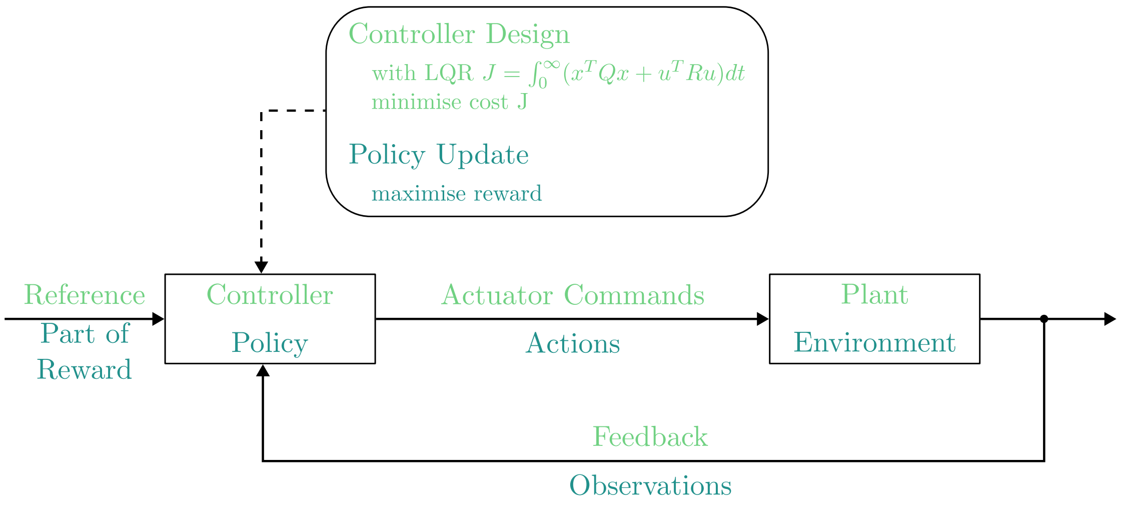


Figure 2.3: A comparison of optimal control and RL, adapted from [11]. The classic control loop involves the computation of a control error serving as an input to the controller. In RL, the control error is not explicitly calculated, but intrinsically captured by the DNN.

In control theory, the objective is to design a controller (policy), that maps the observed state feedback (observations) of the plant (environment) together with the reference signal (part of the reward function) to the best actuator commands (actions). The controller design can be thought of as a one-time policy update. One common approach to design an optimal controller is by minimising a cost functional, like in linear-quadratic regulator (LQR) design. Since cost is the negative of reward, the maximisation of rewards, which is the central optimisation objective in RL, is essentially solving the same problem.

In RL, an algorithm (agent) tries to learn the optimal control behaviour over time, instead of having the designer explicitly solving it. Another difference concerns knowledge about the system to be controlled: In control theory, a control engineer typically requires a model of the system, including an operating point, to solve the control problem. In contrast, the RL algorithm learns the right parameters on its own without specific a priori knowledge of the system dynamics.

In optimal control, the control engineer can influence the system behaviour by adjusting the cost function. Adjusting the weightings to states and inputs in LQR is similar to reward engineering in RL. However, a key distinction is the arbitrariness of the reward function. The reward function is not constrained to quadratic functions and can have any functional form of the current state, previous state, and the action just taken, as stated in (2.2).

# 3 Reinforcement Learning Algorithm

This chapter discusses the theoretical background of the Proximal Policy Optimization (PPO) algorithm [12], the model-free RL algorithm used in this thesis. The PPO algorithm is compatible with DNNs, making it suitable for solving complex problems with high-dimensional observation and action spaces. Compared to other policy optimisation algorithms, like Trust Region Policy Optimization (TRPO) [13], PPO is more stable during training and does not require extensive hyperparameter tuning. PPO has demonstrated strong empirical performance across different environments and benchmark tasks. At the time of writing this thesis, PPO is the state-of-the-art policy optimisation algorithm.

Implementations of popular RL libraries contain a lot of *code-level* optimisations. These implementation details dramatically change the agent's performance [14]. In this work, the PPO implementation of Stable-Baselines3 (SB3) [15] is utilised.

## 3.1 Background - Policy Optimisation and Q-Learning

RL algorithms can be categorised by their learning objectives, including policy $\pi$, value function $V(s)$, action-value function $Q(s,a)$ and/or a model of the environment. In model-free RL, the primary methods are policy optimisation algorithms and Q-Learning algorithms. Figure 3.1 provides a non-exhaustive overview of the most popular model-free RL algorithms.

### Policy Optimisation

Policy optimisation methods explicitly represent the policy as $\pi_\theta(a|s)$. The objective is to maximise a specific performance measure, $J(\pi_\theta)$, which is typically the cumulative reward. The policy parameters $\theta$ are optimised through gradient ascent on the performance objective, $\nabla_\theta J(\pi_\theta)$, referred to as the *policy gradient*. A policy optimisation algorithm updates the policy solely based on the data collected under the current policy version, resulting in a reduced sample efficiency compared to other RL algorithms. The core concept of policy gradients is presented in Section 3.2.

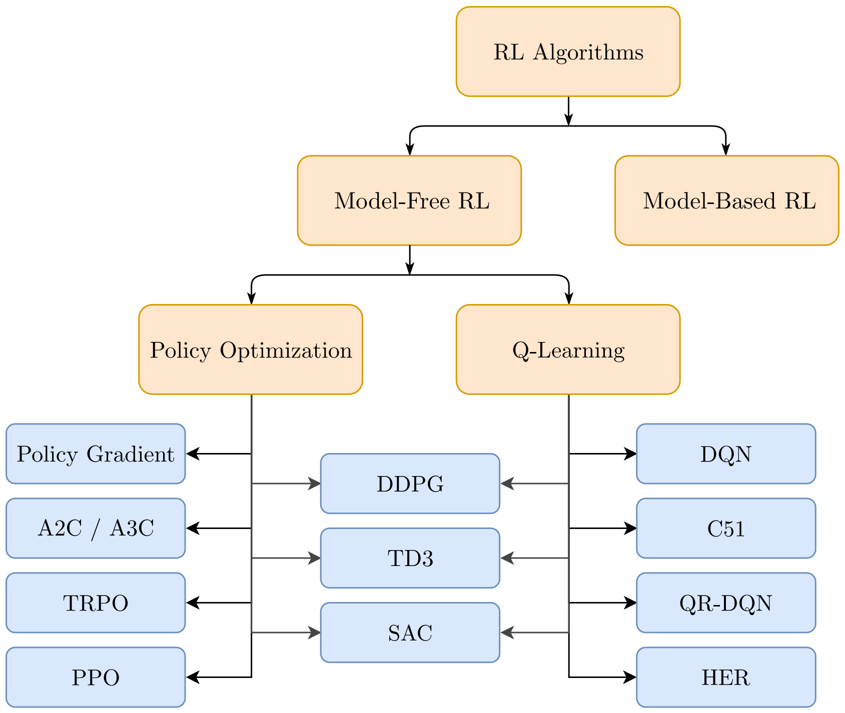


Figure 3.1: Overview of different RL algorithms, adapted from [10].

### Q-Learning

Methods in the family of Q-Learning algorithms aim to estimate the optimal action-value function by iteratively updating their Q-values. These values represent the expected return for taking specific actions in specific states. The Q-Learning agent selects the action with the highest Q-value for the current state

$$a(s) = \arg\max_a Q(s,a) \tag{3.1}$$

in order to receive the most expected return. During training, actions are selected according to (3.1) with a probability of $1-\epsilon$, and randomly selected with a probability of $\epsilon$, with $\epsilon \ll 1$. This $\epsilon$-greedy policy helps balancing between exploring unknown actions and exploiting known ones.

Q-learning algorithms typically learn in an off-policy way, including a replay buffer to reuse past experience instead of solely relying on the most recent data. The Q-Learning algorithm updates its Q-values according to

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)]\,, \tag{3.2}$$

where $\alpha$ is the learning rate, $r_{t+1}$ the received reward and $\gamma$ the discount factor. The difference between $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ and the expected return $Q(s_t, a_t)$ is called temporal difference. A larger temporal difference leads to a stronger update signal for the Q-values.

## 3.2 Policy Gradient

In this section, adapted from [10], the derivation of the policy gradient is presented. Considering both the policy and the environment as stochastic, the probability for a single $T$-step trajectory $\tau$ following policy $\pi_\theta$ is given by

$$P(\tau|\pi_\theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)\,. \tag{3.3}$$

The product on the right-hand side contains two probability distributions.

- $P(s_{t+1}|s_t, a_t)$: The probability of the environment to transition to a specific next state $s_{t+1}$, given that the environment is currently in state $s_t$ and the agent takes action $a_t$.
- $\pi_\theta(a_t|s_t)$: The probability of taking a specific action in a given state depending on the parameters $\theta$.

The expected return of policy $\pi_\theta$ is given by

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)] = \int_\tau P(\tau|\pi_\theta)G(\tau)\,, \tag{3.4}$$

i.e., the probability of a trajectory $\tau$ occurring under policy $\pi_\theta$, multiplied by the return of this trajectory $G(\tau)$, and integrated over all possible trajectories. Since the overall goal in RL is to find a policy that maximises this return, the central optimisation problem is formulated as

$$\pi^* = \arg\max_{\pi_\theta} J(\pi_\theta)\,, \tag{3.5}$$

where $\pi^*$ represents the optimal policy. The intuition is to alter the policy parameters $\theta$ in order to increase the probabilities of trajectories associated with higher rewards. This optimisation of the policy parameters is performed using gradient ascent

$$\theta_{k+1} = \theta_k + \alpha\, \nabla_\theta J(\pi_\theta)|_{\theta_k}\,, \tag{3.6}$$

with policy parameters $\theta_k$ and $\theta_{k+1}$ before and after the policy update, respectively. The learning rate $\alpha$ determines how much to change the policy parameters $\theta_k$ in the direction of the policy gradient $\nabla_\theta J(\pi_\theta)$. Applying the gradient with respect to $\theta$ to (3.4) yields

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \underset{\tau\sim\pi_\theta}{\mathbb{E}} [G(\tau)] = \int_\tau \nabla_\theta P(\tau|\pi_\theta) G(\tau) \,. \tag{3.7}$$

With the well-known *log-derivative trick*, the gradient of $P(\tau|\pi_\theta)$ with respect to $\theta$ follows

$$\nabla_\theta P(\tau|\pi_\theta) = P(\tau|\pi_\theta) \nabla_\theta \log P(\tau|\pi_\theta) \,, \tag{3.8}$$

which gives

$$\nabla_\theta J(\pi_\theta) = \int_\tau \left[ P(\tau|\pi_\theta) \nabla_\theta \log P(\tau|\pi_\theta) G(\tau) \right] . \tag{3.9}$$

In (3.7), the integration of the gradients of probabilities is performed, whereas in (3.9), the integration considers the probabilities themselves. With this result, the policy gradient itself can be written in expectation form

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}} \nabla_\theta \log P(\tau|\pi_\theta) G(\tau) \,, \tag{3.10}$$

which is crucial in sample-based methods like RL. Proceeding from (3.3), the logarithm of the probability of a trajectory yields

$$\log P(\tau|\pi_\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T} (\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)) \,. \tag{3.11}$$

In (3.11), only the last term depends on the policy parameters $\theta$. Thus, taking the gradient with respect to $\theta$ leads to

$$\nabla_\theta \log P(\tau|\pi_\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \,, \tag{3.12}$$

and the policy gradient

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}} \left[ \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G(\tau) \right] . \tag{3.13}$$

So far, the derivation of the policy gradient considers the true expectation, involving all possible trajectories. As this is computationally intractable, the expectation is estimated by a sample mean. Based on a finite set of $N$ trajectories $\mathcal{D} = \{\tau_i\}_{1 \le i \le N}$, the approximate policy gradient yields

$$\nabla_\theta J(\pi_\theta) \approx \hat{g} = \frac{1}{N} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G(\tau) \,. \tag{3.14}$$

If the policy is represented in a way that allows to calculate $\nabla_\theta \log \pi_\theta(a_t|s_t)$, i.e., differentiable with respect to $\theta$, then the policy gradient is computable and the policy can be updated.

When taking an update step of the policy gradient according to (3.13), the logarithmic probabilities of all actions of a specific trajectory $\tau$ are changed according to $G(\tau)$. However, actions should only be reinforced based on post-action consequences, as pre-action rewards are irrelevant. Therefore, the policy gradient can also be expressed as

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[ \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \right] , \tag{3.15a}$$

with the sum of discounted rewards after a point in a trajectory $t$

$$G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}, s_{t'+1}) \,. \tag{3.15b}$$

## Baselines in Policy Gradients

A key property of probability distributions like $\pi_\theta(a_t|s_t)$ is, that the total probability across all possible actions yields

$$\int_{a_t} \pi_\theta(a_t|s_t) = 1 \,. \tag{3.16}$$

Applying the gradient with respect to $\theta$ and using the same trick as in (3.8) leads to

$$\underset{a_t \sim \pi_\theta}{\mathbb{E}} \left[ \nabla_\theta \log \pi_\theta(a_t|s_t) \right] = 0 \,. \tag{3.17}$$

An arbitrary function $b(s_t)$ does not change the expectation in (3.17) and therefore

$$\underset{a_t \sim \pi_\theta}{\mathbb{E}} \left[ \nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t) \right] = 0 \,. \tag{3.18}$$

The function $b(s_t)$ is called a baseline and the result in (3.18) allows to modify the expression of the policy gradient (3.15)

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right] , \tag{3.19}$$

without changing the expectation. A common choice for a baseline is $V^\pi(s_t)$, the on-policy value function, which empirically reduces variance in the gradient estimate, leading to a faster and more stable training process. The advantage function

$$A_t = G_t - V^\pi(s_t) \tag{3.20}$$

relates the obtained return, as a consequence of the selected action, to the average return the agent expects. Intuitively, if the agent gets the return that it expects $A_t = 0$, it should *feel* neutral. In practice, as $V^\pi(s_t)$ cannot be computed, it has to be approximated, typically by a DNN $V_\phi(s_t)$ with parameters $\phi$. To represent the value function of the most recent policy, the value network is updated concurrently with the policy network. Learning value function $V_\phi(s_t)$ is a typical SL task with the objective to minimise the mean square error (MSE) between the output of the value function, i.e., the expected return, and the actually obtained return $G_t$

$$\phi_k = \arg\min_{\phi} \mathop{\mathbb{E}}_{s_t, G_t \sim \pi_k}\left[\left(V_\phi(s_t) - G_t\right)^2\right] , \tag{3.21}$$

with $\pi_k$ denoting the policy at epoch $k$. The minimisation of this MSE objective is done by gradient descent, starting from the value function parameters $\phi_{k-1}$ of the previous epoch. A setting like this, where the agent learns a policy and a value function, is called actor-critic. The policy represents the actor, and the value function criticises the action by comparing the obtained reward with the expected average reward.

