UNIVERSITÄT OSNABRÜCK

Controlling Self-Driving Race Cars with Deep Neural Networks

UNIVERSITY OF OSNABRÜCK

DEPARTMENT OF NEUROINFORMATICS

BACHELOR'S THESIS

Author:
Christoph Stenkamp

Supervisors:
Leon René Sütfeld
Prof. Dr. Gordon Pipa

Osnabrück,
14th September, 2017

Abstract

State-of-the-art self-driving cars rely on handcrafted algorithms to control the cars' movement. As those systems depend on complete understanding of the environment, tactical decisions such as optimizing its speed are not considered as a factor in their driving profiles. In this thesis, a racing simulation will be paired with state-of-the-art deep learning techniques to investigate how to maximize a self-driving car's operational capabilities. Building on a racing simulation given as a game implemented with the Unity game engine, this thesis adds functionality to enable artificial agents to play and learn it. Further, first capable agents that successfully learn the environment are provided. After discussing the implementation of environment and agents, first experiments are conducted concerning the best model and reward-function and the effect of supervised pre-training.

Acknowledgements

This thesis is dedicated to my friends and family.

A big thanks goes to my supervisors as well as my proof-readers, and also to Alex for answering many questions.

I am most grateful to my girlfriend Marie, who had the hardest task of all: Enduring me while writing this thesis, while consistently proof-reading and helping me focus.

Contents

List of Figures

List of Algorithms

List of Abbreviations

The abbreviations used throughout the work are compiled in the following list below. Note that the abbreviations denote the singular form of the abbreviated words. Whenever the plural forms is needed, an *s* is added. Thus, for example, whereas ANN abbreviates *artificial neural network*, the abbreviation of *artificial neural networks* is written as ANNs.

ANN	**A**rtificial **N**eural **N**etwork
API	**A**pplication **P**rogramming **I**nterface
CNN	**C**onvolutional (artificial) **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
DDDQN	**D**ueling **D**eep-**D**ouble**Q**-**N**etwork
DDQN	**D**eep-**D**ouble**Q**-**N**etwork
DDPG	**D**eep **D**eterministic **P**olicy **G**radient - Network
DPG	**D**eterministic **P**olicy **G**radient
DQN	**D**eep-**Q**-**N**etwork
FPS	**F**rames **P**er **S**econd
GUI	**G**raphical **U**ser **I**nterface
MDP	**M**arkov **D**ecision **P**rocess
PG	**P**olicy **G**radient
POMDP	**P**artially **O**bserved **M**arkov **D**ecision **P**rocess
TORCS	**T**he **O**pen **R**acing **C**ar **S**imulator *(software)*

1 Introduction

1.1 Motivation

Self-driving cars are attracting more and more public attention. Precisely 125 years after the first long-distance car ride of *Bertha Benz* in 1888, a *Mercedes* drove the same distance fully autonomously in 2013[1]. The first US driving license was issued to a self-driving car developed by *Waymo* (previously *Google*) already in 2012, with fully autonomous cars without any human interface devices driving on the streets since 2014. As of November 2016, their fleet of self-driving cars had driven more than 3.5 million kilometers fully autonomously on public roads[2]. As of 2017, self-driving *Uber*-taxis are already driving on streets of the USA[3].

The most famous case of self-driving cars is however the autopilot-feature that comes with almost every *Tesla*, that takes away more and more driving tasks from its human driver. Since the company introduced their *autosteer*-feature, crashes were reduced by an impressive 40%[4]. For the aim of this thesis, Tesla is an especially interesting case: Many of their cars are equipped with the capability to drive autonomously, including all the necessary sensors. These sensor are active and produce data even when the AI is not driving, allowing Tesla to collect billions of miles of data[5] to their central hub. Most interestingly, the company incorporates *Deep Neural Networks* on this huge amount of data to improve their self-driving algorithms[6], be it only for visual processing like modelling the scene around the car.

In other news, the research area of *Reinforcement Learning* has achieved incredible progress in the past years. While respective algorithms failed even in simple environments until the publication of the *Deep-Q-Network*[19], more and more environments are played by reinforcement learning agents with superhuman performance[7].

As there are many approaches aiming at transforming virtually driving agents to ones that drive in the real world (see eg. [41]), it is interesting to consider agents that are trained virtually, in the hope of translating their techniques and properties to actual self-driving cars. A huge advantage of training virtually is that it is not necessary to keep up a *zero tolerance policy*. This is especially relevant in the context of race cars, as it allows to focus on *tactical decisions* instead.

---

[1] https://www.mercedes-benz.com/en/mercedes-benz/innovation/autonomous-long-distance-drive/ [accessed 1st September, 2017]

[2] https://static.googleusercontent.com/media/www.google.com/en//selfdrivingcar/files/reports/report-1116.pdf [accessed 1st September, 2017]

[3] https://www.uber.com/cities/pittsburgh/self-driving-ubers/ [accessed 1st September, 2017]

[4] https://electrek.co/2017/01/19/tesla-crash-rate-autopilot-nhtsa/ [accessed 1st September, 2017]

[5] https://electrek.co/2016/11/13/tesla-autopilot-billion-miles-data-self-driving-program/ [accessed 1st September, 2017]

[6] https://www.tesla.com/autopilot [accessed 1st September, 2017], for more information about *Tesla Vision* see https://electrek.co/2016/11/18/tesla-self-driving-demonstration-video-real-time-tesla-vision/ [accessed 1st September, 2017].

[7] In the midst of writing this thesis, an algorithm by *OpenAI* for example defeated dozens of human professionals in the popular multi-player game *DOTA 2* (https://blog.openai.com/more-on-dota-2/ [accessed 12th September, 2017])

1.2   Goal of this thesis

The goal of the work behind this thesis is to *Control Self-Driving Race Cars with Deep Neural Networks*. While this is a very coarse goal, the tasks of the work are clearly defined. First, a given car simulation was to be transformed into a platform that can be learned. Second, agents need to be developed that make good tactical race decisions inside this platform. Third, research questions are answered about specific features of these agents. The following sections will describe the three tasks in more detail.

1.2.1   Driving

As put forward by *Lex Fridman* in his MIT lecture "*Deep Learning for Self-Driving Cars*"[8] the tasks for self-driving cars can be sub-divided into the following categories:

- Localization and Mapping
- Scene Understanding
- Movement Planning
- Driver State

Similar categorization is provided by the developers of the race car simulation *TORCS*[39], which divide the *racing problem* among others into *trajectory planning*, which is finding an optimal trajectory on the fly while driving, and *inference and vision*, the problem of how to infer useful information from high-dimensional input.

State-of-the-art self-driving cars rely on handcrafted algorithms to solve either of these problems individually in highly modularized systems. The problem of driving a car is seperated into tactical decisions, such as what speed to aim for to drive safely or whether or not to overtake, and on the other side operational low-level decisions for the actual motor commands. While many advances on the tactical side are made, offensive tactical profiles require good operational systems. In car racing, be it virtual or real motorsports, extreme tactical profiles are used, as the goal is to perform at the limits of the possible. Such tactical profiles require reliable operational performance.

Especially in virtual car racing, where mistakes are condoned far more than in real life where actual lives are involved, it is interesting to focus on the overall racing problem. In *end-to-end* approaches, this can be summarized as *minimal lap time*: Finding the policy that minimizes the excepted time for a given lap.

While this problem can be solved analytically, it is also interesting to solve the racing problem *on the fly*, where the agent learns over its task only through continuous interaction. If the situation of a single car on the track is given, the problem corresponds to a *partially observed Markov decision process*. A formulation of the environment in such a way allows for *reinforcement learning*, a branch of artificial intelligence where many progresses are made in recent time.

In the course of this thesis, a virtual driving agent was developed that solves a given racing game using recent advances in *deep neural networks* as well as reinforcement learning. The goal of this agent is to achieve the best possible driving policy, advancing as far as possible without crashing, or even minimizing laptime of a given track.

1.2.2   Creating a research platform

Creating an artificial driving agent is impossible without an environment to train the agent on. While numerous environments for car racing already exist (like some environments accessible through OpenAI's *gym* or the *TORCS* platform), in this thesis a proprieatary software will be

---

[8]MIT 6.S094, course website: `http://selfdrivingcars.mit.edu/`

used. The game to be played is a driving simulation programmed with the game engine *Unity 3D*.

The basis of the environment was given by the first supervisor of this thesis as a fully playable game. In the course of this thesis, this game was extended, such that artificial driving agents can communicate with this platform over *sockets*. The game sends high-level as well as low-level information about the state of the game and requires actions back fast from the agent. Additionally, all functionality to easily create agents using the programming language *Python* and the deep learning library *TensorFlow* was implemented. This communciation needs to be as efficient as possible, such that the performed action receives the environment in time.

There are several contrasts to other environments and other approaches that make this one interesting: For example, the game is live, inspectable and a user can intervene into what the agent does at any time. This makes it easy to assess the policy of an agent. Further, if agents specify a *reward* or some measure of *value of state or action*, these values can be inspected – if the state-value or reward is high when driving into a wall at full speed, something is likely to be wrong.

Further, as source code of both game and agent is open and the game is programmed straight forward without unneeded features, it is very easy to change its code, such that any new information an agent could incorporate is easily added to it. This thesis describes the implemented components in detail, inviting interested parties to add functionality at their convenience.

As the implementation of such a research platform is by no means an easy task and requires profound software engineering, a substantial portion of this thesis deals with the specific implementation – this means that this thesis serves further as a manual of the code, as a starting point for further theses.

1.2.3 Research questions

Additionally to implementing platform and agent, several research questions will be crystallized and answered. In doing so, different agents will be developed, and their performance compared. The main answered questions are:

- How different models perform in comparison, and specifically if discretizing the action-space impairs performance

- What a good reward function looks like, that rewards the *correct* behaviour at all times (including braking)

- How agents that rely purely on pretraining perform in comparison to reinforcement learning agents

- How to incorporate pretraining into reinforcedly learning agents

1.3   Reading instructions

This thesis is structured as follows:

Chapter 1   begins with the motivation for this topic. Afterwards, the goals of this thesis are presented. In the end, a short summary of the chapters is given as an overview.

