BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

APPLICATION OF REINFORCEMENT LEARNING IN AUTONOMOUS DRIVING

APLIKACE POSILOVANÉHO UČENÍ V ŘÍZENÍ AUTONOMNÍHO VOZIDLA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. DAVID VOSOL**
AUTOR PRÁCE

SUPERVISOR — **Doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Vosol David, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Strojové učení
Název: **Aplikace posilovaného učení v řízení autonomního vozidla**
Application of Reinforcement Learning in Autonomous Driving
Kategorie: Umělá inteligence
Zadání:

1. Prostudujte problematiku posilovaného učení. Seznamte se s dostupným softwarem pro aplikace posilovaného učení a se simulátory prostředí. Porovnejte state-of-the-art algoritmy a dostupné nástroje z hlediska aplikovatelnosti pro realizaci autonomní řízení vozidla.
2. Navrhněte systém pro autonomní řízení vozidla s využitím posilovaného učení. Soustřeďte se na schopnost co nejrychleji projet trať bez kolizí ovládáním brzdy, plynu, řazení a volantu, na základě informací z kamery a jiných senzorů. Uvažujte různé architektury řídicího systému, např. lokální učení a/nebo řízení vs. využití cloudových služeb pro učení a/nebo řízení.
3. Navržené řešení a jeho varianty realizujte s využitím vhodně vybraných prostředků z bodu 1. Pro experimenty a strojové učení využijte vhodně vybraný simulátor prostředí, např. TORCS.
4. Vhodným způsobem demonstrujte funkci vytvořeného systému. Vyhodnoťte dosažené výsledky a vytvořte plakát shrnující tuto práci.

Literatura:

- Sutton, R. S.; Barto, A. G.: Reinforcement Learning: An Introduction, Second edition, The MIT Press, 2018. ISBN 978-0262039246
- Russell, S.: Artificial Intelligence: A Modern Approach, 4th Edition, Pearson, 2020. ISBN 978-0134610993
- Loiacono, D.; Cardamone, L.; Lanzi, P. L.: Simulated Car Racing Championship: Competition Software Manual, arXiv:1304.1672, 2013.
- Weng, L.: Policy Gradient Algorithms, Lil'Log, 2018. URL: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

Při obhajobě semestrální části projektu je požadováno:

- První 2 body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstract

This thesis is focused on the topic of reinforcement learning applied to a task of autonomous vehicle driving. First, the necessary fundamental theory is presented, including the *state-of-the-art* actor-critic methods. From them the *Proximal policy optimization* algorithm is chosen for the application to the mentioned task. For the same purpose, the racing simulator TORCS is used. Our goal is to learn a reinforcement learning agent in a simulated environment with the focus on a future real-world application to an RC scaled model car. To achieve this, we simulate the conditions of remote learning and control in the cloud. For that, simulation of network packet loss, noisy sensory and actuator data is done. We also experiment with the least number of vehicle's sensors required for the agent to successfully learn the task. Experiments regarding the vehicle's camera output are also carried out. Different system architectures are proposed, among others also with the aim to minimize hardware requirements. Finally, we explore the generalization properties of a learned agent in an unknown environment.

Abstrakt

Tato práce se zabývá problematikou posilovaného učení aplikovaného na úlohu autonomního řízení vozidla. Nejprve je probrána nezbytná teorie posilovaného učení, která je zakončena představením nejmodernějších aktor-kritik metod. Z nich je vybrána metoda *Proximal Policy Optimization*, která je následně aplikována na tuto úlohu. Pro tento účel je také zvolen závodní simulátor TORCS. Naším cílem je naučit v simulovaném prostředí agenta autonomně řídit, s ohledem na jeho budoucí aplikaci v reálném prostředí v podobě zmenšeného RC modelu vozidla. Za tímto účelem jsou simulovány podmínky vzdáleného učení a ovládání vozidla v cloudu a to v podobě simulace ztráty paketů s daty od senzorů a aktuátorů nebo simulace zašuměných dat. Také jsou provedeny experimenty s cílem zjistit nejmenší počet senzorů, se kterým je agent schopen se úlohu naučit. Dále je experimentováno s využitím výstupu kamery vozidla. Jsou představeny různé návrhy architektur systému, mimo jiné i se zaměřením na co nejnižší hardwarové požadavky. Na závěr jsou prozkoumány vlastnosti naučeného agenta z pohledu generalizace v neznámém prostředí.

Keywords

reinforcement learning, policy gradients, actor-critic, autonomous driving, TORCS, neural networks, proximal policy optimization, PPO

Klíčová slova

posilované učení, gradientní strategie, aktor-kritik, autonomní řízení vozidla, TORCS, neuronové sítě, optimalizace blízké strategie, PPO

Reference

VOSOL, David. *Application of reinforcement learning in autonomous driving.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Vladimír Janoušek, Ph.D.

Rozšířený abstrakt

Tato práce se zabývá problematikou posilovaného učení aplikovaného na úlohu autonomního řízení vozidla. V první části práce je nejprve uvedena nezbytná terminologie posilovaného učení a následně probrány základní principy a algoritmy. Od *Markovova rozhodovacího procesu*, který je základem všech algoritmů posilovaného učení, přes tzv. *Value-based* metody, kdy zjišťujeme kvalitu jednotlivých stavů ve kterých se agent nachází, případně hodnotíme i akce, které agent v daném stavu provedl. Mezi tyto přístupy patří *Monte-Carlo, Temporální diference a Dynamické programování.* Od těchto metod se poté přesouváme k *Policy-based* metodám, tedy metodám, které optimalizují přímo agentovu strategii. V tomto přístupu se zaměřujeme na vysvětlení algoritmů *Stochastic Policy Gradients*, *Monte Carlo Policy Gradients* (REINFORCE) a varianty využívající tzv. baseline a advantage funkce. Následně také popisujeme využití neuronových sítí v policy-based algoritmech.

Na závěr specifikujeme metody zvané Aktor-kritik, tedy spojení *Value-based* a *Policy-based* metod. Mezi tyto metody patří i algoritmus *Proximal policy optimization* (PPO), který v praktické části také využíváme na úloze autonomního řízení vozidla.

Následně je čtenáři přiblížena problematika autonomního řízení a dostupný simulátor TORCS (*The Open Racing Car Simulator*). Ten je využíván i pro experimentální část. Jsou představeny dostupné senzory a aktuátory, výběr tratí a navrženy odměnové funkce. Následně je specifikován systém pro samotné autonomní řízení a jsou představeny jeho různé architektury. Jako první je představena tzv. *Regular architecture*, ve které agent využívá pouze základních senzorů vozidla, které pak slouží jako vstup pro neuronové sítě Aktor-kritik. Výstupem jsou pak parametry Normálního rozdělení pravděpodobnosti, tedy střední hodnota a rozptyl. Z tohoto rozložení se poté vzorkuje agentova akce. Těmito akcemi jsou zatáčení, plyn a brzda. V této architektuře jsou následně zkoušeny různé topologie neuronové sítě, hyperparametry algoritmu PPO (batch-size, learning rate, aj.), parametry samotného simulátoru (délka epizody, hodnoty odměn a pokut, odměnové funkce, počty a typy senzorů, aj.). To vše s cílem co nejlépe naučit agenta projet danou trať. Jako vyhodnocovací metriky sbíráme údaje o průměrné rychlosti, ujeté vzdálenosti, celkové hodnotě odměn, nebo také počet nárazů vozidla a hodnotu agentovy entropie. Následně je agent také testován na nových, pro agenta dosud neznámých tratích, s cílem ověřit jeho schopnost generalizovat.

Dále je představena tzv *ConvNet architecture*, ve které agent využívá také výstupu z kamery. Experimenty jsou rozděleny na dvě části, v první agent využívá pouze výstupu z kamery a druhé, kdy agent využívá jak výstupu z kamery, tak také základních senzorů. Zde jsou zkoušeny různé architektury konvoluční neuronové sítě, pro zpracování výstupu z kamery. Jako výstup takové sítě je poté tzv. “Features vector”, který je následně konkatenován k vektoru dat ze senzorů. Tento nově vzniklý vektor je potom vstupem do sítě aktor-kritik.

Jako poslední je představena tzv. *Hybrid architecture*, ve které reagujeme na empiricky získané poznatky, kdy agent není schopen se naučit úlohu autonomního řízení pouze na základě výstupu z kamery. To ani v případě využití úprav kamerového snímku pro zachycení dynamiky vozidla. Mezi těmito úpravami je například odečtení aktuálního a předchozího snímku, nebo jejich spojení. V tomto novém přístupu tedy nejprve naučíme agenta na klasických senzorech. Takto naučeným agentem poté vygenerujeme dataset ve formě: výstup z kamery – hodnoty senzorů. Tímto datasetem následně naučíme konvoluční neuronovou síť v klasickém přístupu učení s učitelem předpovídat hodnoty senzorů na základě výstupu z kamery. Tímto způsobem se nám úspěšně podařilo naučit agenta autonomně řídit.

Následně byly provedeny experimenty s cílem simulovat podmínky reálného světa. V našem případě simulace ztráty paketů při architektuře systému, kdy výpočet a učení agenta probíhá v cloudu a s agentem je komunikováno pouze bezdrátově prostřednictvím síťového protokolu UDP. Tato architektura je pojmenována jako *Cloud architecture*. Kromě ztráty paketů je také simulováno zašumění dat a to jak sensorických ve směru (agent – cloud), tak dat pro aktuátory (cloud – agent).

Všechny tyto snahy jsou prováděny s cílem co nejlépe připravit agenta posilovaného učení v simulovaném prostředí, za použití minimálních hardwarových požadavků tak, aby byla možná budoucí aplikace takového agenta v reálném prostředí v podobě zmenšeného RC závodního modelu auta. Na základě experimnetů jsme zjistili, že je agent schopen autonomního řízení vozidla pouze pomocí kamery a také pouze za použití jediného senzoru, který představuje jednoduchý LIDAR senzor. Také se chová stabilně při simulaci vnějších vlivů a dokáže poměrně úspěšně generalizovat v nových prostředích. Takový agent by měl poté být s co nejmenším dodatečným úsilím schopen autonomně řídit reálný RC model na reálné trati. Vzniklý systém by se poté v budoucnu mohl zúčastnit například mezinárodní univerzitní soutěže pro autonomní řízení RC modelů, soutěže *NXP Cup*.

Application of reinforcement learning in autonomous driving

Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of Mr. Doc. Ing. Vladimír Janoušek Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
David Vosol
May 15, 2022

Acknowledgements

I would like to thank my supervisor Mr. Doc. Ing. Vladimír Janoušek Ph.D., my family and also my girlfriend who has supported me throughout the making process of this work.

Contents

1 Introduction **3**

2 Reinforcement learning **5**
2.1 Introduction . . . 5
2.2 Theory . . . 8
2.2.1 Markov Process . . . 8
2.2.2 Markov Decision Process . . . 12
2.3 Value-based methods . . . 14
2.3.1 Dynamic Programming . . . 15
2.3.2 Monte Carlo methods . . . 16
2.3.3 Temporal Difference methods . . . 16
2.3.4 Value-based methods and Function approximators . . . 18
2.4 Policy-based methods . . . 20
2.4.1 Stochastic Policy Gradient methods . . . 20
2.4.2 Monte-Carlo Policy Gradient (REINFORCE) . . . 21
2.4.3 Actor-Critic Policy Gradient methods . . . 22
2.4.4 Trust-Region methods . . . 24

3 Autonomous driving and system design **33**
3.1 Autonomous driving . . . 33
3.2 Simulation environment . . . 34
3.2.1 TORCS . . . 35
3.2.2 Sensors and Actuators . . . 39
3.2.3 Reward shaping . . . 43
3.2.4 Performance metrics . . . 45
3.2.5 The Algorithm . . . 46
3.3 Implementation details . . . 47
3.3.1 Algorithm implementation . . . 47
3.3.2 Neural network architectures . . . 49
3.3.3 Cloud architecture . . . 55

4 Experiments and results **57**
4.1 Computational Hardware . . . 57
4.2 Subject of experiments . . . 57
4.3 Experiments . . . 58
4.3.1 Hyperparameters . . . 59
4.3.2 Differences in performance . . . 62
4.3.3 Initialization and Instability . . . 65

4.3.4 Comparison of network sizes 67
4.3.5 Reward functions 70
4.3.6 Number of sensors required 72
4.3.7 Cloud architecture 74
4.3.8 Example of perfect training and inability to learn 76
4.3.9 Generalization 79
4.3.10 Hybrid architecture 81
4.4 Summary and further outlook 82

5 Conclusions **84**

Bibliography **85**

A Installation and run of the program **89**

Chapter 1

Introduction

With the technological advances in recent decades such as chip miniaturization, improvements in sensor technology and most importantly rapid progress in artificial intelligence, it is estimated that by the year 2035 the first fully autonomous vehicles will be offered to the public. Modern Machine learning has played a major role in recent improvements in autonomous driving systems. Especially due to the increased reliability of the vehicle's perception and prediction within its environment. Nowadays, an autonomous vehicle is able to recognize traffic lights and signs and behave in accordance with them, follow road lanes, automatically park and unpark itself. Some technological companies already offer partial autonomous driving systems, e.g. Waymo founded by Google or Tesla automotive company.

The field of Machine learning also includes algorithms and methods in which an agent is trying to achieve a set goal by interacting with an environment. The agent can receive either a reward or penalty for its actions, which depend on a given environment. Such methods are called Reinforcement learning. They have received a great deal of attention in academic research over the past years. It is mainly due to the rapid advances in artificial neural networks, or in short Deep learning, which modern Reinforcement learning algorithms heavily utilize.

This thesis is concerned with the application of such a Reinforcement learning algorithm on the task of autonomous vehicle driving. For this purpose, the simulation environment TORCS has been chosen. *The Open Racing Car Simulator* enables the agent to observe through vehicle's sensors and camera output its environment and by actuators such as steering, acceleration and braking allows the agent to interact with the environment. The agent must learn a reward-based mapping between the sensor inputs and the vehicle dynamics. This mapping will then represent agent's behavioral strategy, called policy. When such a policy is learned, it then serves as vehicle's decision making brain and allows it to drive autonomously without further need of external help.

This thesis sets the goal to not only build a Reinforcement learning agent that can operate a vehicle in a simulation environment, but has the ambition in future work, to apply such a learned agent to a real world scaled RC model car. For that, simulation of different hardware architectures, noisy sensory and actuator data or network packet loss will be introduced. Also, the focus on minimal hardware requirements will be important, as the on-board embedded system is typically of low performance. The other architectural approach is controlling the vehicle wirelessly by a cloud service.

For the Artificial Intelligence part, the *Proximal Policy Optimization* (PPO) will be used, which is an on-policy, model-free, actor-critic method, also currently considered as the *state-of-the-art* Reinforcement Learning algorithm. The algorithm will operate on continuous domain, both for states and actions. Necessary improvements of the PPO will be done, in order to be able to operate on the task of autonomous driving. We also propose multiple architectures of the PPO agent and three reward functions, as the TORCS environment does not specify them directly. Our goal is also to obtain an agent with high performing policy in terms of proposed performance metrics, but at the same time encourage its generalization property in new, unknown environments.

Detailed experiments will be carried out for the comparison of different combinations of approaches and settings mentioned above as well as the thorough interpretation of its results.

The structure of the thesis will follow this order. In the 1st Chapter, the introduction and goals are set. In the 2nd Chapter, the theoretical foundations and detailed description of PPO algorithm are stated. In the 3rd Chapter, the design of autonomous driving system and its variants are introduced, followed by the experiments and test results in Chapter 4. Finally, in Chapter 5, the conclusion and outlook for the future research work is drawn.

Chapter 2

Reinforcement learning

The goal of this chapter is to provide a thorough introduction to the theoretical foundations of Reinforcement learning. Section 2.1 first provides an intuitive introduction to the topic and terminology, Section 2.2 describes and formalizes the problem of Reinforcement learning with *Markov theory* in detail. The definitions and derivations from Section 2.2 form the basis for the description of the algorithms of Reinforcement learning in Section 2.3 and Section 2.4 All algorithms relevant for the practical part of this thesis are presented and explained there. Section 2.3 deals with *value-based* algorithms. These create the basis for understanding the *policy-based* and *actor-critic* algorithms presented in Section 2.4. *Policy-based* algorithms have the nice property of being applicable in continuous action spaces - for actions with real valued output. Thus, they are also useful for autonomous driving.

2.1 Introduction

Thinking about the nature of learning, a first intuitive idea of an approach to learning may be learning through interaction with our environment. A toddler learns to grasp objects or look around at loud noises without being explicitly taught to do so by an adult. Instead, the toddler associates the actions taken with the reactions of the environment, learns which consequences follow from which actions, and which actions to perform to achieve a particular goal. *Reinforcement learning* (RL) is the algorithmic modeling of this interactive learning. An agent is located within an environment and selects actions within it. As a result, the agent receives a positive, neutral or negative reward from the environment as well as feedback about the new state of the environment. The agent's goal is to select actions that maximize the sum of rewards in the long-term. Formally, the interaction between agent and environment in RL can be described as follows: At a discrete time-step $t \in \mathbb{N}$, an agent receives a representation of the state of the environment S_t, which describes the current state of the environment. Based on information from S_t, the agent selects an action A_t to change the state of the environment it is in. The environment subsequently generates a reward $R_{t+1} \in \mathbb{R}$ for the agent's action for the next time-step $t + 1$, describes its changed state S_{t+1} and returns these two pieces of information to the agent. The dynamics by which the environment generates reward and state are determined by a *transition model* of the environment. The agent's goal is to learn a policy for selecting actions that maximize the sum of individual rewards over T time-steps.[42] [40] [46]

The loop of this agent-environment interaction can be described as a sequence of states, actions, and rewards that they are associated with and is generatively written in the form:

$$(S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, ...) \tag{2.1}$$

This sequence provides exactly the information that an RL agent can use to learn a policy. How the agent implements this algorithmically is explained in Section 2.3 and Section 2.4.

Such a sequence can be finite if the environment has a terminating state, i.e., a state that has only itself as a subsequent state, with a reward of zero. An example of a finite sequence of states, actions, and rewards is, for instance, the player-game machine interaction. A *state-action-reward* sequence can also be infinite, for example in a space-probe interaction. In the context of the present work, a finite sequence is generated by interaction of a vehicle with its environment, such as the driving route and the presence of other vehicles. In contrast to supervised learning, RL does not require annotated example data, but the data results from the interaction between agent and environment. RL also differs from unsupervised learning, since the goal is to maximize the sum of rewards and not to discover structures in the data. [42] [20]

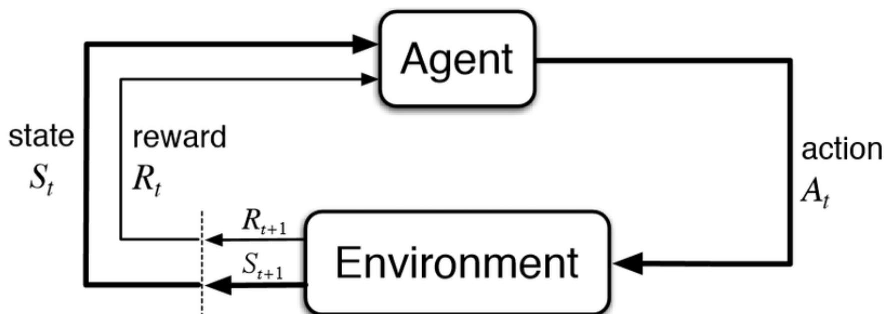

Figure 2.1: The main reinforcement learning interaction loop, Source: [42]

In addition to a representation of the interaction scheme of agent and environment, a consideration of the internal processes in agent and environment is also necessary, in order to be able to describe RL algorithmically in a thorough way. In particular, the functioning of an agent is considered, since it is the entity that has to develop a policy within an environment to maximize the sum of rewards. The dynamics of the environment are mostly considered as given and thus are not changeable. An RL agent algorithm can use one or more of the following functions to find a policy that maximizes the sum of rewards: a state-value function, a policy function, or an agent-internal model of the transition model of the environment. These agent's functions are computed from the data of one or more sequences of interactions, for example as the one described above. A state-value function describes how good it is for an agent to be in a certain state within the environment. While an immediate high reward may seem desirable for the agent in the short term, it is usually better for the agent to abandon it and instead look for a high long-term reward, from which he will strive much more in the long-run. [40] [42]

The long-term expected reward of a state is expressed by the *state-value* function. An agent can thus implicitly select those actions that lead to states for which the *state-value* function assumes a high value. Agents that exclusively use a *state-value* function are called *value-based*. In addition to *state-value* functions, there are *action-value* functions, which are defined similarly and will become important in subsection 2.2.2

A *policy* function describes which action an agent should perform in a certain state. An agent can have different policy functions in different interaction sequences, i.e. it can follow different strategies. The optimal policy function is such a policy-function that is more profitable in the long-term than all others. The goal of all agents is to find this optimal policy function for all states. In contrast to a state-value function, the policy function does not describe the quality of a state, but which action should be executed in a state. *Value-based* agents can implicitly infer a policy, but do not explicitly compute it. Agents that explicitly compute a policy function are called policy-based. Whereas agents that compute a policy function and a state-value function at the same time are called *Actor-critic* methods. An internal model of the environment transition model can be computed by an agent to predict how the environment will behave when the agent performs a particular action in a particular state. The agent can use this internal model of the environment to predict future states and future rewards. RL agents that use such an internal model are called *model-based*. An insight into model-based RL agents is given by Sutton and Barto in *Reinforcement Learning - An Introduction* [42] and in David Silver's lectures [40], or Sergey Levine's course on Reinforcement Learning [22], because in this thesis we are mostly focused on the model-free approach.

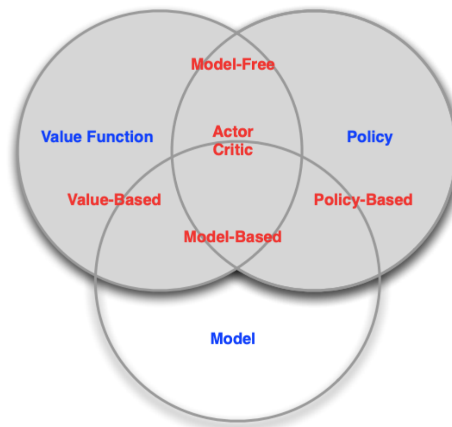

Figure 2.2: Classification of Reinforcement Learning algorithms, Source: [40]

A classification of RL agents into categories, based on these different computable approaches, can thus be made. Such a classification is shown above in Figure 2.2. According to it, the RL agents can be categorized based on the type of approach which take to solutions to sequential decision problems, where an agent must pursue a goal over many time-steps. The algorithms presented in subsection 2.3.1 are planning algorithms, which are model-based, thus they know the transition model from the beginning. If the transition model is not known, RL agents can be used to learn the return maximizing policy. In both cases, the environment can be modeled mathematically using *Markov decision process* (MDP).

The above computable components of an RL agent can be defined on an *MDP* and computed by algorithms. In subsection 2.3.2 and subsection 2.3.3, algorithms for value-based RL agents are presented, and in section 2.4 , algorithms for *policy-based* and *actor-critic* agents are discussed.

2.2 Theory

For the description of an environment of an agent, within an RL framework, the Markov theory can be used. Based on it, further algorithms can be derived, having the main objective of maximizing the sum of rewards from every single time-step individually. In order to be able to model such an environment as a *Markov decision process*, the environment has to be fully observable and number of actions and states have to be finite.

From the interaction scheme of agent and environment presented in section 2.1 it can be seen that a formal model must be able to represent states, rewards and actions. For this purpose, subsection 2.2.1 first shows how Markov processes can represent the states of an environment, later we introduce the rewards and returns. Lastly, we present the Markov decision processes, which can represent states, rewards, and actions. [40] [42]

2.2.1 Markov Process

Markov process (MP) can be used to formally describe the states of an environment. They also describe the dynamics with which the environment determines the state transitions. In such a Markov process, an agent can only observe the changing states of the environment and thus has no influence on the state of the environment. Markov processes are characterized by two properties. First, state transitions are not deterministic, i.e. they are influenced by randomness. Therefore, states are modeled as realizations of random variables, defined below. Second, the current state of the environment is enough for the selection of the future state, so that all previous states should not be considered. [20] [34]

Random variable and Probability function

The first property of Markov process states that each concrete state of an environment is the realization of a discrete random variable. Thus, a set V contains all states of an environment. A discrete random variable X is then able to assume, some state with a certain probability $\mathbb{P}(X = x)$ where $x \in V$. A state can be understood as a realization of a random experiment, which with a certain probability the environment assumes. This can be then written in terms of Probability function. [34]

$$\mathbb{P}(X = X(\omega)) = \mathbb{P}(X = x) \tag{2.2}$$

The repeated successive execution of a random experiment can be represented as a sequence of random variables. Such a sequence is then called a stochastic process, since each member of the sequence is a random variable $X_t(\omega)$ where $t \in \mathbb{N}$ and ω is the elementary outcome of all the possible outcomes Ω. An example of such a stochastic process can be sequence: $X_t(\omega), X_{t+1}(\omega), ..X_n(\omega)$, where a single term can be shortened to X_t.

Stochastic process

A stochastic process is defined as a collection of random variables defined on a common probability space.

Often in stochastic processes the probability that the environment assumes a certain state depends on the realized states of previous random variables. For example, if the weather forecast is assumed to be a stochastic process, then the yesterday's weather may still have an influence on tomorrow's weather. To represent this causality complicates the modeling of stochastic processes, so that with definition of *Markov property* the dependence

of future states is assumed only on the current state. This is the second important property of Markov process. [40]

Markov property

Definition of Markov property uses conditional probability formalism and can be formulated as, „The future is independent of the past, given the present.“ The stochastic process has Markov property if and only if for all $t \in N$ holds: $\mathbb{P}(X_{t+1}|X_t) = \mathbb{P}(X_{t+1}|X_1, ..X_t)$.

The Markov property has some advantages in practical RL such as the uniqueness and distinctiveness of states and also can be used to exactly formulate the probability of a state transition, which is defined as: [20] [40]

$$\mathcal{P}(x'|x) = \mathbb{P}(X_{t+1} = x'|X_t = x) \tag{2.3}$$

Given $|V| = n$ possible subsequent states, a state $x \in V$ also has n transition probabilities. Thus, if an environment in RL has n states, the environment can map the probability of transition from x to x' by an entry in a square $n \times n$ matrix. Together with the modeling of states as a realization of random variables, the definition of Markov process can be expressed.

The Markov process is a stochastic process with Markov property and is described as tuple $(\mathcal{S}, \mathcal{P})$ for which holds:

- $\mathcal{S} = s_1, s_2, ..s_n$ is a finite set of states
- $\mathcal{P}$ is an $n \times n$ transition matrix with probability values $\langle 0; 1 \rangle$

The elements of $\mathcal{P}$ are defined as: $\mathcal{P}(i|j) = \mathbb{P}(S_{t+1} = s_i|S_t = s_j)$ where $i, j \in (0, N)$

Single element of the matrix then represents the probability with which the random variable S_{t+1} assumes the state s', if $S_t = s$ is given. Each row of the transition matrix then gives for a state, a probability function over all subsequent states. On a Figure 2.3 bellow an example of terminating Markov process can be seen. [34] [7] [17]

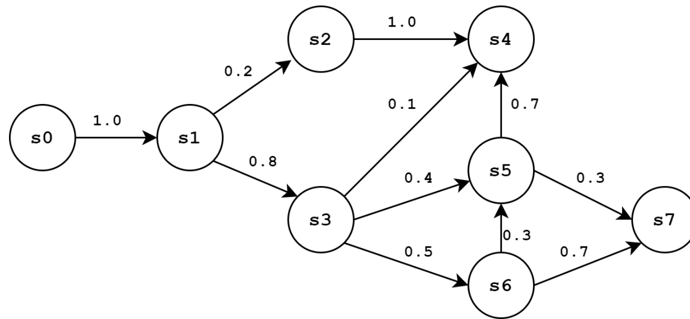

Figure 2.3: Markov process example, on the edges are transition probabilities, nodes represent states, starting in state s_0.

An agent in a Markov process, can observe the sequence of states. This sequence in RL is then called an episode or later in the thesis a trajectory. Because of the probabilities used in Markov processes, they are not always the same. Thus, for the Markov process from an

example on Figure 2.3, these finite sequences of states could be observed by an agent, if we assume that the state s_0 was initial. Episode 0: (s_0, s_1, s_2, s_4), Episode 1: $(s_0, s_1, s_3, s_5, s_7)$, and so on.

The episodes shown here as examples only contain the states. In the context of autonomous driving, a completed lap on a race track could be considered as an episode, where the visited states can be expressed by the sensory data observed by the agent's vehicle. Now for the complete description of an environment within RL framework, the actions and rewards need to be defined.

Rewards

Rewards can be defined as an extension to Markov process. This is then called *Markov reward process* (MRP) and is defined below. The main objective of an RL agent is to maximize the sum of rewards from each time-step. In the example of Markov process on Figure 2.3, the agent can observe different episodes, but has no tools to determine how good or bad an episode actually was. When the sum of rewards is high, then the episode can be considered as a good one. This sum is then called the return. With returns, we are now able to exactly measure this quantity of „goodness“. It even becomes possible to measure how good or bad a single state is, by calculating a state-value function. In the next subsection, this function will then be extended by actions, so that an agent can actively transition to good states in order to maximize the return.

Markov reward process is a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$. The set of states $\mathcal{S}$ and transition matrix $\mathcal{P}$ are similar to those in Markov process.

- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is a reward function, where for state s the expected reward as: $\mathcal{R}(s) = \mathbb{E}\left(R_{t+1} \mid S_t = s\right)$ is defined.
- $\gamma \in (0; 1)$ discount factor parameter

The reward function specifies how much reward an agent expects from the environment for a given state. The reward function is based on an expectation because the concrete reward values are for a probabilistic state transitions. Accordingly, scalar reward values would be added to the edges in Figure 2.3 for MRP. The sum of rewards weighted by probabilities then corresponds to $\mathcal{R}(s)$. So, if an agent is in a state s at time t, the agent receives reward R_{t+1} at time $t + 1$, when it transitions to a subsequent state s'. Rewards of an episode can then be described as a sequence $(R_1, R_2, ..R_t)$. The sum of this sequence is called the *return* and will be defined now. [40] [42] [17] [46]

Expected Return

The return G_t is the sum of discounted rewards obtained in a single episode by an agent.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + .. = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

The calculation also includes the discount factor γ. As the return is an infinite series, the discount factor is in an interval $\langle 0, 1 \rangle$, then the following can occur. If it is equal to one, then the value of the series goes to infinity. Only in the case of always terminating episodes the agent can calculate the return. If it is below one, then the return has a finite value, so the agent can determine how good or how bad an episode was. Not only is the discount

factor useful mathematically, but it is also useful for tuning of an agent for the benefits of rewards. If early rewards in an episode are more important than later ones, then it should correspond to a value close to zero. If the rewards express monetary gains, then it is this case, because early rewards earn additional interest. In contrast, the closer it is to one, the more important later rewards are. For example, for the PPO agent to be evaluated, γ has the value 0.99.