## Gaussian Policy

The Gaussian policy is a common stochastic policy for continuous action spaces. It maps observations to mean actions $\mu_\theta(s)$ and standard deviations $\sigma_\theta(s)$. The actual actions are sampled from a parameterised normal distribution

$$\pi_\theta(a|s) = \frac{1}{\sqrt{2\pi}\sigma_\theta} e^{-\frac{1}{2}\left(\frac{a-\mu_\theta}{\sigma_\theta}\right)^2} . \tag{3.22}$$

Typically, the policy is represented as a DNN, so the mean value $\mu_\theta$ and the standard deviation $\sigma_\theta$ are nonlinear functions of the weights and bias of this DNN. Using calculus to manually compute partial derivatives is possible since the policy is differentiable, but cumbersome. Special frameworks, like PyTorch [16] or TensorFlow [17], offer automated gradient calculations. Like in other ML tasks, a loss function is defined

$$L = \log \pi_\theta(a_t|s_t) G(\tau)\,, \tag{3.23}$$

and the computer does the differentiation automatically. The loss function (3.23) is **not** a typical loss function like in SL, where it indicates how well the training is going. A falling loss indicates a successful training, as it describes the difference between the predicted labels and the true labels. In policy gradient methods, however, there is no intuition behind the loss function (see Figure 6.4). The only measure, indicating a successful training, is the average return.

## 3.3 Proximal Policy Optimisation

This section covers the main ideas of the DRL algorithm PPO, introduced by Schulman [12]. Summarising Section 3.2, the policy gradient can be written as

$$\hat{g} = \hat{\mathbb{E}}_t \left[ \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \,, \tag{3.24}$$

where $\hat{\mathbb{E}}_t$ indicates the empirical average of a finite batch of samples, similar to (3.14), and $\hat{A}_t$ the advantage function (3.20). In implementations like SB3 [15], that use automatic differentiation software like PyTorch [16], it is sufficient to construct an objective function, also called loss function, whose gradient is the estimator of the policy gradient

$$L^{\mathrm{PG}}(\theta) = \hat{\mathbb{E}}_t \left[ \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \,. \tag{3.25}$$

The policy parameters are updated through gradient ascent (3.6), with the learning rate $\alpha$ being a measure of how much the policy is changed in parameter space. However, even a minor change in the parameter space can have a significant impact, as a single bad policy update step may let the policy performance collapse. In the following epoch, the trajectories are collected under this bad policy, which can cause the agent to fail learning by getting stuck in an unusable area of the environment's state space.

TRPO [13], the direct predecessor of PPO, uses a special constraint on the size of the policy update to avoid this kind of collapse.

$$\text{maximise} \quad \hat{\mathbb{E}}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \tag{3.26a}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t \left[ \bar{D}_{KL} \left( \pi_\theta(\cdot|s_t) || \pi_{\theta_{\text{old}}}(\cdot|s_t) \right) \right] \leq \delta \tag{3.26b}$$

The *surrogate* objective in (3.26a) is a measure of how policy $\pi_\theta$ performs relative to the old policy $\pi_{\theta_{\text{old}}}$, based on data from the old policy. It approximates the actual performance difference and, for reasonably small policy updates, (3.26a) is not much different than the *original* objective function (3.25). In fact, differentiating both (3.26a) and (3.25) gives exactly the same gradient. The innovation is (3.26b), which represents a constraint in terms of the Kullback–Leibler divergence $\bar{D}_{\text{KL}}(\cdot||\cdot)$. This measure, illustrated in Figure 3.2, indicates how two probability distributions differ from each other. The constrained optimisation problem (3.26) ensures that the new updated policy does not deviate too much from the old policy in terms of action probabilities.

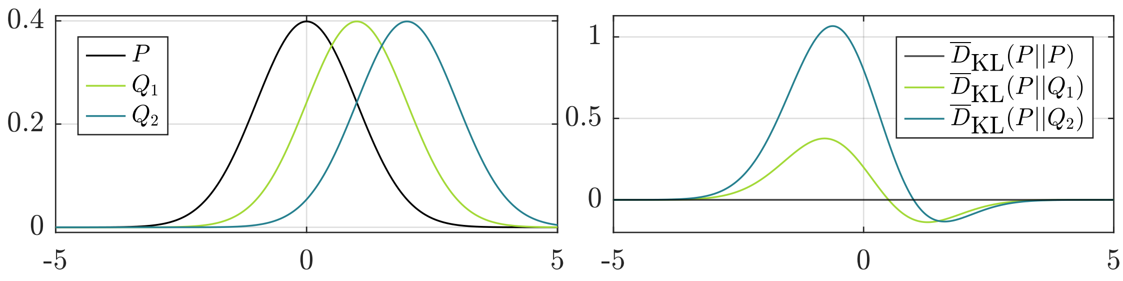


Figure 3.2: Kullback–Leibler divergence $\bar{D}_{\text{KL}}(\cdot||\cdot)$ as constraint in TRPO's optimisation problem. The Gaussian distributions $P$, $Q_1$ and $Q_2$, plotted on the left half, have mean values of $\mu_P = 0$, $\mu_{Q_1} = 1$ and $\mu_{Q_2} = 2$, respectively, and an equal standard deviation of $\sigma = 1$. The corresponding graphs of $\bar{D}_{\text{KL}}(\cdot||\cdot)$ are potted on the right half, where $\bar{D}_{\text{KL}}(P||P)$ aligns with the abscissa.

Despite its theoretical guarantees, TRPO faces practical challenges arising from its computational complexity of solving a constrained optimisation problem. On the other hand, PPO is a relatively simple algorithm that includes the extra constraint (3.26b) directly in the optimisation objective. Schulman introduced two versions of PPO in his work [12]. The more popular, easier to implement, and empirically better performing PPO-Clip is discussed here.

With the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, the objective function of the PPO algorithm can be written as

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[ \min \left( r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right) \hat{A}_t \right) \right] . \tag{3.27}$$

Clip range $\varepsilon$ determines how much two consecutive policies can differ in terms of action probabilities, commonly within the range $\varepsilon \in [0.1, 0.3]$. The first term inside the min-operator in (3.27) is equivalent to TRPO's objective (3.26a). The second term, $\text{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right) \hat{A}_t$, is similar to the first, but comprises a clipped version of the probability ratio $r_t(\theta)$. It acts as a regularisation mechanism by constraining the policy updates. The advantage estimate $\hat{A}_t$ can be both positive and negative, which changes the effect of the min operator. Figure 3.3 illustrates the objective function (3.27) for a single time step, called term, as a function of the probability ratio $r_t(\theta)$. The loss function $L^{\text{CLIP}}(\theta)$ is the empirical average of many of these terms. A positive advantage, i.e, an action achieved a better return than expected, is plotted on the positive ordinate, a negative advantage, i.e., an action yielded a worse return than expected, is plotted on the negative ordinate, respectively.

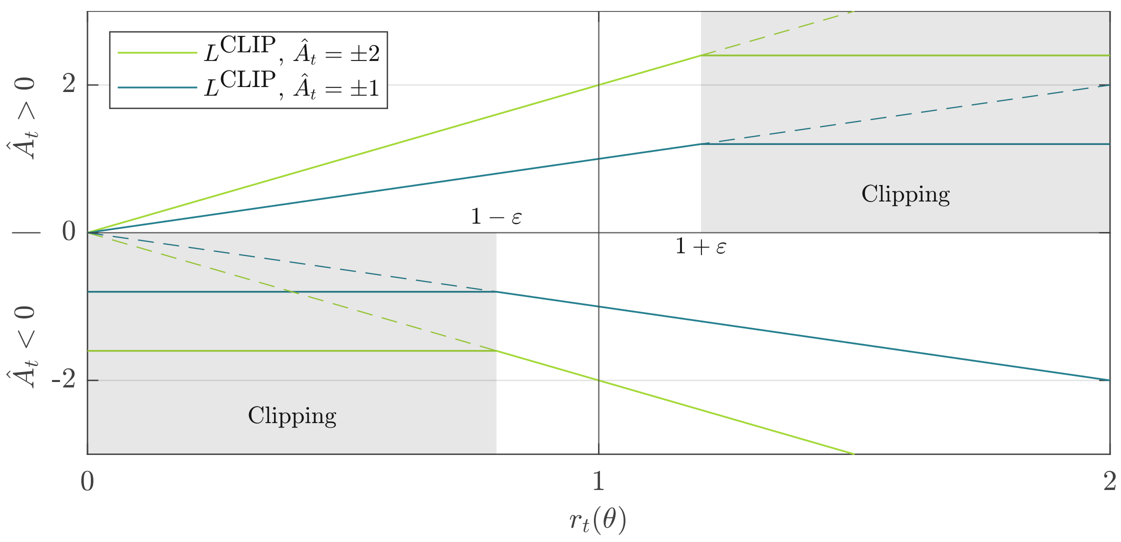


Figure 3.3: PPO's objective function for a single time step, adapted from [12], with clip range $\epsilon = 0.2$. When a data sample resides within the (grey) clipping area, the corresponding gradient is 0, signifying that the sample has no influence on the policy update process.

The following combinations of $r_t(\theta)$ and $\hat{A}_t$ can occur:

- $\hat{A}_t > 0$:
  The action produces a better-than-expected return, which is an incentive to increase the probability of it, characterised by a positive gradient. However, when $r_t(\theta)$ gets too high, the objective function flattens out and has zero gradient. This happens, when this action is already much more probable $r_t(\theta) > 1 + \varepsilon$ under the current policy than it was under the old policy. The objective function is clipped to ensure that the action update remains within a reasonable range.

- $\hat{A}_t < 0$ and $r_t(\theta) < 1 - \varepsilon$:
  This case corresponds to a bad action, characterised by a negative advantage, that is much less probable $r_t(\theta) < 1 - \varepsilon$ under the current policy than it was under the old policy. The clipping mechanism ensures that the probability is not reduced even further since the gradient is zero in this region.

- $\hat{A}_t < 0$ and $r_t(\theta) \gg 1$:
  Under the current policy, the probability of taking a bad action, characterised by a negative advantage, has increased. The negative gradient allows to undo the last policy update by reducing the probability of such an action again. It is the only region where the unclipped part of the objective (3.27) has a lower value than the clipped version and thus gets returned by the min operator.

The final objective function, which is optimised, comprises the clipped PPO objective (3.27), augmented by two additional terms

$$L^{\mathrm{PPO}}(\theta) = \hat{\mathbb{E}}_t \left[ L^{\mathrm{CLIP}}(\theta) - c_{\mathrm{vf}} L^{\mathrm{VF}}(\theta) + c_{\mathrm{ent}} S[\pi_\theta](s_t) \right] . \tag{3.28}$$

The term $L^{\mathrm{VF}}(\theta)$ represents the MSE value function loss, as stated in (3.21), while $S[\pi_\theta](s_t)$ denotes an entropy bonus to influence the agent's exploration behaviour. The value function coefficient $c_{\mathrm{vf}}$ and entropy coefficient $c_{\mathrm{ent}}$ regularise the importance of these terms relatively to the clipped PPO loss (3.27).

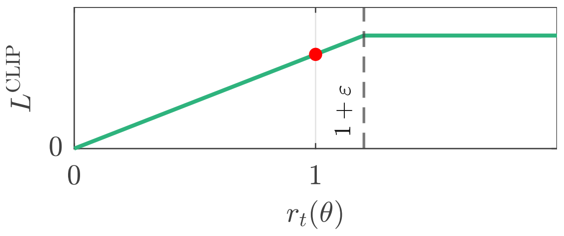
## Implementing PPO Algorithm

The PPO algorithm is relatively easy to implement, i.e., it requires only a few changes compared to implementing a standard policy gradient (3.25). The pseudocode of PPO, given in Algorithm 1, gives a brief step-by-step description of how this algorithm works.