Chapter 2   provides an extensive theoretical foundation of reinforcement learning. The first section details how to correctly formalize an environment as a Markov decision process. Afterwards, Q-learning will be explained. This leads over to the *Deep Q Network*, a recent advance in deep learning. Subsequently policy gradient techniques will be introduced, most notably a learning technique termed *Deep DPG*. The chapter concludes with an overview of exploration techniques.

Chapter 3   gives an overview of related work in this field. At first, other frameworks for reinforcement learning and racing simulations will be introduced. Afterwards, a coarse overview of the state-of-the-art in machine learning for self-driving cars, in real life as well as in simulations, will be given.

Chapter 4   details the architecture of the implementation provided in the course of this thesis. It will start with the general characteristics and design decisions, before providing a detailed explanation of the source code of environment and agent. The explanations are in enough detail to understand the complete code developed in the scope of this thesis. Any reader not interested in the details of the game can skip sections 4.2.1 and 4.2.2

Chapter 5   explains the results of the implementation in the form of the features, models and agents that were developed on top of the platform from chapter 4.

Chapter 6   presents the performance of the agents from chapter 5 and analyzes them to answer the research questions as stated in chapter 1.

Chapter 7   discusses the results of this thesis. For that, the chapter is divided into three parts, where the first part debates the platform and the second part the agents. Both of these parts also discuss open questions and future research directions. The last section of this chapter ends this thesis with the conclusion.

2 Reinforcement learning

The task at hand was not only to provide a reinforcement learning agent, but also to convert a game itself into a respective environment. In this chapter I will go into detail about reinforcement learning in general, giving insights on the specific approaches chosen. The explanations will be given from a theoretical perspective at first and in detail associated with specific implementations later on.

2.1   Reinforcement learning problems

Machine Learning can be subdivided into three main categories: supervised learning, unsupervised learning and semi-supervised learning. The first deals with direct classification or regression using labelled data which consists of pairs of datapoints with their corresponding category or value. Unsupervised lerning algorithms cluster the data without any prior knowledge, for example by grouping objects by similarity in their properties. Semi-supervised learning entails all techniques which involve a *weak teacher*.
In this thesis, a certain kind of semi-supervised learning will mainly be considered: *Reinforcement learning* (**RL**). In RL, instead of labels for the data, its weak teacher only provides feedback on actions performed by the learner. Its framework can be understood by means of a decision maker (*agent*) performing in an *environment*. The agent makes observations in the environment (its input), takes actions (output) and receives rewards dependent on its actions. In contrast to classical machine learning approaches, the agent is also responsible for *exploration*, as it has to acquire its knowledge actively. Thus, a reinforcement learning problem is given if the only way to collect information about the *underlying model* of the environment is by interacting with it. As the environment does not explicitly provide actions the agent has to perform, its goal is to figure out the actions maximizing its cumulative reward until a distinctively defined *training episode* ends.

Markov Decision Processes

In the classical RL approach, the environment is divided into discrete time steps t. For every of those steps, the environment is expressed via a state s_t. Further, an environment allows for an agent's actions a_t. The future state s_{t+1} may be indeterministic, but depends on the history of previous states $s_0, .., s_t$ as well as the action a_t. For an environment to correspond to a *Markov Decision Process* (**MDP**), the *Markov property* must hold, which means that a state s_{t+1} depends only on the current state s_t and current action a_t: $p(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_0, a_0, .., s_t, a_t)$.
Formally, a MDP is a 5-tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, consisting of the following:

$$\mathcal{S} - \text{set of states } s \in \mathcal{S}$$
$$\mathcal{A} - \text{set of actions } a \in \mathcal{A}$$
$$P(s'|s,a) - \text{transition probability function from state } s \text{ to state } s' \text{ under action } a : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$$
$$R(r|s,a) - \text{reward probability function for action } a \text{ performed in state } s : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$$
$$\gamma - \text{discount factor for future rewards } 0 \leq \gamma \leq 1$$

Both the state transition function and the reward function may be indeterministic, which means that neither reward nor the following state are in complete control of the decision maker. Because of that, only expected values are examinable, depending on the random distribution of states. As the distribution of rewards depends deterministically on the state transition dynamics, a transition from s to s' under action a always yields the same reward r. I will refer to the actual result of a state transition at discrete point in time t as s_{t+1} and to the result of the reward function as r_t. If no point in time is explicitly specified, it is assumed that all variables use the same t.

An *offline learner* receives the problem definition as input in the form of a complete MDP, where the only task left is classifying actions yielding high rewards from actions giving suboptimal results. The task for an *online reinforcement learning* agent is a lot harder, as it has to learn the MDP itself via trial and error. In the process of reinforcement learning, the agent will encounter states s of the environment while performing actions a. Throughout this interaction, it receives rewards r, in part depending on its action.

In many RL problems, the full state of the environment is not known to the agent, and it only perceives an observation depending on the environment: $o_t := o(s_t)$[1]. This is referred to as *partial observability*, and the corresponding decision process is a *partially observed MDP*.

Additionally, the agent knows when a final state of the environment is reached, terminating the current training episode. For the agent, an episode therefore consists of a set of observations, actions and rewards ($\mathcal{S} \times \mathcal{A} \times \mathbb{R}$) until at time t_t some terminal state s_{t_t} is reached:

$$Episode := \big((s_0, a_0, r_0), (s_1, a_1, r_1), (s_2, a_2, r_2), .., (s_{t_t}, a_{t_t}, r_{t_t})\big)$$

Value of a state

In the process of reinforcement learning, the agent learns to perform as well as possible in the previously unknown environment. To do that, it uses an action-policy π, depending on a set of parameters θ. A policy performs a mapping from states to actions, which in the case of a *deterministic* policy leads to a distinct mapping $\pi_\theta(s) = a$. Though a stochastic policy is possible, it will not be considered for now[2]. As the agent does not have supervised data on which actions are optimal, it must learn some kind of measure for the value of being in a certain state or performing a certain action.

The commonly used measure for the value of a state when using policy π can be calculated by the immediate reward this state gives, summed with the expected value of the discounted future reward the agent will achieve by continuing to follow its policy π from this state on:

$$V^\pi(s_t) := \mathbb{E}_{s \sim \rho^\pi}\left[\sum_{t'=t}^{t_t}(\gamma^{t'-t} * r_{t'}^\pi)\right] \tag{2.1}$$

---

[1] From now on, when the state of the environment is meant, it will be explicitly referred to as s_e, while s is reserved for the agent's obvervation of the enviroment $o(s_e)$. Likewise, $\mathcal{S}$ will refer to the set of observations.

[2] Over its dependence on a, the result of both the reward function and the state transition function depend on π. To be explicit about that, I will refer to a reward dependent on π as r^π and a state transition dependent on π as s^π. If state or reward depends on an explicit action instead, I refer to it as r^a and s^a. Whenever not necessary for clarity, I will also drop π's dependence on θ.

It can easily be seen that the definition of value allows for a recursive definition:

$$V^\pi(s_t) = r_t^\pi + \gamma * \mathbb{E}_{s \sim \rho^\pi} \left[\sum_{t'=t+1}^{t_t} (\gamma^{t'-t} * r_{t'}^\pi) \right]$$
$$= r_t^\pi + \gamma * V^\pi(s_{t+1}) \tag{2.2}$$

This relation is known as the *Bellman Equation*, which allowed for the birth of dynamic programming[3]. While the state distribution depends on the agent's policy, it may be indeterministic[4]. The discounted state visitation distribution, which assigns each state a probability of visiting it according to policy π, is denoted ρ^π.

The actual, underlying value of a state $V^*(s)$ could accordingly be defined as the value of the state when using the best possible policy, which corresponds to the maximally achievable reward starting in state s_t:

$$V^*(s_t) := max_\pi V^\pi(s_t)$$

While *passive learning* simply tries to learn the value-function V^* without the need of action selection, an *active reinforcement learner* tries to estimate a good policy that can actually reach those high-value states. If the value of every state is known, the optimal policy can be defined as the one achieving maximal value for every state of the MDP: $\pi^* := argmax_\pi V^\pi(s) \forall s \in \mathcal{S}$.

Value of an action

While the definition of a state-value is useful, by itself it does not help an agent to perform optimally, as neither the successor function $P(s'|s, a)$, nor the reward function $R(r|s, a)$ is known to the agent. While so-called *model-based* reinforcement learning (also referred to as *Certainty Equivalence*) tries to learn both of those explicitly to reconstruct the entire MDP, *model-free* agents use a different measure of quality: the *Q-value*. It represents the expected value of performing action a_t in a state s_t, subsequently following the policy π:

$$Q^\pi(s_t, a_t) := \mathbb{E}_{s \sim \rho^\pi} \left[r_t^{a_t} + \gamma * V^\pi(s_{t+1}^{a_t}) \right] \tag{2.3}$$

with the Q-value Q^* of the optimal policy accordingly

$$Q^*(s_t, a_t) = \mathbb{E}_{s \sim \rho^\pi} \left[r_t^{a_t} + \gamma * V^*(s_{t+1}^{a_t}) \right]$$
$$= max_\pi Q^\pi(s_t, a_t)$$

For the Q-value, the Bellman equation holds as well: If the correct Q-value under policy π, $Q^\pi(s_{t+1}, a_{t+1})$, was known for all possible actions at time t, then the optimal action is the one maximizing the sum of immediate reward and corresponding Q-value. According to the *Principle of Optimality*[5], the value of the decision problem at time t can be re-written in terms of the immediate reward at t plus the value of the remaining decision problem at $t + 1$, resulting from the initial choices:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s \sim \rho^\pi} \left[r_t^{a_t} + \gamma * Q^\pi(s_{t+1}, \pi(s_{t+1})) \right] \tag{2.4}$$

---

[3]Dynamic programming is another solution strategy for MDPs. In contrast to RL it requires the complete MDP as input, thus corresponding to an offline learner.

[4]That is one of the reasons to discount future rewards: The agent cannot be fully sure if it actually reaches the states it strives for. Also, using the discounted reward generally helps making the agent perform good actions as quickly as possible.

[5]The definition of Bellman's principle of optimality states that *"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision"* (quote [5]).