The return can be used to determine the sum of the discounted rewards of an episode. The state-value function according to definition below, can be used to determine how high the expected long-term return is, starting from a certain state. Due to the probabilities of state transitions, the episodes that always start with a particular state does not need to be always the same. Therefore, the expected return of a particular state is the expected value of the conditional density function over the probabilities of the returns for a state. The return G_t can thus be treated mathematically as a continuous random variable. [40] [42] [20]

State-value function

If an agent knows from the state-value function what the long-term expected return of each state of an environment is, then it can infer which state it should transition to, in order to maximize the return of an episode. In this case, the agent has to move to the state that has the highest long-term expected return. But in an MRP, there are no actions using which an agent could transition to different states. For that we need to define actions, which is done in the MDP introduction.

If the agent observes a sequence of states and rewards, it can remember the subsequent rewards for each state, calculate the return of an episode from them, and thus iteratively adjust the probabilities for the higher occurrence of a return over several episodes. If the number of episodes goes to infinity, the estimated probabilities converge to the true probabilities and the long-term expected return of a state is found. Intuitively, the more episodes and consequent returns an agent observes, the better it can estimate how good a state is. This estimative way of determining the state-value function, or the action-value function are going to be introduced in the next subsection. We will also show how the state-value function can be determined by a recursive iterative approach. This iterative approach originates from Bellman's equation and the long-term return must satisfy it. From this equation, an iterative calculation rule can be derived for finding that long-term reward of a state. To derive Bellman's equation, a few derivations are necessary. According to definition of state-value function: [42] [20]

$$
\begin{aligned}
v(s) =& \mathbb{E}[G_t | S_t = s] \\
=& \mathbb{E}[R_{t+1} + \gamma G_t | S_t = s] \\
=& \mathbb{E}[R_{t+1} + \gamma v\left(S_{t+1} = s'\right) \mid S_t = s] \\
=& \mathcal{R}(s) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s) v\left(s'\right)
\end{aligned}
\tag{2.5}
$$

Finally from that can be derived equation written below. The last derivation expresses that the long-term expected return of a state depends only on the immediate reward and the long-term expected return of the subsequent state.

To arrive at the equation, the definitions of state-value function and the explanation, that the expected value of $v(S_{t+1} = s')$ is the average of all $v(s')$ weighted by transition probabilities, can be used. If the equation is further extended to include actions, the most

important equation in all of Reinforcement learning emerges: the *Bellman equation*. It expresses that if an agent wants to know the long-term expected return from a state, it only needs to add together the reward of the state and the long-term expected return of the next state. [17] [46]

2.2.2 Markov Decision Process

Unlike Markov process or Markov reward process, an agent in *Markov decision process* (MDP) has the possibility to take control over the state transitions of the environment through actions. In an MDP, an agent can co-decide which state the environment should transition to next. The goal of an agent is to transition to states using actions so that the return is maximized according to definition of Expected return. [40]

The Markov decision process is Markov reward process with actions. It is an tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ defined as:

- $\mathcal{S} = s_1, s_2, .., s_n$ a finite set of states
- $\mathcal{A} = a_1, a_2, .., a_m$ a finite set of actions
- $\mathcal{P}$ is an $m \times m \times n$ transition matrix where an element:
  $\mathcal{P}(s'|s,a) = \mathbb{P}\left(S_{t+1} = s' \mid S_t = s, A_t = a\right)$ gives the probability of the state transition from s to s' when action a is chosen
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, where for state s, action a, the expected reward $\mathcal{R}(s,a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$ is defined.
- $\gamma \in (0;1)$ discount factor parameter

How an agent determines which action it performs at a time-step in a state is described by the policy function. This policy function specifies which action an agent performs and can be deterministic or stochastic. First, the stochastic case is considered in definition below. [40] [20] [42] [17] [46]

Stochastic Policy function:

$$\pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \tag{2.6}$$

Accordingly, the entire behavior of an agent, which describes with which probabilities which actions are executed in which states, can be described by an $|\mathcal{S}| \times |\mathcal{A}|$ matrix. Each entry of the matrix is a policy function $\pi(a_i|s_j)$ and $i \in N, j \in M$. This matrix is then called a policy π. [40] [42]

Sum over all possible actions in a given state must be equal to one. The entries of the matrix are unchanged across all time-steps of an episode. This means that the underlying policy of different episodes can be different, but does not change during an episode. For this reason, the state-value function from definition 2.5 must be adapted for MDPs, since only the long-term expected return, starting from a state, should be determined with respect to a behavior of the agent described by a policy. [40]

State-value function:

$$\begin{aligned} v_\pi(s) =& \mathbb{E}_\pi[G_t|S_t = s] \\ =& \sum_a \pi(a|s) \sum_{s',r} \mathcal{P}(s', r|s, a)[\mathcal{R} + \gamma v_\pi(s')] \end{aligned} \tag{2.7}$$

The definition 2.7 states, that only the rewards of episodes that were based on the same policy of the agent may be included in the expected value, which contrasts with the definition 2.5. The $v_\pi(s)$ also satisfies the Bellman's equation. [40]

Action-value function:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.8}$$

If an agent has a policy π, is in a state s at time-step t, and performs action a, the action-value function $q_\pi(s, a)$ represents the expected long-term return for this state and action under the current policy. Thus, the action-value functions can be described as a table, where each entry describes how profitable it is in the long-term for the agent to perform a particular action in a particular state. In contrast, the state-value function can be defined as 1D vector, where each element describes the long-term return of a state. The relationship of state-value function and action-value function can consequently be defined as:

$$v_\pi(s) = \max_a q_\pi(s, a) \tag{2.9}$$

The table of action-value functions is also called Q-Table. A policy and the action-value functions of an agent can both be represented by $|\mathcal{S}| \times |\mathcal{A}|$ matrix. Implementations of RL algorithms then mostly use only the matrix of action-value functions and infer the policy from them. The goal of an agent is to maximize the return. In the above context, this means to find a policy that yields as much return as possible. For this, we first note what it means for one policy to be better than another. Let π and π' be two different stochastic policies. Then π' is called better or equal to π exactly if $v'_\pi(s) \geq v_\pi(s)$ holds for all states. It can be shown that there is always at least one policy that is better or equal to all other policies. This policy is then called the optimal policy π_*. If an agent uses the optimal policy, then for all states and actions the agent will use the optimal state-value function and the optimal action-value function: [40] [42] [29] [17]

$$v_*(s) = \max_\pi v_\pi(s) \text{ and } q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.10}$$

Conversely, if the optimal action-value function is found, the optimal policy can be derived. The optimal policy function $\pi_*(a|s)$ always has a selection probability of 1 for such an action for which $q_*(s, a)$ is maximal for a given state. [40] [42]

$$\pi_*(a \mid s) = 1 : \text{ if } a = \arg\max_a q_*(s, a), \text{ else } 0 \tag{2.11}$$

It also follows that the optimal policy is deterministic, meaning for the same state the optimal policy function always selects exactly the same action. In contrast, Definition 2.6 allows for stochastic behavior of the policy. The transition from stochastic to deterministic policy is explained by the greedy selection of actions in the last equation. As will be shown in the following sections, all RL algorithms presented here use a stochastic policy, since it allows exploration of the state space. Policies of value-based algorithms tend to

be nearly deterministic. Policies of policy-based algorithms can remain stochastic if the optimal policy is stochastic too. Both classes of algorithms have in common that they attempt to determine π_*, because an MDP is said to be solved if such a policy is found. In each case, the path to determining an agent's optimal policy in an environment is done by an iterative procedure in which the agent repeatedly tries a policy, learns from failure, and derives a new, improved policy. [40] [42] [22] [17]

Bellman's optimality equation

Just like the state-value functions, the optimal state-value function $v_*(s)$ must also satisfy Bellman's equation. It was already said that from Bellman's equation iterative computational rule, the long-term expected return of each state can be derived. Since it is now also required that an agent can include actions into its behavior, the state-value function for MRP is extended to include actions. For this, Equation 2.9 first states that the optimal state-value function of a state under a policy is equal to the value of the return-maximizing action of the action-value function for a fixed state. [42] [40] Formally, this means:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}} q_{\pi_*}(s, a) \\
&= \max_{a} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a)] \\
&= \max_{a} \sum_{s', r} p(s', r | s, a)[r + \gamma v_*(s')]
\end{aligned}
\tag{2.12}
$$

This equation derives the recursive relation to the subsequent state, so that the reference to the optimal policy can be omitted. The last derivation is called Bellman's optimality equation for $v_*(s)$. Similarly, for action-value function can be derived too. Unlike above, here, the relation to rewards and the expected return is immediate: [42]

$$
\begin{aligned}
q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a' | S_t = s, A_t = a)] \\
&= \sum_{s', r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')]
\end{aligned}
\tag{2.13}
$$

Backup diagrams are great for intuitive derivation of the optimality equations. They are presented in detail in *Sutton and Barto*[42] and *Silver*[40]. In the following sections, RL algorithms will be presented and derived, based on the definitions presented in this subsection. They are able now to compute the optimal policy π_*, which translates to obtaining the maximal return during the interaction with the environment.

2.3 Value-based methods

In this section, agent algorithms are presented that first iteratively compute the state-value functions or the action-value functions of the states and actions, respectively, in order to derive continuously improving policies from them. Therefore, these algorithms are called value-based. Subsection 2.3.1 deals with planning algorithms where transition probabilities and reward function must be known. Subsection 2.3.2 presents algorithms that are used in MDPs with finite episodes. Subsection 2.3.3 explains algorithms for MDPs with infinite episodes. Subsection 2.3.4 shows how function approximators, such as neural networks, can be used instead of state-action look-up matrices. Finally, thoughts on the value-based algorithms are presented.

2.3.1 Dynamic Programming

If the transition probabilities and the reward function of an MDP environment are known, then an agent can compute an optimal policy using *dynamic programming* (DP). Dynamic programming is a class of planning algorithms that consists of the evaluation of a given policy (policy evaluation) and the subsequent improvement of the policy (policy improvement). The goal of these two steps is to infer $v_*(s)$ or $q_*(s,a)$ for all the states and actions in a procedure called *Generalized Policy Iteration*. Subsequently, using these to infer the optimal policy. For the algorithms presented in here and in subsection related to MC methods, we will assume the simplest case of the procedure, which is characterized by always determining the state-value or the action-value function in the Policy Evaluation step before computing a new, improved policy in the Policy Improvement step. Other variants of the Generalized Policy Iteration procedure only approximate the state-value or action-value function and can thus compute the optimal policy faster. In the policy evaluation step, DP first computes the state-value function for every state and under current policy. This is iterated until convergence to $v_k(s) \approx v_\pi(s)$ with: [40] [42] [22].

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\big(\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) v_k(s')\big) \tag{2.14}$$

for all states. This equation is the analogous equation to Bellman's optimality equation for $v_*(s)$ in iteration form. [40] [42] Once $v_\pi(s)$ is found for a given policy the action-value function can be determined.

$$q_\pi(s,a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) v_\pi(s') \tag{2.15}$$

This equation is then the Bellman's equation for $q_\pi(s,a)$ and can be used in the Policy Improvement step to create a new policy. It should also be ensured that the new, deterministic policy is better or equal to the old policy, when for all states applies: [42] [40]

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{2.16}$$

This means that choosing action $a' = \pi'(s)$ in state s under policy π produces a better result than action $a = \pi(s)$. Next, $q_{\pi'}(s,a)$ is computed for the new policy π' in the Policy Evaluation step and improved with $\pi'(s) = \arg\max_a q_{\pi'}(s,a)$. If Policy Evaluation and Policy Improvement steps are executed enough times and no more improvement of the old policy over the new policy is found, then optimal q_* and π_* are found. [40] [22] [42]

Well-known algorithms in DP which implement this Generalized Policy Iteration are the *Policy Iteration algorithm* and its special case the *Value Iteration algorithm*. They use the equations Bellman's equations for $v_*(s)$ and $q_*(s,a)$ and a variant of the Generalized Policy Iteration method, where $v_\pi(s)$ does not have to be determined. The iteration in the Policy Evaluation step is immediately followed by the Policy Improvement step. An obstacle for the practical application of these algorithms is the frequent lack of knowledge about the environment dynamics. As the agent does not know all the states of the environment already. It has to gradually explore the environment in order to know its surroundings as it is not known right from the beginning. This issue is addressed in the following subsections. [40] [42] [20]

2.3.2 Monte Carlo methods

Unlike DP, *Monte Carlo* (MC) methods can be used if the transition probabilities and the reward function of the environment are unknown. With MC methods, an agent can infer the optimal policy π_* from the optimal action-value function $q_*(s, a)$ estimated by experience. Just like algorithms in DP, MC methods use the iterative procedure Generalized Policy Iteration to incrementally infer π_*. For this, first a finite trajectory $(S_0, A_0, R_1, S_1, A_1, .., R_n, S_n)$ is generated. The actions are then determined and selected by a stochastic policy $\pi(a|s)$, whose evolution is explained below, and the states and rewards come from the unknown environmental dynamics. From this episode, G_t is then computed for all state-action pairs reached. This is followed by the policy evaluation step: [42] [40]

$$q_{k+1}(s, a) = q_k(s, a) + \alpha(G_t - q_k(s, a)) \tag{2.17}$$

Where, $\alpha \in (0; 1)$ is a learning rate and the return G_t gives the sum of discounted rewards starting from action a in state s. Consequently, the term $G_t - q_k(s, a)$ corrects the value of $q_{k+1}(s, a)$ in the direction of the target G_t.

The index $k \in \mathbb{N}_0$ then gives the current episode and in the limiting case applies again $q_k(s, a) = q_\pi(s, a)$. Since MC methods do not require knowledge of the environment dynamics, a policy for episode generation in MC methods should be non-deterministic, otherwise it is not guaranteed that all states and actions are explored. With a stochastic policy, actions can be randomly selected and such a policy is therefore used by MC methods for exploration. This approach is called the ϵ-greedy approach for exploring all states and actions in an MDP. Accordingly, a new policy π' is explored by MC methods with: [22] [40] [42]

$$\pi'(a \mid S_t) = \begin{cases} \frac{\varepsilon}{|\mathcal{A}|} + 1 - \varepsilon & \text{if } a = A^* \\ \frac{\varepsilon}{|\mathcal{A}|} & \text{if } a \neq A^* \end{cases} \tag{2.18}$$

where $A^* = \arg\max_a q_\pi(S_t, a)$. The problem of exploring all states and actions in an unknown MDP is a major problem in RL and is referred to as the *exploration vs exploitation* dilemma. Also, in the case of applying the ϵ-greedy approach to exploration, a greedy policy improvement theorem ensures that the new policy is better than the old one. The process of adjusting action-value values in the Policy Evaluation step and determining a new policy in the Policy Improvement of π_* step are referred to as training the RL agent. Equation 2.17 shows that MC methods use the reward G_t of an episode. This can only be calculated if the episode is already terminated. Consequently, this means that MC methods can only be used in environments with finite, i.e. always terminating episodes. [42] [40] [17] [46]

2.3.3 Temporal Difference methods

Temporal difference (TD) methods, unlike MC methods, can also be used in environments with infinite episodes. The mechanism that makes this possible is called *bootstrapping*, determines $v_{k+1}(s)$ or $q_{k+1}(s, a)$ from the immediate reward plus $v_k(s')$ or $q_k(s', a')$, respectively, and is already applied in a similar way in DP through Bellman's equations. Unlike algorithms of DP, however, TD methods do not depend on the environment dynamics $\mathcal{P}$ and $\mathcal{R}$. Like DP and MC methods, TD methods also use the iterative Generalized Policy Iteration procedure to determine the optimal policy. The policy evaluation step differs in TD methods from MC methods, but the policy improvement step is the same for both with the ϵ-greedy approach. The difference between TD methods and MC methods in the

evaluation step can be most easily illustrated by the iterative calculation of the state-value function of a policy with the following equation: [40] [42] [22]

$$v_{t+1}(s) = v_t(s) + \alpha(R_{t+1} + \gamma v_t(s') - v_t(s)) \tag{2.19}$$

where the term $(R_{t+1} + \gamma v_t(s') - v_t(s))$ is called *TD-error* and the term $G_t^{(1)} = R_{t+1} + \gamma v_t(s')$ is called *TD-target*.

This equation illustrates the difference between TD and MC methods. TD methods only have to wait until the next time-step before adjusting $v_{t+1}(s)$, whereas MC methods usually have to wait until the end of the episode, for the computation of G_t. Therefore, the index t now corresponds to the time-step. More generally, the TD target can be expressed as the n-step return, formulated as: [42]

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ..\gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n} = s(n))$$

(2.20)

where for limiting case of n nearing infinity, the TD methods become MC methods. TD methods that use $G_t^{(1)}$ are called TD(0). There are methods, where the specific number of steps n in $G_t^{(n)}$ do not have to be specified. These methods are called TD(λ), we will not discuss its concept here, just for now. Later, when we will be discussing the PPO algorithm, this concept will be used for definition of *Generalized Advantage Estimation* (GAE), one of the major aspects of PPO. We can mention, that the concept is known as exponentially weighed average.

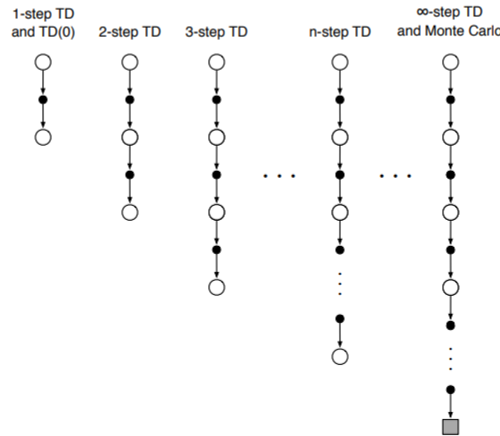

Figure 2.4: Here in graphics we can see the difference between MC and TD methods (n-step TD prediction), Source: [42]

The TD methods use the action-value function for the evaluation step and iteratively determine $q_\pi(s, a)$. A TD method that implements this practically is called *SARSA* and adjusts the action-value function in the evaluation step for a state, action pair at each time-step of the episode as:

$$q_{t+1}(s,a) = q_t(s,a) + \alpha(R_{t+1} + \gamma q_t(s',a') - q_t(s,a)) \tag{2.21}$$

[42] The name *SARSA* comes from the generated sequence of transitions: (s, a, r, s', a'). This sequence is generated by the environment dynamics and the current ϵ-greedy policy. Once $q_{t+1}(s,a)$ is computed, the new policy π' is determined at time-step $t+1$. These iterations are repeated until optimal action-value and optimal policy functions are found. This calculation of action-value finds application in the SARSA(0) algorithm. Analogous to TD(λ) there are also SARSA(λ) methods. [42]

SARSA is referred to as an on-policy algorithm, because it evaluates and improves the same policy that is used to select the next action. In contrast, there are off-policy algorithms that evaluate and improve one policy, but use a different policy for selecting the next action. One of the reasons can be an introduction of a new version of an agent with policy π, which wants to learn or even improve the behavior from an old agent with good performing policy μ. But still, they can be the very same policy. As an example an off-policy TD algorithm can be considered the *Q-learning*.[29] Consider we have a policy $\mu(a|s)$ that determines the actual actions a and a'. Then we have second policy $\pi(a^*|s)$ which is used to compute the TD target. Then, in the evaluation step, at each time-step of the episode, the action-value function can be calculated as: [40] [42]

$$q_{t+1}(s,a) = q_t(s,a) + \alpha(R_{t+1} + \gamma \max_{a^*} q_t(s',a^*) - q_t(s,a)) \tag{2.22}$$

[40] In Q-learning in contrast to SARSA, the action maximizing action a^* of all action-value functions at a fixed s' is always chosen when adjusting the action-value function for a (s,a), regardless of which action a^* of policy π was chosen. As in the SARSA algorithm, in Q-learning the computation of $q_{t+1}(s,a)$ is followed by the policy improvement step with the ϵ-greedy approach with respect to π and optionally μ as well. [40] [17] [46]

2.3.4 Value-based methods and Function approximators

The *Monte Carlo* and *Temporal difference* algorithms already described are referred to as *Tabular Methods*, since the central building block is the $|\mathcal{S}| \times |\mathcal{A}|$ matrix of the action-value returns. However, if the number of states or actions becomes very large, this matrix becomes also very large. An example of this is an agent for the autonomous guidance of a vehicle, which can determine the state of the surrounding environment by a camera output. Then, each pixel of the camera image represents three states in the RGB space. In addition to the high memory requirements of such a matrix, it will be nearly impossible for an agent with MC or TD methods, to evaluate all states in order to derive a policy. [20]

Approximate Solution Methods emerge as a solution approach to the problems of Tabular Methods, which are both more memory efficient and can generalize. Such Approximate Solution Methods use parametrizable function approximators such as decision trees, regression methods, or neural networks, instead of a matrix, to determine the action-value returns. If neural networks with hidden layers are used as function approximators, the resulting algorithms are classified as *Deep Reinforcement Learning* (DRL). Unlike Tabular Methods, Approximate Solution Methods use parametrized function approximators: [42] [29]

$$\hat{v}(s,\boldsymbol{w}) \approx v_\pi(s) \text{ or } \hat{q}(s,a,\boldsymbol{w}) \approx q_\pi(s,a), \text{ where } \boldsymbol{w} \in \mathbb{R}^d, d \in \mathbb{N} \tag{2.23}$$

The vector $\boldsymbol{w}$ then represents the weight vectors of the neural network. If a loss function $E(\boldsymbol{w})$ is defined over this weight vector, then in the iterative *Stochastic gradient descent* (SGD) procedure, the weight vector can be updated, as long as the error continuously decreases i.e. until: (s, a, $\boldsymbol{w}$) $\approx q_\pi(s, a)$

As a loss function here the *Mean square error* (MSE) can be used, which is typically used in regression optimization problems. Thereby T is the set of training data from which the target variable y is taken. However, unlike in classical supervised learning, no information is available about the target variable, here $y = q_\pi(s, a)$, so the target is assumed to be the loss function. Thus, if function approximators are applied to MC methods and SGD is assumed as the optimization algorithm, the loss function for one-step SARSA is defined as: [34] [42] [11] [22]

$$\overline{MSE} = \frac{(R_{t+1} + \gamma\hat{q}(s', a', \boldsymbol{w}) - \hat{q}(s, a, \boldsymbol{w}))^2}{2} \tag{2.24}$$

The following rule is used to adjust the weights for one-step SARSA:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t + \alpha(R_{t+1} + \gamma\hat{q}(s', a', \boldsymbol{w}) - \hat{q}(s, a, \boldsymbol{w}))\nabla\hat{q}(s, a, \boldsymbol{w}) \tag{2.25}$$

The update is done in a way that each vector's component that is responsible for a part of the error, should be updated accordingly. A high share on the error exists if the slope of the error with respect to the particular component is large. The weights are adjusted until the difference between two subsequent updates is less than some defined ϵ value (typically small number). Then we can say that $\hat{q_*}$ is found, from which the optimal policy can be derived. Often also early-stopping is being used in practical deep learning instead of ϵ.

So now that the vectors of state-action values are not a look-up table anymore, they can be written as follows and are therefore approximated by the neural network.

$$\hat{\boldsymbol{y}} = \begin{pmatrix} \hat{q}(s, a_1, \boldsymbol{w}) \\ \vdots \\ \hat{q}(s, a_n, \boldsymbol{w}) \end{pmatrix} \tag{2.26}$$

Also in the policy improvement step, the ϵ-greedy approach for the action selection can be applied. [11][40] [6] [17] [46] [29]

Thoughts on value-based methods

All algorithms already presented are based on the Generalized Policy Iteration and thus are easy to understand and implement in the same way. The disadvantage of this simplicity is that they must first compute an action-value function in order to infer an improved policy, without it, it is not possible. The main goal of an RL algorithm is to determine the optimal policy so that an agent can maximize the return. Therefore, an agent is primarily interested in policies and state-value functions and action-value functions are only a means to obtain it. If an ϵ-greedy approach is applied to the vector $\hat{\boldsymbol{y}}$ above, the computational costs to calculate an improved policy can increase very heavily for a very large number of actions. Also, to find the action with the largest long-term expected return, all elements of the vector would have to be iterated. In the extreme case, the action space can be even continuous. Thus, the algorithms already presented do not show an efficient solution to this problem.

Next issue is, due to the vanishing coefficient in the ϵ-greedy approach, the initially stochastic policies tend towards deterministic ones, i.e. $\pi(a|s)$ is always either 1 or 0. This is a problem at environments, which are not fully observable. The value-based algorithms then cannot be used at all. In these cases, a stochastic policy would fit the problem more, since it always includes randomness in its policy. [42] [40] [22] [34]

2.4 Policy-based methods

Policy-based algorithms parametrize the policy directly, similar to the parametrization of state-value and action-value functions in subsection 2.3.4. This section deals with algorithms and methods that learn a parametrized policy. In subsection 2.4.1, stochastic policy gradient methods are introduced and explained. Building on this, subsequent subsections introduce various stochastic policy-gradient algorithms. Subsection 2.4.3 explains the idea behind actor-critic methods and subsection 2.4.4 explains the trust-region methods and the one policy gradient algorithm (PPO) that is used in Chapter 4.

2.4.1 Stochastic Policy Gradient methods

All algorithms which use a parametrized policy are called Policy gradient methods. We can introduce the parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d, d \in \mathbb{N}$ of a policy. Then, extending Definition 2.6, we can derive a formula as:

$$\pi(a|s, \boldsymbol{\theta}) = \mathbb{P}(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$$

is the probability that action a is executed in state s with parameters $\boldsymbol{\theta}$ at time-step t of π_θ. If a performance measure $J(\boldsymbol{\theta})$ is introduced on the parameter vector, then it can be formulated that policy gradient methods, using the gradient ascent procedure, can evaluate the parameter vector iteratively with: [40] [22] [42]

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha \nabla J\left(\boldsymbol{\theta}^t\right) \tag{2.27}$$

The derivation of a loss function of performance measure for policy gradient methods not as easy as in the case of value-based methods. For this, similar to the distinction between MC methods and TD methods, we need to differentiate between finite and infinite episodes. For both cases, the theorem on policy gradients shows how to compute the $\nabla J(\boldsymbol{\theta})$. [40] [42]

In the case of finite episodes, we can say that: the goodness of the parameter vector is measured by the return of the first state of the episode. The policy gradient for finite episodes can be then formulated as: [42]

$$\begin{aligned}
\nabla v_{\pi_\theta}(s) &= \nabla\left(\sum_a \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_\theta}(s, a)\right) \\
&= \sum_a \left(\nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_\theta}(s, a) + \pi(a \mid s, \boldsymbol{\theta}) \nabla q_{\pi_\theta}(s, a)\right)
\end{aligned} \tag{2.28}$$

$$\vdots$$

$$= \sum_s \sum_{k=0}^{\infty} \mathbb{P}\left(s_0 \rightarrow s, k, \pi_\theta\right) \sum_a \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_\theta}(s, a) \tag{2.29}$$

where $\mathbb{P}(s \rightarrow s', k, \pi_{\boldsymbol{\theta}})$ is the probability of a state transition from s to s' in k steps under policy $\pi(a \mid s, \boldsymbol{\theta})$. This probability thus serves as a weighting of the gradient of the return. Moreover, if $\eta(s) = \sum_{k=0}^{\infty} \mathbb{P}(s \rightarrow s', k, \pi_{\boldsymbol{\theta}})$ is defined, it can be further written:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}}(s) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_\theta}(s, a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \\
&= \mathbb{E}_\pi \left(\sum_a q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla \pi(a \mid s, \boldsymbol{\theta}) \right)
\end{aligned} \tag{2.30}$$

(Proof of Policy Gradients theorem - for episodic case)[42]

The equation describes that the gradient of $J(\boldsymbol{\theta})$ is proportional to the probability sum of the $\sum_s \mu(s)$, which indicates the probability that the agent is in state s, multiplied by the sum weighted gradients of the probabilities for the selection of actions. The $\sum_s \eta(s)$ is the proportionality constant.

Practically, it provides the insight, that when the agent is in a state, it should move in the direction of weights that represent the largest increase in the probability of selecting an action in that state, weighted by the expected return for that state and action. [42] [22]

In the case of infinite episodes, the performance measure $J(\boldsymbol{\theta})$ is the average reward per time-step. The gradient $\nabla J(\boldsymbol{\theta})$ can then be determined by the policy gradients for infinite episodes theorem:

$$\nabla J(\boldsymbol{\theta}) = \sum_s \mu(s) \sum_a \nabla \pi(a \mid s, \boldsymbol{\theta}) q_{\pi_\theta}(s, a) \tag{2.31}$$

and thus the calculations of a gradient for infinite case corresponds to the one of episodic case. [42] [40]

2.4.2 Monte-Carlo Policy Gradient (REINFORCE)

The Monte-Carlo Policy Gradient algorithm is a purely policy-based algorithm. Similarly to the classical MC methods, it uses the return of complete finite episodes for the parameter adjustments. The algorithm is called REINFORCE. In an Algorithm 1 we show a pseudo-code for it. Here, the return is gradually expanded with each time-step, unlike in MC methods. [34]

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha \gamma^t G_t \nabla \ln \pi \left(a \mid s, \boldsymbol{\theta}^t\right)$$

This weight adjustment ensures that the weight vector θ^t is adjusted with the product of return G_t and the gradient vector $\nabla \ln \pi(a \mid s, \boldsymbol{\theta})$, which indicates the largest increase in the probability of selecting action a in state s. So the return G_t serves here as a scaling factor

of the gradient. The higher the return, the greater should be the probability of selecting an action a in state s by policy $\pi_{\boldsymbol{\theta}}$. [42] [22]

Algorithm 1: Monte-Carlo Policy Gradient - REINFORCE

Input: differentiable policy $\pi(a \mid s, \boldsymbol{\theta})$
Output: policy parameters θ
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
while *True* **do**
  Generate an episode $s_0, a_0, r_1, ..s_{T-1}, a_{T-1}, r_T$ using policy π_θ
  foreach *time-step t=0, .. ,T-1* **do**
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t + \alpha\gamma^t G_t \nabla \ln \pi\left(a \mid s, \boldsymbol{\theta}^t\right)$
  end
end

The algorithm uses as optimization the stochastic gradient ascent, since the weights are adjusted at each time-step. For the gradient $\nabla \ln \pi\left(a \mid s, \boldsymbol{\theta}^t\right)$, depending on whether the environment is discrete or continuous action space, the policy parametrizations explained in previous subsection can be used. The disadvantage of this algorithm is the high variance of the returns between time-steps and the associated „slowness“ of the learning process. The reason for the variance is the unadaptive and loose formulation of the scaling factor. [20] [42]

2.4.3 Actor-Critic Policy Gradient methods

The long-term expected return for an action in a state, $q_{\pi_\theta}(s, a)$, visible in the proof of Policy Gradients, is a scaling factor for the largest increase in the probability of selecting such a given action in a given state. In the previous subsection, the scaling factor is the return G_t. In order to improve the slowness of learning and the high variance of the REINFORCE, the theorem on policy gradients is extended to include a baseline $b(s)$ that acts individually for each state, thus acts adaptively on the scaling factor. It can be written as: [42] [22] [41]

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \left(q_{\pi_\theta}(s, a) - b(s)\right) \nabla \pi(a \mid s, \boldsymbol{\theta}) \tag{2.32}$$

If the baseline is zero for all the states, the initial equation 2.30 is again obtained. For example, a vector of constants could be also used as a baseline. If a function approximator with bootstrapping is used as a baseline, such as TD(0) or SARSA(0), the concept of Actor-Critic methods can be formulated. Here, the Critic computes the scaling factor and the actor adjusts the policy parameter $\boldsymbol{\theta}$. For the Actor network, the weight adjustment formula is then as follows: [40] [22]

$$\begin{aligned}\boldsymbol{\theta}^{t+1} &= \boldsymbol{\theta}^t + \alpha\left(q_{\pi_\theta}(s, a) - b(s)\right) \frac{\nabla \pi\left(a \mid s, \boldsymbol{\theta}_t\right)}{\pi\left(a \mid s, \boldsymbol{\theta}_t\right)} \\ &= \boldsymbol{\theta}^t + \alpha\left(R_{t+1} + \hat{q}\left(s', a', \mathbf{w}^t\right) - \hat{q}\left(s, a, \mathbf{w}_t\right)\right) \frac{\nabla \pi\left(a \mid s, \boldsymbol{\theta}_t\right)}{\pi\left(a \mid s, \boldsymbol{\theta}_t\right)}\end{aligned} \tag{2.33}$$

where

$$\frac{\nabla \pi\left(a \mid s, \boldsymbol{\theta}_t\right)}{\pi\left(a \mid s, \boldsymbol{\theta}_t\right)} = \nabla \ln \pi\left(a \mid s, \boldsymbol{\theta}_t\right)$$

In equation above, the target $R_{t+1} + \hat{q}\left(s', a', \mathbf{w}^t\right)$ is assumed as a scaling factor and the baseline $\hat{q}\left(s, a, \mathbf{w}^t\right)$ tries to counteract the strongly fluctuating target values. The Critic thus provides the scaling factor to the Actor. The Algorithm 2 shows the pseudo-code for such an Actor-Critic algorithm. [42]

Algorithm 2: Actor-Critic (one-step)

Input: differentiable policy parametrization $\pi(a \mid s, \boldsymbol{\theta})$
Input: differentiable action-value parametrization $\hat{q}(s, a, \mathbf{w})$
Output: policy parameter θ and action-value weights
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and action-value weights $\mathbf{w} \in \mathbb{R}^{d}$
while *True* **do**
  Observe state from environment
  Sample action from $\pi(a \mid s, \boldsymbol{\theta})$
  foreach *time-step t=0, .. ,T-1* **do**
    Take action a, observe s' and r from environment
    Sample next action a' from $\pi(a' \mid s', \boldsymbol{\theta}_t)$
    $\delta_t \leftarrow r_t + \gamma \hat{q}\left(s', a', \mathbf{w}_t\right) - \hat{q}\left(s, a, \mathbf{w}_t\right)$
    $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \delta_t \nabla \ln \pi\left(a \mid s, \boldsymbol{\theta}_t\right)$
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \beta \delta_t \nabla \hat{q}\left(s, a, \mathbf{w}_t\right)$
    $a \leftarrow a'; s \leftarrow s'$
  end
end

Just like n-step TD methods, there are also n-step Actor-Critic algorithms, which are fundamentally explained and detailed in [42] Sutton and Barto and [22] Levine.