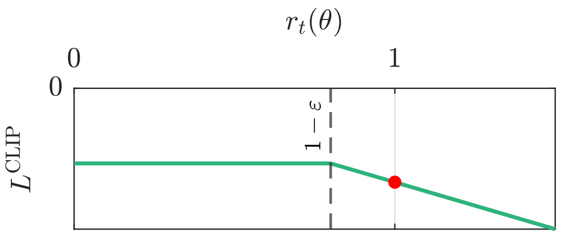
**Algorithm 1** PPO, Clipped (adapted from [12])

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $j = 1, 2, \dots$ do
3:     Collect set of trajectories $\mathcal{D}_j$ by running $\pi_{\theta_{\text{old}}}$ in the environment
4:     Compute advantage estimates $\hat{A}_t$
5:     for $k = 1, 2, \dots, K$ do
6:         Evaluate $\pi_\theta$ on $\mathcal{D}_j$, compute $r_t(\theta)$
7:         Compute and optimise surrogate loss $L$
8:     end for
9:     $\theta_{\text{old}} \leftarrow \theta$
10: end for
```

The set of trajectories in line 3 contains state-action pairs and rewards. With the latter, the advantage estimate $\hat{A}_t$ is computed. On the first iteration of the loop in line 5, the current policy $\pi_\theta$ **is** the old policy $\pi_{\theta_{\text{old}}}$, resulting in a probability ratio of $r_t(\theta) = 1$. This corresponds to the situation illustrated by the red dots in Figure 3.4. The gradient will increase the probabilities of good actions, and decrease the probabilities of bad actions, as illustrated in Figure 3.4a and Figure 3.4b, respectively. This results in the updated version of the current policy $\pi_\theta$. The following iterations through the loop in line 5 are different. The updated current policy $\pi_\theta$ is evaluated on the state-action pairs of the old policy $\pi_{\theta_{\text{old}}}$. Given the state and the action, the probability of that action under the current policy, as a consequence of the state, is calculated, which is then used to calculate the probability ratio $r_t(\theta)$. Depending on the value of $r_t(\theta)$ and the sign of $\hat{A}_t$, the data point either contributes (no clipping) or does not contribute (clipping) to the subsequent policy update.



(a) PPO Clip, $\hat{A}_t > 0$.



(b) PPO Clip, $\hat{A}_t < 0$.

Figure 3.4: Plots showing one term of the objective function (3.27) as a function of the probability ratio $r_t(\theta)$, for both positive (left) and negative (right) advantages, adapted from [12].

# 4 Modelling

This chapter introduces the simulation environment, which includes both a vehicle model and a driver model. First, a brief introduction is provided, followed by the physical equations to derive an analytical representation of the environment.

## 4.1 Overview

There are numerous options for modelling a car, depending on the relevant phenomena that the model needs to describe. The complexity of the model varies based on the level of accuracy necessary for the particular simulation. Simple quarter-vehicle models are capable of describing suspension behaviour or the contact between tire and road, while more complex four-wheel models can account for the interaction between all four wheels and the vehicle body. Section 4.2 introduces a two-wheel vehicle model comprising a semi-empirical tire model: the *Pacejka Magic Formula* [18].

The driver model's task is to ensure that the vehicle tracks the reference trajectory accurately. Represented by the driver model described in [19], a realistic driver behaviour is obtained. This model is augmented by incorporating an additional *look ahead* component. In [20], various approaches to this look ahead component are mentioned, and specifically, the *look ahead along the longitudinal axis of the vehicle* method is utilised in this work. The process of developing and tuning the model describing the driver's behaviour is outlined in [3]. The results of this process are presented in Section 4.4.

The system dynamics is described as state space model $\dot{x}_i = f(\mathbf{x}, \mathbf{u})$, where each state derivative $\dot{x}_i$ is an explicit function of the state vector $\mathbf{x}$ and input vector $\mathbf{u}$. This representation, as an initial value problem, enables the use of explicit ordinary differential equation (ODE) solvers, such as the Runge-Kutta method or the forward Euler method, to numerically integrate the equations of motion.

## 4.2 Vehicle

The two-wheel vehicle model, illustrated in Figure 4.1, describes the relevant lateral vehicle dynamics of regular cornering and the powerslide. The coordinate system is located at the center of mass (COM) of the vehicle. The x-axis aligns with the longitudinal axis of the vehicle, the y-axis points to the left in relation to the driving direction, and the z-axis points upwards, following the conventions of a Cartesian right-hand system. The front wheels and rear wheels on their respective axle are represented as one wheel. The COM is located in the plane of the road, ensuring constant vertical tire forces $F_{z,i}$ and eliminating dynamic axle load distributions. The model consists of three components: front wheel ① with rolling radius $r_{\text{front}}$ and reduced moment of inertia $I_{\text{front,red}}$, rear wheel ② with rolling radius $r_{\text{rear}}$ and reduced moment of inertia $I_{\text{rear,red}}$, and vehicle body ③ with dimensions $l_{\text{front}}$ and $l_{\text{rear}}$, total vehicle mass $m$ and moment of inertia with respect to the vertical axis $I_z$.

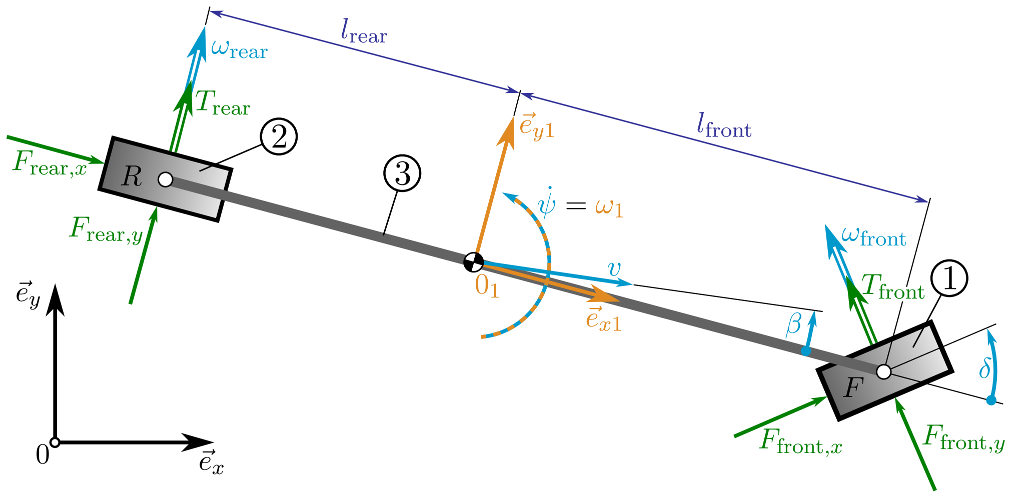


Figure 4.1: Two-wheel vehicle model with independently driven front and rear axle.

The state vector $\mathbf{x}_V$ and input vector $\mathbf{u}$ of the vehicle are defined as follows

$$\mathbf{x}_V = \begin{bmatrix} \beta & \dot{\psi} & v & \omega_{\text{front}} & \omega_{\text{rear}} \end{bmatrix}^T , \tag{4.1}$$

$$\mathbf{u} = \begin{bmatrix} \delta & T_{\text{front}} & T_{\text{rear}} \end{bmatrix}^T . \tag{4.2}$$

Here, $\beta$ denotes the vehicle side slip angle, $\dot{\psi}$ its yaw rate, $v$ the speed of the COM, $\omega_{\text{front}}$ and $\omega_{\text{rear}}$ the angular speeds of front and rear wheel, $\delta$ the steering angle and $T_{\text{front}}$ and $T_{\text{rear}}$ the drive torques to front and rear axle.

The equations for the balance of momentum and the balance of angular momentum regarding the entire system yield

$$\begin{aligned}
\vec{e}_{x1}:&\quad m[\dot{v}\cos\beta - v(\dot{\beta}+\dot{\psi})\sin\beta] = F_{\text{front},x}\cos\delta - F_{\text{front},y}\sin\delta + F_{\text{rear},x}\,, && (4.3)\\
\vec{e}_{y1}:&\quad m[\dot{v}\sin\beta + v(\dot{\beta}+\dot{\psi})\cos\beta] = F_{\text{front},x}\sin\delta + F_{\text{front},y}\cos\delta + F_{\text{rear},y}\,, && (4.4)\\
\vec{e}_{z1}:&\quad \ddot{\psi}I_z = (F_{\text{front},x}\sin\delta + F_{\text{front},y}\cos\delta)l_{\text{front}} - F_{\text{rear},y}l_{\text{rear}}\,, && (4.5)
\end{aligned}$$

where the tire forces $F_{\text{front},x}$, $F_{\text{front},y}$, $F_{\text{rear},x}$ and $F_{\text{rear},y}$ are oriented as illustrated in Figure 4.1. The equations for balance of momentum for front axle and rear axle lead to

$$\dot{\omega}_{\text{front}}I_{\text{front,red}} = T_{\text{front}} - r_{\text{front}}F_{\text{front},x}\,, \tag{4.6}$$

$$\dot{\omega}_{\text{rear}}I_{\text{rear,red}} = T_{\text{rear}} - r_{\text{rear}}F_{\text{rear},x}\,. \tag{4.7}$$

Rearranging the system of nonlinear first order ODEs, from (4.3) to (4.7), leads to the explicit expression $\dot{\mathbf{x}}_V = \mathbf{f}(\mathbf{x}_V, \mathbf{u})$

$$\begin{bmatrix} \dot{\beta} \\ \ddot{\psi} \\ \dot{v} \\ \dot{\omega}_{\text{front}} \\ \dot{\omega}_{\text{rear}} \end{bmatrix} = \begin{bmatrix} -\dot{\psi} + \frac{F_{\text{front},x}\sin(\delta-\beta)+F_{\text{front},y}\cos(\delta-\beta)-F_{\text{rear},x}\sin\beta+F_{\text{rear},y}\cos\beta}{mv} \\ \frac{(F_{\text{front},x}\sin\delta+F_{\text{front},y}\cos\delta)l_{\text{front}}-F_{\text{rear},y}l_{\text{rear}}}{I_z} \\ \frac{F_{\text{front},x}\cos(\delta-\beta)-F_{\text{front},y}\sin(\delta-\beta)+F_{\text{rear},x}\cos\beta+F_{\text{rear},y}\sin\beta}{m} \\ \frac{T_{\text{front}}-r_{\text{front}}F_{\text{front},x}}{I_{\text{front,red}}} \\ \frac{T_{\text{rear}}-r_{\text{rear}}F_{\text{rear},x}}{I_{\text{rear,red}}} \end{bmatrix}. \tag{4.8}$$

### 4.2.1 Tires

The tire forces are modelled by the semi-empirical *Pacejka Magic Formula* [18]. According to the two-wheel vehicle model, the forces acting on the left and right front tires, as well as the left and right rear tires, are summed up on their respective axes. The tire model represents the tire forces as a nonlinear function

$$F_{i,x|y} = f(F_{i,z}, s_{i,x}, \alpha_i, \mu)\,, \tag{4.9}$$

with vertical tire force $F_{i,z}$, tire longitudinal slip $s_{i,x}$, tire side slip angle $\alpha_i$ and friction potential of contact between tire and road $\mu$. Subscript $i = \{\text{front}, \text{rear}\}$ indicates front tire and rear tire, respectively. Both $s_{i,x}$ and $\alpha_i$ are derived from kinematic relations between tire and road, illustrated in Appendix A.1.

## 4.3 Path

The objective of this section is to obtain a set of equations describing the location of the vehicle. This is achieved by utilising a curvilinear coordinate system, where the vehicle's position and orientation is determined relatively to the reference trajectory. The coordinate system includes an x-axis aligned with the tangent to the reference trajectory, a y-axis pointing towards the vehicle's COM, and a z-axis oriented upwards, following the conventions of a Cartesian right-hand system.

In the plane, the position of a rigid body is specified by three independent coordinates. The state vector describing this position

$$\mathbf{x}_P = \begin{bmatrix} s_s & s_n & \alpha \end{bmatrix}^T \tag{4.10}$$

comprises the arc length $s_s$, the lateral distance $s_n$ and the angle from the tangent of the trajectory to the vehicle's longitudinal axis $\alpha$ . The coordinate system and $\mathbf{x}_P$ are illustrated in Figure 4.2.

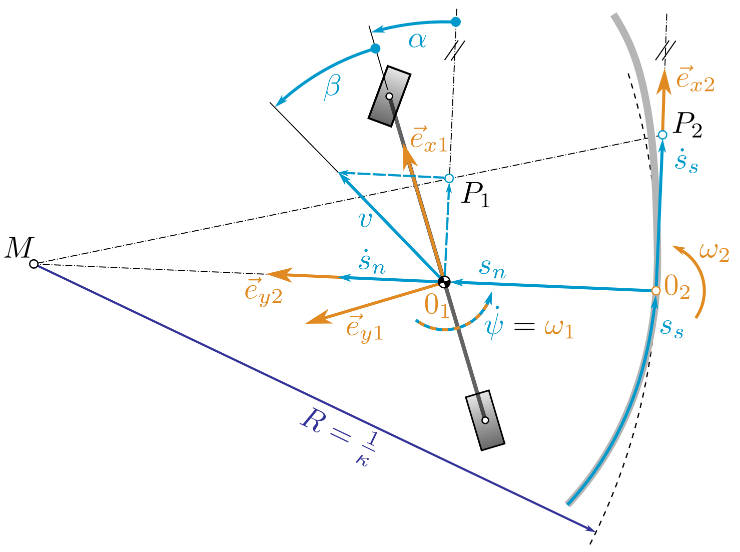


Figure 4.2: Vehicle's location and orientation relative to a given path.

Considering the right triangles $M0_2P_2$ and $M0_1P_1$, the dynamics of $s_s$ and $s_n$ can be expressed as

$$\dot{s}_s = v\frac{\cos(\alpha+\beta)}{1 - s_n\kappa}\,, \tag{4.11a}$$

$$\dot{s}_n = v\sin(\alpha+\beta)\,, \tag{4.11b}$$

with curvature $\kappa$ as the inverse of radius $R$. The difference between the vehicle's yaw rate $\dot{\psi}$ and the rotation of the coordinate system $\omega_2 = \frac{\dot{s}_s}{R}$ yields the dynamics of $\alpha$

$$\dot{\alpha} = \dot{\psi} - \frac{v\kappa\cos(\alpha+\beta)}{1-s_n\kappa}\,. \tag{4.11c}$$

## 4.4 Driver

Most powerslide controllers described in literature are primarily designed for autonomous drifting. However, in this thesis, it is essential to consider the presence of a human driver in the vehicle. This consideration is crucial because the controller, referred to as the agent, responsible for stabilising the vehicle, must account for the behaviour of a human driver. In [3], the design process of the driver model is discussed. The controller derived from this model consists of a feedforward (ff) component and a feedback (fb) component, with their corresponding contributions $\delta_{\text{ff}}$ and $\delta_{\text{fb}}$ to the steering angle $\delta = \delta_{\text{ff}} + \delta_{\text{fb}}$ of the vehicle.

### 4.4.1 Feedforward Controller

The feedforward controller maps the vehicle side slip angle $\beta$ to the feedforward portion of the steering angle $\delta_{\text{ff}}$. The signal passes through an additional first-order transfer function $G(s) = 1/(1+Ts)$, with time constant $T$. Consequently, an extra state variable, denoted as $x_{\text{ff}}$, is included to represent these controller dynamics within the equations describing the state space model

$$\dot{x}_{\text{ff}} = a x_{\text{ff}} + b(\beta + \Delta_{\text{ff}})\,, \tag{4.12a}$$

$$\delta_{\text{ff}} = c x_{\text{ff}}\,. \tag{4.12b}$$

In (4.12), the term $\Delta_{\text{ff}}$ represents a small constant used for mapping the vehicle side slip angle $\beta$ to the feedforward steering angle $\delta_{\text{ff}}$. The feedforward controller's objective is to ensure that the front wheel remains roughly parallel to the target trajectory of the vehicle. The constants $a$, $b$ and $c$ are selected to match the behaviour of the chosen transfer function.

### 4.4.2 Feedback Controller

The feedback component of the driver controller, which is implemented as a real Proportional Derivative (PD) controller with dead time, considers the vehicle's lateral deviation from the target trajectory. However, instead of directly accessing the actual lateral distance $s_n$, the control error is derived from the modified lateral distance $\tilde{s}_n$. From the driver's perspective, this modified lateral distance $\tilde{s}_n$ is determined by visually projecting ahead along the vehicle's longitudinal axis for a given look-ahead time $t_{\text{look}}$ resulting in a look-ahead distance $x_{\text{look}} = v t_{\text{look}}$. By subtracting the steady-state lateral distance $\tilde{s}_{n,0}$ from this projected distance, the driver obtains a measure of the current lateral position relative to the desired path. These considerations are summarised in Figure 4.3.

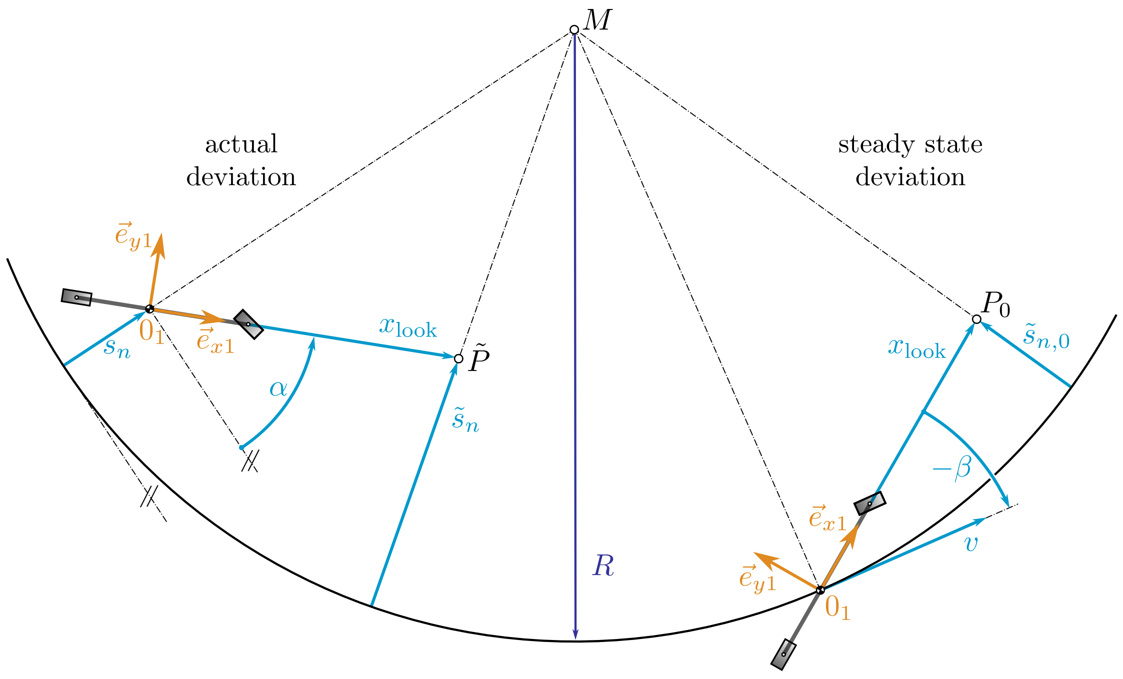


Figure 4.3: Driver's look ahead along the vehicle's longitudinal axis.

Applying the cosine theorem to triangles $0_1\tilde{P}M$ and $0_1P_0M$ in the left and right halves of Figure 4.3, respectively, leads to the lateral distances

$$\tilde{s}_n = R - \sqrt{(R - s_n)^2 + x_{\text{look}}^2 - 2x_{\text{look}}(R - s_n)\cos\left(\frac{\pi}{2} - \alpha\right)}\,, \tag{4.13a}$$

$$\tilde{s}_{n,0} = R - \sqrt{R^2 + x_{\text{look}}^2 - 2Rx_{\text{look}}\cos\left(\frac{\pi}{2} + \beta\right)}\,. \tag{4.13b}$$

The difference between (4.13a) and (4.13b), denoted as $e_{\mathrm{fb}} = \tilde{s}_n - \tilde{s}_{n,0}$, represents the control error that serves as input to the real PD controller. The feedback controller's transfer function is given by

$$G(s) = K_P \frac{1 + T_V s}{1 + T_N s} e^{-\tau s} \,, \tag{4.14}$$

where $K_P$ represents the proportional gain, $T_V$ and $T_N$ are the time constants, and $\tau$ is the dead time. This transfer function characterises the dynamics of the PD controller used in the feedback component of the driver controller. Transfer function (4.14) can be transformed into a state space representation by introducing the state variable $x_{\mathrm{fb}}$. Omitting the dead time component, the state space equations follow

$$\dot{x}_{\mathrm{fb}} = \frac{e_{\mathrm{fb}} - x_{\mathrm{fb}}}{T_N} \,, \tag{4.15a}$$

$$\delta_{\mathrm{fb}} = K_P [x_{\mathrm{fb}} + \frac{T_V}{T_N} (e_{\mathrm{fb}} - x_{\mathrm{fb}})] \,. \tag{4.15b}$$

The overall driver state can be summarised to

$$\mathbf{x}_D = \begin{bmatrix} x_{\mathrm{ff}} & x_{\mathrm{fb}} \end{bmatrix}^T \,. \tag{4.16}$$

# 5 Gym Interface

This chapter introduces OpenAI Gym [21], an open-source Python library that serves as a framework for RL. This library provides a standardised interface for the interaction between agents and environments and includes a range of pre-built environments that developers can use to test and benchmark their RL algorithms. Additionally, it facilitates the creation of custom environments, where RL algorithms can be trained on, as done in this work. The model described in Chapter 4 is implemented as an *Env*, a high-level Python class representing the environment within the RL theory. The DRL algorithm PPO, covered in Chapter 3, is used to solve the environment, specifically the PPO implementation of SB3 [15].

## 5.1 Gym Env Overview

Each environment model implemented as a Gym environment inherits from the abstract base class for RL environments in the OpenAI Gym library. A Gym environment includes the methods shown in Figure 5.1, with the first three (`__init__`, `reset`, `step`) being mandatory.

The `__init__` method, also known as constructor, is a special method in Python classes that is automatically called when a class object is created, like `env = CustomEnv(...)`. This method defines the environment parameters, observation space and action space. At the beginning of each episode, the `reset` method is called to reset the environment to an initial state. The first observation of the episode is then returned to the agent. The agent and the environment interact through the `step` method. When the agent sends an action, the environment transitions to a new state. In return, the environment provides feedback to the agent, including a new set of observations, the reward and the information if the episode has ended. The optional `render` method creates a graphical visualisation of the environment transitions. The typical use case is the evaluation of a trained policy. To save computational resources, this method should not be called during training. The optional `close` method closes any open computational resource that was used by the environment, like a rendering used for visualisation.