As the value of a state is defined as the maximally achievable reward from that state, the relation between Q and V can be expressed as

$$V(s_t) = max_{a_t}Q(s_t, a_t) \tag{2.5}$$

Quality of a policy

Any agent's goal is to find a policy that can follow the trajectory of the state distribution with the highest expected reward. If Q-values are known, the optimal policy can be defined as the one maximizing the state-action value for each state:

$$\pi^* = argmax_a Q^*(s, a) \forall s, a \in \mathcal{S} \times \mathcal{A} \tag{2.6}$$

This policy guarantees maximum future reward at every state. Note however, that finding $argmax_a Q(s, a)$ is only easily possible if $\mathcal{A}$ is discrete and finite

As for the actual performance of a policy, a useful measure is the *performance objective $J(\pi)$*, which stands for the cumulative discounted reward from the start state using the respective policy. To measure the performance objective, it is necessary to integrate over the whole state space $\mathcal{S}$ with each state s weighted by its distribution due to π. As only non-stochastic policies are considered here, integration over the action space $\mathcal{A}$ is not necessary. The integral can, as shown by [29], be expressed by the expectation of the value of states following the distribution $s \sim \rho^\pi$:

$$\begin{aligned} J(\pi) &= \int_{\mathcal{S}} \rho^\pi(s) V^\pi(s) ds \\ &= \mathbb{E}_{s \sim \rho^\pi} \left[V^\pi(s) \right] \\ &= \mathbb{E}_{s \sim \rho^\pi} \left[Q^\pi(s, \pi(s)) \right] \end{aligned} \tag{2.7}$$

In short, this relation states the higher its expected Q-value for the states it visits, the better the policy. As however the underlying Q-function is not known, an agent must estimate it. As further the Q-value always also depends on the agent's policy, it must learn a policy π simultaneously to the corresponding estimates of Q^π.

If an agent knew Q^*, the *greedy* policy to always take the action yielding the highest value Q^*-for every state would be guaranteed to be optimal[6].

If the action-space $\mathcal{A}$ is finite and discrete, it is trivial to calculate the respective greedy policy from a given Q-function. As this is the easier scenario in comparison to its counterpart *policy gradient techniques*, it will be handled first in the course of this thesis.

In the case of a finite and discrete action-space, it is thus enough for a model-free RL agent to find a maximally precise estimate of Q, gradually improving its estimates and thereby its policy. To do that, it does not need to explicitly learn the reward- and transition function, but instead can model the Q-function directly. In RL settings with a highly limited amount of discrete states and actions, the respective Q-function estimate can be specified as a lookup table, whereas otherwise the function is estimated using a nonlinear function approximator.

In the following sections, the problem of finding a maximally precise estimate of Q^π will be covered. The agent's approximation of Q^π will be denoted Q^θ and greedy policy basing on this approximation is coherently as π_θ.

---

[6] In fact, the agent cannot act only according to the greedy policy, as it will need to *explore* the environment first. The problem of exploration will be considered later in this chapter.

2.2 Temporal difference learning

Throughout the reinforcement learning process, the agent collects more information about the environment, continually updating its estimate Q^θ of Q^π using samples from its episodes of interacting with it. To know how to improve its estimates, the agent must assess their quality. If the underlying Q-value Q^π was known, the agent could define an optimizable *loss function*, for example in the form of the squared difference $(Q^\theta - Q^\pi)^2$. As the agent does however not know Q^π, it performs *iterative approximation* to estimate it with continually *more informed samples* stemming from interaction with the environment.

Using the recursive definition of a Q-value given in the Bellman equation 2.4 allows for a technique called *temporal difference learning*[31]: At time $t+1$, the agent can compare its estimate of the Q-function of the last step, $Q^\theta(s_t, a_t)$, with a new estimate using the new information it gained from the environment: r_{t+1} and s_{t+1}. Because of the newly gained information from the underlying MDP, the new estimate will be closer to the actual function Q^π than the original value:

$$Q^\pi(s_t, a_t) = r_t + \mathbb{E}_{s \sim \rho^\pi} \left[\gamma * max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1}) \right] \tag{2.8}$$

$$\approx r_t + \gamma * r_{t+1} + \mathbb{E}_{s \sim \rho^\pi} \left[\gamma^2 * max_{a_{t+2}} Q^\theta(s_{t+2}, a_{t+2}) \right] \tag{2.9}$$

Keeping in mind that Q^θ is only an estimator of Q^π, it becomes clear that equation 2.9 is closer to the actual Q^π, as it incorporates more information stemming from the model itself.

The mean-squared error of equation $r_t + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ (the *temporal difference*) is generally known as the *temporal difference error*. In temporal difference learning, it gets minimized via iterative approximation. Even though $r_t + \gamma * Q^\theta(s_{t+1}, a_{t+1})$ also uses an estimate, it contains more information from the environment, and is thus a *more informed guess* than $Q^\theta(s_s, a_s)$. That makes it reasonable to substitute the unknown $Q^\pi(s_{t+1}, a_{t+1})$ by $Q^\theta(s_{t+1}, a_{t+1})$.

It is noteworthy, that each update of the Q-function using the temporal difference will not only affect the last prediction, but all previous predictions.

SARSA

The new knowledge about the environment can be incorporated in different ways, one of which is called SARSA. An agent using this algorithm samples a full tuple of $\langle s_t, a_t, r_t, s_{t+1}, a_{t+1} \rangle$ from its interaction with the environment, to then calculate the temporal difference error in non-terminal states as $TD := (r_t + \gamma * Q^\theta(s_{t+1}, a_{t+1})) - Q^\theta(s_t, a_t)$. SARSA is an example of *on-policy* temporal difference learning: In on-policy learning, the agent uses its own policy in every estimate of the Q-value. An important flaw of this method is, that it easily gets stuck in local optima if the agent's policy is not stochastic.

Q-learning

The *Q-learning* algorithm [35] stands in contrast to SARSA. When calculating the temporal difference error (its *learning step*), a Q-learning agent does not need to sample the action a_{t+1} because it calculates the Q-update at iteration i using the best possible action in state s_{t+1} according to the greedy policy $\pi_{greedy}^\theta(s) = argmax_a Q^\theta(s, a)$[7]. This policy does however not correspond to the one that is used to generate samples: In its *inference* step, Q-learning agents must utilize a stochastic policy in order to escape local optima and explore its environment. This policy is generally defined as a *soft* version of the greedy policy in question, which adds noise according to a certain definition (see section 2.5).

---

[7]A slight deviation from this is *double-Q-learning*, an architecture I will go into detail about later.

As a Q-learning agent thus uses a different policy to evaluate its Q-value and to act upon, Q-learning is considered an *off-policy* algorithm. A Q-learning agent learns about the deterministic policy $\pi := \pi_{greedy}^{\theta}$ while following a stochastic policy $\beta := soft(\pi_{greedy}^{\theta})$.

As the previous definition of Q-values was only correct in non-terminal states, a case differentiation must be introduced between terminal- and non-terminal states. In the following, y_t will stand for the updated estimate of the Q-value at t, sampling the necessary states, rewards and actions from interaction with the environment (which results in something very similar to the formula used in [20]). To express its dependence on the policy paramters θ, it will be superscripted:

$$y_t^{\theta} = \begin{cases} r_t & \text{if } t = t_t \\ r_t + \gamma * max_{a'} Q^{\theta}(s_{t+1}, a') & \text{otherwise} \end{cases} \quad (2.10)$$

The temporal difference error for time t is accordingly defined as

$$TD_t := y_t^{\theta} - Q^{\theta}(s_t, a_t) \quad (2.11)$$

A Q-learning agent must thus observe a snapshot of the environment, consisting of the following input: $\langle s_t, a_t, r_t, s_{t+1}, t+1 == t_t \rangle$ (where the last element is the information whether state s_{t+1} was a terminal state). That information is then used to calculate the temporal difference error.

In very limited settings, using the above error straight away allows for the update-rule in simple Q-learners: Consider an agent, specifying its approximation of the Q-function (its *model*) with a lookup-table Q^{θ}, initialized to all zeros. It is proven by [36] that for finite-state Markovian problems with non-negative rewards the update-rule for the Q-table (with $0 \leq \alpha \leq 1$ as the *learning rate*)

$$Q_{i+1}^{\theta}(s_t, a_t) \leftarrow \alpha * \left(r_t^{a_t} + \gamma * Q_i^{\theta}(s_{t+1}^{a_t}, a_{t+1}) \right) + (1 - \alpha) * Q_i^{\theta}(s_t, a_t) \quad (2.12)$$

converges to the optimal Q^*-function, making the greedy policy π^* optimal[6].

When using off-policy algorithms with π as the policy we learn about and β as the policy we act upon, our performance objective $J(\pi)$ (equation 2.7) must change, as it must incorporate that while the value of a state is calculated using π, the distribution of states follows from stoachastic policy β:

$$\begin{aligned} J_{\beta}(\pi) &= \int_{\mathcal{S}} \rho^{\beta}(s) V^{\pi}(s) ds \\ &= \mathbb{E}_{s \sim \rho^{\beta}} \left[Q^{\pi}(s, \pi(s)) \right] \end{aligned} \quad (2.13)$$

The learning step of reinforcement learning consists of two steps: *policy evaluation*, where the agent evaluates its current policy according to the knowledge gained from the environment, and, based on that *policy improvement*. In the standard Q-learning for discrete $\mathcal{A}$ considered so far, those steps are interleaved, leading to a form of *generalized policy iteration*: the Q-learner learns its action-value-function and its policy simultaneously. After updating its Q-function estimate via the temporal difference error, the agent updates its policy β as a *soft* version of the greedy policy $\pi_{i+1}(s) := argmax_{a'} Q^{\pi_i}(s, a') \forall s \in \mathcal{S}$. It is worth mentioning that this approach is generally limited because the operation $argmax_{a'} Q(\cdot, a')$ can only trivially be found in settings where $\mathcal{A}$ is finite and discrete, as it requires a global maximization over all possible actions. This problem is circumvented by splitting up policy evaluation and policy

improvement, as done by the architecture for continuous actions that will be explained in section 2.4.1.