To further reduce the variance instead of the action-value function, an *Advantage* function can be used. The Advantage function then indicates whether and by how much action in state is better or worse compared to all other possible actions in that state. If advantage for a given action is greater than one, that action is on average better than all other actions. The Advantage function can be then calculated by a function approximator with: [22] [28] [17] [46] [40]

$$\begin{aligned} a_{\pi_{\boldsymbol{\theta}}}(s, a) &= q_{\pi_{\boldsymbol{\theta}}}(s, a, \boldsymbol{\chi}) - v_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{w}) \\ &= R_{t+1} + \gamma \hat{v}\left(s', \mathbf{w}\right) - \hat{v}(s, \mathbf{w}) \end{aligned} \tag{2.34}$$

Then, if we plug it into the equations for the policy gradient and the weight adjustment, we obtain:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left(\nabla \ln \pi(a \mid s, \boldsymbol{\theta}) a_{\pi_{\boldsymbol{\theta}}}(s, a)\right) \\ \boldsymbol{\theta}^{t+1} &= \boldsymbol{\theta}^t + \alpha\left(R_{t+1} + \gamma \hat{v}\left(s', \mathbf{w}^t\right) - \hat{v}\left(s, \mathbf{w}^t\right)\right) \nabla \ln \pi\left(a \mid s, \boldsymbol{\theta}^t\right) \end{aligned} \tag{2.35}$$

This algorithm is then called *Advantage Actor-Critic* (A2C) and has overall much better performance than original Actor-Critic with a baseline. [22] [28]

Continuous Action-space

Policy Gradient methods can be used in environments with discrete as well as with continuous action space. Value-based methods have the disadvantage that they can only be used

in environments with discrete action space. Whether a given policy gradient method can be used on discrete or continuous action space is determined by the policy parametrization. In the case of a discrete action space, a numerical value $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ can be computed foreach state-action pair. The computation of these values can be done, for example, via a neural network, similar to equation 2.26. An exponential *Softmax* distribution: [42] [22]

$$\pi(a \mid s, \boldsymbol{\theta}) = \frac{\exp(h(s, a, \boldsymbol{\theta}))}{\sum_b \exp(h(s, b, \boldsymbol{\theta}))} \tag{2.36}$$

can then specify the probability with which an action should be selected. In the case of a large or even continuous action space, instead of evaluating individual actions, parameters of distribution functions such as the Normal distribution are computed. The parameters of the Normal distribution are the mean $\mu \in \mathbb{R}$ and the variance $\sigma^2 > 0$.

The Normal distribution is defined on $\mathbb{R}$, as it is a continuous action space. Mean and variance, can then be parametrized and thus approximated with: [34]

$$\pi(a \mid s, \boldsymbol{\theta}) = \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \boldsymbol{\theta}))^2}{2\sigma(s, \boldsymbol{\theta})^2}\right) \tag{2.37}$$

From the last equation it can be seen that the policy-parameter vector can be described by $\boldsymbol{\theta} = (\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma)$. Its parameters can be then computed by one neural network. This allows the implementation of continuous actions for stochastic policies. [42] [40] [20]

2.4.4 Trust-Region methods

All the policy gradient methods mentioned so far use the approach of optimizing the objective function such as an MSE, by minimizing the loss and updating the weights vector in an iterative way via *Stochastic gradient descent*. This should continuously lead to a learned, well performing model. The gradient descent optimizes via first order derivatives, this approach can be called a line search. It follows the direction of the highest gradient with fix-sized step.

The other approach, which optimizes second order derivatives is called a trust-region. Instead of going in a line step improvements, in a trusted regions it first defines a region where it is safe to be, in terms of the value of the gradient and the local optimal point within the region is found. This way it better approximates the function's optimal solution.

In supervised learning, if the SGD during an optimization steps too far in the direction of the biggest gradient, it does not matter that much, because it has lot of sample data, waiting to be evaluated. Thus, fixing the optimization solution during next steps, stepping back in the opposite direction. In RL, if the optimization „over-steps“ and misses the optimal value, poor policy is obtained. This eventually leads to a very poor results from which the RL algorithm is usually unable to recover. Considering this, the trust-region methods are in general much more stable and potentially yielding higher returns. There is a trade-off for the much higher stability property, which will be discussed later. [22] [36] [32]

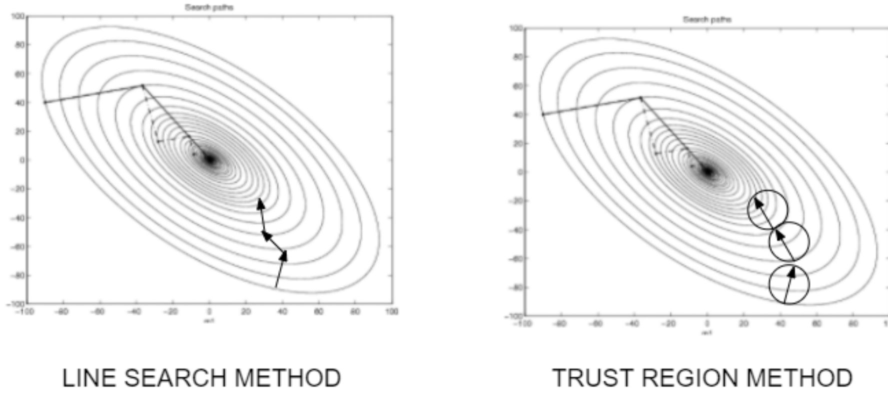

Figure 2.5: Line search vs Trust regions, Source: [6]

The general RL objective is already a pretty complex function, thus a use of some less complex function is viable. The trust-region works in a way, where it calculates first the region where it can trust the less complex function, which has similar value solutions within some region as the original objective. This way, it sets the region boundaries, where the trust-region method is looking for the next optimal gradient value. This setup then forms a constrained optimization problem, where it optimizes an objective with respect to the trust-region boundary.

The second order approximation of a function can be done, for example by a Taylor polynomials in quadratic form, then the subject of optimization would look something like:

$$f(x) \approx \tilde{f}(x) = f(\mathrm{c}) + \nabla f(c)^T (x - c) + \frac{1}{2!}(x - c)^T H(c)(x - c) \tag{2.38}$$

where the $\tilde{f}$ is the Taylor polynomial to be approximated to the f, c is a point where it approximates, and H is a Hessian matrix.

Usually, the trust-region can be a hyper-sphere, constrained as follows:

$$||x - c||_2 \leq \delta \tag{2.39}$$

where δ is the maximal boundary of the trust-region.

In order to find the optimal gradient value within the trust-region, the objective function has to have a Hessian matrix inside, consisting of second order partial derivatives. This would already be a hard problem for some small optimization problem. If we consider neural network, consisting of thousands of parameters, it is obvious that such a calculation of the second order derivatives is pretty computationally expensive.

Nevertheless, finding of the trust-region and the complete optimization process can be formalized in an Algorithm 3 as:

Algorithm 3: Generic Algorithm of Trust-region method

```
Initialize δ, x*_0, n = 0
while not converged do
    n ← n + 1
    Solve x*_n = argmin_x f(x) subject to ||x − x*_{n−1}||_2 ≤ δ
    if f̃(x*_n) ≈ f(x*_n) then
        Increase δ
    end
    else
        Decrease δ
    end
end
```

This idea of second order optimization is a well-known mathematical problem, as well as the method of trust-regions. Though, in the field of RL it is a completely new idea. The algorithm that we will discuss in a subsequent subsection, the *Trust-Region Policy Optimization* (TRPO), uses exactly this novel approach. [36] [20]

Trust-Region Policy Optimization

There are some limitations in Policy gradients that the *Trust-region policy optimization* addresses. They are data inefficient as they only use collected trajectories once and then they need to collect new ones, for later policy updates. This also induces high variance, because with bad estimates, bad policy is generated, which consequently generates bad trajectories, resulting in the whole policy not being stable. In order to achieve stable updates, the θ parameters of the policy network must be updated directly, using the previously learned policy π_{old}. Therefore new policy will not be „too far away“ with its estimates from the old policy, thus will update only within the trusted region.

This idea of policies not far from each other comes from an *Importance sampling* [42], This approach is also sometimes called the *Surrogate loss*, this is the case in the original TRPO article as well. [36]

$$\begin{aligned} E_{x\sim p}[f(x)] &= \int f(x)p(x)dx \\ &= \int f(x)\frac{p(x)}{q(x)}q(x)dx \\ &= E_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] \end{aligned} \tag{2.40}$$

If we have two distributions $p(x)$ and $q(x)$ and we want to calculate the expectation of the function $f(x)$ by following the function $f(x)$. In this case $q(x)$ is the old policy and $p(x)$ is the new policy. The new trajectory is sampled from the $p(x)$ but since due to the learning process it is noisy, so we use the old policy $q(x)$ to estimate the total reward. Then two cases are possible $\frac{p(x)}{q(x)} > 1$, meaning there is high variance, so the current policy is far from the old policy. Second, $\frac{p(x)}{q(x)} < 1$ means, there is low variance, thus updating new policy by utilizing the previously learned policy induces control of the variance. [22] [6]

The gradient of the objective function is only accurate close to the current policy. So the new policy should not be too different from the old one. This can be considered as an optimization problem within a constraint of a trusted region.

For that the *Kullback-Leibler* (KL) [34] divergence can be used. KL divergence can assign a value of a distance when comparing two distributions, i.e. it measures how much are two distributions different.

$$D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{2.41}$$

which subsequently in terms of policy distributions translates to:

$$D_{KL}\left(\pi' || \pi\right)[s] = \sum_{a \in \mathcal{A}} \pi'(a \mid s) \log \frac{\pi'(a \mid s)}{\pi(a \mid s)} \tag{2.42}$$

It also encourages exploration of the new policy. Since if the probability for action in an old policy is low, you can assign higher probability in new policy, without increasing the divergence. The objective function with KL penalty as a constrain of TRPO algorithm then can be defined as follows:

$$\begin{gathered} \max_\theta \mathbb{E}_t\left[\frac{\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}{\pi_{\theta_{\text{old}}}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)} A_{\pi_{\theta \text{ old}}}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] \\ \text{subject to } \mathbb{E}_t\left[KL\left(\pi_{\theta_{\text{old}}}(\cdot \mid \mathbf{s}) \| \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right)\right] \leq \delta \end{gathered} \tag{2.43}$$

Note, that for the calculation of KL divergence, the Hessian vector product (using conjugate gradient) of second order derivatives is necessary. To understand this topic fully, good mathematical background is necessary. Therefore, further information about the complete calculation and proof can be found in the TRPO article [36]. A drawback of this approach is the high computational complexity due to the optimization of natural gradient, i.e. gradient under constraint. For that, standard gradient descent optimization methods like *Stochastic gradient descent* cannot be used. Even though, the TRPO does not directly calculate the Hessian but rather approximates its values, it is still a very difficult algorithm to fully understand to, or even to implement it properly. The main contribution of the TRPO is the clever application of techniques from different science areas into the field of RL. [6][20][32][36]

For completeness, here in a pseudo-code the Algorithm 4 is presented in the same way as in the TRPO article:

Algorithm 4: Trust-region policy optimization

for *iteration=1,2...* **do**
  Run policy $\pi_{\theta_{old}}$ in environment for T time-steps;
  Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$;
  Compute policy gradient g;
  Use conjugate gradient to approximate $F^{-1}g$ (Hessian vector product);
  Do line search on Surrogate loss and KL constraint;
end

These drawbacks of TRPO are addressed by another trust-region method called *Proximal policy optimization*. It is also the algorithm that is used during the experiments in this work.

Proximal Policy Optimization

PPO is an Actor-critic On-policy algorithm which simplifies the TRPO, but behaves in the same way. Published in 2017 and since then it is considered as a *state-of-the-art* Policy gradient algorithm for RL.

PPO has three main goals to achieve. First is to make the code cleaner and less math dependent. Second is to reduce the computational complexity, thus making the learning process faster. Last one, is to use fewer hyperparameters, so the complexity related to finding the correct values of hyperparameters is reduced as well.

PPO consists of two different versions. The first one uses similar approach as the TRPO, it is trying to address the constrained optimization problem, so that it can be computed using standard gradient optimizers, such as an *Stochastic Gradient Descend*. It still uses the computationally expensive KL divergence, but introduces new technique, the adaptive KL-penalty. This allows to adaptively change the influence of the KL by creating a penalty for it. [38] [6] [22] [20]

Unconstrained optimization using KL-Penalty: This is the objective function to be optimized, it is almost the same as the TRPO objective, but now unconstrained:

$$\underset{\theta}{\operatorname{maximize}} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta (a_t \mid s_t)}{\pi_{\theta_{\text{old}}} (a_t \mid s_t)} \hat{A}_t - \beta \mathrm{KL} \left[\pi_{\theta_{\text{old}}} (\cdot \mid s_t), \pi_\theta (\cdot \mid s_t) \right] \right] \tag{2.44}$$

In the PPO it is called the L^{KLPEN} and it uses the adaptive β parameter to increase or decrease the influence of KL divergence.

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta (a_t \mid s_t)}{\pi_{\theta_{\text{old}}} (a_t \mid s_t)} \hat{A}_t - \beta \mathrm{KL} \left[\pi_{\theta_{\text{old}}} (\cdot \mid s_t), \pi_\theta (\cdot \mid s_t) \right] \right] \tag{2.45}$$

This way it deals with the constrained optimization from TRPO, it is a common trick in the field of optimization, and is done most of the time by using *Lagrandian Duals*. The β acts as a scaling factor and in PPO is also adaptive, since it is already a complex task to decide on an exact correct value of the β. It works as follows. If KL is too small, its boundaries can be released a little, thus β is reduced. Oppositely, if KL is too large, the penalty should be increased for the following trajectories, thus β is increased. It follows this:

$$\begin{aligned} &\text{Compute } d = \hat{\mathbb{E}}_t \left[\mathrm{KL} \left[\pi_{\theta_{\text{old}}} (\cdot \mid s_t), \pi_\theta (\cdot \mid s_t) \right] \right] \\ &\quad \text{If } d < d_{\text{targ}} / 1.5, \text{then: } \beta \leftarrow \beta / 2 \\ &\quad \text{If } d > d_{\text{targ}} \times 1.5, \text{then: } \beta \leftarrow \beta \times 2 \end{aligned} \tag{2.46}$$

where d_{targ} is the target KL distance that was initially set and the d is the current KL distance.

Clipped Surrogate Objective: The downside of the KL-Penalty approach is, that it still needs the computationally expensive calculation of the KL. To address this issue, second version of the PPO is proposed. The Surrogate objective denoted as L^{CPI} (Conservative policy improvement). Next we can call $r_t(\theta)$ the ratio between new and old policy, concept known from the Policy Gradient proof in 2.4.1, which results in:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{2.47}$$

Then we can write the L^{CPI} objective as:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right] \tag{2.48}$$

If we then plug it into the final objective function $L^{CLIP}(\theta)$ we get:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \operatorname{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{2.49}$$

What it does is, that it is clipping the policy update if the ratio between new and old policy gets too far from each other. It substitutes the function of KL divergence from the first objective, but without the computational expenses that go with it. Here it uses simple *min* and *clip* operators. It restricts the ratio not to go under $1-\epsilon$ and above $1+\epsilon$, if the two policy distributions are getting too far from each other. This clipping function is further visualised on a Figure 2.6.

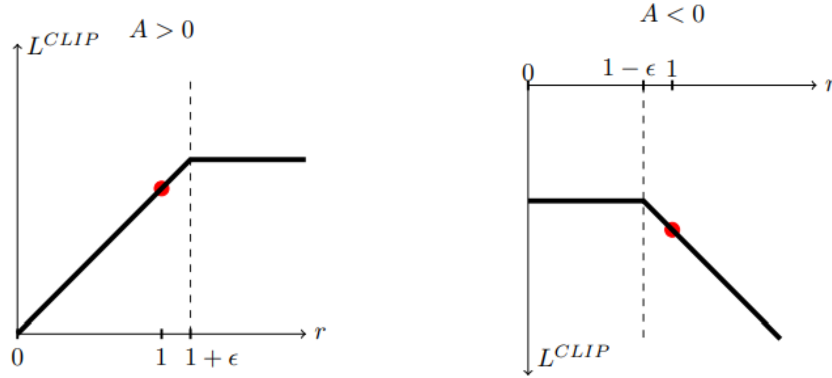

Figure 2.6: Clipping of PPO - typically $\epsilon = 0.2$, A is the advantage function, Source: [38]

Figure shows the clipping of positive advantage (left), and clipping if the advantage is negative (right). We can see that L^{CLIP} is a lower bound on L^{CPI}, with a penalty for having too large of a policy update. So it makes our policy really pessimistic, about the future update, being really conservative about updating too much. Rather than mathematical reasons like at TRPO, this PPO's objective L^{CLIP} is empirically chosen, with the focus on the most similar behavior as the KL divergence of two policy ratios (Surrogate function). It was shown that it has similar or better performance than the PPO with KL-penalty.

More intuition behind it is, that if $A > 0$, we want to update just a little bit, so that the action happens more often. Similarly, when $A < 0$ we want to decrease the possibility of that action just a bit. When we look at the right most region of the graph, the ratio r is high, so the last policy update made that action a lot more probable, but at the same time the advantage is negative, so this last policy update is worsening our policy, therefore we want to „undo“ the last update, proportionally to how bad the action was in the first place. We want to move in the opposite direction of the gradient step. This is also the only region where the original „unclipped“ $r_t(\theta)\hat{A}_t$ has a lower value than the clipped objective

and thus gets returned by the *min* operator.

Then the full objective function that will be implemented is defined as follows:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 L_t^S \left[\pi_\theta\right](s_t) \right] \quad (2.50)$$

Where the $L_t^{CLIP}(\theta)$ is our newly created clipping objective, the $L_t^{VF}(\theta)$ is the squared error loss for the critic and the $L_t^S(\theta)$ is entropy bonus that ensures sufficient exploration of the policy. Maximizing its entropy, forces the policy to wide spread across the possible actions, resulting in the most unpredictable outcome, i.e. it drives the policy to be more random at the beginning, until the other parts of the main objective take over and start dominating the resulting policy update.

This forms the final objective which is used by the second version of the algorithm. The c_1 and c_2 are hyperparameters that restrict the influence of its respective terms. In the original implementation its values are set as: $c_1 = 0.99$ and $c_2 = 0.001$, which were empirically measured for best performing results. [38] [32]

The loss function for the critic is defined as:

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{\text{targ}} \right)^2 \quad (2.51)$$

where the V_t^{targ} is the target value and $V_\theta(s_t)$ is the value function generated by the network.

The Algorithm 5 for PPO with Clipped objective can then in a simplified pseudo-code be defined as [22]:

Algorithm 5: PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ

for *k=0,1,2...* **do**

Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using advantage estimation algorithm (GAE)

Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \underset{\tau \sim \pi_k}{\mathrm{E}} \left[\sum_{t=0}^{T} \left[\min\left(r_t(\theta)\hat{A}_t^{\pi_k}, \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t^{\pi_k} \right) \right] \right]$$

end

GAE:

If we take a closer look at the advantage estimation used within PPO algorithm 5, it uses the principals of TD-λ returns. This approach is called the *Generalized Advantage Estimation* (GAE). It tries to solve the bias-variance trade-off when using an approximation of a value function, as could be seen in Actor-Critic algorithms discussed earlier. [6] [22] [37] [38]

If we take the advantage estimate from Equation 2.34:

$$\hat{A}(s,a) = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{2.52}$$

this can be considered as a 1-step advantage estimate. If we expand this to more steps, consequently we get:

$$\begin{aligned}
\hat{A}_t^{(1)} &:= \delta_t^V && = -V(s_t) + r_t + \gamma V(s_{t+1}) \\
\hat{A}_t^{(2)} &:= \delta_t^V + \gamma \delta_{t+1}^V && = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
\hat{A}_t^{(3)} &:= \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V && = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3})
\end{aligned} \tag{2.53}$$

Finally, if we further rewrite this in the context of n-step advantage estimate, we get:

$$\hat{A}_t^{(n)} := \sum_{l=0}^{n-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \tag{2.54}$$

This reduces the bias, since it less depends on the estimate of a value function, but at the same time it increases variance, as now we rely completely on the extra n-step expected return estimates summed together. For example other policy gradient algorithm, A3C uses 5-step estimate for its advantage. The GAE does not want to explicitly set an exact number of the steps, so a nice workaround is to use the technique form λ-returns and eligibility traces. The solution is then exponential average across all the steps.[37]

The equation for GAE using exponentially-weighted average is then defined as follows:

$$\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} :=& (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \ldots\right) \\
=& (1-\lambda)\left(\delta_t^V + \lambda\left(\delta_t^V + \gamma \delta_{t+1}^V\right) + \lambda^2\left(\delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V\right) + \ldots\right) \\
=& (1-\lambda)\left(\delta_t^V\left(1 + \lambda + \lambda^2 + \ldots\right) + \gamma \delta_{t+1}^V\left(\lambda + \lambda^2 + \lambda^3 + \ldots\right)\right. \\
=& \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned} \tag{2.55}$$

The series can be rewritten into a single sum. Now λ is the exponential discount factor that controls the bias-variance trade-off. Note, that if $\lambda = 0$, we are left with the TD advantage estimate (high bias, low variance) and if $\lambda = 1$, this is the equivalent of n-step estimate for the extended advantage estimation (low bias, high variance). The optimal value chosen for PPO is then $\lambda = 0.95$.

In the case of PPO, only a segment of length T time-steps of an episode is used for GAE estimation, which should be always $T << T_{episode}$. Then the advantage estimate can be written as:

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{2.56}$$

Other Policy Gradient methods

As a side note we can also mention other *state-of-the-art* algorithms and concepts used within the RL domain. The algrithms are for example A3C, DDPG and SAC. By the way, before we commited to the PPO algorithm in this thesis, we were really considering the use of SAC. As it is more efficient in terms of data samples and connects the on-policy actor-critic and off-policy approach.

Asynchronous Advantage Actor-Critic (A3C), is a policy gradient method with a special focus on the learning running in parallel. In A3C, the critics learn the value function while multiple actors are trained in parallel and are synchronized by global parameters once in a while. Thus the A3C should work well in a parallel setup. [20] [47]

Deep Deterministic Policy Gradients (DDPG), is a model-free off-policy actor-critic algorithm, combining DPG with DQN. The DQN stabilizes the learning of Q-function by experience replay buffer and the frozen target network. DDPG works in continuous space due to the actor-critic framework while a deterministic policy is being learned. For better exploration of the policy, a noise is added to it. The DDPG does also do soft updates ("conservative policy iteration") on the parameters of both actor and critic networks, in a way that the target network is constrained to change slowly whereas in DQN the target network is kept frozen for a period of time. [22] [47] [23]

Soft actor critic (SAC) - is an off-policy actor-critic model which incorporates the entropy measure of the policy into the reward to encourage exploration and ensure stability. Then the policy should act as randomly as possible while being able to get trained successfully. It uses separate policy and value function networks for the actor-critic setup. It also adds off-policy property by being able to reuse previously collected data, just like the experience replay buffer in DDPG. This supports the efficiency of the algorithm. In its objective function there are two objective one is the maximization of expected return and the second one is the maximization of entropy. [20] [47]

Great review of other Policy Gradient methods can be found at *Lilian Weng Blog* [47].

Chapter 3

Autonomous driving and system design

The aim of this chapter is to describe the various experiments that are attempting to solve the task of autonomous driving in a simulation environment TORCS. These experiments are then evaluated and analyzed in Chapter 4 with respect to achieve the objectives described in Chapter 1. The specifics of the experiments as well as the variants of it and the design of the whole system, are discussed in more detail in this chapter too. Initially, the description of autonomous driving task is presented first, following the requirements of a reinforcement learning agent and the description of simulation environment are outlined. After that, the experiment setup is introduced.

3.1 Autonomous driving

The theme context of this work is autonomous driving. This means that an RL agent is developed in order to autonomously drive a vehicle within a simulation environment. That is why it is necessary to first introduce how autonomous driving is understood by international taxonomy standards. In 2014, The Society of Automotive Engineers (SAE) [4] created a six-level classification standard of vehicle automation. This was done in order to estimate possible legal consequences for vehicles with high degree of autonomy. This classification includes not only the lateral and longitudinal guidance of a vehicle, but also other aspects of driving such as parking, pedestrian-vehicle interaction, or driving in urban traffic. All of these conditions are present only in the fifth, highest level. The lower the level the least of these conditions are fulfilled. The range begins at level zero, meaning „No automation“ and ends with level five, meaning „Full Automation“. The detailed description of the levels is listed below.

- **Level 0** - The human driver does all the driving. No assistance is provided.
- **Level 1** - An advanced driver assistance system (ADAS) on the vehicle can sometimes assist the human driver with either steering or braking/accelerating, but not both simultaneously.
- **Level 2** - An advanced driver assistance system (ADAS) on the vehicle can itself actually control both steering and braking/accelerating simultaneously under some circumstances. The human driver must continue to pay full attention ("monitor the driving environment") at all times and perform the rest of the driving task.

- **Level 3** - An automated driving system (ADS) on the vehicle can itself perform all aspects of the driving task under some circumstances. In those circumstances, the human driver must be ready to take back control at any time when the ADS requests the human driver to do so. In all other circumstances, the human driver performs the driving task.
- **Level 4** - An automated driving system (ADS) on the vehicle can itself perform all driving tasks and monitor the driving environment – essentially, do all the driving – in certain circumstances. The human need not pay attention in those circumstances.
- **Level 5** - An automated driving system (ADS) on the vehicle can do all the driving in all circumstances. The human occupants are just passengers and need never be involved in driving.

The other available taxonomy standards include *The U.S. National Highway Traffic Safety Administration* NHTSA [1], *Federal Highway Research Institute* in Germany BASt [2], and *Verband der Automobilindustrie* VDA. [19] The difference between them is visible only in the last two levels of Full automation, differing in terms of full automation during high-risk situations. In the SAE, in the fourth level, the vehicle encountering a high-risk situation is required to safely come to a stop in a secure place, typically a road shoulder, while in the fifth level the vehicle does not need any interference by the human driver and can operate fully on its own under any conditions. In comparison in the BASt classification the fifth level is missing, therefore the fourth level is considered the highest. Though, that does not mean the vehicle is driverless, the classification just does not take into account the possibility of no driver. [2] [1]

The vehicle controls which are to be operated by the RL agent in this work consist of lateral and longitudinal guidance on the road. In order to succeed the task, the RL agent will have to learn the acceleration, braking and steering control of the vehicle independently. If we want to compare it to the classifications above, it can be done to only a limited extent, as the official standards take into account all the different tasks. Of course the ideal goal would be to meat the requirements for the level five of SAE classification, at least in terms of the lateral and longitudinal guidance of a vehicle. The reasons for only focusing on the vehicle guidance are mainly because our initial goal is to prepare an agent for an RC model car application, secondly most of the freely available simulation environments only offer vehicle guidance as well. Also available literature is mostly focused on similar task. [10][16][43] This leads to the specification of the simulation environment that will be used in our experiments and will be discussed in the next subsection. [45][13][21]

3.2 Simulation environment

The reasons that the available literature focuses mostly on a simulation instead of the real-world vehicle and environment is quite simple. It is due to the reward function, which plays a crucial role in the learning process of the RL agent. The agent needs to fail a lot in order to learn some useful policy. That would mean that the real-world vehicle would encounter a lot of damage during the training process, possibly destroying itself without learning anything. More common way of doing things is to first learn the agent in a synthetic environment and later, when the RL agent is trained, just fine-tune it in the real-world environment.

If we think again about the reward function, it is much easier to manually define a reward function for the longitudinal and lateral guidance than for all the other aspects of

real driving behavior on public roads, including parking, pedestrian-vehicle interactions and the traffic signs and signals. Nonetheless these limitations do not represent a reduction in a complexity and usability of RL agents as such. After what was stated above, the racing simulators meet these requirements quite well. Also in the literature the use of racing simulators as the environment is mostly used.

3.2.1 TORCS

The one, very popular lightweight racing simulator used in the machine learning research is *The Open Racing Car Simulator* (TORCS). It is free, open-source racing game and thus will be used in this work as well. The use of TORCS is also convenient for the reasons of future application of the learned agent in real-world environment. Particularly to the scaled RC model car, which will then possibly be capable of racing on a custom track. [48] [25]

To mention some alternatives to the TORCS simulator, there is also simulator called CARLA - which simulates the real urban traffic conditions, but the requirements for computing performance are very high. The same applies for other urban traffic simulators, which in addition are not even freely available e.g. *Nvidia Drive Sim*. As next, we will describe the TORCS environment in a more detail.