```
class CustomEnv(gym.Env):

    def __init__(self, parameters):

    def reset(self):
        return observation, info

    def step(self, action):
        return observation, reward, done, info

    def render(self):

    def close(self):
```

Figure 5.1: Structure of a Gym environment.

To modify an existing environment without changing the core implementation code, wrappers offer a suitable solution. To wrap an environment, it must be instantiated first. Together with possible parameters, this environment is passed to the wrapper's constructor `wrapped_env = Wrapper(env, parameters)`. Section 5.3 introduces the utilised wrappers for the powerslide environment discussed in Section 5.2.

## 5.2 PowerslideEnv

The implemented structure of the `PowerslideEnv` is visualised in Figure 5.2. To ensure the readability of the code, the model's individual parts, discussed in Chapter 4, pose an own class. These classes are instantiated in the `__init__` method of the main class, with the relevant parameters passed to their constructors, as shown by the dashed lines in Figure 5.2. During environment transitions, represented by the `step` method, these classes interact with the main class, as illustrated by the solid lines in Figure 5.2. In `Driver`, the lateral deviation (4.13) and the steering angle (4.15) are calculated. Class `Tire_Forces` calculates the input variables (A.1)-(A.6) for the *Pacejka Magic Formula* [18] (4.9), which is implemented in `Pacejka_Tire`.

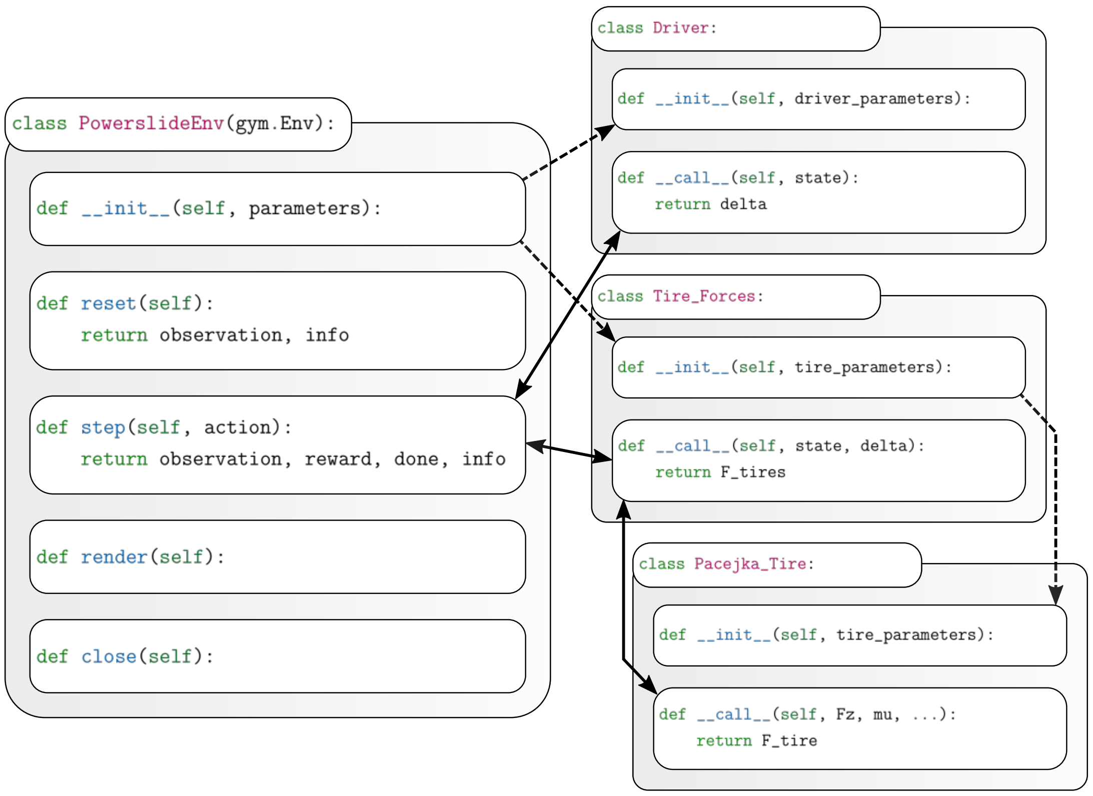

class Driver:
def __init__(self, driver_parameters):
def __call__(self, state):
return delta
class PowerslideEnv(gym.Env):
def __init__(self, parameters):
def reset(self):
return observation, info
def step(self, action):
return observation, reward, done, info
def render(self):
def close(self):
class Tire_Forces:
def __init__(self, tire_parameters):
def __call__(self, state, delta):
return F_tires
class Pacejka_Tire:
def __init__(self, tire_parameters):
def __call__(self, Fz, mu, ...):
return F_tire

Figure 5.2: Implementation structure of the PowerslideEnv class.

### 5.2.1 __init__ - Method

This function is automatically called when an object is created from the PowerslideEnv class. It takes the following arguments:

```
def __init__(self, start_mode, track, beta_target, render_mode):
```

The `start_mode` argument specifies whether the vehicle starts in the regular cornering mode or powerslide mode at the start of an episode. With the argument `track` the range of possible track conditions is determined, including the road radius $R$ and the friction potential $\mu$. The argument `beta_target` sets the range of possible target side slip angles of the vehicle. The Boolean `render_mode` argument indicates whether rendering is on or off.

In the PowerslideEnv constructor, all parameters of vehicle, tire, driver are set, and the simulation settings are initialised. The ODE solver's step size $\Delta t_{\text{solver}} = 0.001\,\text{s}$ ensures accurate numerical integration while the controller's step size $\Delta t_{\text{controller}} = 0.01\,\text{s}$, which corresponds to the step method, aligns with the electronic control unit (ECU) sampling rate on the real-world test vehicle. A single call to the step method corresponds to ten calls of the ODE solver.

Further, the set of valid actions and observations is defined. In Gym, these mathematical sets are referred to as spaces and every Gym environment must have the attributes action_space and observation_space. The Box space is used for continuous actions and observations. This space does not only specify the dimensions of actions and observations, but also defines the lower and upper bounds for each individual element. The action_space is determined based on the electric drive of the test vehicle. Considering a maximum available drive torque of 3000 Nm on the front axle and 4000 Nm on the rear axle, this space is defined as

```
low_acts = np.array([0, 0])
high_acts = np.array([3000, 4000])
self.action_space = spaces.Box(low=low_acts, high=high_acts)   .
```

The lower limit of 0 Nm indicates that the agent cannot apply any brake torques. The observation_space, similar to the action_space, is selected based on the test vehicle, specifically the sensors available in the vehicle. It consists of the state vector of the vehicle (4.1), the steering angle $\delta$ and the vehicle side slip angle reference signal. The values for the corresponding lower and upper bounds, given in Appendix A.2, are chosen reasonably. Their numerical impact on the training is discussed in Section 5.3.

### 5.2.2 reset - Method

This method is called to bring the environment back to an initial state, typically after an episode has ended. New tuples of path radius $R$, friction potential $\mu$ and target vehicle side slip angle $\beta_{\text{target}}$ are set. For exploratory reasons, the initial speed is selected randomly within a reasonable range and the other states are calculated referring to a steady state of regular cornering or powersliding according to the start_mode argument. This results in the environment's initial state vector comprising vehicle states (4.1), path states (4.10) and driver states (4.16)

$$\mathbf{x} = \begin{bmatrix}\beta & \dot{\psi} & v & \omega_{\text{front}} & \omega_{\text{rear}} & s_s & s_n & \alpha & x_{\text{ff}} & x_{\text{fb}}\end{bmatrix}^T . \tag{5.1}$$

The initial observation, which is returned to the agent, is a subset of (5.1) augmented by the vehicle side slip angle reference

$$\mathbf{o} = \begin{bmatrix}\beta & \dot{\psi} & v & \omega_{\text{front}} & \omega_{\text{rear}} & \delta & \beta_{\text{target}} & \beta_{\text{inter}}\end{bmatrix}^T , \tag{5.2}$$

where $\beta_{\text{target}}$ denotes the steady-state powerslide vehicle side slip angle and $\beta_{\text{inter}}$ the predefined curve of how to reach $\beta_{\text{target}}$. Starting from the initial side slip angle $\beta$, $\beta_{\text{inter}}$ describes a ramp function with a slope of $-35\,°/4\,\text{s}$ terminating at $\beta_{\text{target}}$. The slope's value is derived from expert knowledge.

### 5.2.3 step - Method

The `step` method represents the interaction between agent and environment. According to its policy, the agent sends an action, as reaction to the previous observation, to the environment. The environment transitions to a next state consequently and provides feedback to the agent, consisting of the next observation, the immediate reward, a termination signal and possible additional information:

```
def step(self, action):
    return observation, reward, done, info .
```

The termination of the episode can be caused either when the vehicle leaves the track or when its side slip angle is out of range.

- $|s_n| > s_{n,\text{max}}$: The vehicle deviates from the reference path by a distance greater than $s_{n,\text{max}}$.
- $|\beta - \beta_{\text{inter}}| > \beta_{\text{dev,max}}$: The vehicle side slip angle $\beta$ deviates from the side slip angle reference $\beta_{\text{inter}}$ by more than $\beta_{\text{dev,max}}$.

If one of these conditions is met, the `step` method will return `done=True`. This early termination of the episode prevents the agent from exploring areas of the state space that would contribute irrelevant training data to the policy update process.

### 5.2.4 render - Method

This optional method helps evaluating a policy. Figure 5.3 shows a snapshot of the `render` method. The current location and orientation of the vehicle according to the path and the reference vehicle side slip angle can be observed.

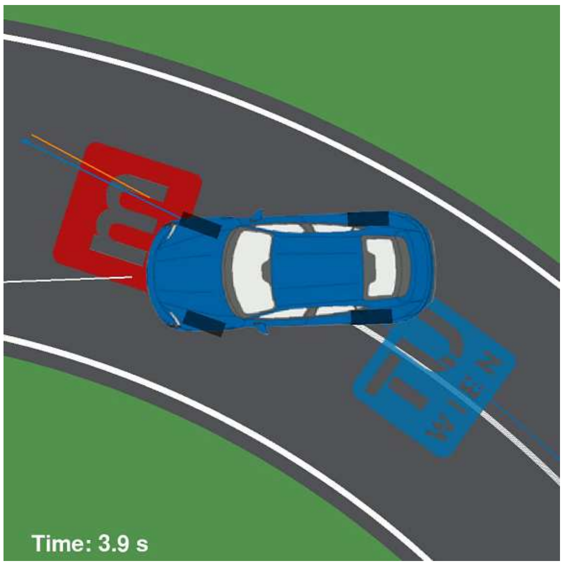


Figure 5.3: Screenshot of environment rendering. The lines in front of the vehicle indicate its side slip angle tracking behaviour. Path tracking is symbolised by the line behind the vehicle.

### 5.2.5 Powerslide-Reward

The shape of the reward function $R$, specifies the desired behaviour of the agent. The vehicle should track the reference side slip angle while following a reference path. Thus the reward function

$$R\left(\beta, s_n\right) = \sum_{i=1}^{2} w_i R_i = w_{\text{slip}} R_{\text{slip}}\left(\beta\right) + w_{\text{path}} R_{\text{path}}\left(s_n\right), w_i \in [0, 1] \tag{5.3}$$

is defined as a weighted sum of the reference vehicle side slip angle tracking and the trajectory following reward $R_{\text{slip}}$ and $R_{\text{path}}$, respectively. $R_{\text{slip}}$ is assigned a higher weight, empirically $w_{\text{slip}} = 0.75$. Reward terms $R_i$ are defined as bell-shaped functions

$$R_i = \exp\left(-c_i \Delta_i^2\right), \tag{5.4}$$

with shaping parameters $c_i$, vehicle side slip angle tracking error $\Delta_{\text{slip}} = \beta - \beta_{\text{inter}}$ and path following error $\Delta_{\text{path}} = s_n$. As illustrated in Figure 5.4, reward function $R > 0$ and the gradient $\nabla R$ exists and is finite. High $|\Delta_i|$ result in vanishing gradients, however these situations are avoided by terminating the episode as stated in 5.2.3. At every time step $t$, the reward $r_t = R\left(\beta_t, s_{nt}\right)$ is positive, encouraging the agent to prolong the episode.

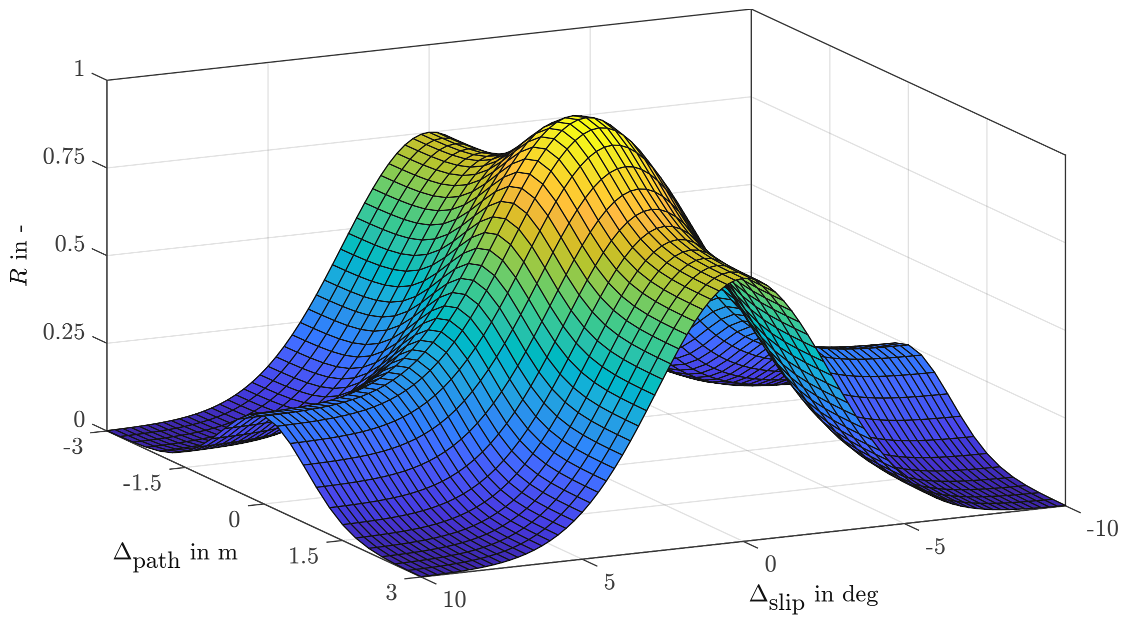


Figure 5.4: Reward function $R$ penalising vehicle side slip angle tracking error $\Delta_{\text{slip}}$ and path following error $\Delta_{\text{path}}$.

## 5.3 Wrappers

Wrappers provide a convenient way for modifying an existing environment implementation, without changing the codebase. Besides a `Reward` Wrapper customising the reward function, the environment described in Section 5.2 is wrapped by the following two.

### `TimeLimit` Wrapper

If none of the termination conditions described in Section 5.2.3 is ever fulfilled, the episode would last forever. The `TimeLimit` wrapper truncates the episode when a maximum number of steps is reached. The reason for truncation and the important difference to termination is discussed in Section 2.2.

### `NormaliseSpace` Wrapper

For numerical reasons, it is recommended that the inputs and outputs of a DNN have similar magnitudes, typically within the range $[-1, 1]$. In Section 5.2.1, the `action_space` and `observation_space` define the numerical bounds of the individual actions and observations. According to these bounds, the `NormaliseSpace` wrapper scales the original observation of the environment and rescales the network output.

# 6 Training Procedure

This chapter covers the relevant parts of the RL training procedure. It includes the initialisation of environment and agent, as well as the monitoring of the training process. Computational resources of the Vienna Scientific Cluster (VSC) accelerated the training processes, and thus had a major impact on this work.

## 6.1 Training Setup

To optimise performance, training data is collected from 40 environments running in parallel. Provided by SB3 [15], a wrapper called *SubprocVecEnv* distributes each environment into its own process. This results in a notable speedup, especially for computationally complex environments like the `PowerslideEnv`, described in Section 5.2. Practice guidelines from SB3 [15] mention, that the number of environments should not exceed the number of logical computing cores, for performance reasons. Since the training is carried out on the VSC, sufficient cores are available. Applying the `TimeLimit` wrapper, described in Section 5.3, the truncation limit of each environment is set to 4000 steps which corresponds to a maximum episode length of 40 s. This is long enough to see if the agent is able to stabilise the powerslide.

Initialising the agent involves configuring it with a set of hyperparameters. A set that performed successfully is given in Appendix A.3. Further, the network architecture of the actor and critic network are defined. While the dimensions of the input and output layer are determined by the observation space and action space, respectively, the number and size of the hidden layers can be adjusted. A Multilayer Perceptron (MLP) with 3 hidden layers, each consisting of 16 neurons resulting in more than 700 parameters to be tuned, has demonstrated good results, as shown in Chapter 7.

### 6.1.1 Callbacks

Callbacks are functions that are called on specific events during training. SB3 [15] provides a set of useful callbacks and also allows the creation of custom callbacks. Two callbacks utilised in this work are outlined in the following.

### EvalCallback

Included in the callback collection of SB3, this callback periodically evaluates the performance of an agent in a separate evaluation environment. In contrast to the training period, where actions are sampled stochastically from Gaussian distributions, the evaluating period is characterised by actions that are selected deterministically, i.e., the mean vales of the Gaussian distributions.

By aligning the evaluation frequency with the epoch length, the evaluation occurs after each policy update. During evaluation, the average sum of rewards obtained under the current policy is calculated. Whenever this result surpasses the previous highscore, the policy undergoing evaluation is saved as the new best model.

### TensorBoardCallback

TensorBoard is a visualisation toolkit provided by [17] that can monitor the training progress in real-time. By default, training variables such as the losses of the policy gradient and the value function, frames, i.e., number of observations per second, and information about the rollout, like the mean episode length and the mean sum of rewards, are displayed. The toolkit can also be used to monitor further variables by applying a customisable callback. Some of the resulting graphs are discussed in Section 6.2.

## 6.2 Training Process

For documentation and traceability purposes, essential information of the current training and its associated environment is saved at the beginning of the training, e.g., the hyperparameters, reward function and observation space. This helps to keep track of the changes made between different training sessions. Once training has started, its progress can be tracked by a TensorBoard session. To initiate this session, the following command must be executed in the command line interface:

```
tensorboard --logdir_spec NAME1:PATH/TO/FOLDER, NAME2: ...
```

After execution, accessing the provided link below will open the TensorBoard session:

```
TensorBoard 2.14.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

### 6.2.1 Training Metrics

In this section, training metrics from two different training sessions are compared and discussed. The first training session, denoted as T1 and represented by the green colour, is successful, while the second, denoted as T2 and represented by the blue colour, does not achieve satisfactory results. Both training sessions are initialised under identical conditions, including the same reward function, hyperparameters, and other relevant settings, with the sole exception being the difference in the number of hidden layers. In training T1, an MLP with 3 hidden layers is trained, while in training T2, the number of hidden layers is 2. The number of neurons per layer remains identical at 8 and the Exponential Linear Unit (ELU) function serves as activation function. The figures in this section illustrate some of the metrics provided by TensorBoard. These metrics are plotted against the number of environment steps, where the distance between two points along the abscissa corresponds to the size of the rollout buffer.

## Training Success

The graphs in Figure 6.1 provide an overview of the training success. While training T1 shows improvement over time, the progress of training T2 is poor. The maximum episode length in the powerslide environment is constrained by the truncation limit of 4000 steps. With the weightings $w_i$ of the reward function (5.3) satisfying $\sum w_i = 1$, the reward per time step is limited to the range $[0, 1]$. Consequently, the average episode length is an upper bound to the average return. The RL algorithm PPO demonstrates stable learning behaviour, as the learning curve does not collapse, once high average returns are received.

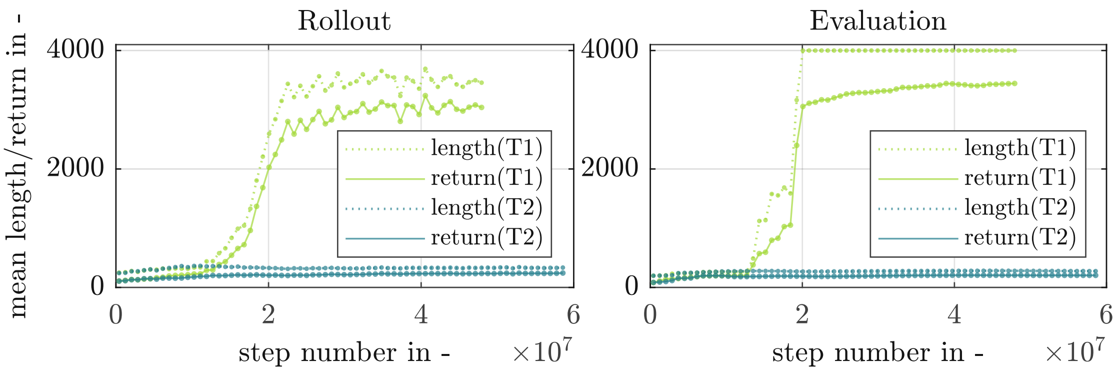


Figure 6.1: Average length and achieved cumulative reward per episode. The rollout period with stochastic actions is shown on the left and the evaluation period with deterministic actions on the right.