Even if $\mathcal{A} \subseteq \mathbb{N}^n$, a Q-learner that uses tables as Q-function-approximator reaches its limits really fast, as the state space $\mathcal{S}$ may also be continuous or simply too big for a table to be useful. If that is the case, an update rule like in equation 2.12 becomes irrelevant quickly.

Instead, a better idea is to use the definition of the temporal difference error to define a loss function, which is to be minimized throughout the process of RL. A commonly used loss function is the *L2-loss*, because it allows for gradient descent, updating the parameters of the Q-function into the direction of the newly acquired knowledge. In this case, it may also be useful to calculate the loss of a batch of temporal differences simultaneously. The L2-loss for batch $batch$ with model-parameters θ_i, making up the policy π_i is thus defined as the following:

$$L_{batch}(\theta_i) := \mathbb{E}_{s,a,r \sim batch}\left[\left(y_{batch}^{\theta_i} - Qbatch^{\theta_i}(s,a)\right)^2\right] \tag{2.14}$$

2.3 Q-learning with neural networks

To understand this section, basic knowledge on how *Artificial Neural Networks* (**ANN**s) work and what they do is presupposed. Specifically, knowledge of *Convolutional Neural Networks* (**CNN**s)[40], mainly used in image processing, is required. As mentioned before, it is (in theory) not only possible to use a Q-table to estimate the Q^π-function, but any kind of function approximator. Thanks to the universality theorem, it is known that ANNs are an example of such[8]. The defining feature of ANNs in comparison to other machine learning techniques is their ability to store complex, abstract representations of their input when using a *deep* enough architecture.

2.3.1 Deep Q-learning

While for a Q-table the states and actions of the Markov decision process must be discrete and very limited, this is not the case when using higher-level representations like neural function approximators. If the agent's observation of a state of the game is high-dimensional (like for example an image), the chance for an agent to have the exact same observation twice is extremely slight. Instead, an Artificial Neural Network can learn a higher-level representation of the state, grouping conceptually similar states, and thus generalize to new, previously unseen states.

Deep-Q-Network (**DQN**) refers to a family of off-policy, online, active, model-free Q-learning algorithms for discrete actions using Deep Neural Networks. Using ANNs as function approximators for the agent's model of the environment requires a loss function depending on the Neural Network parameters, specified by θ[9]. As in the tabular case, the policy is straightforwardly defined depending on the $argmax_a$ of the Q-function. The update rule in Deep Networks depends on the gradient with respect to its loss, $\nabla_{\theta_i} L(\theta_i)$. As the DQN-architecture only considers discrete actions, there is one change that can be made in the definition of the Q-function: DQNs only receive the state as input, whereas the network returns a separate Q-value for each action $a \in \mathcal{A}$. This speeds up the inference, as one forward step is enough to calculate the Q-value of all actions in a certain state.

---

[8]For a proof of the universality theorem, I refer to chapter 4 of Michael A. Nielsen's book "*Neural Networks and Deep Learning*", Determination Press, 2015. The referred chapter is available at `http://neuralnetworksanddeeplearning.com/chap4.html` [accessed on 25th August, 2017]

[9]When neural network parameters are assumed, I will use the notation $Q(s, a; \theta_i)$ instead of $Q^\pi(s, a)$.

While there are attempts to use Artificial Neural Networks for Q-learning from as early as 1994[25], these showed success only in very limited settings, as they missed some key components of modern Deep-Q-Networks.

In standard online RL tasks, the update step minimizing the loss specified in equation 2.14 is performed not for a batch, but for each time t right after the observation occured to the agent. In those situations, the current parameters of the policy determine the next sample the parameters are trained on. It is easy to see that those consecutive steps of MPDs tend to be correlated: It is very likely that the maximizing action of time t is similar to the one at $t + 1$. Consecutive steps of an MDP are not representative of the whole underlying model's distribution. ANNs require independent and identically distributed samples, which is not given if the samples are generated sequentially. As shown by [13], the update using gradient descent is prone to feedback loops and thus oscillation in its result, never converging to an optimal estimate of the Q^π-function.

It was not until *Deepmind*'s famous papers in 2013[19] and 2015[20], that those issues were successfully addressed. One important step when using ANNs instead of Q-tables is to perform stochastic gradient descent using minibatches. In every gradient descent step of the neural network, neither only the last temporal difference error TD_t is considered (leading to oscillations), nor the entire sequence $TD_0, .., TD_{t_t}$ (because batch updates are not time-efficient in ANNs). Instead, as usual when dealing with ANNs, minibatches are sampled from the set of all observations. When performing the gradient descent step, the weights for the target y_t are fixed, making the minimization of the temporal difference error a well-defined optimization problem (with clear-cut target values as in supervised learning) during the learning step.

The two important innovations introduced in the DQN-architecture were the use of a *target network* as well as the technique of *experience replay*, which in combination successfully solved the problem of oscillating and non-converging action-value functions, even though still no formal mathematical proof of convergence is given.

Experience replay

As mentioned above, learning from only the most recent experiences biases the policy towards those situations, limiting convergence of the Q-function. To address this issue, the DQN uses an experience replay memory: Every percept of the environment (the $\langle s_t, a_t, r_t, s_{t+1}, t + 1 == t_t \rangle$ - tuple) is added to a limited-size memory of the agent. When then performing the learning step, the agent samples random minibatches from this memory to perform learning on a maximally uncorrelated sample of experiences. In the original definition of DQN, those minibatches are drawn uniformely at random, while as of today, better techniques for sampling those minibatches are available (see eg. [26]), increasing the performance of the resulting algorithm significantly.

Target networks

During the training procedure, the DQN-algorithm uses a separate network to generate the target-Q-values which are used to compute the loss (equation 2.14), necessary for the learning step of every iteration. Intuitively speaking, this is necessary because the Q-values of the *online network* shift in such a way, that a feedback loop can arise between the target- and estimated Q-values, shifting the Q-value more and more into one direction. The risk of such feedback-loops is decreased by using of a separate *target network* for calculating the loss. This is only periodically updated with the weights of the online network used for the policy to reduce the risk of correlations in the action-value Q_t and the corresponding target-value y_t (see equation 2.10).

The use of these two techniques leads to the Q-learning update rule which uses a loss as put forward in [20]:

$$L_i(\theta_i) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\left[\left(r + \gamma * max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s_t, a_t; \theta_i)\right)^2\right] \qquad (2.15)$$

Where i stands for the current network update iteration, θ_i^- for the current weights of the target network (updated every C iterations to be equal to the weights of the online network θ_i), $Q(\cdot, \cdot; \theta)$ for the Q-value dependent on an ANN using the weights θ, $\mathbb{E}[\cdot]$ for the expected value in an indeterministic environment, D for the contents of the replay memory of length $|D|$ containing $\langle s_t, a_t, r_t, s_{t+1}\rangle$-tuples, and $U(\cdot)$ for a uniform distribution.

As is the case with the experience replay mechanism, the usage of a target network was improved as well – modern algorithms do not perform a hard update of the target network every C steps, but instead perform *soft target network updates*, where every iteration, the weights of the target network are defined as $\theta_i^- := \theta_i * \tau + \theta_i^- * (1 - \tau)$ with $0 < \tau \ll 1$, first introduced in [15]. This improves the stability of the algorithm even more.

As a pseudocode for the DQN-architecture is already stated in the corresponding paper [20], listing it again here would be superfluous. Instead, this thesis compares the pseudocode with the code of the given implementation using *Python* and *Tensorflow* in appendix A.1. Note that the printed version has minor differences to the actual implementation for visualization purposes.

2.3.2 Double-Q-learning

It is well known that Q-learning tends to ascribe unrealistically high Q-values to some action-state-combinations. The reason for this is, that to estimate the value of a state s_i it includes a maximization step over estimated action values $Q(s_i, a)$ where naturally, overestimated values are preferred over underestimated values. It is not possible that Q-value-estimates are completely precise: estimation errors can occur due to environmental noise, inaccuracies in function approximation[10] and many other issues. Because Q-learning uses the maximum Q-value of state s_{i+1} in every estimate of the value of state s_i, only those estimates are propagated where the noise of the estimation is in a positive direction. Because of that, state s_{i-1} will have the accumulated upward noise from both state s_i and s_{i+1}, and so forth, which leads to unrealistically high action-values. While this would not constitute a problem if all Q-values would be uniformly overestimated, [10] showed that its very likely that the Q-value $Q(s_i, a)$ for only some actions a is overestimated – which changes the result of the $argmax_a$ operation and thus leads to biased policies and a worse performance.

The solution suggested by [10] is called *Double-Q-learning*. In its original definition without using Neural Networks as function approximators, a double-Q-learner learns two value-estimates in parallel, letting each experience update only one of them at random. In the learning-step, one function is used to determine the greedy policy, while the other is used to determine the value of this policy[11].

---

[10]Consider a flexible ANN trained on only a small sample so far - the ANN will overfit by covering all samples precisely. This overfitting leads to steep curves, over- or underestimating many values in between.

[11]*"In DoubleQ, we still use the greedy policy to select actions, however we evaluate how good it is with another set of weights"* (quote from [10]). The intuition behind that is, that the probability of both value-functions always overestimating the same actions is almost zero. The authors proved this intuition by finding that the lower bound of an overestimation is zero in the case of Double-Q-learning: $max_{a'} Q^\pi(s, a') - V^*(s) = 0$.

Deep-DoubleQ-Learning (**DDQN**) takes the Double-Q idea to the existing framework of Deep-Q-learning: Overestimations are reduced by decomposing the max-operation into *action selection* and *action evaluation*. Instead of introducing a second value function, DDQN re-uses the target-network of the DQN architecture in place of the second value function. To calculate target-values in the learning-step of a DDQN, the online network is used to choose an action, whereas the target network is used to generate the target Q-value for that action (instead of using the max-operation as in DQN). As shown by [10], DDQN improves over DQN both in terms of value accuracy and in terms of the actual quality of the policy.