TORCS is modular, discrete-time simulator which simulates the aspects of vehicle dynamics, kinematics, fluid and thermo-dynamics. It also features different driving modes. One of them is called „Practice mode" and will be used throughout the experimental part of this thesis. It is basically the solo-driving mode without any opponent. The other mode is the „Quick race" mode. Differing in the addition of opponents. These opponent vehicles are operated internally be the TORCS game mechanics. The main focus will be on the solo-drive mode, where the RL agent will learn the longitudinal and lateral guidance of the vehicle on a given track. The mode including opponents is omitted, as our goal is not to learn an agent how to compete with opponents but mainly how it succeeds in the driving task in general.[48] [26] [33]

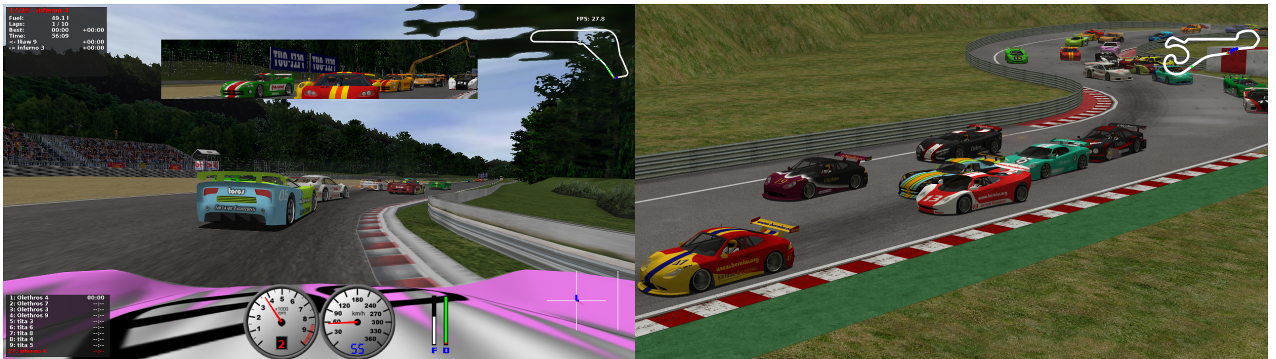

Figure 3.1: TORCS in game screenshots, Source: [48]

The TORCS environment includes about 40 race tracks, which are divided into three categories. The interesting one for us is the first category, Race Tracks. These consist of real and artificial asphalted race tracks, which correspond the most to the real-world conditions. The other category is Dirt tracks, which are mostly curvy and short in length

and simulate the off-road driving experience. The last one can be described as oval-shaped circuits, they are asphalted as well, but due to its topology, they are not interesting for our experiments. The agent would possibly quite easily learn to race such a trivial shaped track with just one type of curve. Thus the tracks selected for our experiments all belong into the first category.

There were eight tracks selected, that will be participating in the experiments, the decision takes into account the difficulty of each of the tracks, consisting of number of curves, its radius, the total length of the track and the maximal speed that the racing vehicle can achieve during the race on such a track. The tracks with its short description can be seen in a list 3.2.1 below.

- **E-road** - 3260m, width 15m - high speed track, combination of curves and straights
- **E-Track-2** - 5380m, width 12m - lot of curves with one straight
- **E-Track-4** - 7041m, width 15m - high speed long track with few curves
- **Forza** - 5784m, width 11m - very fast and smooth circuit, real-world track in Monza

The main track used for most of the training experiments is the *E-Road* and *E-Track-2* track, which have the perfect balance between its length, number and type of curves and the overall composition of these elements, helping the agent to learn a versatile driving style, that can be later benchmarked on the rest of the tracks. Tracks that were used also for testing but are not depicted on a Figure 3.2 are *E-Track 3*, *Michigan* - which is an oval shaped track, *G-Track-2* and *G-Track-3*.

The car chosen for our experiments is called within the simulator as **car7-trb1**, which is a default racing sport car. As most of the cars have similar parameters in terms of top speed, weight or aerodynamics, the selection of the type of car is not significant and therefore will not be studied in our experiments. Though, the initial idea was to use different cars throughout the learning process, thus inducing the robustness of agent's policy, making it more usable in terms of generalization, when moved into real-world conditions.

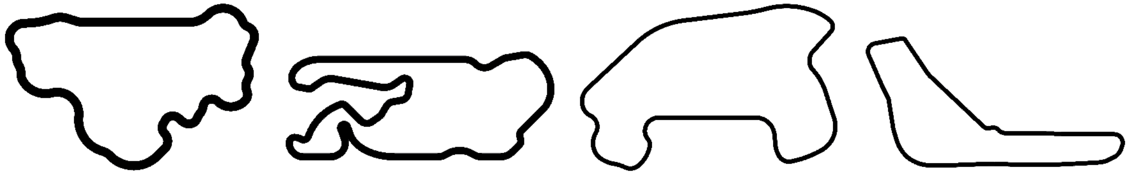

Figure 3.2: The TORCS tracks chosen for the experiments. From left to right: E-Road, E-Track 2, E-Track 4, Forza, Source: [48]

Now we should introduce the usability for experiments of the TORCS environment. Due to the fact that TORCS was first developed as a regular game, its internal architecture allows the potential RL or any other machine learning agent to access its internal information about the state of the simulation. These can be the exact position of the car, its velocity, the information about the track itself or about each opponent's vehicle. This would not only compromise the results of such experiments but would not in any way simulate the real-world conditions.

Since there used to be quite popular TORCS Championship, where machine learning researchers used to compete with its agents, there had to be created different architecture of the simulator, in order to achieve truly equal conditions for all the competing teams. This is typically done by client-server architecture, which was done also in this case by Loiacono et al.[25] The TORCS environment acts as a server and each agent then communicates with the environment as a client. The communication is based on UDP protocol. The vehicle within the TORCS environment is located on the server side and the RL agent on the client's side is sending the control commands to the vehicle controller. In return the vehicle is sending its sensory data about the perceived environment back to the client. This sensory data is being send by the server every 20ms. Then the client has 10ms to send a packet consisting of the control commands back. This strict separation of the simulation environment and the agent's implementation allows for the use of different programming language or computing architecture of the client, as the UDP interface is uniform across different platforms and languages. [48] [25] [24] [10]

The client-server architecture can be seen on a Figure 3.3 below.

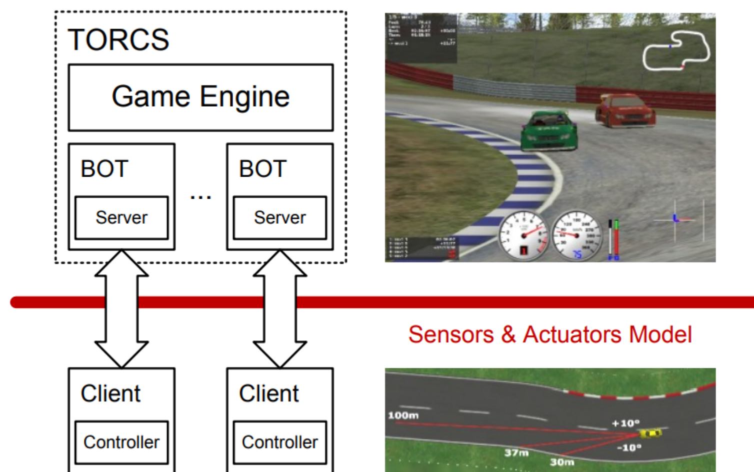

Figure 3.3: TORCS Server architecture from Championship manual [25]

One such a client available online is the *SnakeOil* client, which is a lightweight program written in Python by C. Edwards [9]. It allows the UDP communication with the TORCS server environment and through it a control and basic settings of the simulation can be set. It first opens a UDP socket and then the server side is started. Even though this abstraction can be seen as enough, there is another layer available between the client and the RL agent, which will also be used in this work. It is the *GymTorcs* [8] environment, which allows us to use the standardized *OpenAI Gym* environment, which is used all over machine learning research. This allows us to further focus just on the implementation of the RL agent and do not bother with the complications related to the setup of the TORCS environment and all its caveats related to the communication.

There were of course many adjustments done to the *SnakeOil* client and also to the *GymTorcs* framework to function correctly and in alignment with our needs for many specific experiments. One of them is the functionality of the camera and its use with other sensors, which is not officially supported by the TORCS. Other can be the different way of sending the sensory data to the client due to different learning approaches used or the

simulation of the communication failures or the sensor/actuator noisy data, that mimic the real-life conditions related to one of the objectives of this work, that is train and prepare the RL agent in a simulated environment with the focus to simulate multiple possible conditions, so the agent could be without huge modifications deployed into the real-world scenario.

OpenAI Gym and PyTorch

If we step back and focus on the *OpenAI*, in 2016 the company released an open-source Python library called *Gym*. It unifies the research simulation environments for RL agents. It had the goal of simplifying and unifying the agent's interaction, as well as it allowed broader public to use and test the same environments as the research community. Most importantly, it brought to the field some sort of standardization for the benchmarking of RL agents and the structure of simulation environments in general. This allows the researches to compare different RL agents on similar tasks, without worrying about the environment's implementation differences. An alternative to the *OpenAI Gym* can be the locomotion simulation environment *Mujoco*, which features different n-legged robots and humanoids. Coincidentally in 2021 the OpenAI bought the *Mujoco* simulator and according to the company, it plans to make it freely available as part of the *OpenAI Gym* library. [31]

Some well-known tasks in Gym can be mentioned, such as the Cart-Pole environment by Sutton and Barto, Mountain-Car, Lunar Lander or Atari Games. If we recall the *Gym-Torcs* environment, it is not natively included in the Gym, but similar environment had been created, which follows the basic structure of Gym environment, which can be seen in Algorithm 6 below. The implementation of *GymTorcs* is located in `gym_torcs.py` file within the thesis implementation source files. It starts the *SnakeOil* client, which opens a connection to the TORCS server. The *GymTorcs* interaction with the environment is then standard, just like interaction with any other Gym environment and its game's implementation.

As had been mentioned earlier, for the implementation of our RL agent - algorithm PPO, the use of function approximators will be in the form of neural networks. Since our *SnakeOil* and *GymTorcs* subprograms are implemented in Python, our RL agent will be implemented in this language as well. Therefore there is a need for search of a specialized Deep Learning library available also for this language. The available options are *Keras, TensorFlow, Theano, Caffee, PyTorch* and surely many others. The selection came to the last one mentioned, the PyTorch library. Released by AI Research at Facebook in 2016. The reasons behind the selection of PyTorch are mainly, some previous personal experiences with this framework, also it is the only machine learning framework that we had been presented during our study at FIT. According to the GitHub statistics, the currently most used machine learning library for that purpose. Also in the research community, PyTorch is by many considered as the current standard go-to library for the field of machine learning research and business sphere. For example the car manufacturer Tesla uses for its FSD

Autopilot software PyTorch as well.

Algorithm 6: OpenAI Gym standard environment pseudo-code

```
Initialize an environment by gym.make()
for episode=1,2...N do
    Observe initial state from env.reset()
    while True do
        Select action from policy
        Get next state, reward, done from env.step( action )
        if done then
            break
        end
        Assign to current state the next state
    end
end
```

[31]

As we can see from the pseudo-code above the most important function of the environment is the `env.step()`, which takes the agents action as an argument, makes a step within the environment and then returns the next observed state, reward related with this transition and boolean value whether the episode has ended or not. The rest of the code is a simple cycle, iterating over the number of episodes and while loop within an episode, representing the individual time-steps. The next section focuses on the next state values, which are in our TORCS environment represented by the sensory values observed by the vehicle within the simulation environment. The next subject discussed are the possible action values represented by vehicle's build-in actuators. The section 3.2.3 then writes about the reward function and the related topic of reward shaping. [8]

3.2.2 Sensors and Actuators

The information about the current state of the environment is in the form of sensory data, captured by vehicle's sensors within the TORCS server. Therefore it is the only way for the agent to know what is in the simulation happening around him. For the task of lateral and longitudinal guidance it is necessary to consider the type and amount of sensors which will be used and thus would be enough for the agent to learn well-performing policy. It is actually a frequent subject of research in autonomous driving. It also depends on the level of autonomy targeted. [35] [26] [33]

One of the well performing sensor is the LIDAR. Based on the reflection intensity of a laser beam it scans the environment and after post-processing the point-cloud can be created, representing the detailed 3D scan of the environment. One of the downsides of this sensor is its extremely high cost. Another very helpful sensor is radar, it operates on similar principles as LIDAR, but with sound waves. Its advantage is, that it is in comparison with the LIDAR much cheaper and smaller sensor. Last of the important sensors is the camera, this sensor is probably the cheapest one and thus, heavily used in the industry. Also the advances in supervised learning, enabled camera sensors to get relatively smart and can be used to describe the surroundings in a form of complex vector spaces. For example the Tesla company says, that the future of self-driving cars is in the solitary use of camera sensors. Also they are strictly against LIDAR technology, as due to its high cost, such vehicles using

it will never be mass produced. [44]

According to research from University of Michigan [35], which compared the LIDAR, radar, camera sensor technology for different tasks of autonomous driving, e.g. object recognition and classification, it says that in an effort of achieving Level 5 of SAE classification for autonomous driving, all mentioned sensors have to work in combination with vehicle-to-vehicle communication, and only then they would be able to succeed in all required tasks.

The RL is an end-to-end approach for autonomous driving, so no further division into layers for specific tasks, as can be seen in other approaches. The RL agent receives all sensory data necessary and then through *trial and error*, continuously learns a policy. Thus, implicitly learning the effect of lane lines during driving, the presence of other vehicles, detection of obstacles, etc.

For our use in this thesis the selection of sensors had been considered in regards with the reality. For example the `distFromStart`, representing the distance measured from the start line to the vehicle position on the track, is not the most realistic type of a sensor that can be found in real-world autonomous vehicles. With this in mind, in the Table 3.1 below are showed all the sensors that are available in the TORCS environment. The ones highlighted are sensors that were selected for the use in our experiments. One important information about the `vision sensor`, representing the front camera input. As this is not an official sensor in terms of the TORCS environment [25], the simulator does not allow usage of other sensors, when `vision` sensor is enabled. This limitation had been successfully bypassed in the code, so in the experiments we were able to experiment with different approaches in terms of selected sensors, including the combination of camera and regular sensors, as well as the comparison of solo regular sensors or solo camera output.

We also investigated the minimal required set of regular sensors, with which the agent was still able to learn a successful policy. We were also experimenting with different camera output, so that the agent would not only have the information about its position, position of the lane lines but also have some information about its motion and velocity, which is crucial when learning only from camera sensory data. But this will be in more detail reviewed further in this thesis.

To further divide selected sensors, the ones describing the state of the vehicle can be `speed` sensors, `RPM` and `WheelSpinVel`. The ones describing the state of the environment can be considered as `track` sensor, `opponent` sensor or `focus`. The `track` and `focus` sensor can be then considered as a low-resolution LIDAR sensors. The position of the vehicle via GPS is not used in this work as the TORCS architecture does not provide this type of information. We should also mention, that some of these sensors will be used for creation of the reward function, which will be discussed later in this chapter.

Name	Range	Description
angle	[-$\pi, \pi](rad)$	Angle between the car direction and the direction of the track axis.
curLapTime	[0, $\infty](s)$	Time elapsed during current lap.
damage	[0, $\infty](pts)$	Current damage of the car (the higher is the value, the higher is the damage).
distFromStart	[0, $\infty](m)$	Distance of the car from the start line along the track line.
distRaced	[0, $\infty](m)$	Distance covered by the car from the beginning of the race.
focus	[0, 200] (m)	Vector with 5 values, each representing the distance from the vehicle to the lane boundary within 200 m. Unlike the track sensor, the focus sensor observes a range of only 5 degrees. However, the agent can once a second ask for specific field of view, thus it is the subject of learning.
fuel	[0, $\infty](l)$	Current fuel level.
gear	{-1,0,1..6}	Current gear: -1 for reverse, 0 for neutral, and 1 to 6 for regular gear.
lastLapTime	[0, $\infty](s)$	Time to complete the last lap.
opponents	[0, 200] (m)	Vector with 36 values, each representing the distance to the nearest opponent, within 200 m. The sensor covers the entire area around the vehicle.
racePos	{1,2,..N}	Position in the race w.r.t. other vehicles.
rpm	[0,$\infty](rpm)$	Number of rotation per minute by the vehicle engine.
speedX	(-$\infty, \infty)(km/h)$	Speed of the vehicle along the longitudinal axis of the vehicle.
speedY	(-$\infty, \infty)(km/h)$	Speed of the vehicle along the lateral axis of the vehicle.
speedZ	(-$\infty, \infty)(km/h)$	Speed of the vehicle along the Z axis of the vehicle.
track	[0, 200] (m)	Vector of 19 values, each representing the distance from the vehicle to the lane boundary, within 200 m. An are of 180 degrees in front of the vehicle is scanned with a resolution of 10 degrees.
trackPos	(-$\infty, \infty)$	Distance between the vehicle position and the center of the lane. Zero if vehicle is in the center -1 is the left lane line, 1 the right lane line.
wheelSpinVel	[0, $\infty](rad/s)$	Vector with 4 values, each representing the radial speed of a wheel.
z	[-$\infty, \infty](m)$	Distance of the car mass center from the surface of the track along the Z axis.
vision	(0, 255)	Tensor of dimension 64x64x3, representing RGB pixel input from a camera. NOTE that vision can be enabled only by forbidding sensory data from every other sensor mentioned.

Table 3.1: Table with sensors available in the TORCS environment [25]

In the Figure 3.4 below it can be seen the vehicle's perception of the environment and its position via some of the sensors, especially the `angle, track and focus` sensor. This is the set of sensors with which the experiments were initially started and then empirically were lowered to a minimal possible functioning set.

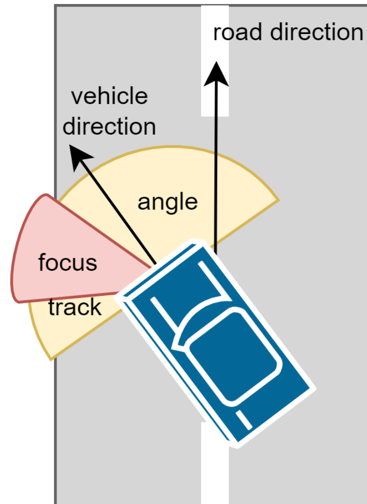

Figure 3.4: Visualisation of the officially available „vision" sensors in the TORCS environment

So as for the agent to learn autonomous driving, the lateral and longitudinal guidance has to be controlled via vehicle's actuators. In the table 3.2 below we can find information about all actuators provided by the TORCS environment. Again, the ones highlighted, are the ones that were chosen for the application during our experimental part of this thesis. The remaining sensors, if not controlled by the agent are then automatically controlled by the TORCS. The reasons for not using the clutch and gearing is quite simple. Modern cars, especially those that would in the future be fully autonomous are usually already right now electric, therefore no need for manual transmission. The next reason is in regards to this thesis, that is the RC model for which we are trying to learn our agent, is using regular electric DC motor, thus it would make no sense to train our agent also with the clutch and gearing actuators enabled. This should not make this task any more trivial, it is still a hard problem to solve, as all the actuators used have continuous action space, i.e. combination of their possible values is almost infinite.

Name	Range	Description
accel	[0,1]	Virtual gas pedal (0 means no gas, 1 full gas).
brake	[0,1]	Virtual brake pedal (0 means no brake, 1 full brake).
clutch	[0,1]	Virtual clutch pedal (0 means no clutch, 1 full clutch).
gear	-1,0,1,..,6	Gear value.
steering	[-1,1]	Steering value: -1 and +1 means respectively full right and left, which corresponds to an angle of 0.366519 rad.
focus	[-90,90]	Focus direction in degrees.
meta	0,1	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

Table 3.2: Table of possible actuator values, highlighted ones were selected for the experiments

3.2.3 Reward shaping

Reward shaping is an important discipline within the field of RL. It is not unusual that the environment does not include explicitly defined reward function, thus it is up to the researcher to carefully design it on its own. This is the case with TORCS environment as well. From the Algorithm 6 describing the Gym environment procedure, it can be seen that the `env.step()` takes an action as an argument and in return returns the new state of the environment and the reward obtained from this transition. The new state represents a vector of sensor values, meaning each element of a vector is from a single sensor. Thus it suggests, that we could map some specific sensory values directly to the reward function. For example the position of a vehicle to stay within the road. So the `trackPos` sensor giving us the info how far from the lane line the car is, should stay within defined boundaries. Once exceeding it, the agent obtains a negative reward. In a similar way, the other sensors can be mapped with the reward.

It is important to mention that the reward shaping is an intensive research topic, as the relation with a reward obtained is directly influencing the performance of the agents policy. If a reward is too high, then the agent will next time most likely use the same action in a given state. If the reward is too low, then the agent will probably choose a different action next time visiting the state. This is typical across all RL algorithms, thus it is very important to define a good reward function.

In this section we will propose three reward function designs plus so-called terminal conditions for an agent, which determine the end of an episode. For example when the agent's vehicle crashes, or goes in a wrong direction, this action will lead to a high negative reward, thus will terminate the agent's episode. These terminal conditions are defined in a Table 3.3 below. All reward functions defined in this section will then be empirically evaluated, measured in terms of performance metrics and comprehensively discussed.

If an agent learns a good policy and none of the terminal conditions occur, then the termination of an episode will happen after 1600 time-steps or less, depending on the type of experiment. These values were empirically measured. A well trained agent, for most of the tracks, was able to drive 3 continuous laps before an end of the episode.

Event	Reward	Terminal
collision	-100	yes
out of track	-70	yes
stopped moving	-1	no
driving backward	-80	yes
progress too small	-10	no

Table 3.3: The proposed terminal conditions and their penalty values, where some of them also end an episode.

During the early experiments, it was found that in order for the agent to explore the environment as much as possible, especially during the initial episodes, it was better that not all the terminal conditions mentioned should lead to an end of the episode. Otherwise the results were quite poor and the agent's episode was on average only a few time-steps long. That meant the agent only knew the beginning of the track, thus was prone to overfitting on that part. Later when it discovered further parts of the track, it was unable to overcome such an overfitting, which resulted in a poor policy, incapable of further learning.

Now we will propose the first continuous reward function used during our experiments, this one is based on Wang et. al. [45] and [10] It is defined as:

$$\mathcal{R}_1 = \text{SpeedX} \times \cos(\varphi) - \text{SpeedX} \times \sin(\varphi) - \text{SpeedX} \times \text{TrackPos} \tag{3.1}$$

It is defined by considering the maximization of longitudinal velocity SpeedX $\times$ $cos(\varphi)$, minimization of lateral velocity SpeedX $\times$ $sin(\varphi)$ and to maintain the agent in the center of the track. Thus any movement in the lateral direction is then considered as undesirable and has a negative effect on the reward. This is also supported by the third term of the reward function, the `TrackPos` sensor. The first two terms of the equation can be easily derived by using trigonometry, where the direction of the vehicle and the direction of the road are two sides of a right-angled triangle. This is also the most used reward function during our experiments. [24] [17] [33]

$$\begin{aligned} \cos(\varphi) &= \frac{\text{Longitude}}{\text{SpeedX}} \\ \text{Longitude} &= \cos(\varphi)\ \text{SpeedX} \\ \\ \text{and subsequently:} & \\ \sin(\varphi) &= \frac{\text{Latitude}}{\text{SpeedX}} \\ \text{Latitude} &= \sin(\varphi)\ \text{SpeedX} \end{aligned} \tag{3.2}$$

The second reward function mentioned is from [13]. The value of the `TrackPos` sensor is used as well, but this time it is subtracted from the angle $cos(\varphi)$ and then the result is weighed by the `SpeedX` sensor value.

$$\mathcal{R}_2 = \text{SpeedX} \times (\cos(\varphi) - \text{TrackPos}) \tag{3.3}$$

There is also third reward function proposed, but it is only used in experiments comparing the overall capability of the agent to learn a successful policy with each reward functions. That is, measuring the effect of the reward function on the learning process of an agent. It is defined as follows: [13]

$$\mathcal{R}_3 = \text{SpeedX}\left(\cos\varphi - \frac{1}{1+e^{-4(|\text{TrackPos}|-0.5r_w)}}\right) \tag{3.4}$$

It is a smooth reward, penalizing lateral distance with a sigmoid function. Where the r_w term is the road width. [13]

It is obvious that these reward functions have direct influence on the lateral and longitudinal guidance of a vehicle, but the terminal conditions specified in Table 3.3 are of the same importance. Especially the amount of penalty with which the agent is „awarded". If any of them is too low, the agent would not stop doing such actions, contrary if the value is too high, the agent would not learn a good policy, as its overall reward would be high negative value, suggesting the whole episode was bad. Also the termination of an episode of each terminal condition had to be empirically studied, so the agent would explore the environment enough and was not cut off too early. This happened especially with the „`progress too small`" penalty.

3.2.4 Performance metrics

When we take a look at the RL algorithm's learning process, it is supposed that the policy continuously improves over a period of several episodes, with respect to the return value. It is expected that during the first phase of the learning process, the policy won't be good and the agent will have problems with controls of the vehicle and would mostly only explore the environment. Later, after enough exploration is done, the agent continuously adjusts its internal parameters, shifting into the exploitation phase. During that period it improves its policy, lowering the entropy for its actions, thus making noticeable progress. This process continues until it possibly reaches a high return yielding policy, in terms of the driving capabilities of the agent.

In order to find a suitable performance metrics for checking the quality of an RL agent, learning the task of autonomous driving, or more specifically the lateral and longitudinal guidance of a vehicle, literature was studied for the most common ones. Here are the three most common metrics listed: [24] [17] [33]

- Total return achieved by the agent controlled vehicle per episode.
- Distance reached by the agent controlled vehicle per episode.
- Average speed of the agent controlled vehicle per episode.

In the literature also the performance of a human driver is sometimes considered and compared with the result of the RL agent. This comparison might be interesting to see, but due to the settings of the experiments it is not really the main objective of investigation.

It is easy to deduce that these metrics will improve over the time, as the agent's policy gets better and better, and explores the state space more and more. Depending on the setup of specific experiments, the training period will be between 600 to 3000 episodes in length. These were empirically measured and mostly manually stopped after reviewing the results, as due to the behavior specifics of the RL optimization it would not be correct to use approaches from supervised learning in the form of early-stopping. The early-stopping method is used to finish the training before the agent seems to be over-fitted, so typically

when some metric, usually accuracy reaches a specific level. The RL optimization is usually oscillating within a range of values, so no strict limit for a metric should be used, as the same performance level is usually reached later again.

In order to be able to evaluate the metrics properly, another condition has to be met. We need to propose a type of reference frame within which we can compare two learned agents. Two agents can have the same total return, but this does not mean that they behave the same or that they are of the same quality. One can be able to successfully and repeatedly drive to the half of the track in a high speed and without any collision, whereas the other one can for example for the first time finish a whole lap, but in a slow speed or with many accidents. For these reasons the agent will have the necessary condition to at least once finish a complete lap of the racing track. This would then serve as a reference frame for further interpretation of an agent's learned and measured performance.

One other metric suitable for the PPO algorithm can also be the entropy of its policy. A policy has a maximal entropy when all policies are equally likely and minimal when one of the action's probability of the policy is dominant. So entropy metric is used as an index of „healthy“ training, where the entropy should be continuously getting towards lower values. When the entropy is getting higher values, we can deduce that the learning is not advancing at all, but is actually worsening, suggesting we should stop the experiment and discard its results. When entropy converges to a certain value and does not improve further, we can consider the training as finished, as the agent would most probably not improve any more.

Also for the agent to be confidently considered as learned, it also means that the agent is able to correctly generalize.

The agent's learning will be performed completely on a single track, mainly *E-road* and *E-Track 2*. For the testing, the performance will be measured on a different, for the agent yet unknown track. If it is able to correctly generalize, it should immediately get high returns for the unknown track. In other case, it means that the agent should be kept learning for a longer period of time or that it was over-fitted. This problem of generalization of RL agents is also an active area of research. Some techniques from supervised learning can be used, such as batch normalization or usage of dropout layers within the neural networks. The performance metrics used for the testing, will be the same as those used during the learning period.

Luckily, in most of the experiments the agent was not over-fitted for one track, thus the use of techniques preventing from over-fitting were not necessary.

3.2.5 The Algorithm

The algorithm primarily used in this thesis is the *Proximal Policy Optimization* (PPO) 2.4.4. Since its policy has an output in continuous action space domain, it is well suited for the vehicle's actuators, that are defined over real-valued intervals. We will use them in its original unchanged form, as it was proposed in the paper. [25] The PPO version with the clipped objective will be used, in order to avoid calculations of the KL-divergence, which requires lot of performance for its calculation and thus it is not in our interest to use it with a relatively low performing hardware, which is also representing the vehicle's on-board integrated microprocessor. [38]

The algorithm will use multiple neural networks, depending on the type of experiment. When dealing with only sensory data, there will be two networks design used, one representing the policy and outputting the means and variances for actions and the second one used for outputting the state-value for a given state. When the camera output will be used

there will be as well a separate convolutional network extracting features from a vehicle's view image. These features will be then concatenated with the other sensory data and would be fed into the two regular networks for state-value and policy.

One other variant will be also tried, that is to collect a pairs camera output - sensory data for a given state and in this way create a dataset. The agent for such collection will be an already trained one, ensuring that the dataset would consist of samples from a complete race track. Then a separate convolutional network would be trained as a regular neural network in supervised learning, outputting sensory data for a given camera image. This trained network would then serve as a „simulator" of regular sensors and therefore would be used as a substitute of them. This sensory output would then be an input into the regular two networks mentioned in the beginning of this paragraph. In this way, if training is successful, we could get rid off completely of the sensors, thus relying solely on the camera output.

That would be especially useful for the RC model, where there would not be a need for other real sensors to be installed, but the camera. Although it would be a real surprise if this last mentioned setup would work. Mainly because a learned agent would not struggle in the initial state of learning, so no weird turns, slow speeds or unpredictable behavior would be contained within the dataset. Maybe if we plug this learned convolutional network into an already enough trained agent, trained by only sensory data, then this problem might not be present. So the complete agent would be firstly trained on sensory data, which would then be replaced by „camera output to sensory data" network, and then fine-tuned. This approach would then might work.

3.3 Implementation details

In this section we will discuss the implementation details and provide more information about the technical aspects of experiments and what is used for the collection of experiment's results and how they will be compared. We will also dive more into the neural networks architecture and the whole data pipeline from the environment to results.

3.3.1 Algorithm implementation

As a first thing we have tried to implement the PPO algorithm all by our-self. These efforts were not long after abandoned. It was not possible to implement the algorithm in order to work even in the slightest form. After that, we have encountered several implementations available online and from those, three good ones were chosen. After an extensive experimentation where some of them did not even work at a basic tasks from *OpenAI Gym*, so the simplest one working was chosen. That could now be declared as our baseline implementation. Surely, it did not meet our requirements in terms of quality of results at basic *OpenAI Gym* environment tasks, nor the „readiness" for our experiments with TORCS environment, but at least it worked - meaning the agent was getting better during the learning sessions. Such an implementation was then hugely changed and improved.

One of the main changes was the support for not equally long episodes, which does not seem as much, but it is crucial for our task. As the agent while driving can crash at the first corner and the waiting till the end of an episode would be meaningless, but also would waste our computational resources. The only requirement for the length of an episode is, that it has to be bigger than our batch size. Which for most of the experiments was empirically

discovered and set to 32. It is due to the style of training of the neural network. This will be also elaborated more later in the text.

The next quite important improvement was the possibility to use both numerical values from sensors and also the images from vehicle's camera, both separately and at the same time. Not forgetting to mention the possibility of using the convolutional neural network for generation of sensory data.