During evaluation, as shown in the right half of Figure 6.1, the agent typically performs better compared to the rollout period, shown in the left half of the same figure. This difference arises due to the fact that during the rollout period, the agent explores by randomly sampling actions from a Gaussian distribution, whereas during evaluation, it exploits the best known actions, represented by the mean values of the same distribution. For the successful training T1, the increase in the sum of rewards until environment step $2 \times 10^7$ is primarily due to the rising episode length. All evaluations from that step onwards share the same episode length of 4000. The increasing performance, expressed by the sum of rewards, from then on is the result of a better tracking performance regarding the vehicle side slip angle $\beta$ and the path $s_n$ according to the reward function (5.3).

## Agent and Environment

While Figure 6.1 gives a general overview of the training progress, it does not offer insights into the agent's behaviour or the corresponding responses of the environment. To address this issue, additional information is provided by Figure 6.2 and Figure 6.3. Importantly, these graphs present average values over an entire rollout, the temporal behaviour cannot be visualised using TensorBoard. To illustrate an episode over time, a separate evaluation, provided in Chapter 7, is necessary.

Figure 6.2 illustrates the mean drive torques of front axle (left) and rear axle (right). The first values of the plots are located in a similar region. Caused by the initialisation of the MLP's parameters, the untrained MLP outputs action values close to 0. According to the `NormaliseSpace` wrapper provided in Section 5.3, the action values around 0 are transformed to the mean values of the corresponding action space bounds given in Section 5.2.1. Therefore, the initial values of $T_{\text{front}}$ and $T_{\text{rear}}$ are located in the proximity of 1500 Nm and 2000 Nm, respectively. In the case of the successful training T1, the mean values of $T_{\text{front}}$ and $T_{\text{rear}}$ decrease during the training process while those of training T2 remain at higher levels.

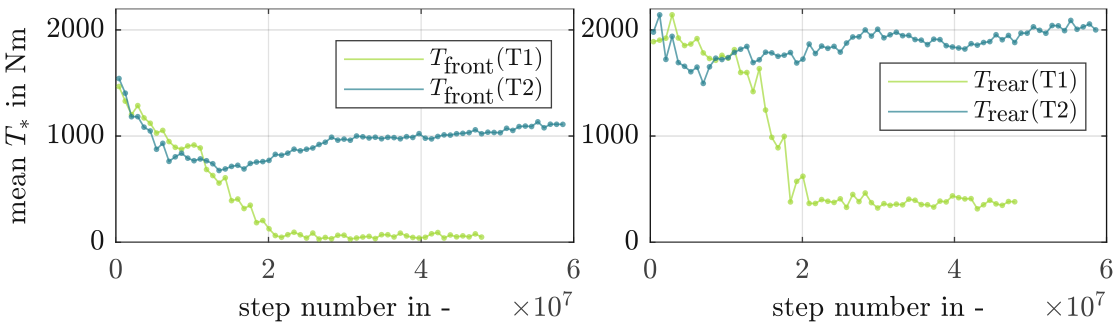


Figure 6.2: Average drive torques $T_*$ applied to front axle (left) and rear axle (right) during the rollout period.

According to the reward function (5.3), the goal is to track the reference vehicle side slip angle, here $\beta_{\text{target}} = -35°$, while also keeping the lateral distance to the path $s_n$ minimal. Figure 6.3 presents the corresponding mean values. While both trainings initially show similar tracking performance due to similar agent behaviour, training T1 better fulfils the criteria than training T2 as the training progresses. Without looking at the temporal behaviour during one episode, the agent's behaviour in training T2 can be interpreted as follows: The vehicle side slip angle is increased in absolute terms, as shown in the left graph of Figure 6.3. However, compared to the successful training T1, Figure 6.2 reveals that the applied drive torques are relatively high, which results in an increased vehicle speed. At a higher speed, it is impossible to follow the path, and the vehicle leaves the track to the outside. The right graph of Figure 6.3 illustrates the vehicle's average location. Every episode starts with the vehicle near to the middle of the track $|s_n| < 0.25\,\text{m}$. Negative values of $s_n$ correspond to mean vehicle position outside the middle of the track.

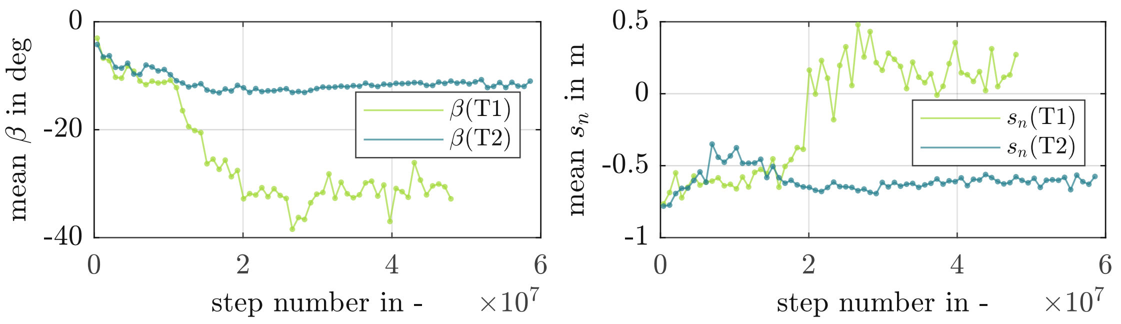


Figure 6.3: Average vehicle side slip angle $\beta$ (left) and location $s_n$ (right) during rollout period.

## Other Training Metrics

Apart from measuring the training success and monitoring the agent's and environment's behaviour, it is important to consider additional metrics. The left graph in Figure 6.4 gives insights in how fast the training is running. It represents the average number of observations the agent receives per second, averaged over an entire update step, which consists of generating experience by interacting with the environment, updating MLP parameters, and evaluating the current policy. The difference between training T1 and training T2 arises due to the evaluation phase. As shown by the right graph of Figure 6.1, an episode of training T1 lasts longer than one of training T2, leading to in increase of the overall policy update tiem of training T1.

In SL, a decreasing loss indicates a successful training, as the loss typically corresponds to the MSE of the actual DNN output compared to the target output. The loss function is defined on a fixed data distribution, which is independent of the parameters to be optimised. This is different toRL, where data is generated by the most recent policy. Therefore, there is no intuition behind the loss function when dealing with RL, as the loss does not measure the performance. The right graph in Figure 6.4 shows the result of PPO's objective function (3.28).

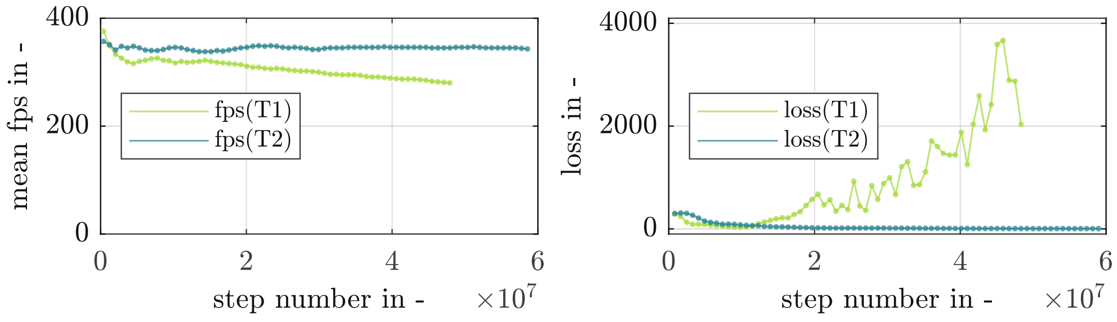


Figure 6.4: Average number of observations per second that the agent receives during rollout period (left) and result of the objective function (right).

# 7 Results

In this chapter, the evaluation of a successfully trained agent is discussed. For this purpose, the trained policy network is tested in different scenarios, including variations of the target vehicle side slip angle and initial conditions in Section 7.1, as well as driver and friction potential variations in Section 7.2. According to the evaluation callback described in Section 6.1.1, the best policy during training is taken to perform these tasks. Appendix A.4 provides an illustration of the policy in the closed control loop. The agent was trained on a circular path with radius $R = 60\,\text{m}$ and friction potential $\mu = 0.21$. Any plot in this chapter demonstrates the performance of the same policy network, which is an MLP consisting of 3 hidden layers, each with 16 neurons per layer and ELU activation function. An illustration of the MLP is given in Appendix A.5, and the utilised set of training hyperparameters is provided in Appendix A.3. In the remainder of this chapter, side slip angle refers to the vehicle side slip angle.

## 7.1 Target Side Slip Angle and Initial Condition Variation

To evaluate the policy performance, the control behaviour regarding two different target side slip angles, each with two different initial speeds, is analysed. Table 7.1 summarises these 4 scenarios. The target side slip angles stated there fall outside the range of target side slip angles during training, where they were constrained to $[-42\,°, -28\,°]$. The results are plotted in Figure 7.1. Regardless of the initial conditions, the vehicle settles at the same steady state for a target side slip angle $\beta_{\text{target}} = -45\,°$, represented by scenarios S1 and S2. Except for scenario S4, the policy manages to stabilise the powerslide at the desired side slip angle $\beta_{\text{target}}$.

| | $\beta_{\text{target}} = -45\,°$ | $\beta_{\text{target}} = -25\,°$ |
|---|---|---|
| $v_0 = 20$ km/h | Scenario S1 | Scenario S3 |
| $v_0 = 34$ km/h | Scenario S2 | Scenario S4 |

Table 7.1: Summary of 4 different evaluation scenarios. The scenarios differ in terms of the target side slip angle $\beta_{\text{target}}$ and the initial speed $v_0$.

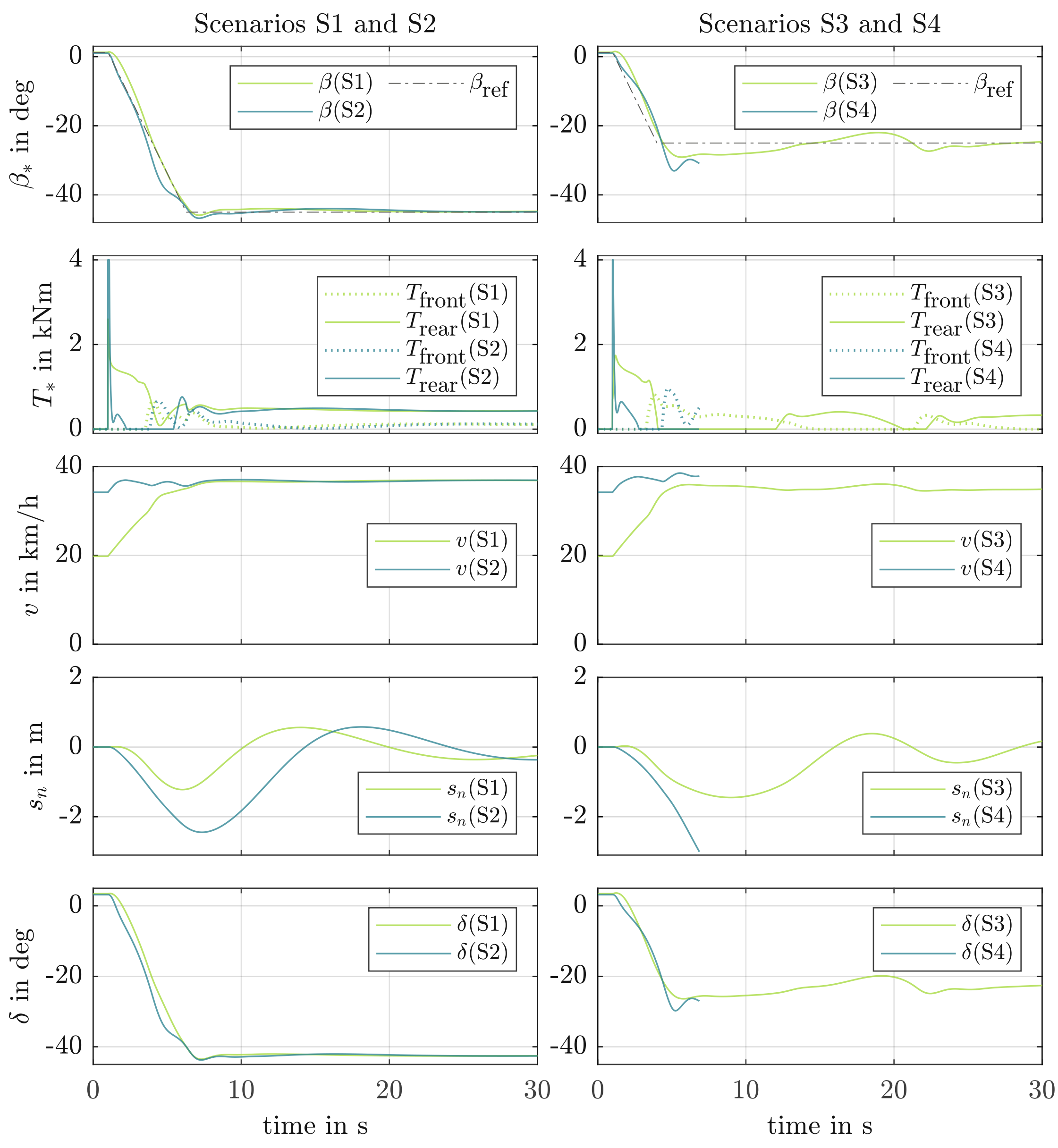


Figure 7.1: Evaluation scenarios comprising different target side slip angles $\beta_{\mathrm{target}}$ and initial conditions $v_0$. The plots include side slip angle $\beta$ together with the reference side slip angle $\beta_{\mathrm{ref}}$, drive torques $T_*$ as actions applied by the agent, speed $v$, lateral distance to the path $s_n$ and steering angle $\delta$. Scenarios S1 and S2 with $\beta_{\mathrm{target}} = 45\,°$ are plotted on the left half, while scenarios S3 and S4 with $\beta_{\mathrm{target}} = 25\,°$ are plotted on the right half.

Figure 7.2 illustrates the handling diagram, visualising the side slip angle $\beta$ and normal acceleration $a_n$. Comparing the two steady states, illustrated by the crosses with scenarios S1 and S2 converging at $\beta = -45\,°$ and scenario S3 at $\beta = -25\,°$, respectively, reveals, that the latter has a lower normal acceleration $a_n$, and therefore a lower vehicle speed, which is also evident in the plots in the third row of Figure 7.1. The starting point of scenario S4 and its target end point, which is also the end point of scenario S3, are characterised by almost identical normal accelerations. Since the agent can only apply positive drive torques to the axles to increase the side slip angle in absolute terms, the two objectives expressed by the reward function (5.3) are conflicting. It is not possible to track the reference side slip angle while also following the path appropriately. Further, the agent has no access to the current lateral distance to the path $s_n$, only indirectly by the driver's reaction on the steering wheel. As a result, the vehicle gets too fast and consequently leaves the track to the outside.