When using Double-Q-Learning, the target-value of the calculation of the temporal difference error (y_t^θ from equation 2.10) must thus be re-defined as

$$y_t^\theta = \begin{cases} r_t & \text{if } t = t_t \\ r_t + \gamma * Q(s_{t+1}, argmax_{a'}Q(\ s_{t+1}, a'; \theta_i); \theta_i^-) & \text{otherwise} \end{cases} \tag{2.16}$$

It can be seen in appendix A.1 how small the actual change to normal DQN is: The only difference is the usage of the target network in line 21 of page 85 (marked in blue).

2.3.3 Dueling Q-learning

In many situations encountered during Q-learning, the value of all possible actions a_t in a state s_t is almost equal. Consider a simulated car, driving with full speed towards a wall, already so close that breaking or steering cannot stop the car from driving into the wall. As all actions will end the episode by the car hitting the wall, all actions will have roughly the same Q-value.

The idea of the *dueling architecture*[34] for DQN is to learn the Q-function more efficiently. For that, it splits up the Network into a *value stream* and separate *advantage streams*. As mentioned earlier, though the ANN resembles the $Q^\theta(s,a)$-function, the actions are not input to the network, but instead it outputs $|\mathcal{A}|$ Q-values, one for each action. A *Dueling Deep-DoubleQ-Network* (**DDDQN**) works by splitting an early layer of its corresponding network in half. One of the resulting streams is the value-stream, which results in one value, intuitively corresponding to the value ($V_{s_t}^\theta$) of a state s_t, irrespective of the action that can be taken in this state. The other stream is the advantage stream, which outputs $|\mathcal{A}|$ values, standing for the *Advantage* (A) of taking a certain action in this state – it can be seen as a relative measure of the importance of each action. The relation between Q, V and A is the following:

$$Q(s,a) = V(s) + A(s,a)$$

In the very last layer of the ANN those two streams are combined again, so that a DDDQN also outputs one Q-value for each action. Thanks to this, any DDQN can be transformed into a DDDQN by simply changing the structure of the used ANN. It is important to mention however, that the last layer cannot simply calculate $V(s) + A(s,a)$ – If that was done, then the network could simply set the V-stream to a constant, negating any advantage gain of splitting them up in the first place. However, as explained in equation 2.5, it holds that $argmax_{a'}Q(s,a') = V(s)$ – when using deterministic policies, the value of the state corresponds to the Q-value of the best action that can be taken in this state. Therefore, it must be the case that the $argmax$-action has an advantage of zero. This can be used to calculate the $Q(s,a)$ using the value-stream and the advantage-stream according to the following equation (where θ corresponds to the shared weights, θ^A to the weights specific to the advantage-stream and θ^V to those specific to the value-stream):

$$Q(s,a;\theta,\theta^A,\theta^V) = V(s;\theta,\theta^V) + \big(A(s,a;\theta,\theta^A) - max_{a'}A(s,a',\theta,\theta^A)\big)$$

As it turned out that normalizing with respect to all actions lead to better performance, the actually used normalizer is $\frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \theta^A)$ instead. The advantage gained by splitting into two streams can be described as the following: In the learning step of a DQN, the derivative is taken with respect to difference in the expected Q-value of an action and the better estimate of that Q-value. In normal DQNs, the network can thus only update the parameters responsible for one of its outputs. A DDDQN learns more effectively, as it learns a shared value of multiple actions: A temporal difference in one Q-value likely changes the value-stream of the network, which also changes the Q-value of other actions. Thanks to this, learning generalizes better across actions, without any changes to the underlying reinforcement learning algorithm. As shown by the authors [34], the difference is especially drastic in situations where many actions have similar outcomes.

In appendix A.1, an exemplary source code of the implementation of an agent using Python and TensorFlow is listed. The agent stands alone without an environment, and some crucial parts of it, like an implementation of its memory, are left out. Further, the printed version has minor differences to the actual implementation for visualization purposes. However, all aspects mentioned in the previous sections are included, which is the algorithm of the actual Q-learning as described above and in [10], as well as the computation graph of the network using precisely the DDDQN-architecture as described above and in [34]. In the first two pages, the necessary definitions of agent and network structure are introduced, before in page 85 there is an actual comparison between the pseudocode of a DDDQN (as found in [20], with the changes from [10] and [15] incorporated and marked in blue) and its correspondences in the actual code, namely the `__init__` , `inference` and `q_train_step` -functions of the model-class. Each line of the pseudocode (to the left) corresponds precisely to the respective line of the actual python-code (to the right).

2.4 Policy gradient techniques

Policy Gradient (**PG**) techniques [32] are in principle a far more straight-forward approach to reinforcement learning than temporal difference-methods like Q-learning. The idea behind PG techniques is really simple: The policy $\pi_\theta(a|s) = \mathbb{P}[a|s; \theta]$ of an agent is explicitly modeled using a differentiable function approximator, like a neural network with weights θ. While approximating state-action value is not needed, a policy's performance could be assessed as in equation 2.13.

PG methods optimize for a different quality than Q-learning algorithms: the *advantage A* of an action. It can be defined analogously to the value of a state in equation 2.1, with the difference that this time, the value of the following states is not modelled and can only be calculated in retrospect after having measured all of the respective rewards from the environment:

$$A_t := \sum_{t'=t}^{t_t} (\gamma^{t'-t} * r_{t'}) \tag{2.17}$$

To use the policy gradient, the agent must calculate the advantage of every state it visited so far, to set this advantage as the gradient of the output of its policy function of the corresponding state, to then train the network using backpropagation with respect to this gradient. If the reward is positive, gradient *ascent* will make the network more likely to produce this action in this state. If the gradient was negative, it will lead to gradient *descent*, discouraging the network to repeat this action in the given state. Thus, the network will learn to repeat actions giving high rewards and to avoid those with negative rewards – this gradient corresponds to the loss $\sum_t A_t \, log \, p(a_i|s_i)$: If A_t is positive, we want to increase the probability of action a_t for

state s_i, and decrease it otherwise.

The main problem of purely using this approach is however the huge amount of exploration that is needed – the agent does not have any knowledge about what states are good or bad, but only increases the probability of individual actions that turned out to have a good score. Because of this, it can find good policies only by chance, after millions of iterations. If the agent knew the value of a state, it could optimize its policy in the direction of actions that it knows produce a high-valued state. For that, the policy network must get the gradient information of the action from something estimating this value-function, giving rise to *actor-critic architectures*.

2.4.1 Actor-Critic architectures

As explained, the previously introduced DQN-algorithm learns via temporal differences the state-action value $Q^{\pi_{greedy}}(s, a)$ for the states it encountered with its current policy π. Being a kind of *generalized policy iteration*, the policy is improved in the same step by defining $\pi_{i+1}(s) := soft(argmax_a Q^{\pi_{greedy}}(s, a) \forall s \in \mathcal{S})$. While discretizing the action space to ease the calculation of this maximum, it gives rise to the *curse of dimensionality*, especially when the discretization is fine grained – an iterative optimization process like the $argmax$-operation would thus likely be intractable. An actor-critic algorithm is a hybrid to combine the PG method and the value function. As hinted at before, it consists of two components, one estimating the policy and another one to improve it.

An alternative to discretizing continuous action-spaces $A \subseteq \mathbb{R}^n$ is moving the policy into the *direction of the gradient of Q*. For that, it is necessary to model the policy explicitly with another function approximator. This gives rise to *actor-critic* architectures, where both policy and Q-function are explicitly modeled: The *critic* corresponds to a Bellman-function-approximator, using temporal differences to estimate the action-value $Q^{\pi}(s, a)$[12]. In contrast to the the familiar DQN however, the policy is now explicitly modeled by the *actor*. In the case of a stochastic policy, it would be represented by a parametric probability distribution $\pi_\theta(a|s) = \mathbb{P}[a|s; \theta]$, however here we only consider the case of deterministic policies $a = \pi_\theta(s)$, which takes the necessity of averaging over all possible actions when calculating its performance objective according to equation 2.7 or equation 2.13. Note however, that using deterministic policies will (again) lead to the necessity of off-policy algorithms, as a purely deterministic policy does not allow for adequate exploration of state-space $\mathcal{S}$ or action-space $\mathcal{A}$. Thus, to measure the performance of our policy, we must use function 2.13, which averages over the state distribution of our behaviour policy $\beta \neq \pi$.

To train both actor and critic, actor-critic algorithms rely on a version of the *policy gradient theorem*, which states a relation between the gradient of the policy and the gradient of its performance function. The idea behind actor-critic policy gradient algorithms is to adjust the parameters θ of the policy in the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$, as moving uphill into the direction of the performance gradient corresponds to maximizing the global performance of the policy[13].

---

[12]In that, it corresponds precisely to the previous approach. However, as we now allow for a continuous action-space $\mathcal{A}$, the network cannot return multiple Q-values at once, for each action $a \in \mathcal{A}$. Instead, the actions must also be inputs to the critic, which then outputs one $Q(s, a)$-value.

[13]The original policy gradient, introduced in [32], assumes on-policy learning with a stochastic policy. However, to derive the stochastic policy gradient, one must integrate over the whole action-space, making its usage less efficient and requiring more training than the DPG, introduced here.

Deterministic Policy Gradient

The idea in the *Deterministic Policy Gradient* (**DPG**) technique is to use a relation between the gradient of the (deterministic) policy (represented by the actor), and the gradient of the action-value function Q. The existence of a relation is easy to see, because as introduced in equation 2.13, the performance of our policy π is measured using Q. The DPG technique is the policy gradient analogue to Q-learning: It learns a deterministic greedy policy in an off-policy setting, following a trajectory of states due to a noisy version of the learned policy.

The *off-policy deterministic policy gradient theorem*, put forward in [29], states the following relation between the gradient of the performance objective of a policy $J(\pi)$ (see equation 2.13) and the gradients of the policy-function π and the action-value function Q.

$$
\begin{aligned}
\nabla_\theta J_\beta(\pi_\theta) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) ds \\
&= \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \pi_\theta Q^\pi\left(s, \pi_\theta(s)\right)\right] \\
&= \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)\big|_{a=\pi_\theta(s)}\right]
\end{aligned}
\tag{2.18}
$$

There are two important things about this relation: first, it can be seen that the policy gradient does not depend on the gradient of the state distribution. Second, the approximation drops a term that depends on the action-value gradient, $\nabla_\theta Q^\pi(s,a)$. Since the positions of the optima are however preserved, the term can be left out, making the approximation no less useful. Both claims are shown in [29].