Also the possibility of pre-training or fine-tuning a model from previous runs. The implementation is fully customizable through a configuration file, where all the hyper-parameters of PPO algorithm, the neural networks, as well as the type of experiment, architectures, including the settings of TORCS environment can be easily set before each experiment. Also from each run multiple metrics and information about the run are saved, including the source code at that stage of time and models of an agent at different phases of the training session. This is done by an open-source platform called *MLflow* [27]. Which can then also be used for comparison of individual runs, based on hyper-parameters or specific metrics and can be viewed through a web browser.

MLflow

The *MLFlow* platform runs on a localhost, where it creates a server which serves as the machine learning tool for gathering experiment's data as well as the tool for management and comparison of the different experiments. The *MLflow* also enables the user to watch all the statistics and metrics of an experiment run in real-time, as it plots new data every 10 seconds. This is especially useful when some of the runs are going poorly in terms of performance, so the user is able to easily see and analyze the data and eventually finish the experiment prematurely. This saves a lot of time during the experimental phase of the project. *MLflow* also allows for addition of description of each experiment, so the user can comment with specific insights that he encountered during the run. This then helps during the evaluation part, as remembering hundreds of runs is not in a human capabilities.

The *MLflow* offers three main methods that take care about the collection of data from an experiment.

`mlflow.log_param()` logs a single key-value param in the currently active run
`mlflow.log_metric()` logs a single key-value metric
`mlflow.log_artifact()` logs a local file or directory as an artifact

The names of the methods are quite a self-explanatory, but in short we can mention their use in this work. With the `mlflow.log_param()`, we were collecting typically before the experiment had started, all the hyper-parameters or other special values, that had an importance and were fixed-values, totalling around 36 values. With the `mlflow.log_metric()`, we were collecting all the changeable values, such as the metrics: rewards, returns, average speed, distance raced, entropy, neural network loss and more. Totally around 22 metrics were tracked during each experiment. Lastly with the `mlflow.log_artifact()` method, we were collecting the source code of each run, this was especially helpful during the initial testing of the functionality of the algorithm and the whole experiment pipeline. Also the configuration file was saved this way, because not all parameters were saved directly. Also the neural network's models were periodically saved by it.

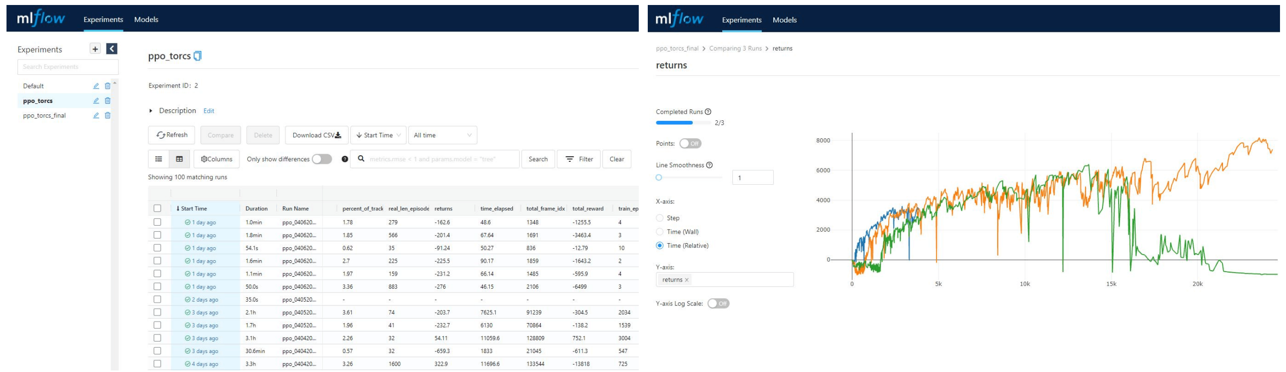

Figure 3.5: *Mlflow* - screenshot of the web application user interface/ Left: overview of available finished and running experiments. Right: detail of a comparison of multiple experiments on a return metric

Lastly, within the *Mlflow UI*, the graphs can be easily viewed and compared across the runs and more importantly, these can be then saved in a form of an image, thus removing the need of usage of third-party graph plotting library, e.g. *matplotlib* for Python. But if needed, all the collected metrics are saved in an `csv` file, thus it can be used for further processing or serve as an input into some other plotting tool or custom code.

3.3.2 Neural network architectures

In the subsection 3.2.5, we have already briefly introduced the neural network setup, which we will further discuss here.

As it was mentioned in subsection 2.3.4, modern Deep RL algorithm uses as the function approximator a neural network. Here we should introduce the architectures of our networks used in this thesis. The PPO algorithm is an actor-critic method, where each of the components needs a neural network, the actor for outputting the mean and variance for a Normal distribution of agent's actions and the critic for outputting the estimated state-value function value.

Regular Architecture (sensors)

In the chapter discussing experiments we will briefly show results of experiments targeting the depth and number of neurons within these two networks. When working architecture was found, it was kept unmodified for the rest of the experiments. Such an architecture is visualized on a Figure 3.6. This is also the setup where the input vector is only consisting of sensory data, more specifically in this case the *track* sensor and *speedX*, the speed in forward direction. In the experiments this architecture will be refereed to as the *Regular Architecture* setup. The networks have two hidden fully-connected layers, these are simple networks, which use as an activation the *ReLU*, only difference is the last layer of Actor network, where in order to ensure correct interval of the actions, the *hyperbolic-tangent* activation is used. The intentions behind the use of such a small networks is mainly the speed of processing and the fact that these network work only over numerical values, few inputs from sensors and only two outputs for actions, the architecture does not have to be anything more complicated. Empirically the 512 neurons within the hidden layers were working great with low processing times, so there was no need for changing it to even bigger architectures. On the contrary, smaller networks (64, 128 and 256 neurons in hidden layers)

worked sometimes even better than bigger ones. The other options that were tried was just 1 hidden layer. It was shown that these sizes are not enough for the agent to learn a good policy.

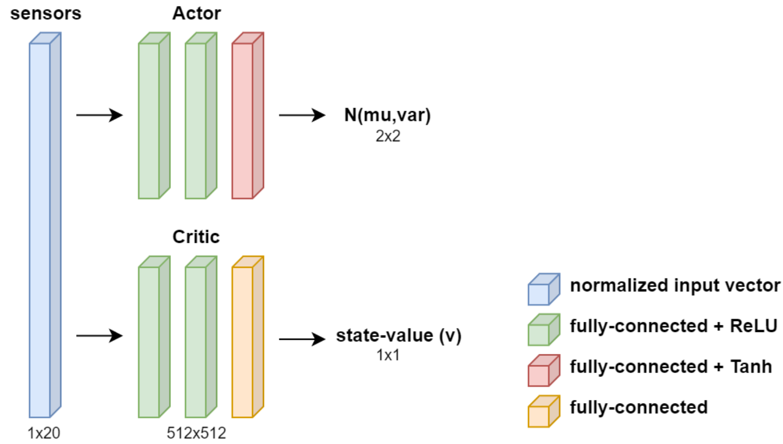

Figure 3.6: 5 The architecture of the Actor-Critic network, which uses sensory data as an input.

ConvNet Architecture (sensors + camera)

As a next architecture, we will also use the camera output, that is, we will need some convolutional layers inside our network topology. This architecture will be later refereed as *ConvNet Architecture*. It consists of convolutional layers, as well as pooling layers, for decreasing the image resolution. As an output from the convolutional network will be 1D features vector that will be concatenated with regular sensory data and then would be fed into the Actor-Critic networks. The architecture of these networks will be the same as in case of the *Regular Architecture* setup. In a more detail the architecture can be seen on a Figure 3.7.

There was as well experimented with different number and types of layers, but these experiments were not as comprehensive as in the case of Actor-Critic networks. It is because this thesis is mainly about Reinforcement Learning and the PPO state-of-the-art algorithm and its performance and possible applications, and not that much about convolutional neural networks and finding of its perfectly working topology. There are other papers focused solely on that topic. We experimented with 7 possible CNN architectures, but when a functioning one was found, it was kept for the rest of the experiments.

This is also supported by the fact, that the limitations of the TORCS environment, allow for the input image to be maximally 64x64x3, which is a relatively small size in comparison with modern CNN's capabilities, which can operate on a much higher input image resolutions. Where the amount of data calls for a careful and more sophisticated design of the network's architecture. The presented CNN architecture takes the input image and continuously lowers the image resolution, so that it can learn specific features within the image. It goes from 3 input channels, representing the RGB channels into the final 24 channels, where each of them should learn a specific detail, such as the lane lines, and the position of the vehicle itself. Then these feature maps are squeezed into fully-connected layers, which should dense the knowledge from an image, resulting in an

feature vector output, that is then concatenated with the normalized sensor vector from the TORCS environment. This resulting vector is then fed into the Actor-Critic networks.

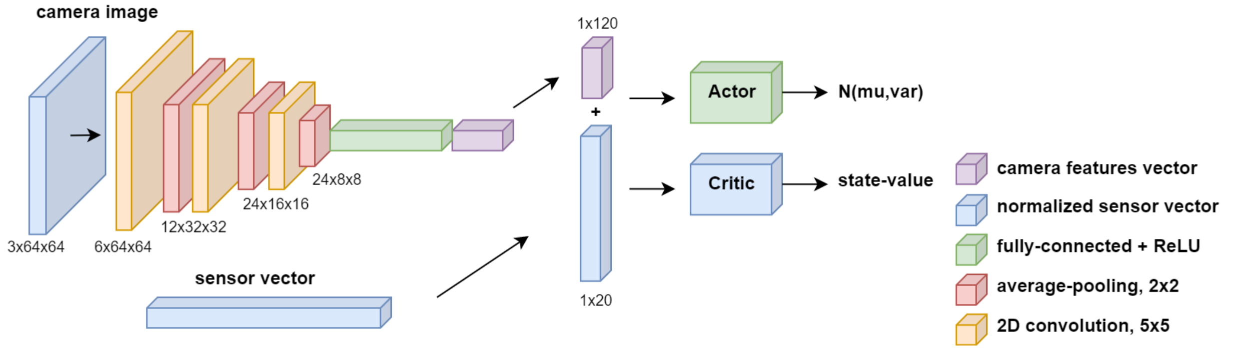

Figure 3.7: Scheme of the CNN features vector creation, concatenated with sensor vector, which is then fed to the Actor-Critic network architecture

The version of architecture, where only camera output is used, that is without the sensory data from the TORCS was also tried, but this approach seemed as a dead end. It was not possible to train such a network enough, so that the agent would have any significant progress. After few experiments, this approach was abandoned and was substituted by this combined architecture, where sensors and camera image work together as an input for the PPO agent.

Input image

During the experiments only with camera sensor, few adjustments were tried, in hope of achieving some better results. The idea behind these adjustments was the agent's inability to recognize a vehicle's motion from a single image. That is, same camera output can be produced by a vehicle driving at high speed, but also by a vehicle being completely stopped and without movement. The lack of such information in the agent's input should play a great role in its resulting poor performance. For that matter, three different adjustments of the input image for *ConvNet* networks were proposed. These ideas are based on a paper [18].

First one, simply concatenates a camera output from a previous state, that is, we are saving previous state image into a buffer. This image is then concatenated to the current one. This way the *ConvNet* gets the necessary information about the motion of a vehicle.

Although, the network gets twice as much data as in the case of a single image input, these data are mostly the same. Meaning, great part of these two images is the same, as the vehicle does not travel great distance between two states, so that the image difference would be great as well. This leads to some level of redundancy in the network's input.

The other approach tries to deal with this issue, but not all the way. We propose an absolute difference image input. That is, image from current state is subtracted from the previous state's image. This subtraction is then put into absolute value, so every pixel is bounded in the interval $\langle 0; 255 \rangle$ (before normalization). This approach reduces the redundancy of the data, but also removes great portion of the information that can be squeezed out of it by the *ConvNet* network. It also does not change the input dimension, as the input stays at $64 \times 64 \times 3$, as a regular camera output image. [14]

To address this, third and last adjustment is proposed, and can be easily deduced by the two already mentioned adjustments. That is, merging together those two ideas. To a standard camera output, we concatenate an absolute difference image (current state - previous state). This way the network gets the standard data from the regular image, but also the information about vehicle's motion from the absolute difference image. Although the input resolution grows in size ($64 \times 128 \times 3$) by this, the amount of data in the absolute difference image is mostly none. As most of the same image features are canceled out, and only the difference caused by the vehicle's motion is kept.

With these three proposed adjustments was experimented as well, and the results will be discussed in more depth in the next chapter.

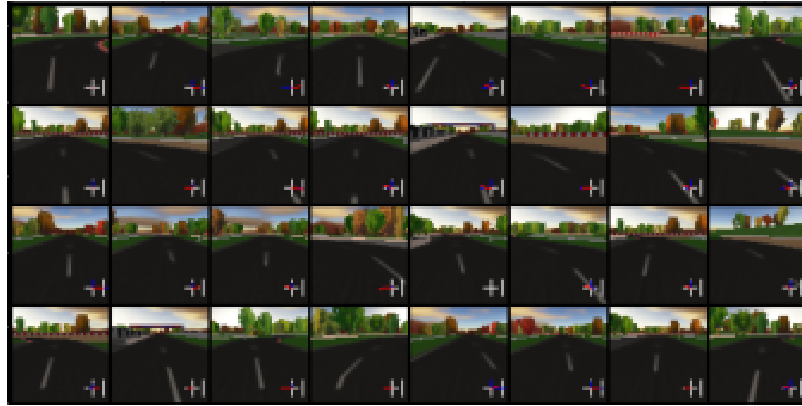

Figure 3.8: Example of the regular camera images of dimension 64x64x3 pixels

If we get back to the different network architectures, the last one that will be mentioned here, tries to focus on the problem of the agent's inability to solely rely on the camera image data as well, but from a different perspective than the manipulation of the input image. The architecture and setup is a bit more complicated that the two previous architectures, but has the most promising results in terms of real-world application.

Hybrid architecture (camera to sensors)

The problem that we are trying to solve by this setup is, that the sensory data, mainly the *track* sensor is just too specific and too much relies on the TORCS environment. The goal of this setup is, that only the camera output will be enough for the agent to learn a good policy. The real camera sensors are cheap and easy to operate in the real-world environment, whereas specific sensors will be not only more expensive, but also quite hard to install in the same manner as they are in the TORCS. Meaning the 19-values long *track* sensor as the agent's low-res LIDAR/radar sensor, in the field of view of -45 deg to 45 deg, is much more complicated to recreate on the RC model. Whereas install one camera sensor to a correct position on the car is much easier.

If we consider the facts, that the agent could not learn just from the image data, but was quite easily learned from sensory data, then the idea of merging these two approaches arises. This *Hybrid architecture*, as we will be calling it, is trying to achieve just that. We will now introduce this architecture's setup. Few diagrams will be needed for better understanding of the attempted setup.

Figure 3.9 shows how a regular agent is learned through only sensory data. From the dimensions of objects coming from TORCS environment, we can easily deduce, that 3x64x64 object is the camera output and the 1x20 object is the sensor vector, both depicted in a blue color. The camera output is not used in this learning setup. The process was already described in *Regular Architecture*, so we just summarize it quickly here. TORCS environment outputs the state, consisting from sensory and camera data, the sensory data are normalized and are fed into the Actor-Critic networks, which produce state-value estimation and mean and variance for agent's actions. These trajectory data go after each episode to the PPO algorithm which in an iterative process continuously learns an agent. It does it in a way that it samples actions which maximize the agent's expected returns. Every single sampled action goes back to the TORCS environment, resulting in the agent's vehicle moving. When the agent's policy converges to a certain state, we consider our agent as learned.

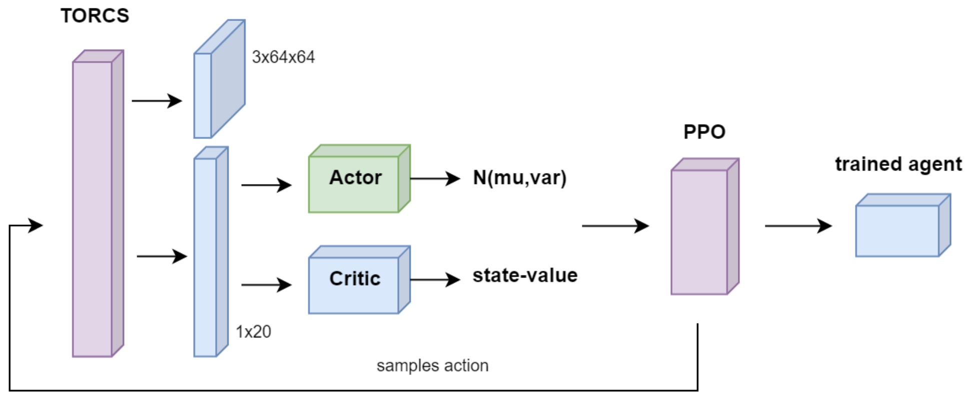

Figure 3.9: Phase one - the learning process of the agent, using only the sensory data. Complete standard interaction loop, used in big portion of the experiments

This learned agent is then used in a different setup, where we collect a dataset - sensory vector and camera output pairs. Since the agent is learned, it should have no problem of achieving great distances on the track, resulting in a few complete laps driven on the track. This way we collect a dataset consisting uniformly of different parts of the track. If we have used an unlearned agent, or we collected the pairs during training, we would most likely have a big amount of samples from the first few meters of the track and only a small percentage would be from later parts of the track. Which is not optimal. This approach was tried and empirically was found that the agent could learn how to drive in the first small part of the track, but was unable to understand the rest of the track. The mentioned approach of using already learned agent fixes the problem.

When a dataset of satisfactory length is collected, we can start a second phase of this *Hybrid architecture* setup. A separate convolutional neural network will be trained in a supervised-learning manner. We will feed the camera output into the CNN and as a ground truth will serve the sensory data from the pairs, collected before. The CNN will produce prediction of the very same sensory data. This is quite a standard task for the supervised learning. As the loss function was used the Mean Square Error metric, Adam as the optimizer and learning rate of 0.001. This setup is also pretty standard as well. This whole supervised learning process is depicted on a Figure 3.10 below. It includes also the

dataset collection from the trained agent and as an output of the process the trained CNN model is showed.

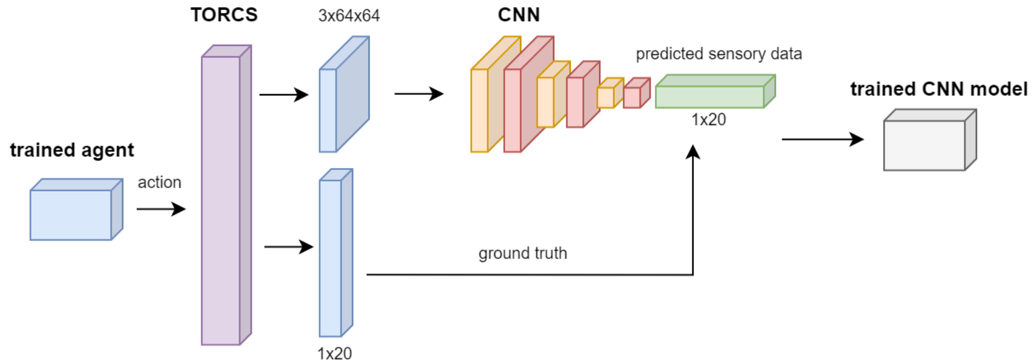

Figure 3.10: Phase two - the dataset collection and CNN learning process

Here on a next Figure 3.11 the details of the CNN used is shown. It is the same network architecture as in the *ConvNet architecture* setup. The only difference is the dimension of the output vector, where in the *ConvNet* setup, the network outputs a features vector of 120 floating-point numbers. Here, the vector consists of 20 normalized floating point numbers, representing the sensory values. One other architecture was tried as well, but the performance was quite similar. So this Figure serves rather for illustration purposes, as the architecture of the network for supervised-learning is not that important in the context of this thesis. That is also why the description of this phase is not as detailed as one might anticipate.

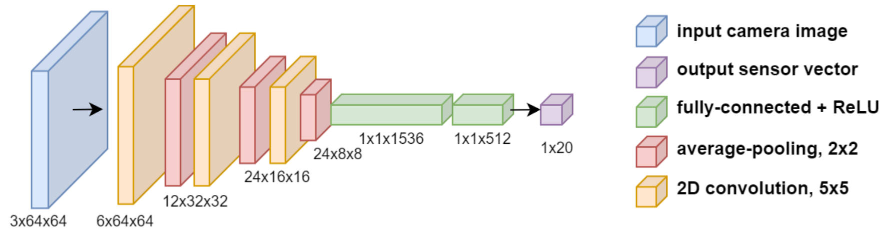

Figure 3.11: Detail of the CNN architecture for the supervised-learning task (prediction of sensors from a camera image)

In the last phase of this *Hybrid architecture*, we will use the learned CNN model and the learned agent. They will work together in the process of autonomous driving task within TORCS environment. The setup is now quite easy to understand. The TORCS environment provides a state as usual, consisting of camera output and sensory data. The sensory data are not used - which is also our goal in this approach. The camera output is fed into the trained CNN model, which should output the predicted sensory values. It has the same dimensions as the trained agent needs and was initially trained on. The agent takes this predicted sensory data and it should output correct action, in order to achieve high returns. This action is then again fed into the TORCS. This also finishes the interaction loop. The Figure 3.12 shows the last phase complete loop of the processing.

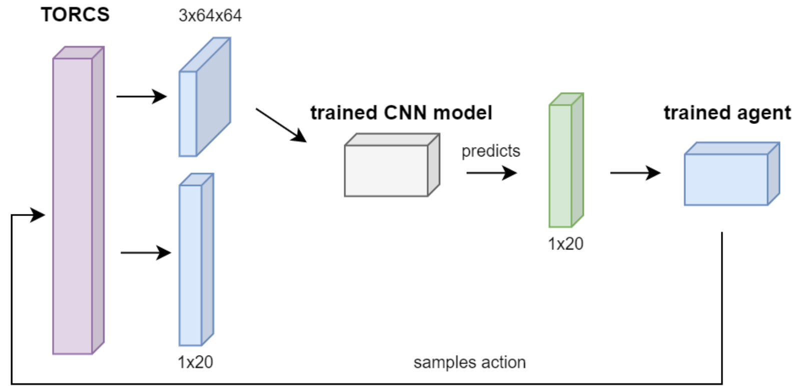

Figure 3.12: Phase 3 - complete loop of trained CNN model and trained agent model

This agent can potentially be trained for a short period of time, in order to get used to the new sensory data - as they will not be 100% the same as the sensory data outputted from the TORCS environment itself. We can also help the agent with this. There are two approaches that were tried in order to successfully transfer the agent from real sensors to the predicted ones by the CNN. First one, the agent during the first phase, would be learned on a sensory data, which were artificially changed. The use of *Gaussian noise* can be mentioned as an example. The other approach is, that the image output, during the dataset collection phase could be altered as well. The use of *Gaussian blur* can be used. These two approaches should ensure, that both the CNN model and the agent should be able to better generalize. Thus both models won't be affected as much by slight differences in the training and validation data. The validation data in this case the sensory data generated by the setup depicted on the Figure 3.12. This also represents the final agent, ready for evaluation, experiments and for testing its ability to correctly generalize on yet unknown race tracks.

With this we finish the description of neural network architectures used throughout the thesis. Last thing remaining for the introduction is the *Cloud architecture*, which is more likely an abstract term and describes the simulation of communication and artificial noise applied to the sensory and action data.

3.3.3 Cloud architecture

The *Cloud architecture* will be part of the experiments, where our main goal is to simulate real-world conditions, where the learning is happening on separate hardware from where the vehicle control hardware is located. Precisely, the RC model only receives the commands for its control of movement, but the learning and processing happens on the server, or in the cloud. These two entities then communicate wirelessly via the UDP network protocol.

These real-world conditions then consist of network packet loss, communication delay, and noisy sensory and action data. In this approach, no additional network architectures are introduced; the only difference is in the data preparation. The data then serve as the input for our agent (state), or input for the TORCS environment (action). Although three approaches are mentioned, only two of them were actually done. The one that was not is the communication delay.

Unfortunately, the TORCS environment works with discrete time and the agent has a 20ms window for the control commands to be sent. During that window, as the time of

simulation is discrete, the environment is not acting but is waiting for the agent's control commands. When the agent misses the window, next discrete step is performed. Thus any delay greater than the length of a window acts as a sort of packet loss, which we do differently, and any delay smaller than the window, results in TORCS environment not noticing any delay at all. This is why the communication delay was not implemented and therefore was not experimented with. Instead it belongs to the category of packet loss, which we have done.

Network packet loss

To simulate the network packet loss, the standard Normal distribution is used, which represents the natural randomness of such an event occurring. It can be arbitrarily set, but for most of the experiments this value was set to 5% of the distribution's value interval. So in other words if the random value sampled from Gaussian distribution with mean 0 and variance 1 is less than lower 5-percentile boundary, the event is triggered. In this case the packet is lost. The lower 5-percentile boundary is in standard Normal distribution at around -1.64. As it has been described above, the TORCS has the timeout for agent's response, as it is discrete time simulation, we „zero" the data within the packet. This data then carry no information, but for the continuity of the optimization they are still being send to the other side. This is done in both directions. From an „agent to TORCS" in packet carrying action values, and for direction „TORCS to agent" in packet containing sensory data - the perceived state. In the case of multiple sensors or camera output, the complete data information is zeroed. [39]

Sensory noise

In the case of sensory noise, almost the same applies as in the packet loss case. Although there are notable differences. Again the standard Normal distribution is used for the event occurrence. The range is the same, 5% of the distribution's value interval. When this is satisfied, then there is a 50% probability for each of the sensors to be affected by the noise. This also applies to the camera output, which is considered as a single sensor. Then, if any sensor has to be affected by the noise, a noise value is sampled from a Normal distribution with a 0 mean and a 0.1 variance. Since all the sensory data are normalized, the maximal noise value can be up to 10% of the sensor's maximal resolution value. In the case of the camera image, for each of the pixels, a different noise value is generated and added.[15] We also did experiments with the 10% probability, to see and compare the differences in influence on the agent's performance.

This sums up the implementation details, or more precisely the details about the experimental setup that was created and focused on.

Chapter 4

Experiments and results

In this chapter we will first mention the hardware that was used for the experiments, next we will briefly remind the different subjects of experiments. This will be followed by experiments itself and its results will be presented.

4.1 Computational Hardware

All experiments had been done on a laptop machine. The operating system used is a Linux Ubuntu 18.04 distribution, working on a Linux kernel 4.15.0-175-generic. The machine runs on *Intel Core i5-8250U* which is a 4 core, 8 threads processor with peak frequency at 3.4GHz and is constructed by 14nm technology. Released in 2017. Its performance is for: Integer Math: 21,201 MOps/sec and for Floating Point Math: 13,018 MOps/sec [12]. Although the laptop contains a graphics card as well, the *NVIDIA GeForce MX150* has in total 384 CUDA Cores, but empirically was shown that the GPU performance is lower than using the processor itself. CUDA Cores are specifically designed for the use in Machine Learning, the hardware is specialized in matrix operations, dot product and cross product mainly. The *PyTorch* library is CUDA-ready, meaning that in a matter if one line of code, the programmer can move any variable into the GPU memory, where it can be further processed by the GPU CUDA cores itself. Since the experiments regarding the performance and speed of matrix calculations showed that the CPU is faster, for the whole experimental phase, the main processing unit was the CPU. [30]

Due to the TORCS simulator limitations, it was not possible to train the agent on a cloud computing service or elsewhere in the cluster. This limitation played its role, especially during the experiments involving camera output. We tried to tackle the obstructions immediately from the start by not designing huge CNN architectures that required a lot of processing power. We focused on smaller architectures, so we were able to train on such a low-performing hardware, that without a doubt, a 5-year old laptop truly is. Due to the great number of different experiments, we made the decision that no experiment should be longer than 14 hours. The median length is around 7 hours for CNN related experiments and around 4 hours for regular network experiments.

4.2 Subject of experiments

In the Chapter 3, we have already mentioned multiple subjects of experiments and how they would specifically look like. In this section we will briefly remind them.

As a first set of experiments we focused on the setting of the PPO algorithm hyperparameters such as the learn rate, mini-batch size, optimal length of an agent's episode, number of update iterations and so on.

Next we mentioned the differences in performance in multiple runs of the same experiment and subsequently the issue that we have encountered during experiments containing camera output and which are directly related to the TORCS implementation limitations.

Then we have discussed the importance of a good initialization of the agent and the eventual instability of the optimization process, especially related to the experiments with *ConvNet architecture*.

Next we compared the different sizes of neural networks used for the actor-critic and how the affect the agent's performance. Then we shortly discussed the two most used network architectures.

For the next set of experiments we reviewed the reward functions and how they affect an agent's driving style.

Then we discovered the minimal set of vehicle's sensors that are necessary in order to be able to successfully train an agent, and also discussed the effect of different sensors on the agent's performance.

As next experiment, we used the *Cloud architecture* in order to simulate the real world conditions such as the packet loss and noisy sensory data, which can occur during the wireless communication between the vehicle and processing hardware in the cloud. These experiments should also show us the stability of PPO and improve the generalization properties of the agent.

Next we have showed an example of perfectly trained agent and presented how such results look like, and also presented an opposite example of the agent that was unable to learn any successful policy. The inability is then related to experiments with agent relying only on camera output or camera output and some additional sensors.

Then we showed results of multiple agents regarding the generalization properties and we talked about the correct time to stop the optimization to prevent overfitting.

As a last set of experiments, we focused on the *Hybrid architecture*, which is an approach that we have proposed in this work and which reacts to the reality, that the agent is not able to learn from only raw camera output. Instead it is able to generate its artificial sensory data which then uses for as the description of the state within the TORCS environment.

4.3 Experiments

In this section we will introduce the results of all the experiments that had been done, we also discuss the issues that occurred during the execution of experiments and we provide deeper understanding why we chose different methods and why we tested various entities more extensively than others. We first begin with the hyperparameter related experiments.

For every experiment the four performance metrics will be depicted. The main metric is the returns value, which is sum of the expected return for a given state across whole episode. The rewards is sum of the rewards across whole episode and average speed and distance raced are obvious. At all four metrics, higher the value, the better.

Little sidenote to the plots showed throughout this section. Usually the values on an x-axis mean number of agent's steps within the environment. On the y-axis, for the returns and rewards these values are unit-less, in case of average speed it is kilometers per hour and for distance raced it is meters. At some of the plots, the x-axis is different, and it is

the relative time - so it also captures the differences in the time elapsed for each episode. Later episodes tend to be longer in time than initial ones and this scale allows us to see it.

4.3.1 Hyperparameters

As the example of experiments related to the optimal PPO hyperparameters finding process, the learning rate related results will be showed. Then we shortly discuss the other hyperparameters such as PPO Update, batch size, episode length and we finish with standard deviation.

Learn rate

Based on the experiments, the learning rate chosen and used for the rest of the experiments is 0.0001. Although the learning process takes more time than for the other higher learning rates, we can clearly see that with a lower learning rate the optimization is much more stable, without unexpected drops in performance and with steady continuous improvements. It also makes more sense, as RL algorithms in general are susceptible to „over-step" the ideal policy value, and also with a really small step a bit further than is optimal in the policy search space, it can result in a completely differently acting policy, which is most of the times much worse. Although PPO's objective function tries to counteract this, by „undoing" the previously made bad decisions, but it is not enough in the case of higher learning rates. This can be also seen on the plots, where we can see a sudden drops in performances for learning rates 0.0003 (orange) and 0.0005 (green).