The plotted steady states, illustrated by the black lines in Figure 7.2, correspond to a fixed drive torque distribution $\gamma = (T_{\text{rear}})/(T_{\text{front}} + T_{\text{rear}}) = 0.81$. The drive torque distributions at the steady states of scenarios S1 and S2 at $\beta = -45\,°$ and scenario S3 at $\beta = -25\,°$, respectively, are slightly different. For this reason, the end points in the plot deviate a little from the dash-dotted line.

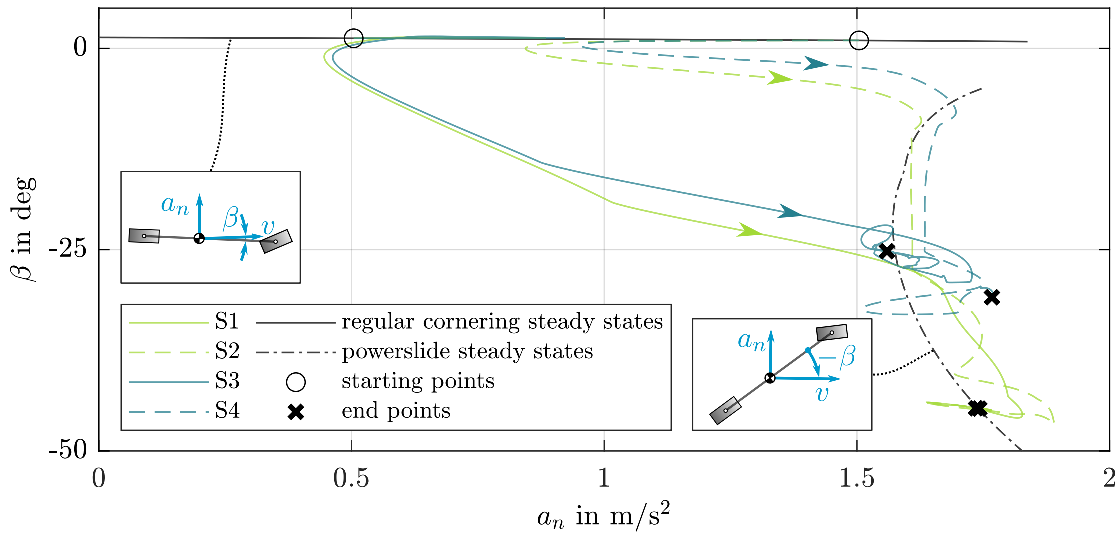


Figure 7.2: Handling diagram illustrating side slip angle $\beta$ and normal acceleration $a_n$. The plotted steady states of powerslide and regular cornvering correspond to a stationary cornering with drive torque distribution $\gamma = 0.81$, radius $R = 60\,\text{m}$ and friction potential $\mu = 0.21$. Scenarios S1 and S3, as well as S2 and S4 share identical initial conditions.

## 7.2 Driver and Friction Potential Variation

While Section 7.1 examines the performance of the trained policy network, this section discusses its limitations. During training, data was collected under consistent track conditions, with radius $R = 60\,\mathrm{m}$, friction potential $\mu = 0.21$ and a single driver model. The performance of the policy is now evaluated using different driver model parameters and different friction potentials $\mu$, and compared to the standard setting the agent encountered during training. Table 7.2 and Table 7.3 summarise these different situations. Compared to the standard driver D0 utilised during training, Driver D1 is characterised by an increase steering bandwidth, while driver D2's steering bandwidth is reduced. Figure 7.3 illustrates the control behaviour for driver variations and friction potential variations. In any non-standard case (D1, D2, F1, F2) the policy fails to stabilise the vehicle at a steady powerslide state.

| | |
|---|---|
| standard driver | D0 |
| increased steering bandwidth | D1 |
| reduced steering bandwidth | D2 |

Table 7.2: Summary of driver variations.

| | |
|---|---|
| $\mu = 0.21$ | F0 |
| $\mu = 0.19$ | F1 |
| $\mu = 0.25$ | F2 |

Table 7.3: Summary of friction potential $\mu$ variations.

By steering aggressively, driver D1 tries to minimise the lateral distance to the path $s_n$. This leads to oscillations of the side slip angle $\beta$ that the policy can not account for. The evaluation stops as the side slip angle $\beta$ gets out of bounds. On the other hand, driver D2 reacts too slow to follow the path. The policy manages to track the side slip angle reference, but the lateral distance to the path $s_n$ becomes too large.

The evaluation results of friction potential variations are plotted on the right half of Figure 7.3. A higher friction potential $\mu$ (F2) leads to a slower changing side slip angle $\beta$. Also the lateral distance to the path $s_n$ remains smaller in the first 10 seconds. However, as the deviation of the the side slip angle $\beta$ compared to the reference gets too high, the policy reacts by applying drive torques to the rear axle. This results also in a higher speed and the vehicle consequently leaves the track to the outside. A lower friction potential (F1) on the other hand, leads to a slightly faster changing side slip angle $\beta$ compared to the standard setting (F0). Due to lower lateral forces, the vehicle also leaves the track to the outside.

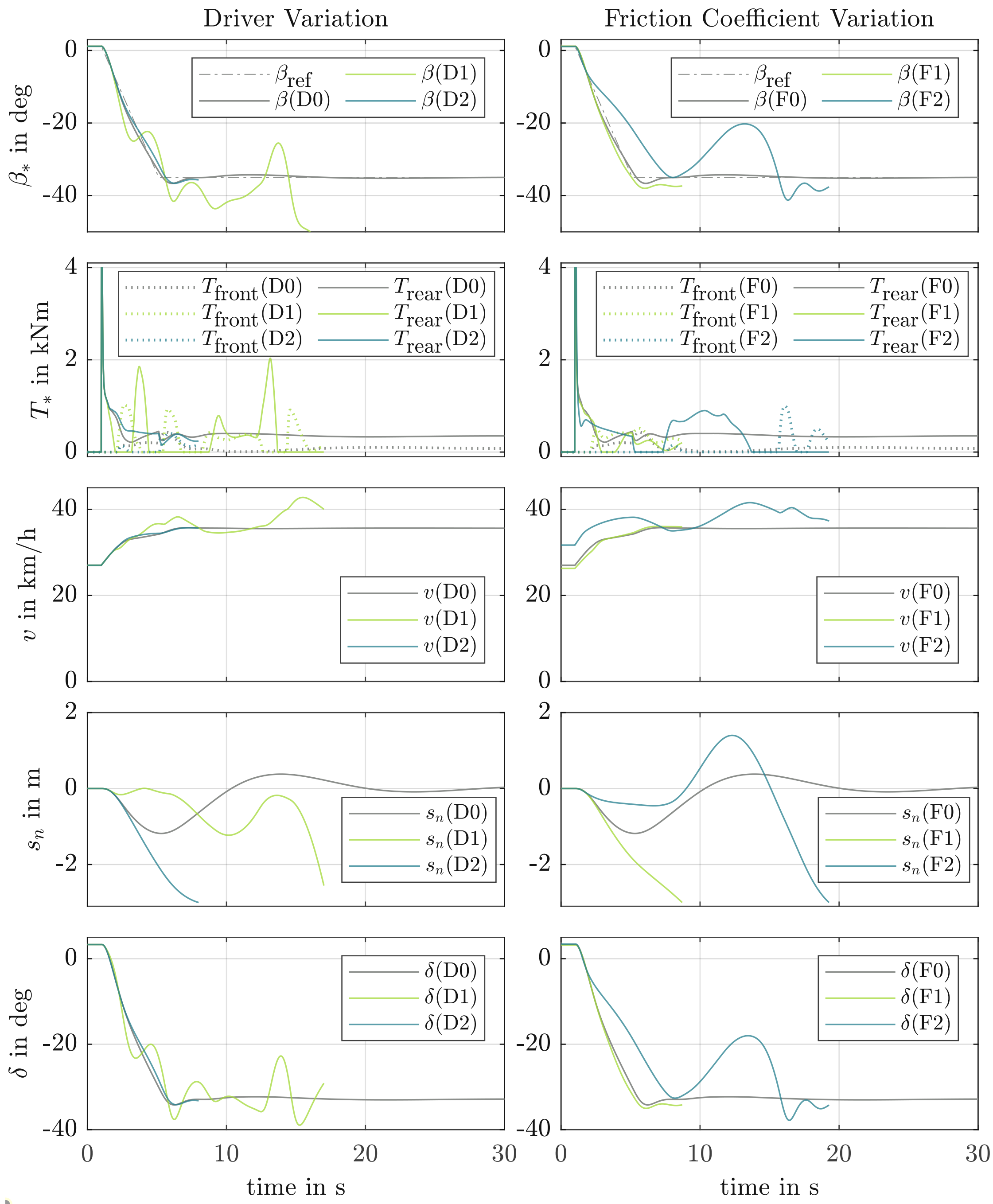


Figure 7.3: Analysis of robustness through variations of the driver behaviour and friction potential $\mu$. The plots include side slip angle $\beta$ together with the reference side slip angle $\beta_{\text{ref}}$, drive torques $T_*$ as actions applied by the agent, speed $v$, lateral distance to the path $s_n$ and steering angle $\delta$. The left half shows different driver behaviours (D1 increased, D2 slower dynamics than D0). The right half compares different friction potentials F1 ($\mu = 0.19$), F2 ($\mu = 0.25$) and F0 (original $\mu = 0.21$).

# 8 Summary and Conclusion

In this work, a powerslide controller for an AWD BEV, with individually driven front and rear axles and a human driver in closed-loop using RL, is designed. While the driver follows a given circular path solely by steering, the ADAS stabilises the powerslide by controlling the absolute drive torque and the drive torque distribution between front and rear axle. The controller, represented by a DNN, is trained in a simulated environment, where a two-wheel vehicle model and a driver model, approximating human driver behaviour, are considered. The control strategy is based on the reward function, similar to a cost function in optimal control. To save computational time, the training process took place on the VSC.

The results of this work indicate that the ADAS is able to stabilise the vehicle's powerslide motion in the presence of a human driver. Unlike traditional control concepts that rely on (the linearisation around) an operating point, the data-driven approach does not require any prior knowledge of the system dynamics in principal. It is noted, that the steady-states of regular cornering are used for the selection of the vehicle's initial conditions, as randomly selecting vehicle states independently of each other would provide artificial non-physical driving situations. Another key difference to traditional controllers concerns the choice of controller inputs. Any measurement that may have an impact on the control strategy can be passed as an observation to the DNN. The RL algorithm will detect the relevant information on its own.

Varying target vehicle side slip angles and changing initial conditions during training increase the controller's generalisation capabilities. The trained controller is able to stabilise the vehicle's powerslide at side slip angles that were not targeted during training. Despite these promising results, there is still room for improvement. While generalisation can be observed regarding the target vehicle side slip angles, the controller's adaption to changing road conditions and driver types is limited.

This work also covers the theoretical background and mathematical fundamentals of a popular DRL algorithm. However, there is no guarantee that the practical application of such an algorithm to a specific problem will succeed. Many aspects can affect the training success, most notably the reward function as part of the central optimisation problem.

While the set of hyperparameters clearly has an impact on training, also less apparent settings, like the observation and action scaling or the maximum number of time steps of an episode, can result in an unsatisfactory control behaviour. Identifying the effects of varying training setup combinations is cumbersome, as even identical settings lead to different controllers due to the stochastic nature of the training process. Similar to the agent's trial-end-error strategy of finding the most effective actions, the ML engineer's task is to determine the best possible training settings.

## 8.1 Outlook

A diversification of the training in terms of driver and road conditions may lead to a better generalising controller and increase robustness. This is particularly important as the next step is to evaluate the ADAS on a real vehicle. Transferring the performance of a controller trained in a simulated environment to a real-world environment may face challenges. The DNN was trained on synthetically generated data and has never encountered real noisy measurements.

Another critical aspect of a real car is potential dead time in the control loop. Just adding dead time to the training process is not assumed to work, as it breaks the assumption underlying the MDP. This issue could probably be tackled by more sophisticated recurrent neural network (RNN) architectures, e.g., Long Short Term Memory (LSTM) networks. With their ability to memorise patterns from long sequences of data, LSTM networks can potentially learn to compensate for dead time in the control loop.

Further work could also focus on stabilising the powerslide not only on circular paths, but also on arbitrary trajectories. This would include a controlled transition from the powerslide back to regular driving.

Incorporating real-world measurements into the training process could further the enhance controller performance. While a real-world RL system poses potential safety issues due to deploying untrained or partially trained and therefore potentially unsafe DNNs to a vehicle, offline RL could be the way to go.

# A Appendix

## A.1 Tire Slip Kinematics

The tire longitudinal slip

$$s_{i,x} = -\frac{v_{i,x} - \omega_i r_i}{v_{i,x}} \tag{A.1}$$

is defined as the negative wheel slip velocity of the contact point $v_{i,x} - \omega_i r_i$ divided by the longitudinal speed component of the wheel centre $v_{i,x}$. The tire side slip angle, denoted as $\alpha_i$, is defined as angle from the tire plane to the velocity vector of the wheel centre. Subscript $i = \{\mathrm{front}, \mathrm{rear}\}$ indicates front tire and rear tire, respectively. The derivation of $s_{i,x}$ and $\alpha_i$ is illustrated in Figure A.1.