The practical implication of this relation is the following:
When updating the policy, its parameters θ_{i+1} are updated in proportion to the gradient $\nabla_\theta J(\pi_\theta)$. In practice, each state suggests a different gradient, making it necessary to take the expectation w.r.t. the state distribution ρ^β:

$$
\theta_{i+1} = \theta_i + \alpha * \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta J(\pi_\theta)\right]
$$

The deterministic policy gradient theorem (equation 2.18) shows that to improve the performance of the policy, it makes sense to move it into the direction of the gradient of Q, where the gradient of Q can be decomposed into the gradient of the action-value with respect to the actions, and the gradient of the policy with respect to its parameters:

$$
\theta_{i+1} = \theta_i + \alpha * \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^{\pi_i}(s,a)\big|_{a=\pi_\theta(s)}\right]
$$

In other words, to maximize the performance of the policy, one can re-use the gradient of those actions leading to maximal Q-values. In practice, the true function $Q^\pi(s,a)$ is unknown and must be estimated.

The off-policy deterministic actor-critic algorithm learns a deterministic target policy $\pi_\theta(s)$ from trajectories generated by an arbitrary stochastic policy, $\beta(a|s) = \mathbb{P}[a|s]$. For that, it uses an actor-critic arcitecture: The critic estimates the Q-function using a differentiable function approximator, using Q-learning as explained in the sections above, with the weights specified as w. The actor updates the *policy* parameters θ in the direction of the gradient of Q (instead of maximizing it globally as in the sections above[14]. Note that for the update of the policy

---

[14]*"The critic estimates the action-value function while the actor ascends the gradient of the action-value function"* (quote [29])

parameters, only the gradient with respect to the weight of the actions as input of the Q-functions is relevant, not the trained weights of the approximator Q^w themselves. Figure 2.1 visualizes the idea behind the algorithm graphically.
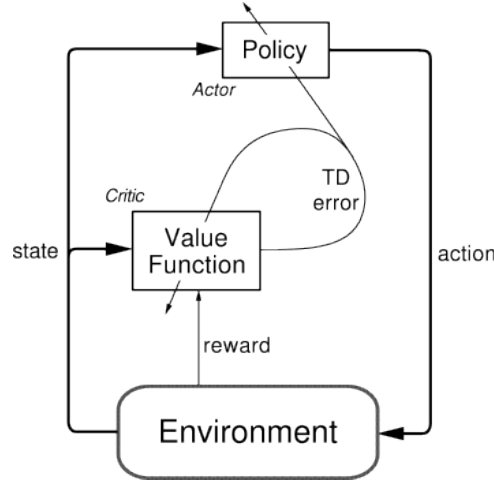

FIGURE 2.1: The actor-critic architecture. Reprinted from [24].

Using the above knowledge, one can derive the *off-policy deterministic actor critic* algorithm. In the first step of this algorithm, the critic calculates the temporal difference error to update its own parameters like in previous sections, and then the actor updates its parameters in the direction of the critic's action-value gradient:

$$TD_i = \mathbb{E}_{s,a,r} \left[\left(r_t + \gamma * Q^{w_i}(s_{t+1}, \pi_\theta(s_{t+1})) \right) - Q^{w_i}(s_t, a_t) \right] \tag{2.19}$$

$$w_{i+1} = \mathbb{E}_{s,a} \left[w_i + \alpha_w * TD_i \nabla_w Q^w(s_t, a_t) \right] \tag{2.20}$$

$$\theta_{i+1} = \mathbb{E}_{s,a} \left[\theta_i + \alpha_\theta * \nabla_\theta \pi_\theta(s_t) \nabla_a Q^w(s_t, a_t) \big|_{a=\pi_\theta(s)} \right] \tag{2.21}$$

With α_w and α_θ as the learning-rates of the critic and the actor, respectively.

As stated by [29], these algorithm may have convergence issues in practice, due to a bias introduced by approximating $Q^\pi(s, a)$ with $Q^w(s, a)$. It is thus important, that the approximation $Q^w(s, a)$ is *compatible*, preserving the true gradient $\nabla_a Q^\pi(s, a) \approx \nabla_a Q^w(s, a)$. This is the case when the gradients are orthogonal, and w minimizes $MSE(\theta, w)$. However, the necessary conditions are approximately fulfilled when using a differentiable critic that finds $Q^w(s, a) \approx Q^\pi(s, a)$.

Deep DPG

The *Deep DPG Algorithm* is an off-policy actor-critic, online, active, model-free, deterministic policy gradient algorithm for continuous action-spaces. The basic idea behind *Deep DPG* (**DDPG**)[15] is combining the ideas of the DQN (section 2.3.1) with the architecture and learning rule using the deterministic policy gradient. For that, they also use parameterized deterministic actor function $\pi_\theta(s) = a$, as well as a critic function $Q^w(s, a)$. As the algorithm is also off-policy, it will learn the policy π_θ, while following a trajectory arising through another, stochastic policy β. This policy will again be a *soft* version of the learned policy π that allows for adequate exploration: $\beta := soft(\pi)$.

The update of the critic is performed analogously to the Q-value approximator in the Deep-Q-Network architecture. A minibatch of $\langle s_t, a_t, r_t, s_{t+1}, t == t_t \rangle$-tuples is sampled from a replay memory of limited size, to then perform Q-learning via temporal differences (see equations 2.14 and 2.15). An obvious difference to the Q-learning in the above sections is however,

that not the greedy $argmax_{a'}Q(s, a')$-policy is used in the determination of the target value, but the agent's own parameterized policy $\pi_\theta(s)$. Just like in Deep-Q-Learning, it is necessary to use target networks to ensure convergence of Q. Lillicrap et. al. [15] were the first to use the previously mentioned soft target updates.

The update of the actor then follows the deterministic policy gradient theorem from equation 2.18: Its estimation of the policy gradient is based on the minibatch-samples used in the critic. For that, it calculates the expectation of the action-gradients it adopted from the critic network. This expectation is an approximation of its policy gradient, allowing the actor to perform a stochastic gradient ascent step to optimize its performance objective.

In practice, it turned out that the usage of target networks for both actor and critic is necessary to ensure stability of the algorithm, such that in practice, there are four different networks: the actors $Q(s, a; \theta^Q)$ and $Q(s, a; \theta^{Q^-})$ as well as the critics $\pi(s; \theta^\pi)$ and $\pi(s; \theta^{\pi^-})$. Incorporating all those changes leads to the following pseudocode excerpt of the agent's learning step, adopted from [15]:

Target-value of the critic:

$$y_t^\theta := \begin{cases} r_t & \text{if } t = t_t \\ r_t + \gamma * Q\big(s_{t+1}, \pi(s_{t+1}; \theta^{\pi^-}); \theta^{Q^-}\big) & \text{otherwise} \end{cases} \tag{2.22}$$

Loss the critic minimizes:

$$L_i(\theta_i^\pi) := \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[\big(y_t - Q(s_t, a_t; \theta^Q)\big)^2\Big] \tag{2.23}$$

Sampled policy gradient the actor maximizes for:

$$\nabla_{\theta^\pi} J_\beta(\pi_\theta) \approx \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Big[\nabla_a Q(s_t, \pi(s_t); \theta^Q)\nabla_{\theta^\pi}\pi(s_t; \theta^\pi)\Big] \tag{2.24}$$

The correspondences of this ANN implementation to the definitions can easily be seen: Equations 2.22 and 2.23 correspond to definitions 2.19 and 2.20, whereas equation 2.24 corresponds to equation 2.21.

For a complete code of the DDPG-implementation as developed in the course of this thesis, it is referred to appendix A.2. Analogously to the DQN-agent, an exemplary source code of the implementation of an agent with a DDPG-model can be found there, using *Python* and *TensorFlow*. The agent stands alone without an environment, and some crucial parts of it, like an implementation of its memory, are left out. On page 88, there is again a comparison of the pseudocode (as provided in [15]) and the correspondences in actual python-code, where each line of the pseudocode (to the left) corresponds precisely to the respective line of the actual python-code (to the right). Note that this listing does not contain a definition of a class *agent*, as it is the same defined as in lines 1-48 of appendix A.1.

2.5 Exploration techniques

There is one main difference between supervised learning and reinforcement learning, mentioned right at the beginning of this chapter: in RL, the only way to collect information about the environment is by interacting with it. It is easy to see, that as long as only deterministic policies $\pi(s) = a$ are considered, the distribution of states that an agent visits is very restricted, as no *exploration* of the state space S of the MDP is given. Once the agent found one trajectory to a terminal state, it will continue *exploiting* this path, which almost gurantees a suboptimal

solution. In order to explore the full state space instead of sticking with the first local optimum found, a stochastic, non-greedy policy is necessary.

In supervised learning it is no problem to shuffle the dataset beforehand, giving the learner independent and identically distributed samples, certainly representative about the entire dataset. In RL however, this is not possible: Consider an artificial agent driving a simulated car around a track – as long as the agent does not perform well enough to reach high speeds, it will not learn anything about such states.

The advantage of off-policy algorithms such as Q-learning and the DPG is, that the policy an agent learns does not need to be the one it follows. The mechanism that ensures exploration can thus be independently handled for both of the techniques with the previously defined $soft(\pi)$-function, that takes as input a greedy, deterministic policy and yields its stochastic counterpart.

A big problem of reinforcement learning is the tradeoff between *exploration* and *exploitation*. At the beginning of training, an agent is supposed to explore the environment as good as it can – in this situation, a highly stochastical exploration function is required. At later stages of training however, once the agent knows most of the state dynamics, exploiting its learned policy becomes more important to generate corresponding training samples (for example high speeds). Because of that, most exploration techniques use an exploration parameter ϵ that decreases over time. This is called *simulated annealing*.

In the following section, I will start with exploration techniques for discrete actions, before then explaining methods that can be used in the case of continuous action spaces $\mathcal{A}$.