Contrary, the learning rate 0.0002 acts during most of the optimization process pretty poorly, suggesting the policy is „over-stepping" too, but continuously, thus not achieving good results. This behavior can be also explained by bad initialization of the networks weights or other initialization variables. This will be showed on a next experiment, how initialization is often quite important for the agent's learning process. We will see multiple runs of the same experiment setup, most of the time with similar results, but some of them will perform very poorly.

From the entropy graph, we can see that the highest learning rate has also the steepest slope, so the policy quickly converges to a certain certainty of its actions, but that includes the unpredictable behavior in terms of performance. Empirically, at entropy around -1.6 is the agent's best performance for a given track, but we usually stopped around -0.3, as it has showed that best performance on a training track implies a subtle overfitting of the agent, thus performing poorly on other tracks. The entropy decrease of learning rate 0.0001 is the slowest, but continuous and mostly we can be pretty sure, that there are not going to be any significant drops in performance during the training.

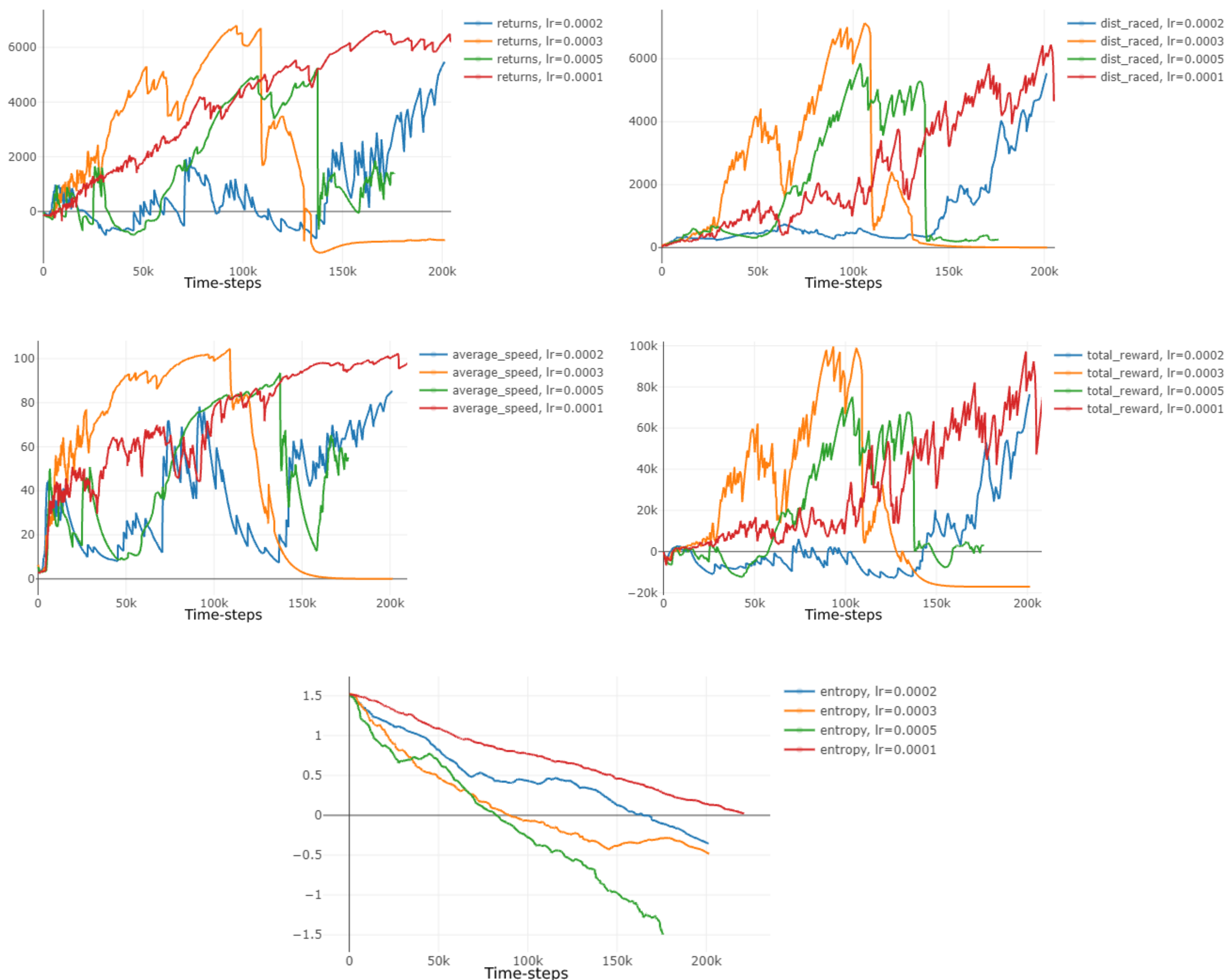

Figure 4.1: Experiment 1 - Finding of optimal learning rate value

If we focus on other hyperparameters regarding PPO algorithm or the optimization setup, we also experimented with few others. Mostly the hyperparameters related purely to PPO were set to the same values as they were mentioned in the PPO article [38].

PPO Update

For example the PPO Updates is a parameter that specifies how many updates of the actor-critic networks are performed at one PPO iteration. Higher values than 20 caused the agent to almost no training at all, whereas lower values caused the optimization to be too slow. The 20 updates per iteration was a sweet spot. The agent is still able to train well, but the training is not that time-consuming. One other reason is, that the PPO is an online algorithm, meaning it is directly learning from the agent's current episode trajectory. So in general it is not very sample efficient algorithm. But we can at least help it in this way and increase the number of PPO Updates. In the paper the value is set from 3 to 15 updates,

depending on the specific learning task. Although we run the agent within a simulation, we still should not „waste“ agent's episode trajectories too extensively.

Batch size

Next focus of the experiments was the mini-batch size. In the paper, they use between 64 to 4096 samples within a single batch. Due to the implementation of our PPO algorithm and due to the nature of the TORCS environment, we were limited in terms of the variable mini-batch size. The size 32 was empirically chosen, based on the speed of the update but also because of the necessary minimal length of an episode. Too small and the update would take too much time, too high and the minimal length of the episode would have to be increased as well. If we imagine that the agent crashes after few time-steps, especially during the initial learning, the episode could not have been terminated and the agent would have to stay in the simulation until the time-steps reach the minimal episode length, while being crashed and not doing anything useful. Then the mini-batch samples would be full of useless data, where the agent does not move, resulting in almost no learning progress, or even degradation of achieved policy performance.

Episode length

Last hyperparameter we focused on, is the maximal episode length. Again, empirically was set to 1600 time-steps. The reasoning behind this value is, that the agent is able to explore the environment for longer, which is really helpful during the first quarter of the learning period. That is the phase where the agent is able drive, but is driving at a low speed. The higher episode length then supports the exploration of the environment, so the agent can get further on the track, thus discovering new, yet unknown profile of the track (new curves, straights, etc). Experiments were done with 350, 768 and 1280 time-steps, which were all too low. Contrary, the 2000 was already too high, as the optimization took too much time, and the issue of sparse reward was taking an effect. If we consider that at most of the tracks, the 350 time-steps were for a pretty well trained agent enough to circle one full lap. On average 1600 time-steps allow the trained agent to drive 3 full laps. The agent usually reaches the maximal episode length right from the start and then at last third of the optimization period, when it is already trained quite well, but is polishing its skills. E.g. tries higher speeds, perfects the turning into the curves, etc, as a result increasing its total distance raced and lowering the lap time.

Standard deviation

To also support the exploration of the environment the standard deviation of the action's probability distribution was set to 0.1, so the agent tries little different actions each time when it appears at the same state. This shows also in the evaluation test of the agent, that no two runs are the same. This is different than with the deterministic RL algorithms, which always use the same action, every time they appear at a given state.

In the table below we can find listed hyperparameters with its values, which, once were found, were used throughout the rest of the experiments 4.1.

Critic Discount	0.5	**Learn Rate**	0.0001
Entropy Beta	0.001	**Mini-Batch Size**	32
Epsilon	0.2	**PPO Updates**	20
GAE Lambda	0.95	**Episode Steps**	1600
Gamma	0.99	**Std. dev. action**	0.1

Table 4.1: Table of default PPO hyper-parameter values

4.3.2 Differences in performance

The next experiment is just for illustration. It is not exactly an experiment, but it is worth mentioning, that the role of good initialization is sometimes crucial for the agent. On a Figure 4.2 we can see 7 runs of the identical experiment. Although the differences are not that great, they can still be seen. Especially the run `nn4`, which has the worst results. Considering that at the same stage of optimization two different runs with same initial conditions can differ in more than 4km in driven distance, or one can have almost double the return, it is significant. So during the experiments it was to us to decide whether each run should be kept running or if it was better to terminate it prematurely. Also the spread of all the runs is quite large.

On other note, it can be noticed certain similarities between some of the graphs. Especially between the distance raced and the total reward. From that we can assume that main part of the agent's high reward is the distance that the agent traveled. On the other hand, if we compare the average speed and returns achieved by the agent, they are also similar. This behavior repeats in other experiments as well. Here it is just nicely shown, on multiple runs at the same time. We can deduce that the overall reward is dependent on the distance, whereas the quality of every state is mostly based on the vehicle's current velocity in that state.

Little side note, just for reference we have showed also the plot of agent's lap time. The times are not in simulation time, but in real-time. They consist also of the time taken for the machine to run the code, but the time is without the PPO update, only the episode simulation. As the computation time is the same for every run, and the version of the code was the same, we can state that the measured time and the lap time, can be interchanged and used for rough measurement of each agent's performance.

These runs were based on the *ConvNet* architecture, so that the agent uses the sensors, in this case `speed, track` and the camera output, which are together concatenated into one feature vector which is the fed into the actor-critic network. The camera output is the version with one single image, without further adjustments. As we can see, although the results are not that amazing, they can still be considered as good.

ConvNet architecture issue

There were huge amount of experiments done in the similar manner, meaning with the *ConvNet* architecture. The unfortunate is, that all these experiments could not be used for this thesis. There was found an issue in the late stage of experiments, regarding the camera output. The TORCS simulator does not include the camera sensor as the „official“ sensor, based on the TORCS Manual [25]. Although the implementation consisted of such a sensor, its usage was very limited. We made it possible to use it in cooperation with the official sensors, as opposing the base TORCS implementation, where the user could choose either he wants to use sensors or the camera sensor.

The unfortunate is, that in order to speed up the experiments, we increased the TORCS simulation speed, from 4x to 100x. We continuously tried higher speeds 8, 16, 32 and then we have reached the 100x speed-up. Everything seemed normal so we continued in all the experiments. Although it had been tried and tested before in low speeds, that the camera output works well. The images from it were multiple times saved and reviewed. But this all happened at low simulation speeds. The problem was, that once the simulation speed reached e.g. 8x the normal speed, the simulation window with the vehicle's view froze. During that time it was assumed that the laptop, where the experiments were done, simply does not have the performance for the simulation to show continuous simulation picture and also that the 100x speed is just too fast for the TORCS to visualize it properly. Plus all the sensors were working properly and the agents during experiments were learning successfully. In fact the camera was outputting only the picture that could be seen in the simulation window. So when the window froze, as well the camera output for the agent's convolutional networks froze. Meaning, the network was getting as the input during the whole simulation period the same image. So the agent was basically learning just from the working sensors and the camera data were only useless data, that it had received every time-step throughout each episode and subsequently each experiment which used the camera output.

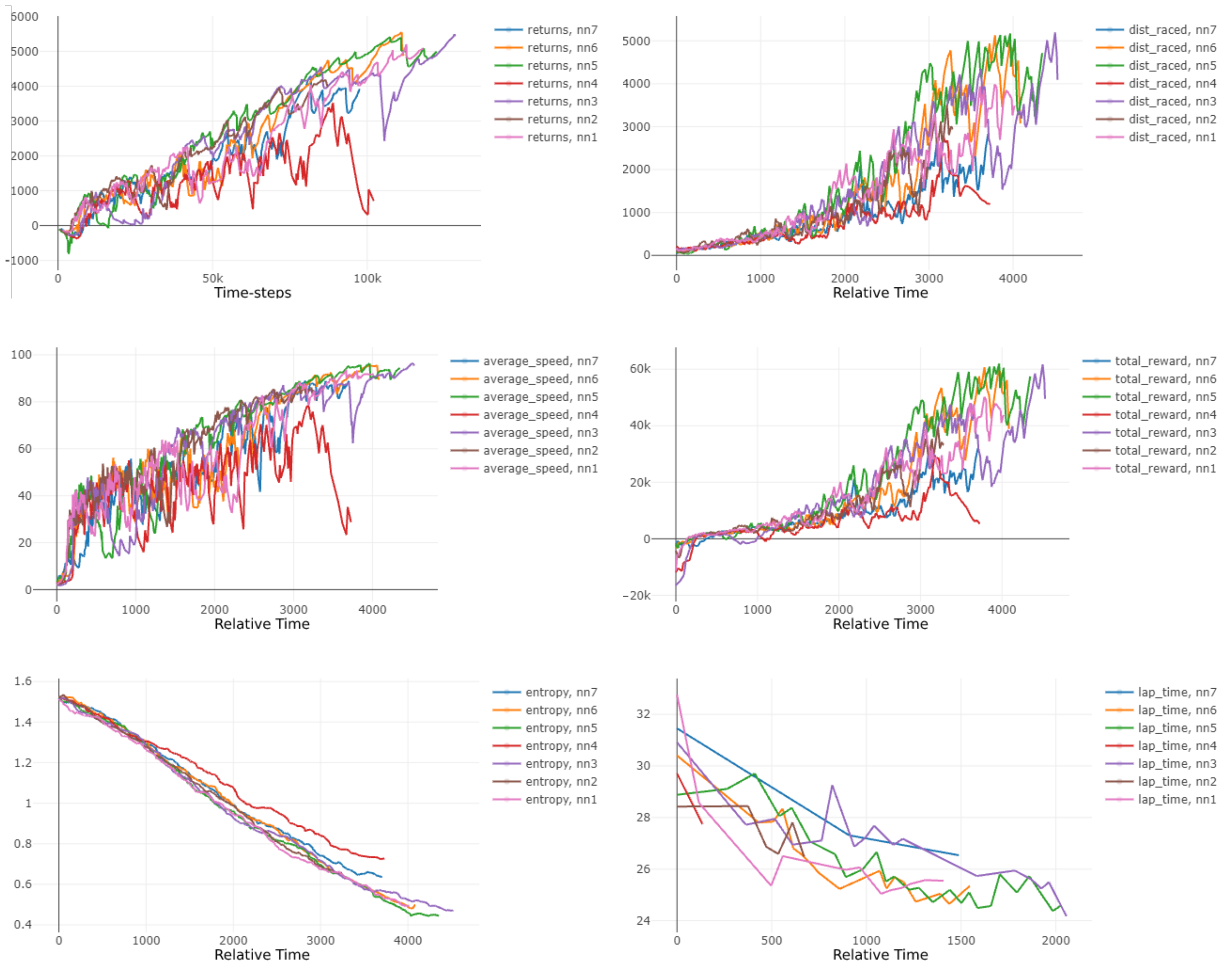

Figure 4.2: Experiment 2 - Initialization, 7 runs of the same experiment (with ConvNet architecture issue)

This was found during the preparation of the last architecture's experiments, where we tried to learn separate convolutional neural network to predict sensor values from the camera image. The architecture is referred as *Hybrid architecture* in the theory section 3.3.2. As we have trained the network from a collected dataset on the Google Colab, for faster learning, the dataset was inspected. Only then we were able to recognize this issue, so good amount of the previous experiments had to be scrapped, including the experiments showed in this section on Figure 4.2, which is discussing the initialization and differences in same setup experiment runs.

It is important to note, that although the PPO had most of the input data with zero information value, it was still able to learn the agent quite well. We can consider the camera data as a great noise experiment, considering that the features vector which was in these experiments created by the convolutional neural network, was of length 120 floats and the useful sensory data were only 20 floats. That is 85% of the input data for the actor-critic

network. This also shows the power of the PPO algorithm, that is able to learn from such a noisy input. Although experiments, which were done in regards to the input/output noise conditions but were done on purpose, are discussed later, when we experiment with the *Cloud architecture*.

4.3.3 Initialization and Instability

This experiment continues with the *ConvNet* models, discussed above, but this time we use the camera image adjustment in a form of absolute image difference. The Figure 4.3 then shows the impact of a really bad initialization. It is even more severe than in the last mentioned experiment. Here we can clearly see, that sometimes the agent just picks up a poor behavior right from the start, possibly caused by the poor initialization and poor selection of actions in the initial phase of the learning. For the comparison the blue and red runs, which behave quite well, had no real struggle to continuously obtain better and better policy, even though the red run had the *Cloud architecture* - which we can consider as a type of additional obstacles for the agent.

Which is the simulation of packet loss and sensory noise both for sensors and for the camera image. So technically it should learn slower and possibly achieve poorer policy in general, though with better generalization properties. The orange and green run, both show what bad initialization does to the agent's performance, but in a slightly different way. The green run just started with poor policy but eventually started to get better and if the optimization lasted longer, it might recovered completely. Whereas the orange run, just picked up poor policy right from the start, with no clear signs of improvement. It appears that it achieved some local optima and was unable to recover from it. In these cases it is better for us to prematurely terminate such a run and start over. Visually, such runs most of the time behave in a way that right from the start of an episode agents start turning and go immediately to the area outside of the road and crash to the barrier or do not move at all, usually use only the break actuator.

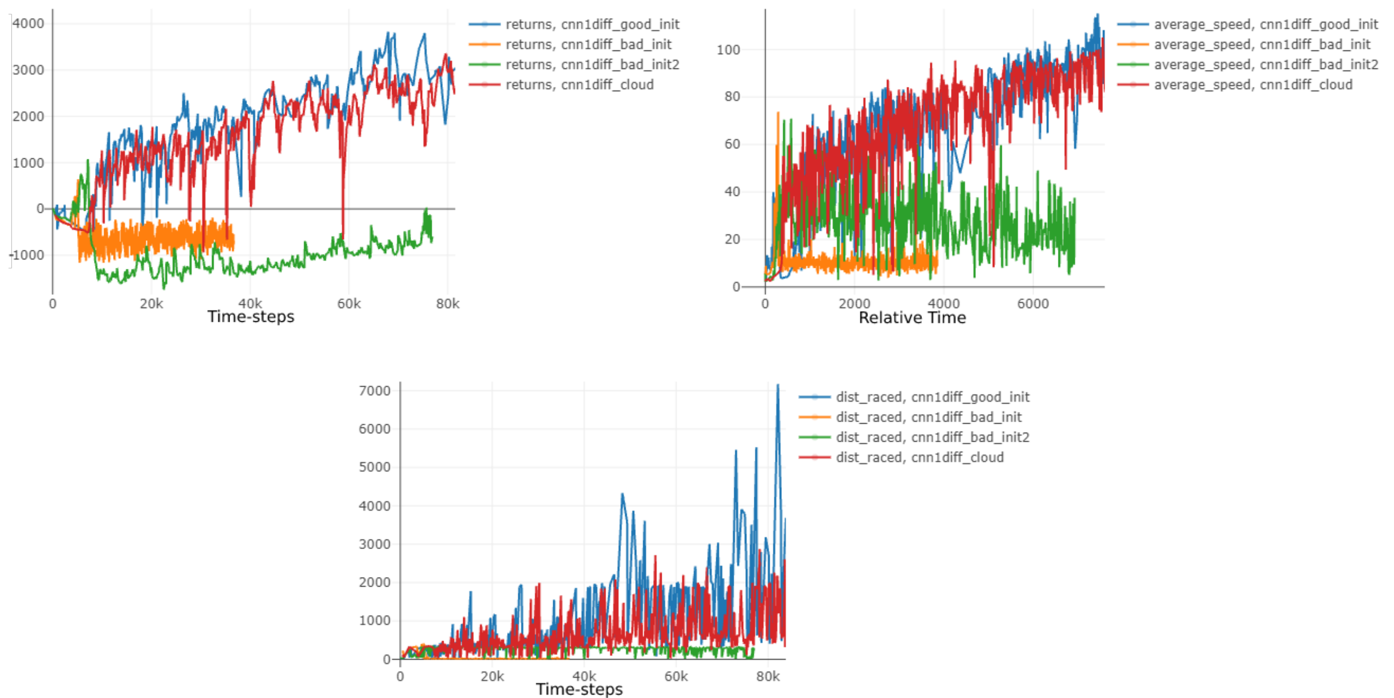

Figure 4.3: Experiment 3 - initialization and instability during optimization

It was also shown that usually a good agent starts with a medium slow speed, and in a few initial episodes continues in a forward direction, exploring the most of the track ahead of him. Usually runs, that start very slow, do not move forward at all, tend to take longer and sometimes are not able to achieve such results, as the medium speed runs. They usually tend to learn the forward motion, but then turn and go outside of the track, and that is the most they can pick up during the learning.

The other topic is the instability of an agent. That is the behavior when the agent picks up a good policy, or is on its way to achieve a good policy, continuously is improving its results, but unexpectedly its performance drops a lot. That can be seen on a Figure 4.4. This run, started just normally, already picking a good habits, thus getting higher returns, but at around 2.5k return and 50k time-steps its performance dropped dramatically to way below zero return value. This agent at that stage of training visually behaves, just like the orange or green one from Figure 4.3. That is the agent is unable to even drive forward, and if it does, it goes straight to the barrier outside of the track. The difference is, that this time, the agent had already picked up some good behavior in its policy, but is temporarily unable to use it. It was not discovered why this happens, but this behavior was already seen during other experiments. Usually such an agent recovers from this drop in a matter of multiple episodes, in this example we can see that it took the agent around 25k time-steps, which is a lot of episodes, to recover from it, but it did it successfully. Then the agent is in a matter of one episode, when it managed to somehow pick up the already learned behavior, able to drive through the track and not crashing in a first few time-steps. As a result it immediately achieves the same returns as it have before the drop of performance.

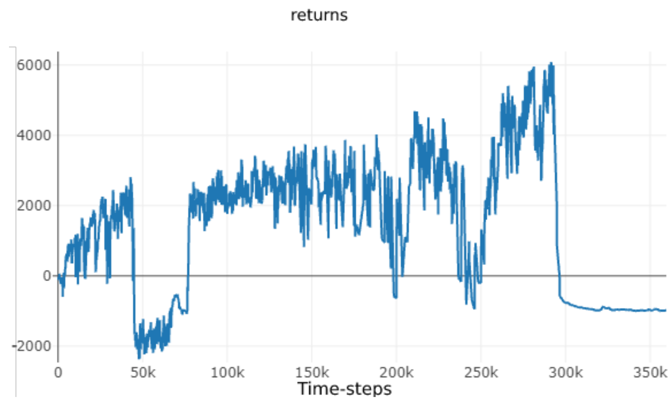

Figure 4.4: Experiment 4 - instability during optimization (ConvNet architecture)

This behavior happened mostly during the *ConvNet* architecture experiments, so it can be related to the camera output issue already discussed. It is possible that suddenly the camera output had changed, meaning the agent was not ready for it, thus good part of its input data got changed and it took some time to adjust to that change of input data. That is one of the possible explanations of such a behavior. Also we can see at this particular run, that the same happened just around the 300k time-step. Although that time the agent was not able to recover from such a drop. This also happened multiple times, usually when the agent got to its highest peak performance and stayed there for a while. Then immediately lost all of its achieved performance. This behavior was more probable at agents with bigger networks architectures and for example agents with 128 neurons in its hidden layers did not suffer from such a behavior. So one explanation can be, when the agent gets well trained, and cannot anymore increase its performance in a way that it achieves higher returns, drives more distance or has higher average speed, it just starts exploring other possible „improvements“ of its behavior, which result in a sudden drop to below zero returns performance. That is the probable explanation of such a behavior. So in order to avoid these, we periodically saved the agent's model, thus when such a drop happened, we could take the last well-behaving model of an agent from the saved models. Usually, when we tried and continued to train such a saved model, the almost same drop happened again, sometimes it was sooner, sometimes later, compared to the original run, but the drop was still present. Also these agents are often not very good in terms of generalization, they tend to perform well on the track that they were trained on, but usually are not that great on other tracks. In comparison with other models that did quite well on the training track, but far from perfect, usually performed much better on new yet unknown tracks, thus better generalized.

4.3.4 Comparison of network sizes

As a next experiment we have tried to focus on the optimal size of neural networks used for the actor-critic. Our goal was to find a size and topology of the networks that work best for the task of autonomous driving. As the actor-critic networks require as an input only numerical values from the sensors, or in case of *ConvNet* architecture, also the features vector extracted from the camera output, all the networks are of the feed-forward type, consisting only of linear layers. As an activation function they use the ReLu activation for the hidden layers. For the output layer, depending on the network, they use hyperbolic

tangent for the actor network, as it outputs the mean and variance of action probability distribution and the actions are in the interval $\langle -1, 1 \rangle$, which is the same as the hyperbolic tangent output. For the critic network, there is no special activation, as the output value should not be limited by anything.

First experiments were done with network topologies consisting of one hidden layer. We started with the 32 neurons in hidden layer size and we gradually increased the value to 64, 128, 256 and 512. All these sizes showed to be insufficient and the agent struggled to learn any good behavior. It took some time to figure this out, that the problem is in the topology of the network and not within other parts of the system. Visually the agent was usually able to drive correctly first few tens or low hundreds of meters, but usually failed at first turn and was unable to pick up the skills required for the correct turning maneuvers.

In the next set of experiments we then increased the number of hidden layers to two. We then started with slightly higher number of neurons than in the previous case. First experiments were done with 64 neurons, then again increased in power of two multiples, such the 128, 256 and 512. This time the agent was finally able to learn some good behavior and its policy was continuously improving as the training process went onward. On a Figure 4.5 we can see picked examples of such a learning. Again we have done multiple runs of different topologies and different settings of the experiment, so here we provide just a small fraction of the results, that are the most representing examples of the common training features and behavior encountered during the experiments.

As we can see, the differences between the topologies are not that obvious, also the smaller networks were able to achieve similar results as the bigger networks. Also we can see the comparison of different set of sensors and their impact on the agent. The runs named with `snrs` at the end, denote the use of all possible sensors described in section 3.2.2. On the other hand the runs without it are using only the `track` sensor. As we can see the difference is almost non-existing, but we will focus to it more in the subsequent experiment.

What we have observed though is the tendency of bigger networks to have a slightly less stable learning process, than the smaller ones, and the sweet spot in terms of size is the 128 neurons version, as it seems it has the most stable learning process and also it takes obviously less time for such a network to be trained, compared to the 512 neuron network. But these differences are not of a big concern.

The other difference though occurs during the late phase of the training, after around 200-300k time-steps. At that time the agent is almost always perfectly trained but it tries to just get better in the driving style, which results in higher average speeds, more distance raced, etc. But this is also usually the time when the agent gets overfitted for the training track. During this time it is much more clear that the bigger networks have the ability to outperform the smaller networks, at least if we compare them by the graph readings. For example for the experiments at Figure 4.5, we used the `e-track-2` track, as we can see, all of the agents achieved little over 6k in returns, which usually does not improve much more. It peaks around 7k, sometimes 8k. But the bigger networks, here the 512 neurons type is able to outperform them by a good margin. For this specific track, it is often able to achieve returns peaking around 10k, usually little less. But it takes almost twice the training time, compared to the results depicted here. As the overfitting is not required nor wanted, we usually did not train the agent any longer than 350k time-steps.

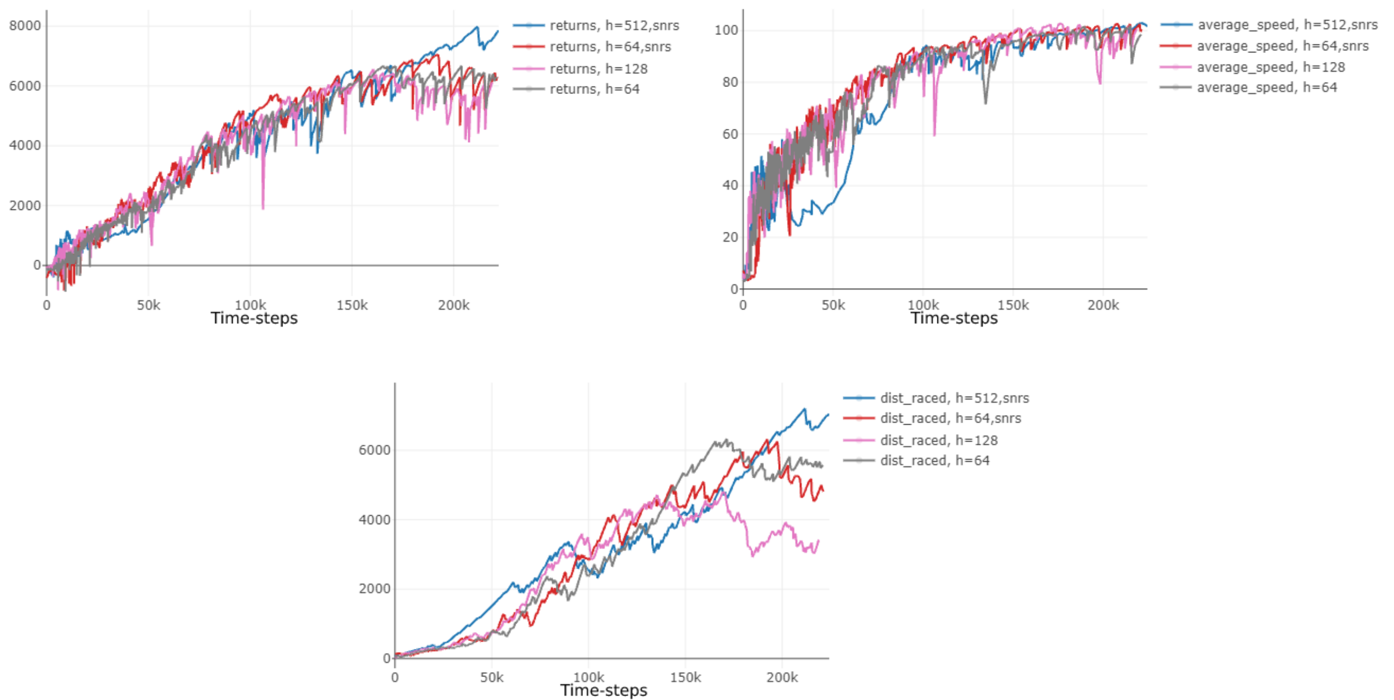

Figure 4.5: Experiment 5 - Comparison of network sizes

Comparison of most used network sizes

On a next set of graphs on Figure 4.6 we can see a direct comparison of 512 neuron and 128 neuron version of the network, which we used the most throughout the experiments. Mostly it was the 512 neurons version, as it turned out that the bigger networks were able to generalize slightly better than the smaller ones, also they were yielding higher returns in general. Even though the results on the training tracks were similar. In this example we have added the graphs of lap times and damage counter. From that we can see a characteristics typical for all the other experiments. During the initial training, the agents make a lot of damage per episode, which is understandable, as they are at that phase yet unable to drive correctly and are just starting to grasp the driving skill. As the training continues the damage count gets lower and ultimately is none, when the agent acquires the basic driving skills. In the later phase from time to time the agents cause some additional damage, meaning they are trying to improve their skills by trying different styles e.g. drive through a specific turn a little differently in order to drive through it in higher speeds and in lower times.