Trigonometric identities of the right triangles $RDP$ and $DFP$ lead to

$$\tan(\alpha_{\mathrm{rear}}) = \frac{l_{\mathrm{rear}} - v/\dot{\psi} \sin\beta}{v/\dot{\psi}\cos\beta} = \frac{l_{\mathrm{rear}}\dot{\psi}}{v\cos\beta} - \tan\beta\,, \tag{A.2}$$

$$\tan\lambda = \frac{l_{\mathrm{front}} + v/\dot{\psi}\sin\beta}{v/\dot{\psi}\cos\beta} = \frac{l_{\mathrm{front}}\dot{\psi}}{v\cos\beta} + \tan\beta\,, \tag{A.3}$$

where $\lambda$ is an auxiliary angle. Since the vehicle body is rigid, the components of the velocities $v$ and $v_i$, pointing in the direction of the vehicle's longitudinal axis, must be equal

$$v\cos\beta = v_{\mathrm{front}}\cos\lambda = v_{\mathrm{rear}}\cos\alpha_{\mathrm{rear}}\,, \tag{A.4}$$

with $v_i$ representing the speed of the front axle and rear axle, respectively. These axle speeds are obtained by combining (A.3), (A.4) and $v_{i,x} = v_i \cos\alpha_i$

$$v_{\mathrm{front},x} = v(\cos\beta\cos\delta + (\sin\beta + l_{\mathrm{front}}\dot{\psi}/v)\sin\delta)\,, \tag{A.5a}$$

$$v_{\mathrm{rear},x} = v\cos\beta\,. \tag{A.5b}$$

The results from (A.5) substituted into (A.1) yield the required tire longitudinal slip.

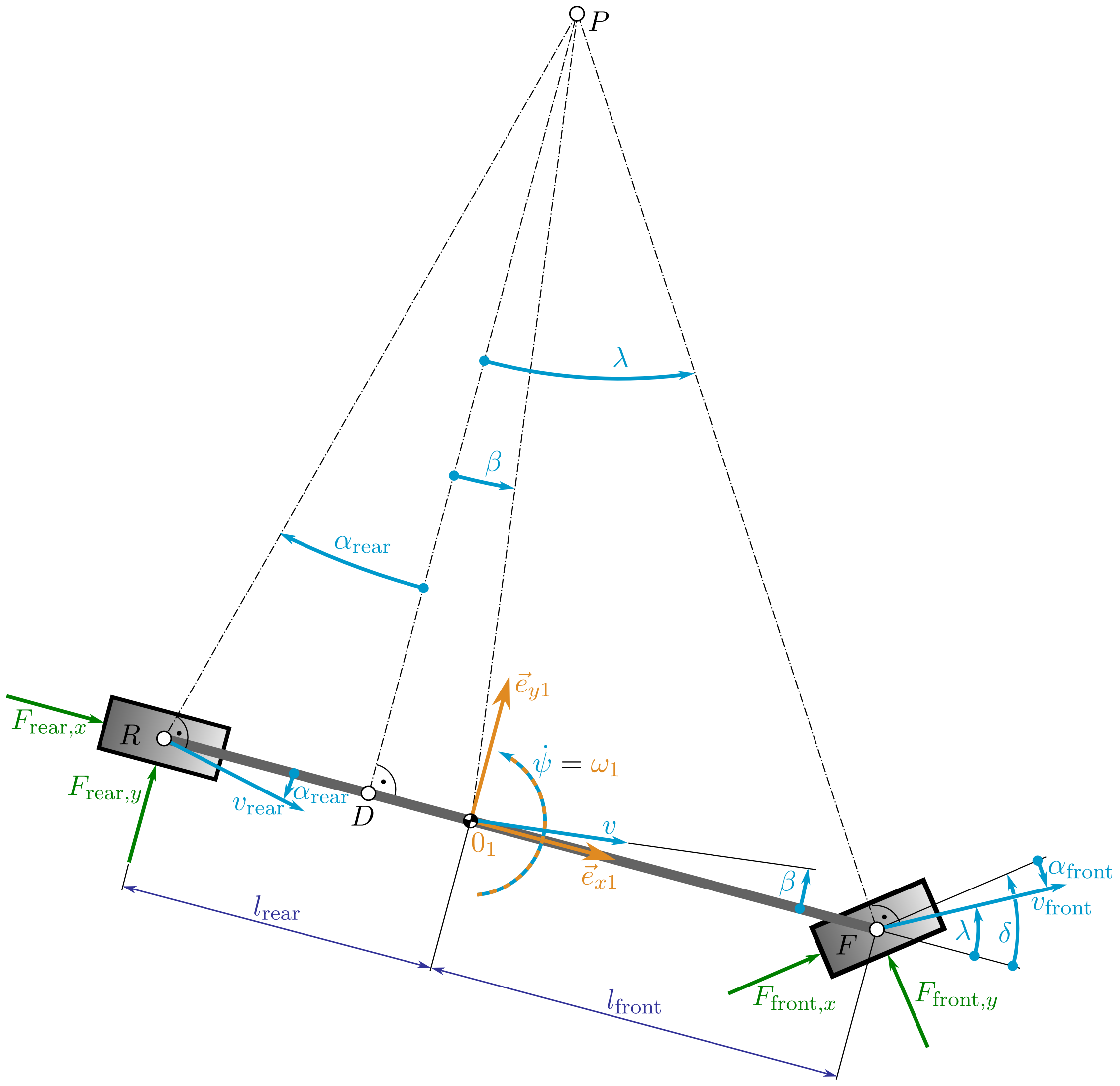


Figure A.1: Tire slip kinematics during regular cornering.

With $\alpha_{\text{front}} = \delta - \lambda$, (A.2) and (A.3), the tire side slip angles follow

$$\alpha_{\text{front}} = \delta - \arctan\left(\frac{l_{\text{front}}\dot{\psi}}{v\cos\beta} + \tan\beta\right) , \tag{A.6a}$$

$$\alpha_{\text{rear}} = -\arctan\left(-\frac{l_{\text{rear}}\dot{\psi}}{v\cos\beta} + \tan\beta\right) . \tag{A.6b}$$

The results of the tire kinematics (A.1), (A.5) and (A.6) serve as input to the tire force model (4.9).

## A.2 Observation Space of PowerslideEnv

| Description | Symbol | Lower Bound | Upper Bound | Unit |
|---|---|---|---|---|
| Vehicle side slip angle | $\beta$ | $-\pi/2$ | $\pi/2$ | rad |
| Yaw rate | $\dot{\psi}$ | $-\pi/3$ | $\pi/3$ | rad/s |
| Speed of COM | $v$ | 5 | 15 | m/s |
| Angular speed of front wheel | $\omega_{\text{front}}$ | 10 | 60 | rad/s |
| Angular speed of rear wheel | $\omega_{\text{rear}}$ | 10 | 60 | rad/s |
| Steering angle | $\delta$ | $-0.9$ | 0.9 | rad |
| Steady-state vehicle side slip angle | $\beta_{\text{target}}$ | $-\pi/2$ | $\pi/2$ | rad |
| Reference side slip angle | $\beta_{\text{inter}}$ | $-\pi/2$ | $\pi/2$ | rad |

Table A.1: Numeric values for the bounds of the observation space.

## A.3 PPO Hyperparameter Set

| Hyperparameter | Value |
|---|---|
| learning_rate | $2 \times 10^{-4}$ |
| n_steps | 10240 |
| batch_size | 5120 |
| n_epochs | 5 |
| gamma | 0.9999 |
| gamma_lambda | 0.98 |
| clip_range | 0.2 |
| clip_range_vf | None |
| normalize_advantage | True |
| ent_coef | $2 \times 10^{-6}$ |
| vf_coef | 0.75 |
| max_grad_norm | 0.6 |
| use_sde | True |
| sde_sample_freq | 128 |
| target_kl | None |

Table A.2: Example hyperparameter setting of SB3's PPO. Their description can be found on https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html#parameters.

## A.4 Deep Neural Network in Closed Control Loop

After training an agent in an RL setting, the DNN representing the policy can be used as a standalone controller. As the DNN parameters do not change anymore, the reward, as central part of the training process, is now disregarded, and also the value function is not required anymore. This phase in development, where capabilities learned during training are put to work, is called *inference*. Figure A.2 visualises the closed control loop. Compared to standard control loops, the control error is not explicitly calculated, but intrinsically captured by the DNN.

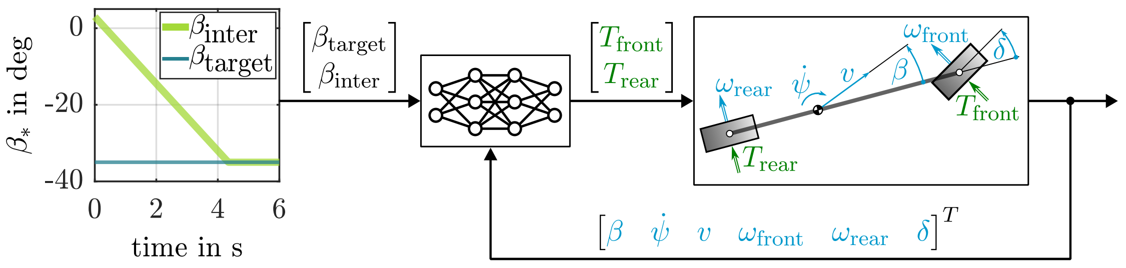


Figure A.2: DNN in closed control loop after training. The vehicle state and the reference vehicle side slip angle serve as input to the controller.

## A.5 Multilayer Perceptron and Activation Function

This section illustrates the network architecture of the DNN evaluated in Chapter 7, an MLP consisting of 3 hidden layers each with 16 neurons per layer and ELU activation function. The size of the input layer and output layer are determined by the size of the observations space and action space, respectively. Figure A.3 visualises the working principle of the MLP.

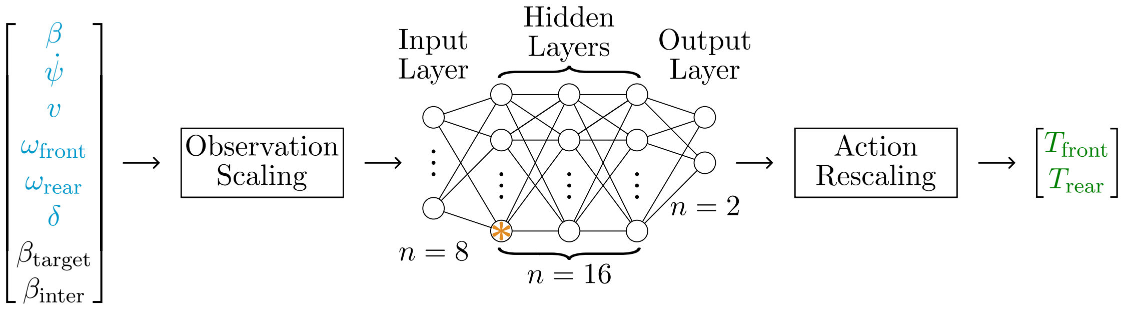


Figure A.3: MLP consisting of 3 hidden layers each with 16 neurons per layer. For numerical reasons, the architecture comprises scaling of observations and rescaling of actions.

Observations are scaled before entering the MLP and the network outputs are rescaled accordingly. Highlighted by the orange asterisk in Figure A.3, this single neuron is exemplarily detailed in Figure A.4. The calculus in a neuron is represented by linear operations followed by a nonlinear activation function. Summarising the neurons of one layer simplifies calculus. The forward pass through one layer can be reduced to a matrix-vector multiplication and a vector-vector summation. The ELU activation function

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0 \end{cases} \tag{A.7}$$

adds nonlinearity to the computation, required to approximate arbitrary functions. The hyperparameter $\alpha$ affects the saturation behaviour and is typically set to a small positive value. For this work the default value of $\alpha = 1$ was used.

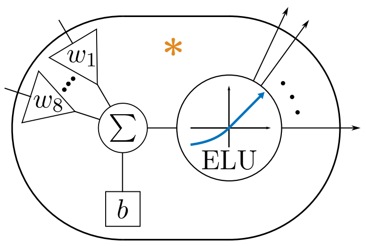


Figure A.4: Operating principle of a single neuron inside an MLP. The outputs of the previous layer's neurons are multiplied with individual weightings $w_i$ and summed up including a bias $b$. The result is passed through the ELU activation function before being forwarded to the neurons of the next layer.

# References

[1] J. Edelmann and M. Plöchl, "Handling characteristics and stability of the steady-state powerslide motion of an automobile", *Regular and Chaotic Dynamics*, vol. 14, pp. 682–692, 2009.

[2] J. Edelmann and M. Plöchl, "Controllability of the powerslide motion of vehicles with different drive concepts", *Procedia engineering*, vol. 199, pp. 3266–3271, 2017.

[3] M. Eberhart, "Theoretische Untersuchung und Entwicklung eines Fahrerassistenzsystems zur Stabilisierung des Powerslides", diploma thesis, TU Wien, 2022.

[4] M. Cutler and J. P. How, "Autonomous drifting using simulation-aided reinforcement learning", in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2016, pp. 5442–5448.

[5] F. Domberg, C. C. Wembers, H. Patel, and G. Schildbach, "Deep drifting: Autonomous drifting of arbitrary trajectories using deep reinforcement learning", in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022, pp. 7753–7759.

[6] M. Acosta and S. Kanarachos, "Teaching a vehicle to autonomously drift: A data-based approach using neural networks", *Knowledge-Based Systems*, vol. 153, pp. 12–28, 2018.

[7] F. Djeumou, J. Y. Goh, U. Topcu, and A. Balachandran, "Autonomous drifting with 3 minutes of data via learned tire models", in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2023, pp. 968–974.

[8] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations", *Advances in neural information processing systems*, vol. 31, 2018.

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

[10] J. Achiam, "Spinning Up in Deep Reinforcement Learning", 2018.

[11] B. Douglas, *What is reinforcement learning?*, YouTube video, 2019.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *CoRR*, vol. abs/1707.06347, 2017.

[13] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization", in *International conference on machine learning*, PMLR, 2015, pp. 1889–1897.

[14] L. Engstrom, A. Ilyas, S. Santurkar, *et al.*, "Implementation matters in deep policy gradients: A case study on ppo and trpo", *arXiv preprint arXiv:2005.12729*, 2020.

[15] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations", *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[16] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019.

[17] M. Abadi, A. Agarwal, P. Barham, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems", *arXiv preprint arXiv:1603.04467*, 2016.

[18] H. Pacejka, *Tire and vehicle dynamics*. Elsevier, 2005.

[19] D. T. McRuer, "Human pilot dynamics in compensatory systems", SYSTEMS TECHNOLOGY INC HAWTHORNE CA, Tech. Rep., 1965.

[20] J. Edelmann, "PKW-Fahrermodell für höhere Querbeschleunigungen", diploma thesis, TU Wien, 2004.

[21] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym", *arXiv preprint arXiv:1606.01540*, 2016.