2.5.1   Exploration for discrete action-spaces

The easiest way to incorporate exploration is the ϵ-*greedy* approach. In this, an agent uses its greedy policy with a probability of $1 - \epsilon$, and a purely random policy with a probability of ϵ. With ϵ decreasing over time, it is ensured to explore the environment at first, to later on exploit the greedy policy.

While ϵ-greedy is the most popular approach (also used in the original DQN), it has some obvious shortcomings. Instead of being purely random, there are properties of an exploration function that may be desirable.

For instance, as in many scenarios a single action is almost irrelevant, *temporally correlated* random actions could help exploring new state trajectories. Doing so also helps against jerking movement.

Another idea is to incorporate the knowledge of an agent not by either performing its best action or a random one by chance, but by preferring ones that the greedy policy would *almost* take. The *Boltzmann exploration* function interprets the individual Q-values as a *softmax*-probability distribution over the actions. In doing so, actions with a high value are more likely to be chosen. To ensure later exploitation, it controls the spread of the softmax-distribution with a parameter τ that is annealed over time. The Boltzmann exploration function is advantagous over ϵ-greedy in situations where certain actions are almost certainly fatal (as for example an extreme turn of a self-driving car when driving at high speeds).

A further property that may be demanded is only incorporating exploration in unknown states while exploiting the learned policy in others. In tabular settings, *count-based* exploration techniques are easily possible. *Model-based Interval Estimation with Exploration Bonuses* (MBIE-EB) algorithms for example solve an extended Bellman-equation that incorporates en exploration bonus for states that have a low state-visit count, driving the agent to *reduce its uncertainty* by visiting less frequently visited states. In high-dimensional, non-tabular settings however, the actual state-visit count becomes almost meaningless, as states are rarely visited more than once.

In [4], the authors propose a *pseudo-count*, derived from a learning-positive density model over the state space. These are subsequently used in a variant of MBIE-EB, adapted with the techniques of the DQN. The used density-model must fulfill certain criteria, as being linearly increasing, generalizing across states and being roughly zero for novel states. In their paper, they use the *Context Tree Switching* density model for two-dimensional images. This density model can also be replaced by a neural network.

Another idea, proposed in [17], is computing the generalized state visit-count from the *feature-space* of the value function. In their algorithm, they propose to use the feature representation of a higher layer of the neural Bellman approximator for the pseudocount – states that have less frequently observed features are deemed more uncertain. Once such a count is given, an exploration-bonus can be incorporated to explore states with infrequent features. As this algorithm computes the employed similiarity measure effienctly, their algorithm is less computationally expensive than [4]. Further, adding it to an existing ANN is rather easy, as its only a function of its final layer – the same feature dimension that is used to measure a state's value is also used to measure its uncertainty. In the approach given in [17], the similiarity measure for the feature-visit density is calculated by the generalized number of co-occurences of its components: It is the product of independent factor distributions over the vector components $\rho_t(\phi) = \prod_{i=1}^{M} \rho_t^i(\phi_i)$, where for the factor model $\rho_t^i(\phi_i)$ a count-based estimator is suggested. This model assigns higher proability to state feature vectors that share features with already visited states. A state-visit pseudocount is calculated from this density model accordingly to [4]. Note that the presented (pseudo-) count based algorithms are straight-forward applicable to continuous action-spaces as well.

2.5.2 Exploration for continuous action-spaces

Switching from discrete to continuous action-spaces allows for new exploration techniques. In the general discrete case, each possible action must be treated as isolated from the others, as no assumption about semantic relatedness of actions (for example by their distance) can be made.

This is different in continuous action-spaces, where a random action can be calculated as a noisy version of the action as provided by the agent's policy. Considering the case of autonomous driving, one output of the network can correspond to the car's steering. Instead of using a purely random action, gaussian noise can be added to this steering command, such that a random action is not too far off the original action – completely random actions are not performed anymore, only noisy versions of the desired ones:

$$soft(\pi^\theta(s_t)) := \pi^\theta(s_t) + \mathcal{N}$$

Selecting noise from this distribution is however still not optimal for self-driving cars or robotics: consecutive actions are selected independently at random, which can lead to them being excessively far from one another, and the control signal discontinuous. It is easy to see that this leads to *jerky* movements, which may ultimately lead to the destruction of an agent or a fast driving car.

A solution for that is *autocorrelated noise*, as for example proposed in [37], which selects the noise based on a stochastic process dependent on a running average. For that, a policy-action is selected as $a_t = \pi^\theta(s_t, \xi_t)$, with $\xi_t \in \mathbb{R}^n$ as the random element. In standard RL processes, ξ_t is identically distributed and stochastically independent for all t, meaning that the noise is not autocorrelated.

The addition of [37] is providing a noise function ξ_t that has the same distribution for each t, with ξ_{t+1} dependent on ξ_t, forcing consecutive actions to be close to each other. For that, it incorporates a moving average:

$$\xi_t = \sum_{i=0}^{M-1} \zeta_{t-i}$$

With M corresponding to the sliding window for the moving average. For $M = 1$, this corresponds to the traditional control policy. It is worth noting, that the variance of noise function is dependent on the temporal discretization – the coefficient needs to be adjusted for different levels of temporal discretization. Wawrzyński [37] provides extensive explanation on how to select parameters appropriately. In general, a control policy based on this has much less jerky movements, even though a fully smooth function is impossible for any discretization of time. Using autocorrelated noise is advantagous in situations where single actions do not have much influence on their own: Due to the temporal correlation, a feedback loop may arise in the noise function, leading to a whole new trajectory of states to be explored.

Similar reasoning leads to the exploration function used in [15]. To incorporate autocorrelated noise, an Ornstein-Uhlenbeck process (see [33]) was used. Such a process was originally created to model the velocity of particles with friction, making it very suitable to for physical processes with inertia. An Ornstein-Uhlenbeck process has mean-reverting properties, which means that it keeps a running average of the *noisestate*, as well as a parameter Θ that decides how fast the *noisestate* reverts towards the desired mean μ. Additionally, it keeps a parameter σ, controlling the volatility. Using this process, an action is calculated like the following:

$$noisestate = \Theta(\mu - noisestate) + \sigma * \mathcal{N}$$
$$a_{noisy} = a + \epsilon * noisestate$$

An advantage of this function is, that it is possible to treat Θ, σ and μ as vectors, providing different properties for each individual action.

3 Related work

The aim of this thesis is to build a good agent for simulated self-driving cars. While this constitutes one task, it requires another one, namely to transform a specific game into a reinforcement learning problem. To show what such environments usually look like, the first section of this chapter elaborates on related work of that domain. The second section will deal with related work in the domain of self-driving cars, starting with real-life scenarios before leading over to algorithms for comparable simulations.

3.1 Reinforcement learning frameworks

The general structure of a reinforcement learning problem as a **POMDP** was outlined in the previous chapter. To summarize, the main interaction between agent and environment is depicted in figure 3.1.
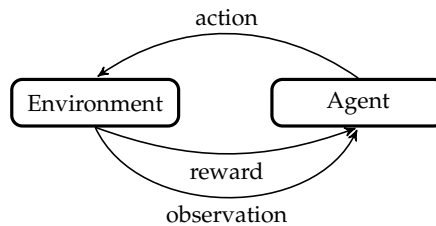
FIGURE 3.1: Interaction between agent and environment in RL

3.1.1 OpenAI gym

When developing RL agents, agent and environment must allow for a dataflow as described in the above figure. In the original Deep-Q-Network [19] as well as in its follow-ups [10, 34], the agents were trained on several ATARI games using the *Arcade Learning Environment* (**ALE**) [3]. This environment converts the ATARI-games into partially observed reinforcement learning problems, therefore providing a simple common interface to over a hundred different tasks. Doing that, it provided the accumulated score so far (corresponding to the reward), the information whether game ended (indicating the end of a training episode), as well as a 160×210 2D array of 7-bit pixels (corresponding to the agent's observation). As the game screen does not correspond to the internal state of the simulator, the ALE corresponds to a POMDP. Environments with discrete actions only are however severely limited, and most of the interesting real-world applications, as for example autonomous driving, require real-valued action spaces. The test scenarios for the Deep-DPG algorithm consisted thus of a number of simulated physics-tasks, using the *MuJoCo* physics environment.

Both of the aforementioned environments are by now, among many others, merged into the *OpenAI gym*[1] environment [8], a toolkit helping reinforcement learning research by including a collection of benchmark problems with a common interface.

---

[1] https://gym.openai.com

The goal of OpenAI gym is to be as convenient and accessible as possible. For that, one of their design decisions was to make a clear cut between agent and environment, only the latter of which is provided by OpenAI. The exemplary sourcecode found in algorithm 1, taken from `https://gym.openai.com/docs`, outlines the ease of creating an agent working in the gym framework.

```python
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
```

ALGORITHM 1: Interaction with the OpenAI gym environment

The code outlines how the general dataflow between agent and environment usually takes place: After a reset, the environment provides the first *observation* to the agent. Afterwards, it is the agent's turn to provide an action. Even though not featured in this simple example, generally performed under usage of the observation. Once an agent has calculated the action and provided it to the environment, it can perform another simulation step, with the tuple $\langle observation, reward, done, info \rangle$, corresponding to s_{t+1}, r_{t+1} and $t + 1 == t_t$ from section 2.2 in addition to debug-information $info$. In the remainder of this work, I will refer to this dataflow as a baseline on how the interaction of environment and agent could look like.

3.1.2 TORCS

TORCS is short for *The Open Source Race Car Simulator*[38, 39]. It is a multi-agent car simulator, used as research platform for general AI and machine learning. The implementation is open source and provides an easy way to add artificial driving agents as components of the game and assess their performance.

On top of TORCS, several APIs exist to provide a common interface for agents, such that they can communicate with the game while being in a separate thread, even for agents programmed in other programming languages. TORCS is also incorporated in OpenAI's gym platform, even though accessing it from there is a non-trivial task.[2].