In this example we also omitted the smoothing of the distance raced graph, so we can see the real nature of an RL algorithm, which almost periodically goes from high distances to almost none - in this case roughly 300 meters. Which is the first turn in the `e-track-2` track. This is usual for almost every agent trained, that before a really good episode, where the agent achieves (learned agent 8-10km in distance), a really low distance is raced, in the low hundreds of meters.

It may be due to the PPO objective function, which basically erases a previous bad step in a policy, by the clipping feature for negative advantage, discussed in 2.6. So the agent picks up a little unwanted behavior within its policy, then acts accordingly poorly.

The PPO „undo's" the policy step, which basically corrects the agents behavior, and the agent is then able to achieve high distances again. This is the probable explanation of such a behavior of the distance raced performance.

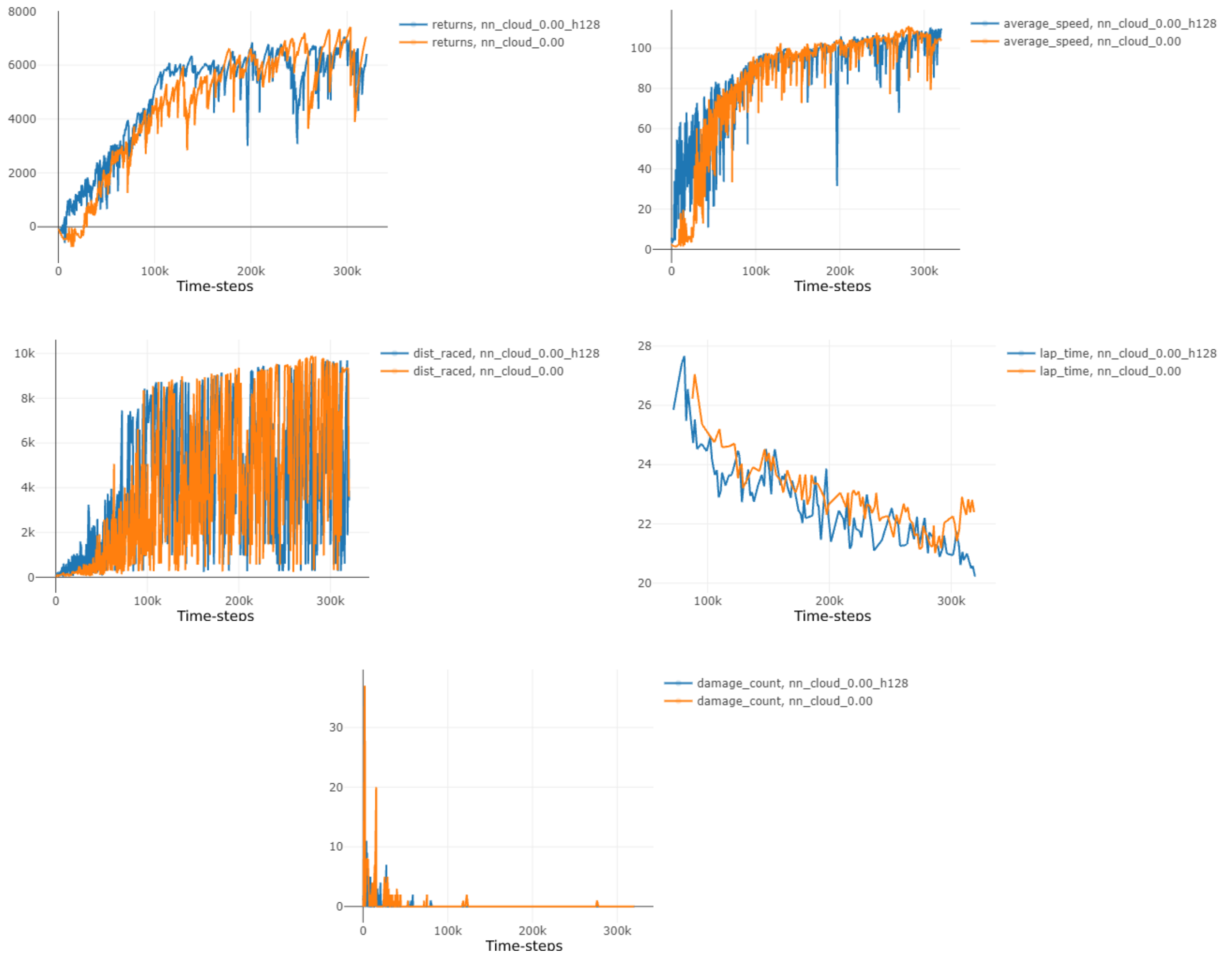

Figure 4.6: Experiment 6 - Small vs big network

4.3.5 Reward functions

If we take a look at the reward functions described in section regarding *Reward shaping* 3.2.3, there were three proposed. This experiment then represents the performance of an agent which receives rewards or penalties based on these reward functions. At first sight we can see on the graphs in Figure 4.7, that the third reward function does not perform well at all. The agent was not able to learn any good behavior, reaching no distance at all. This happened both with *ConvNet* and *Regular* architecture. The reward function used the sigmoid function to smoothen the `TrackPos sensor` role and the track width as a parameter. This reward function was not expected to work well, but this almost non

existing progress of the agent was not anticipated. However, the main focus was to compare the performance of *reward functions 1 and 2*. The first one is being the more complex one, which considers both the forward motion, as well as the „stay in the center“ policy. It supports the agent to stay at the center of the track, any deviance decreases the reward as well as it supports the minimization of speed in lateral direction. This reward function proved to work well with the agent, and it was able to learn good policies.

There is maybe one side effect to this reward function. That is, when the agent's vehicle travels on a straight road, it is not able to simply drive in a direction of the track, instead of it, it „zig-zags“ in the middle of the track, trying to correct its direction by little turns from left to right. That is probably caused by the term decreasing the lateral velocity the $-SpeedX \times sin(\varphi)$ term, which penalizes such a behavior. Other than that, the agent with *reward function 1* works great. Although we must admit, that this type of reward function in a way decreases agent's potential to reach fast lap times, as the optimal travel trajectory on a track is not always in the middle of the road. But in terms of the learning, we can agree that the more sophisticated reward function, the easier and better the agent learns.

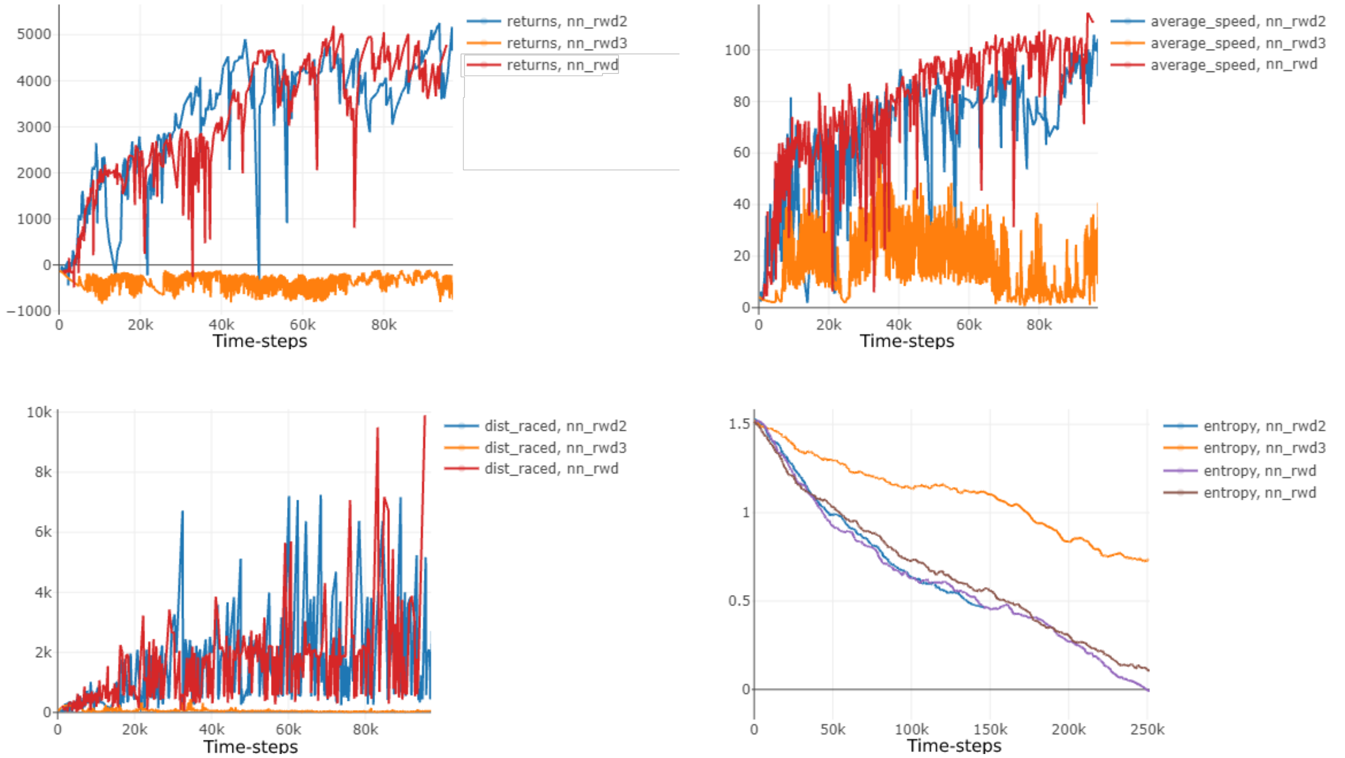

Figure 4.7: Experiment 7 - Comparison of reward functions

As for the *reward function 2*, it is basically the same as the first one, but it does not include the $-SpeedX \times sin(\varphi)$ term, but still with the $-SpeedX \times TrackPos$ term, it penalizes the agent for not being in the center of the track. Although there are differences in the performance of agents using the *reward function 2*. Mainly as the „keep in centre“ policy is not that strict anymore, the agent appears to drive through the curves little bit more smoothly, as it does not try so hard to stay in the centre. But what is maybe more important, on the straight parts of the track, the behavior of „zig-zags“ is almost

eliminated. It is still there, but it is almost not perceivable. Also, when the agent prepares for a subsequent turn, while it is driving through a straight, it does not drive in the centre of the track. Instead it tends to drive on one side, so that the turn is for the agent easier to maneuver. Then such a behavior definitely feels more natural and the overall visual perception of such a driving style feels more race-professional.

Although as the experiments regarding the comparison of reward functions were done in the later stage of the experimental phase, most of the time we used the *reward function 1*. In retrospective, it might be feasible to use instead of it the *reward function 2*, despite the assumption that more sophisticated means better. It is better for the agent, but for the „racing car performance“, we prefer the second one.

Also it might be noticed that on the graph showing distance raced, the red agent, which uses the *reward function 1* - it struggles for some time on the 2km mark, where in reality is a pretty hard right turn on the track. It suggests that it causes problems for the agent to drive successfully through this part with the „stay in the centre“ function term. So the agent with *reward function 2*, which does not include this term, struggles with this part of the track less and is able to faster pick up the correct turning maneuver through it. This also supports the fact, that it probably chooses better trajectory for that specific turn, thus does not have to slow as much as the red agent. The returns are though almost identical, suggesting both *reward functions 1 and 2*, are similarly able to learn the agent to successfully drive the vehicle on a given track.

4.3.6 Number of sensors required

These sets of experiments focus on the minimal number of available vehicle's sensors required for the agent to learn a successful policy. These experiments were performed in order to minimize the set of real sensors which will be needed for the real-world RC model car. As every additional sensor is expensive and its correct mapping into the real world brings the possibility of inaccuracy and unwanted misbehavior by the vehicle. The lower the number of necessary sensors to be installed onto the vehicle, the better for the complete system to function properly. Also we can explore the possibilities of the PPO algorithm in terms of the amount of input data it needs in order to train a successful policy for the task of autonomous driving in a racing environment TORCS.

On the Figure 4.8 we can see the results of such experiments. There were done many more experiments regarding other sensors but they showed that they are not that important and could be easily removed from the agent's vehicle. These sensors were the `damage, rpm, wheelSpinVel, trackPos, focus` and lateral and Z-axis speed. The `trackPos`, which is the sensor measuring the lateral position of the vehicle on the track. If the vehicle is right in the centre of the lane, the sensor's value is zero. For left and right lane it is in the interval $\langle -1, 1 \rangle$. The quite surprising was the possibility of removal of such a sensor. The agent seemed not to struggle at all during the learning without it.

As the most important sensors turned out to be the `track` sensor, the `angle` sensor and the speed in all three dimensions. Based on the experiments we have found that the agent was able to learn almost identically with these three sensors as with all the sensors at once. We then continued removing these sensors one by one, to test whether it is possible to find even smaller set of necessary sensors. As a result, we were able to remove the speed sensors in one set of experiments, also the `angle` sensor in other one. This was also quite a surprise that the agent was able to learn only with one sensor, which is the `track` sensor. On the contrary when we removed this specific one, the agent performed very poorly. It makes a

perfect sense, as we basically removed its only forward vision. The `track` sensor serves as a low-res LIDAR or radar sensor. It covers the frontal aerial field of view $\langle -45\,\text{deg}, +45\,\text{deg} \rangle$ and consists of 19 values. It tells the distance between the vehicle and the boundary of the road. Without this sensor, the agent only perceives the environment through `angle` sensor of the vehicle and its lateral and longitudinal velocity, which is obviously not enough. This run is depicted on the Figure 4.8 in red color. Also if we compare the purple run, which is the agent using all available sensors, it shows us, that the agent's performance is the second lowest. It might be due to the amount of information provided by the sensors, which could be just a little too much for the agent to understand. This behavior was noticeable in other similar experiments.

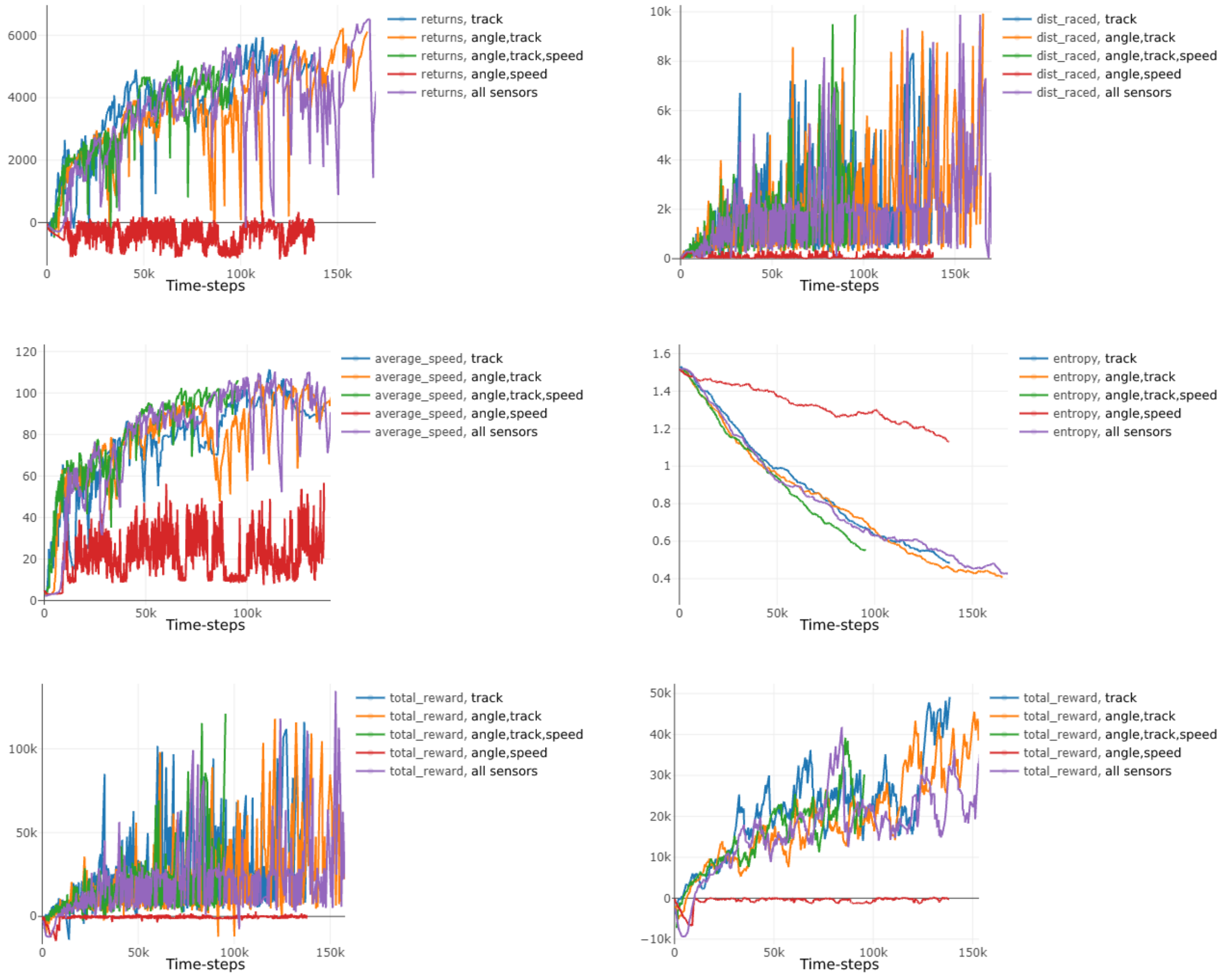

Figure 4.8: Experiment 8 - Number of sensors required

The fact that just the `track` sensor is enough is great for the RC model car, as the total cost of the necessary real-world components is greatly reduced. Also to physically install just one sensor reduces the possibility of mistakes during the process. This idea is

further discussed in the subsequent experiment, where we use the *Hybrid Architecture* and we attempt to completely abandon the use of sensors and only rely on the camera output, but in a different way than one might suspect.

Sensors vs Track sensor

For a completeness we also show a graph of comparison of the all sensors variant and the only `track` sensor variant, both in big and small neural network sizes. As we can see on Figure 4.9 there is almost no difference between the network sizes and the only `track` sensor variant works in both of them. Also in general the results (512 neurons exp.) are pretty identical, this time both of the agents suffered a little from a bad initialization and its policies performed pretty poorly for the first 50k time-steps. But they were able to recover from it very well, resulting in a 6k+ returns and 8k+ kilometers around 150k time-steps, which is still fine. Note that the 64 neurons version denoted *snrs* include only the: `track, trackPos, speed` in 3D and `angle`. It supports the idea that the ability of the agent to learn a good policy with different set of sensors is not related or dependent on the size of the neural network.

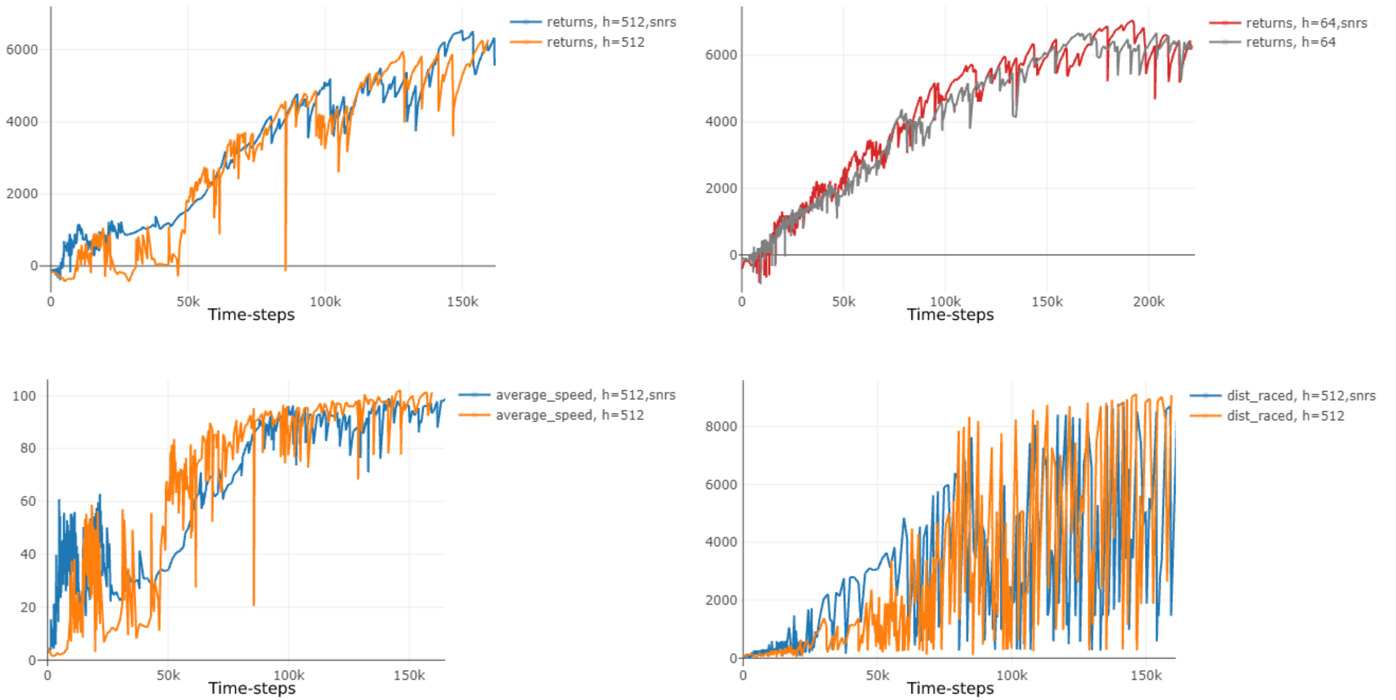

Figure 4.9: Experiment 9 - Combination of sensors vs only track sensor (different network sizes)

4.3.7 Cloud architecture

To further support the effort of successful application of the learned agent into the real-world we have already proposed and discussed the *Cloud architecture*. Here we present the results of the experiments related to it. On a Figure 4.10 we can see results of three runs, where the `pb` in its names means probability. It is in reality the percentile of the standard

Normal probability distribution with which we defined the occurrence of the packet loss or the noise in the sensory data. The results are as expected, the agent with the simulation of events occurring turned off has the best performance, after it the agent with probability 0.05 and subsequently the worse performance has the agent with the highest probability of 0.1. Though, the results of the 0.05 (green) run are great and the agent can be considered fully trained. This suggests that the PPO algorithm performs well, even if its input or output data are corrupted and do not represent the real environment state. It can also be considered that it is in this sense a very stable algorithm and can generalize on the input. The results of the 0.1 (orange) run are not that important, the experiments with it were done just for comparison and to show us the potential further possibilities of the algorithm's stability. Based on [39], the simulated packet loss is usually in articles related to networking and its simulation testing set to 5% anyways.

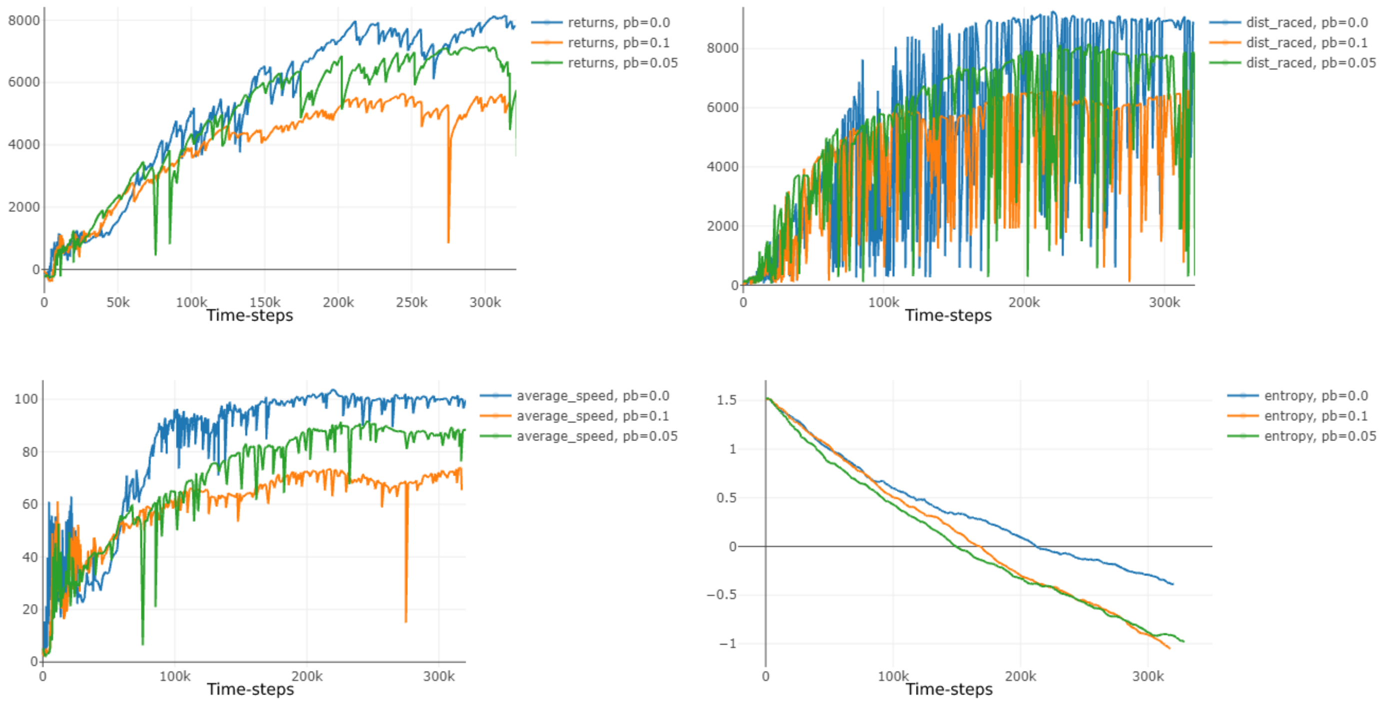

Figure 4.10: Experiment 10 - All Sensors and Cloud architecture, where `pb` is the probability of an event occurring (packet loss, noisy data)

These experiments were done with all sensors available to show the real impact of the *Cloud architecture* simulation. In comparison the results on the Figure 4.11 were done only using the `track` sensor. We can notice the lower performance in general. The most interesting is probably the performance of the 0.1 probability variant, which suffers from the *Cloud architecture* much more than in the version with all sensors. This is because when the event got triggered and the data were corrupted by noise, it always affected the only available sensor, the `track` sensor. It makes sense that the vehicle, with its only „vision“ not working correctly is not going to be able to drive properly. From the previous experiment, when the event got triggered, it only affected some of the sensors by the noise, again based on the probability. But the probability of affecting all the sensors at once was very unlikely.

So the sensory noise probably has greater influence than in the case of the simulated packet loss (no data at all), which is on one hand unexpected and little disappointing but on the other it is great, that the agent can probably handle the complete loss of data quite well.

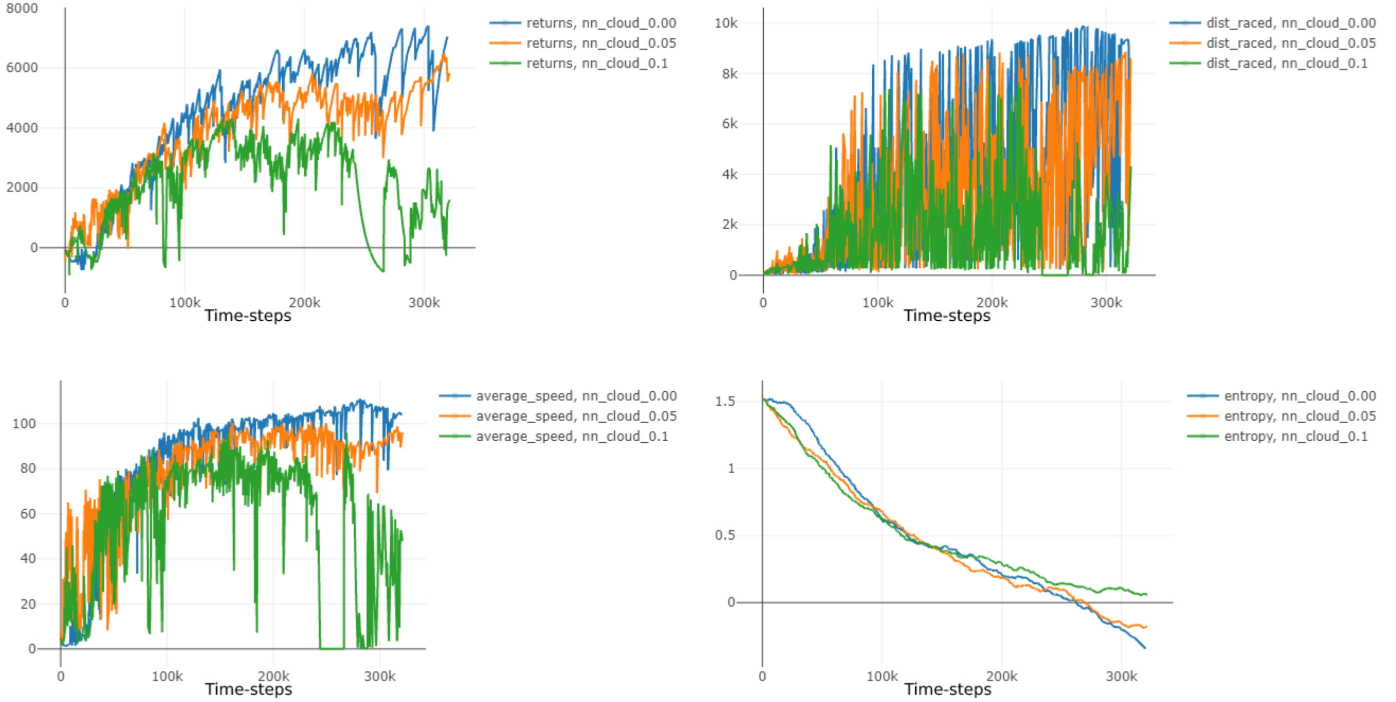

Figure 4.11: Experiment 11 - Track sensor and Cloud architecture, where `pb` is the probability of an event occurring (packet loss, noisy sensor)

These experiments were done on a `e-track-2` race track. The neural networks had 512 neurons in its hidden layers and the rest of the hyperparameters were default.

4.3.8 Example of perfect training and inability to learn

Here is an example of a run on `e-road` track with *Regular architecture* agent. However, it had been pre-trained a little, that is why the returns on the Figure 4.12 do not start from zero but around a 4k mark. This is not important for the purpose why we are showing this run here. We run it for over 1.5 million time-steps which is way over average training time. The network used for this experiment was the 128 neuron version. If we have used another bigger network the results would look a bit different. We have already discussed the issue bigger networks tend to have, that is the instability in later training phases (300k+), as the performance unexpectedly crashes down to negative values and the agent is most of the times unable to recover from it. This is a beautiful example of stability during the training. Since roughly 100k time-steps it continuously hits the 10km mark, overall performance peaks at around 500-600k time-step. The 150km/h average speed and the returns passing 10k marks are the highest that we were able to achieve throughout the experimental phase

of this thesis. Note that the 10km mark is limited by the 1600 time-steps limit for the length of one episode.

This learned agent is for sure overfitted for the training track, so it makes it unusable for later use. We simply wanted to show the maximal performance we were able to achieve. Though the agent performs great, we can still see the high deviation in distance traveled, reaching 10km mark then dropping down to few hundreds meters. We have already discussed the possible causes of this behavior.

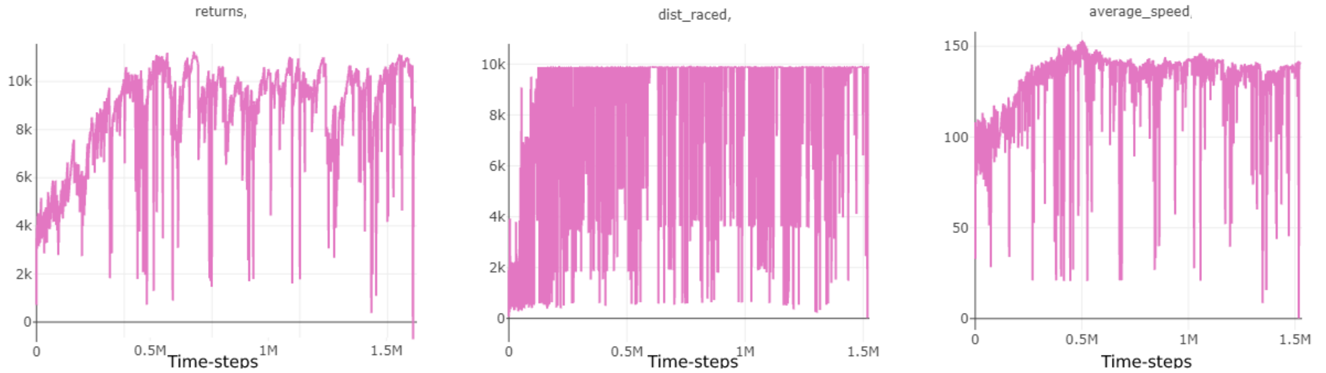

Figure 4.12: Experiment 12 - Run with a great performance (Regular architecture, 128 neurons)

In contrast and as a different example, we next present experiments targeting the inability of an agent to learn, which relies solely on the camera output. The presented results which are depicted on a Figure 4.13 are from the experiment regarding the different convolutional neural network topologies. There we were finding the ideal arrangement configuration of the pooling and convolution filter layers as well as the size of the filter's kernel and the number of such filters. The best performing architecture was then used for the *Hybrid architecture* experiments, which results are presented as a last section of this chapter and finalize the work done in this thesis.

In this section though, the exact architecture is not important, as we want to only demonstrate the inability of such an approach to learn the agent properly. The peak performance is around 900 meters for the green run (architecture 3), and around 2k in returns. But most of the time the run is heavily below this performance. Note that the 150k time-steps took around 9 hours of training, as the simulation speed could be mostly around 2x the real-time speed. The simulator issue was also already discussed at the beginning of this chapter. Although the learning progress is taking place, it is not continuous. Suggesting the unhealthy training process, where the agent is not getting better as time passes. Making this approach unusable for later use.

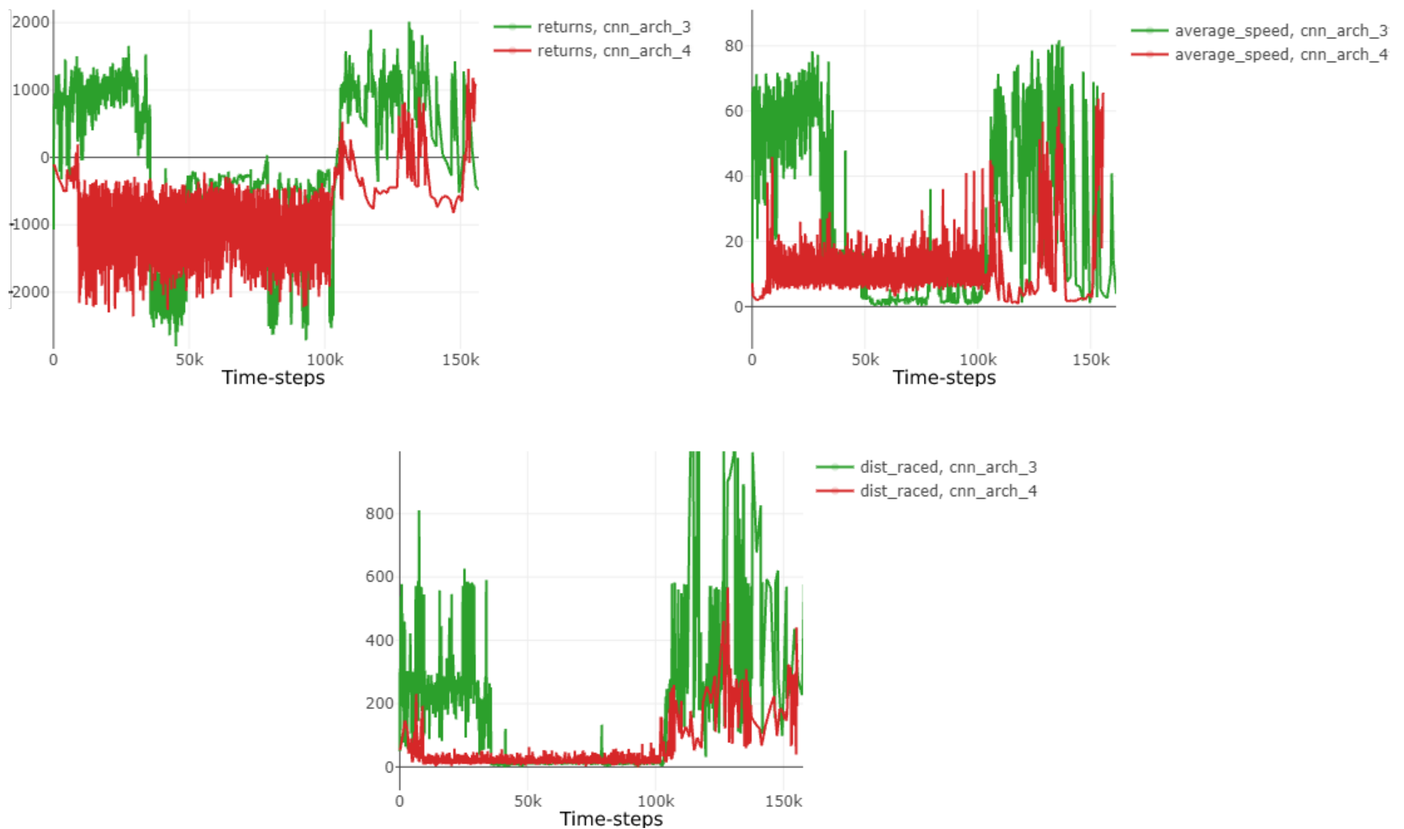

Figure 4.13: Experiment 13 - Agent is unable to learn

As a next example of the same issue we can see another experiments regarding the different image adjustments discussed in section regarding Input image. Again the agent is trained only from camera output data. The results are presented on a Figure 4.16 on the left. The `diff` in the name signifies the use of absolute image difference, the reg signifies the use of regular image without any adjustment and the numerical value in front of it means the number of images used. So the `2diff` variant represents the input image as regular image concatenated with absolute image difference image, which subtracts the current and previous camera image received by the environment.

Again, no significant signs of performance progress, the optimization is unstable and the agent is not getting better or does not keep its performance.

As for the graph on the right, this experiment represents the multiple runs of the architecture 4, which is showed also on the previous Figure 4.13. This represents the architecture that was later used also for the *Hybrid architecture* setup, where we learned such a convolutional network topology, on the image-sensors dataset for the prediction of sensory data.

Here we can see the same lack of performance, but this time most of the runs have negative reward, which usually suggests that the agent acts randomly, without any sign of logical behavior.

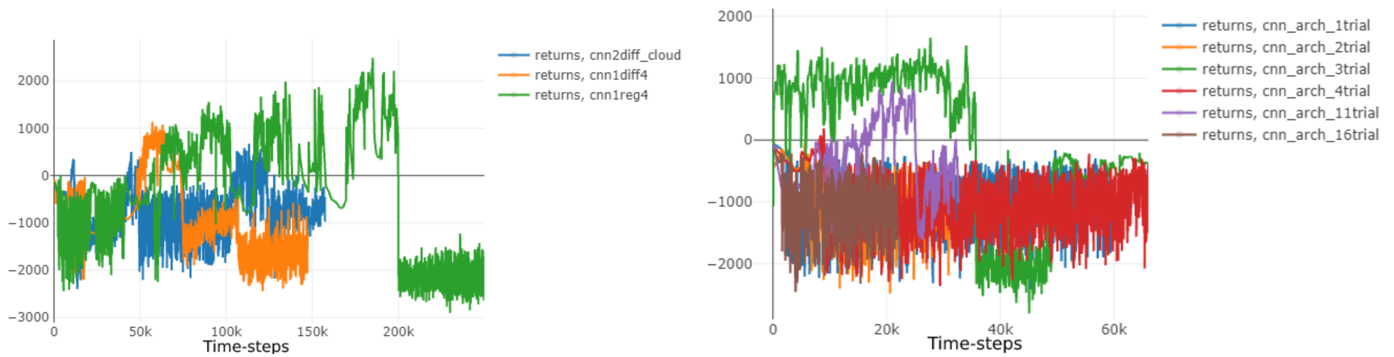

Figure 4.14: Experiment 14 - Agents are unable to learn (ConvNet architecture)

There were also tried approaches which use different sizes of linear layers within the CNN, as well as different sizes of the output feature vector (60,90,120) - which were concatenated to the forward velocity sensory value. That is also the only sensory value that was always used for all other experiments, as it makes it easier for the code implementation to work, but mainly that this velocity is usually always known to the vehicle.

Also the ideas from article [18] were tried, such as the propagation of gradients during the network back-propagation phase, only to the critic network, the already mentioned camera image adjustments, especially the capture of motion within the image by absolute image difference method.

Nevertheless, none of the attempts made this setup to work at all. This brought us the idea to learn a separate convolutional network in a regular supervised learning manner with the already mentioned (camera output, sensory data) pairs dataset and simply predict the sensory data from the camera image. Since we know that the agent is able to learn high return yielding policy by using only the sensors.

4.3.9 Generalization

As it had been already mentioned, the trained agent should be able to generalize on a different track than it was initially trained on. Here we sampled few agents and collected their performance results on a number of different tracks, which are presented in Table 4.2 and Table 4.2. The results are showed in terms of distance traveled per episode and percentage of the track covered by that distance. Since the main metric, the returns are calculated only when the agent is learning, the distance raced is the second most accurate metric, which helps us to show the overall performance quality of agent's policy.

The first table shows the results for two averagely trained agents, one of them being trained with the *Cloud architecture*. For comparison, the second one was trained as a regular agent. The table can also help us to see and decide whether the aspect of simulated cloud conditions allow the agent to generalize better.

Each agent performed around 20 episodes on each of the tracks and then the highest result was recorded into the table. The agents were performing in an evaluation mode, which does not allow the agent to be trained, so it solely relies on the policy trained by the one track used during the training process.

Model / Track	**Avg Trained - Cloud**	**Avg Trained**
Trained on:	e-track-2	e-track-2
Sensors:	**ONLY track sensor**	**ONLY track sensor**
e-track-2	9354m, 174%	9978m, 186%
e-road:	9251m, 284%	7086m, 218%
g-track-3	3724m, 131%	2110m, 74%
forza	6107m, 106%	2789m, 48%
e-track-3	3260m, 78%	2166m, 52%
e-track-4	2532m, 36%	2946m, 42%
michigan	2615m, 113%	4348m, 188%
g-track-2	5753m, 181%	6914m, 217%

If we calculate the mean percentage for each of the agents, it shows us, that the agent trained with the *Cloud architecture* actually performed a little better than the second one (138% vs 128%). If it does really mean that it can indeed better generalize is not sure. There are many aspects that surely influence such results. As an example we can mention the point in time when the training of the agent was stopped, more concretely how overfitted or underfitted was for that training track at that time. As there is no precise way to decide and measure, whether two agents are identically trained. We can stop the training after the same number of episodes or time-steps,or end the training after they reach the same traveled distance or achieve the same returns value, but the level of training congruence will never be accurate.

Nonetheless, if we omit such statements, we can truly deduce that the *Cloud architecture* indeed helps the agent to generalize better on new, yet unknown tracks and that our initial assumptions were correct.

As for the Table 4.2, we present the results for agents trained this time on another track, the `E-road` and with a different focus in mind, the size of the neural network. The same applies here as it was stated above in terms of similarity of the training process. Here, the smallest network seems to be the best in terms of generalization, which is somehow a little unexpected behavior, based on the previous experiments and assumptions. Based on the concrete results though, it achieved at almost every track 100% of the track length or more. This suggests that it knew well all different types of turning maneuvers and was able to correctly apply them in unknown environments. As for the versions with *512, 256* - it seems that it struggled with specific parts of the tracks, as both agents during their 20 episodes scored the same distance almost every time for a given track. As a result, it did not pick up the necessary skills needed to successfully drive through the specific part of the track. As for the *128* neuron variant, we can reason its good performance by the possibility that its training process was stopped at the right time, when the agent was neither overfitted nor underfitted.

Model / Track	Well Trained 512	Well Trained 256	Well Trained 128	CNN Trained (CAM)
Trained on:	e-road	e-road	e-road	e-road
Sensors:	**ALL sensors**	**ALL sensors**	**ALL sensors**	**track, angle, speed, cam**
e-track-2	9277m, 172%	1452m, 27%	9451m, 176%	291m, 5%
e-road:	9884m, 303%	9881m, 303%	9889m, 303%	1835m, 56%
g-track-3	2188m, 77%	730m, 26%	3235m, 113%	350m, 12%
forza	10929m, 189%	1257m, 22%	11266m, 194%	1870m, 32%
e-track-4	2949m, 42%	11800m, 168%	12258m, 174%	2922m, 41%
michigan	2342m, 101%	4848m, 209%	2196m, 95%	1478m, 64%
g-track-2	9657m, 303%	9660m, 303%	9666m, 303%	1356m, 43%

As for the *CNN Trained model*, we mentioned its results here, just so it is visible for comparison and that it was truly unable to successfully learn a good policy. The small distances achieved are also mainly thanks to the enabled sensors and not the camera output at all. The camera output was more of a burden, making the agent's training even more difficult.

4.3.10 Hybrid architecture

The last experiment of this thesis is related to the *Hybrid architecture* proposed and discussed in section 3.3.2. In order to make it work, we had to first collect the dataset, which consisted of around 100k samples. Then the CNN got trained separately on these data. As a label for each sample we first tried to include only the `track` sensor, as it was proven that it is enough sensory data for the agent to get trained. This, however, did not prove to be the same case in this scenario. The agent was not able to get further than few meters on the track, occasionally passing the first turn, which is on the `e-track-2` at around 300 meters. This was the maximum the agent was able to accomplish. On other tracks the results were similarly bad, but usually even much worse. It might be also due to the fact, that the dataset was roughly a third of the size than for the later experiment, at around 35k samples.

Also the experiment where we have tried to learn a new model from scratch, was not successful. This could be caused by the fact, that the dataset was captured by a trained agent, which for example was not almost crashing and typically had different driving style than a newly started agent has. New agent usually for the first number of episodes does not have any idea what it is doing, so it crashes a lot and does unpredictable turns. For these untypical situations there was almost no data in the dataset. So it makes sense that the agent was not able to get trained by this approach, as the neural network was not generalizing well for these types of unpredictable situations.

Later, when we have tried to learn a new agent. We captured into the dataset both trained and untrained agent experience. Around 80% of the dataset was from the trained agent and the 20% was from the untrained one. Then we have trained a usual agent before hand. This agent trained on a bigger number of sensors, concretely `track, angle, speed, and trackPos` sensors. The same sensors (but `speed`) were then also captured during the dataset creation. For the CNN supervised learning task, we used as a loss the MSE (*Mean Square Error*) and a custom metric called *percDiff*. It calculates the cumulative difference between all unique sensory values as:

$$percDiff = \frac{|y - y_{pred}|}{\frac{y + y_{pred}}{2}} \times 100$$

This is calculated for the batch of size 32 and then a mean from an absolute values of *percDiff* is taken. We were able to achieve cumulative sum of 45% for the training set and 44% for the test set for 21 values, which is about 2% average deviation from the single sensory value. For the MSE loss, we have achieved the value of 0.0022. The training was done on a *Google Colab* service in a Python Notebook, which is also part of the code submitted with this thesis.

This way we were quite successfully able to produce sensory data from a camera output image and bypass the fact that the RL agent was not able to learn solely from raw camera output. The results were much better than in the previous case. The agent was immediately able to drive greater distances of around 3km, but yet at some points of the track, especially during a sharp turns, it was not able to properly drive through it. This was probably caused by the lack of computational performance of our machine. As the agent relies completely on the camera output and a much more computation is being done during every time-step, the machine periodically struggled during certain parts of the track. This resulted in a small lag and window image freeze for a short period of time, which was around 5-8 in-game frames. This was usually the cause of the short distances of the agent's vehicle. As when the screen unfroze, the vehicle was already too far into the turn, and it was too late for any correction as the reaction time was very limited. This resulted in an agent being unable to handle the sharp curves, thus crashing into the barrier and the episode has ended.

When this lag happened during a slightly curved turn or on a straight, the agent usually did not have greater issues of handling the lag event well. It surely helped that the initial agent was trained with *Cloud architecture* setup, so that it was used to these types of incidents (missing data) much more, than a regularly trained agent.

As we were unable to obtain a more powerful machine, we could not test the abilities of the agent in more depth. Nevertheless, it is still a great result, considering the amount of problems we have met during the practical part of this thesis.

4.4 Summary and further outlook

For future works, it would be advisable to increase the generalization of the agent, together with further camera output alternation, where the GANs *Generative adversarial networks* could be used to generate a real-world looking image from a simulation environment's image. This concept is already employed by the Tesla Company, so it is feasible. Also, further improvements of the PPO algorithm, mainly the ability to optimize in parallel on multiple processors to speed-up the learning process, should be considered. These attempts were done as well, but were later abandoned, as the computational hardware was not able to handle such tasks sufficiently. Lastly, to concentrate on the physical implementation of the agent's vehicle in a real-world.

Initial experiments with the RC model car were done even before the topic for this thesis was selected, as it was a personal side project of mine. Due to the insufficient knowledge in the electrical engineering field and the lack of skill in the computer modelling (as lot of the custom components had to be 3D printed) and mainly the lack of finance for the additional, necessary, electronic components for further experiments, as they were destroyed a lot during the process, resulted in a halt of this physical implementation of the project.

We then continued working only on the software, more concretely, the machine learning approach to autonomously drive an RC car. Which is also the topic of the thesis.

But surely, the future goal is to return to the hardware side of the project, now finally with a working ML software solution for it.

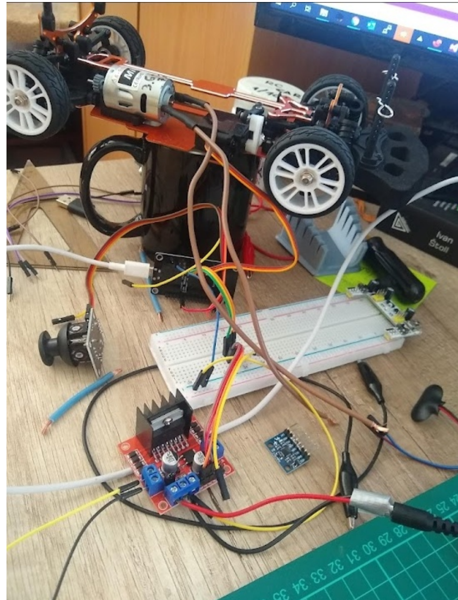

Figure 4.15: Photo taken during the testing of the RC model car, controlled wirelessly via ESP32 and Raspberry Pi

The idea was to use a micro-controller *ESP32* mounted on the car, together with the necessary sensors and a camera, which will then communicate wirelessly with the computer over *MQTT* protocol, where all the processing would be done. The computer would then send back the commands for the actuators. The computer could be either *Raspberry Pi* mounted right on the vehicle (in this case no wireless communication) or communicate with a more powerful computer (referred as the Cloud) wirelessly. The goal would then be for the RC vehicle to be able to drive autonomously on a custom track and participate in a competition, such as the *NXP Cup* organized by *NXP Semiconductors* company [5].

Figure 4.16: Example of the competition cars created by university teams [3]

Chapter 5

Conclusions

In this thesis, the goals stated in Chapter 1 were successfully accomplished. We were able to learn an agent in the task of autonomous driving.

As we were unable to train the agent solely from the camera output, and partly with the proposed *ConvNet architecture*, where creation of features vector from a camera image was concatenated with data from other sensors, the novel approach was proposed. The novel *Hybrid architecture* reacts to the inability by incorporating convolutional neural network learned in a classical supervised learning approach on a camera output - sensory data, dataset pairs. We were able to train such a network to predict sensory data, which then served as an input to an already trained agent by *Regular architecture* approach. In this approach, the agent was learned by regular sensory data, such as the speed, low-res LIDAR sensor (*track*) and angle of the vehicle related to the road direction.

We have also found minimal set of sensors required for the agent from which it could learn a successful policy. It was found that only the *track* sensor is enough for the agent to yield high returns. We have also discovered that smaller networks are better in terms of stability of the optimization process, whereas bigger networks are able to achieve higher returns. As for the reward function, we empirically found that reward function without a term penalizing the „not in the centre“ driving style performed overall better, as the agent was able to discover more human-like driving style and was not unnecessarily forced to drive only in the centre of the road.

In terms of generalization, agents with lower performance tended to generalize better, than agents that could be considered as overfitted for the training track. Also, agents that were trained on a track with suitable track topology were better at generalization, as they had the knowledge of diverse sets of curve types.

The use of proposed *Cloud architecture* and the ability of the agent to perform well under the simulated real-world conditions, also helped the agent to be more stable and better prepared for the real use.

All these findings help us in future work, as we plan to apply such a learned agent to the real-world scaled RC model car. The necessity of only one sensor or the ability to predict sensory data from a camera image greatly reduce the costs and requirements for physical sensor components as well as the need for low-performance embedded hardware, as we have achieved great results by only using small neural network models.

It will be interesting to see, whether these preparatory works of learning the agent in a simulated environment were worth it and will also work in harsh, outdoor conditions.

Bibliography

[1] *The U.S. National Highway Traffic Safety Administration: Automated Vehicles for Safety* [online]. 2016 [cit. 2021-10-05]. Available at: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[2] *BASt: Self-driving cars - assisted, automated or autonomous* [online]. 2020 [cit. 2021-10-05]. Available at: https://www.bast.de/DE/Presse/Mitteilungen/2021/06-2021.html.

[3] *Embedded FI MUNI* [online]. 2021 [cit. 2022-22-04]. Available at: https://embedded.fi.muni.cz/projects/nxpcup.

[4] *SAE International: SAE J3016 - Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles* [online]. 2021 [cit. 2021-10-05]. Available at: https://www.sae.org/standards/content/j3016_202104/.

[5] *NXP Cup* [online]. 2022 [cit. 2022-22-04]. Available at: https://nxpcup.nxp.com/.

[6] Abbeel, P. *Foundations of Deep RL - YouTube Lecture Series* [online]. 2021 [cit. 2021-11-11]. Available at: https://bit.ly/3F5cSfV.

[7] Burget, L. *BAYa - Bayesian models for machine learning - VUT FIT* [online]. 2020 [cit. 2021-12-10]. Available at: https://www.fit.vutbr.cz/study/courses/BAYa/public/.cs.

[8] Dossa, R. F. J. *GymTorcs: An OpenAI Gym-style wrapper for the Torcs Racing Car Simulator*. 2018.

[9] Edwards, C. X. *SnakeOil - Virtual Motor Sports Lubricants (TORCS Client)* [online]. 2015 [cit. 2021-10-08]. Available at: https://xed.ch/p/snakeoil/.

[10] Ganesh, A., Charalel, J., Sarma, M. D. and Xu, N. Deep Reinforcement Learning for Simulated Autonomous Driving. 2016.

[11] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Intel. *Intel i5 8250U - Processor* [online]. 2022 [cit. 2022-02-15]. Available at: https://ark.intel.com/content/www/us/en/ark/products/124967/intel-core-i58250u-processor-6m-cache-up-to-3-40-ghz.html.

[13] Jaritz, M., Charette, R. D., Toromanoff, M., Perot, E. and Nashashibi, F. End-to-End Race Driving with Deep Reinforcement Learning. 2020. Available at: http://team.inria.fr/rits/drl.

[14] Karpathy, A. *Deep Reinforcement Learning: Pong from Pixels* [online]. 2016 [cit. 2022-01-06]. Available at: http://karpathy.github.io/2016/05/31/rl/.

[15] King, A. A. *Modeling the noise: probability basics and the bestiary of distributions* [online]. 2018 [cit. 2022-03-15]. Available at: https://kinglab.eeb.lsa.umich.edu/480/prob/prob.html.

[16] Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A. et al. Deep Reinforcement Learning for Autonomous Driving: A Survey. february 2020. Available at: http://arxiv.org/abs/2002.00444.

[17] Koeller, A. *Evaluation of reinforcement learning algorithms in the context of autonomous vehicles* [online]. 2019. Available at: https://bit.ly/3yBAmcQ.

[18] Kostrikov, I., Yarats, D. and Fergus, R. Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels. april 2020. Available at: http://arxiv.org/abs/2004.13649.

[19] Kuhle, H. *Autonomous Driving* [online]. 2020 [cit. 2021-10-05]. Available at: https://www.vda.de/de/themen/digitalisierung/autonomes-fahren.

[20] Lapan, M. *Deep Reinforcement Learning Hands-On, Second edition*. Packt Publishing, 2020. ISBN 978-1-83882-699-4.

[21] Lee, D.-H. and Liu, J.-L. End-to-End Multi-Task Deep Learning and Model Based Control Algorithm for Autonomous Driving. december 2021. Available at: http://arxiv.org/abs/2112.08967.

[22] Levine, S. *CS285 Deep Reinforcement Learning course - UC Berkeley* [online]. 2021 [cit. 2021-12-19]. Available at: http://rail.eecs.berkeley.edu/deeprlcourse-fa19/.

[23] Li, Y. *Deep Reinforcement Learning: An Overview* [online]. 2018 [cit. 2021-12-15]. Available at: https://arxiv.org/pdf/1701.07274.pdf.

[24] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T. et al. Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*. september 2015. DOI: 10.48550/arxiv.1509.02971. Available at: https://arxiv.org/abs/1509.02971v6.

[25] Loiacono, D., Cardamone, L. and Lanzi, P. L. *Simulated Car Racing Championship: Competition Software Manual* [online]. 2013 [cit. 2021-10-02]. Available at: https://arxiv.org/abs/1304.1672.

[26] Loiacono, D., Prete, A., Lanzi, P. L. and Cardamone, L. Learning to overtake in TORCS using simple reinforcement learning. In: *IEEE Congress on Evolutionary Computation*. 2010, p. 1–8. DOI: 10.1109/CEC.2010.5586191.

[27] MLflow. *MLflow - Open-source tracking platform Documentation* [online]. 2022 [cit. 2021-11-01]. Available at: https://mlflow.org/docs/latest/index.html.

[28] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P. et al. Asynchronous Methods for Deep Reinforcement Learning. february 2016. Available at: http://arxiv.org/abs/1602.01783.

[29] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I. et al. Playing Atari with Deep Reinforcement Learning. december 2013. Available at: http://arxiv.org/abs/1312.5602.

[30] NVIDIA. *NVIDIA CUDA Zone* [online]. 2022 [cit. 2022-02-15]. Available at: https://developer.nvidia.com/cuda-zone.

[31] OpenAI. *OpenAI Gym Environment* [online]. 2021 [cit. 2021-11-01]. Available at: https://gym.openai.com/.

[32] Poupart, P. *CS885 Reinforcement Learning course - University of Waterloo* [online]. 2020 [cit. 2021-12-28]. Available at: https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-spring20/index.html.

[33] Remonda, A., Krebs, S., Veas, E., Luzhnica, G. and Kern, R. Formula RL: Deep Reinforcement Learning for Autonomous Racing using Telemetry Data. arXiv. 2021. DOI: 10.48550/ARXIV.2104.11106. Available at: https://arxiv.org/abs/2104.11106.

[34] Russell, S. *Artificial Intelligence: A Modern Approach, 4th Edition*. Pearson, 2020. ISBN 978-0134-61099-3.

[35] Schoettle, B. SENSOR FUSION: A comparison of sensing capabilities of human drivers and highly automated vehicles sustainable worldwide transportation. 2017. Available at: http://www.umich.edu/~umtriswt.

[36] Schulman, J., Levine, S., Moritz, P., Jordan, M. I. and Abbeel, P. Trust Region Policy Optimization. february 2015. Available at: http://arxiv.org/abs/1502.05477.

[37] Schulman, J., Moritz, P., Levine, S., Jordan, M. I. and Abbeel, P. *High-Dimensional continuous control using Generalized Advantage Estimation*. [cit. 2021-12-29]. Available at: https://sites.google.com/site/gaepapersupp.

[38] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. Proximal Policy Optimization Algorithms. july 2017. Available at: http://arxiv.org/abs/1707.06347.

[39] ScienceDirect. *Packet Loss Probability - an overview | ScienceDirect Topics* [online]. 2022 [cit. 2022-03-15]. Available at: https://www.sciencedirect.com/topics/computer-science/packet-loss-probability.

[40] Silver, D. *Introduction to Reinforcement Learning with David Silver - DeepMind* [online]. 2015 [cit. 2021-11-15]. Available at: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver.

[41] Silver, D., Huang, A., Maddison, C. J. and Guez, A. Mastering the game of Go with deep neural networks and tree search. january 2016. Available at: https://www.nature.com/articles/nature16961.

[42] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction, Second edition*. The MIT Press, 2018. ISBN 978-0262-03924-6.

[43] Tampuu, A., Semikin, M., Muhammad, N., Fishman, D. and Matiisen, T. A Survey of End-to-End Driving: Architectures and Training Methods. march 2020. Available at: http://arxiv.org/abs/2003.06404.

[44] Tesla. *Tesla Autopilot Vision* [online]. 2021 [cit. 2021-12-18]. Available at: https://www.tesla.com/support/transitioning-tesla-vision.

[45] Wang, S., Jia, D. and Weng, X. Deep Reinforcement Learning for Autonomous Driving. november 2018. Available at: http://arxiv.org/abs/1811.11329.

[46] Weng, L. *A (Long) Peek into Reinforcement Learning* [online]. 2018 [cit. 2021-10-14]. Available at: https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html.

[47] Weng, L. *Policy gradient algorithms* [online]. 2018 [cit. 2021-10-14]. Available at: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html.

[48] Wymann, B. *The Open Racing Car Simulator* [online]. 2021 [cit. 2021-09-25]. Available at: http://torcs.sourceforge.net/.

Appendix A

Installation and run of the program

In the files attached to this thesis, in the root directory a `README.txt` file is located, where the complete installation process is described.

For the complete installation, it should be enough to run a shell script `install_script.sh`. The installation was tested on a Linux Ubuntu 18.04, other Linux distributions might require additional steps.

It is necessary to activate the Python environment from root directory by:

```
source ./venv/bin/activate
```

To run the program, following command should be entered from the root directory:

```
python3 run.py -c config.yaml
```

In the same root directory there is `config.yaml` file, which contains the configurable parameters for the experiments, with a description for every parameter.

For the MlFlow server to run, the following command is required to run in separate terminal instance:

```
mlflow ui
```

To access it, enter in an internet browser the localhost address with port 5000:

```
http://127.0.0.1:5000
```

Agent models are located in the `/checkpoints` directory, specific models can be downloaded from the MlFlow UI.

Also an example of a trained agent is visible at: https://www.youtube.com/watch?v=vEEQYFk1Chg

Complete source code of this work is also available at: https://github.com/Vosa23/master_thesis_final