Another approach is given for the *Simulated Car Racing Championship Competition*, as presented in their manual[16]. In that framework, agents communicate over a *UDP*-connection with the game-environment. To do so, the game functions as a server that sends observations of the current game-state to connected agents in an interval of $20ms$. Further, it provides an abstract *BaseDriver* class. Agents that extend this class by implementing the methods to *init* and *drive* can thus communicate as a client with the TORCS-environment, receiving an observation of the game-state and sending their action using a UDP-connection. This approach creates a physical sepearation between game engine and agents, which can thus even run over remote

---

[2]The GitHub-repository `https://github.com/ahoereth/ddpg` [accessed on 7th September, 2017] provides an instruction how to install and use the TORCS-environment in Python using OpenAI gym.

machines. To develop such an agent, no knowledge of the TORCS engine or the game-internal data is necessary. In this thesis, a very similar approach will be taken, where game and agents run in different threads and communicate over sockets.

Further, the game data streamed by this environment is much sparser than what is used in the approach developed in the course of this thesis. It is however worth noting that much of the game data that is streamed from game to agent overlaps with it. The discussed manual [16] contains a table providing detailed overview of the vectors sent to an agent (denoted *sensors*).

3.2 Self-driving cars

As mentioned in the introduction, the overall driving problem can be split into many subcomponents, not all of which are relevant for this thesis. For example, while assessing the *driver's state* is necessary in semi-autonomous vehicles, the used approach does not consider a driver.

There is a lot of progress currently being made in the realm of *scene detection and scene understanding*. While many of these approaches utilize many recent advances of machine learning and artificial neural networks, giving an overview of those would be far beyond the scope of this thesis.

As mentioned in chapter 2, reinforcement learning algorithms are used when the transition dynamics of the environment is unknown. If complete knowledge of the racing problem was given, optimal control motion-planning algorithms could be used to solve the problem of movement planning. An example of an asymptocially optimal algorithm that does so is the sampling-based RRT^* algorithm, short for *rapidly-exploring random trees*. In [12], the authors use this algorithm to generate optimal motion policies, given complete knowledge of the physics and concrete starting conditions. They use the motion planning method to generate optimal trajectories for minimum-time maneuvering of high-speed vehicles, finding the fastest policy that drives without any collisions. In contrast to previous optimal control methods, their system runs in real-time, given enough computing performance.

3.2.1 Supervised learning

Knowing the full underlying physics is nearly impossible, and even if it was known, optimal control is computationally very complex and unlikely to be incorporated in actual driving agents. Therefore, it is interesting to focus on the overall racing strategy in an *end-to-end* fashion, combining trajectory planning, robust control and tactical decisions into one module. Further, it makes sense to learn the problem automatically, without the need of handcrafting a solution for every imaginable situation. The idea of these end-to-end approaches is to automatically learn internal representations of road features, optimizing all processing steps simultaneously. The hope is that the learned features are better representations than hand-crafted criteria like lane-detection, used for the ease of human interpretation. In end-to-end approaches using neural networks, no clear differentiation between feature extraction and controlling can be made as the semantics of the individual network layers remain largely unknown.

One of the first approaches to learn how to drive using an end-to-end neural network is the *Autonomous Land Vehicle In a Neural Network*, short ALVINN[23]. Published as early as 1989, it uses a three-layer neural network to directly learn steering commands from a front-facing 30×32 pixel gray-scale camera as well as a matrix of 8×32 values from a laser range finder as input. The steering-output it produces (it does not learn acceleration or braking) is discretized into a smoothed one-hot vector of 45 units. After training with artificially generated data, it learned with an accuracy of 90% in simulations. In a real testing, it drove a real car for 400 meters at a speed of $1.8km/h$.

A modernized version, doing essentially the same thing with modern techniques and far more computing power is NVIDIA's *End to End Learning for Self-Driving Cars*[7]. In this approach, a convolutional neural network producing direct steering commands from a single front-facing camera was used. For that, a labelled dataset of 72 hours of real driving data was collected to train a 9-layer convolutional network (1 normalization layer, 5 convolutional layer, 3 dense layers), producing the steering command as a single output neuron (hence continuous, but again no throttles or brake). The network was trained supervisedly, minimizing the mean-squared error between the output and the command of the human driver (as saved in the dataset). To remove bias towards driving straight, the training data included a higher proportion of frames representing curves. The performance of the resulting network was tested in a simulation that presented testing data to the network, comparing the produced steering to the real driving command. In testings, the simulated car had statistically two interventions per ten minutes of driving. It is worth noting, that creating a huge labelled dataset postulates no problem in modern times anymore[3].

Table 3.1 summarizes some known supervised approaches to self-driving cars, listing their model, input, output and optimization function.

Project	trained on	model	input	output	optimization function
ALVINN[23]	Manually created dataset	three-layer neural network	30×32 pixel gray-scale camera and 8×32 range finder	continuous steering-command	euclidian distance to recorded action
Nvidia Autopilot[4]	Annotated real-world data	TensorFlow-implementation of [7]	vision of front-facing camera	continuous steering-command	MSE to actual steering
TensorKart by Kevin Hughes[5]	Mariokart 64	Tensorflow-model similar to Nvidia Autopilot	console screen	joystick command as vector	euclidian distance to recorded action

TABLE 3.1: Supervised approaches to learn autonomous driving

3.2.2   Reinforcement learning

While there are many successful approaches that use supervised learning to copy manual steering commands, such approaches have severe limitations. First of all, no statement about their ability to adapt to unknown situations can be made. It is obvious, that it is next to impossible to get enough data of *extreme* situations, in which for example an accident is prevented in the last milliseconds.

Further, it is impossible for a supervised network to become better than its teacher. Especially in the domain of car racing however, it can easily be seen that the ultimate goal is an agent that drives better than its human teacher.

Another factor is, that the presented end-to-end approaches learn in a *short-sighted* manner, where they predict the action solely based on the current observation – without taking into account future implications of their actions. Reinforcement learning in contrast maximizes some long-term reward, trying to predict implications to plan trajectories.

Because of these reasons, it is interesting to look at driving agents that learn through their own interactions with the environment, via the technique of reinforcement learning as described in chapter 2.

---

[3]*Tesla* for example generates thousands of hours of driving data each day: `https://qz.com/694520/tesla-has-780-million-miles-of-driving-data-and-adds-another-million-every-10-hours/` [accessed on 29th August, 2017]

[4]`https://github.com/SullyChen/Autopilot-TensorFlow` [accessed on 20th August, 2017]

[5]`https://kevinhughes.ca/blog/tensor-kart` [accessed on 20th August, 2017]

While reinforcement learning is a promising approach for training autonomous cars, it requires a huge amount of trial and error, which is why it is reasonable to train in simulations, rather than in real life. The presented *DQN* and *DDPG* algorithms require interaction with their environment to calculate their reward, which cannot be provided in real-life situations because of the accident risk.

There are many approaches in recent literature aiming at translating such agents to subsequently perform successfully in real-world situations, as for example [41]. In this paper, the authors propose a neural network that translates the image generated by a race car simulation (specifically, they use the introduced TORCS engine) into a a realistic scene with similar structure, using a network architecture known as *SegNet*[2]. Furthermore, they provide a self-driving agent which uses a discrete version of the *A3C*[18]-algorithm to train throttle, brake and steering-commands, discretized into nine concrete actions. While their result is worse than a supervisedly trained baseline (the dataset of which was deemed the ground truth), a successful driving policy was learned that can adapt to real world driving data.

Available input-data

In simulations, the ground truth of the car's physics can easily be taken as input to an agent. This apparently leads to a far richer set of presentations than what can be utilized in real-life – a possible counter-argument to the adaptability of these implementations to actual real-world scenarios.

However, there are many successful approaches in the literature which learn solely using visual input, comparable to that of a front-facing camera. Further, today's semi-autonomous vehicles have many components that represent a diverse range of possible input. These components include, but are not limited to radar, visible-light-camera, LIDAR, infrared-camera, stereo vision, GPS or audio.

Interesting is for example the 3D-scanning *LIDAR* sensor that can produce very high-level information of the surroundings of the car[6]. In this work *minimap-cameras*, which provide a topview of road ahead of the car (see annotations **H** and **I** of figure B.3 in appendix B) are used. It can be argued that Segnet[2] could be used to convert the result of the respetive sensor into a comparable input. Similar reasoning can be given for many of the other used input-data, which will be explained in section 5.1.

Related implementations

There are several known implementations that perform reinforcement learning on driving simulations. The usual testbed for such implementation is TORCS – in fact, it was the common framework for all found related implementation. Using one and the same environment to train on is a good way to compare actual performances of agents – it is however also worth to use another implementation to test for the algorithm's generality. Table 3.2 shows some known implementations that perform reinforcement learning on driving simulations.

An interesting result is made by Lillicrap et al. [15], which used their algorithm on both low-dimensional input as well as visual input with almost the same average performance. It can thus be assumed that both representations of the agent's *observation* can in general lead to comparable final results.

Note that the the velocity along the track-direction is a popular reward-function, at least in part used in all implementations. While it seems to work reasonably well in combination

---

[6]A video visualizing the data is available under `https://www.youtube.com/watch?v=nXlqv_k4P8Q` [accessed on 10th August, 2017]

[7]for a video of a driving agent, see `https://www.youtube.com/watch?time_continue=4&v=4hoLGtnK_3U` [accessed on 29th August, 2017]

Project	trained on	model	input	output	reward	performance
DDPG [15]	TORCS	DDPG	visual input as provided by TORCS	(throttle, brake, steer) $\subset \mathbb{R}^{n \in \mathbb{N}}$	velocity along the track direction, penalty of -1 for collisions	*some replicas were able to learn reasonable policies that are able to complete a circuit around the track.* (quote [15])
DDPG [15]	TORCS	DDPG	low-dimensional, similar to [16]	(throttle, brake, steer) $\subset \mathbb{R}^{n \in \mathbb{N}}$	velocity along the track direction, penalty of -1 for collisions	
DDPG-Keras-Torcs by Ben Lau[6]	TORCS	DDPG implemented in Keras	angle, 19 range finder sensors, distance between car and track axis, speed along x,y,z axis, wheel rotation, car engine rotation (subset of [16])	(throttle, brake, steer) $\subset \mathbb{R}^{n \in \mathbb{N}}$	velocity along the track-direction minus velocity in transverse direction	reasonable policy after 2000 episodes[7]
A3C [18]	TORCS	Asynchronous advantage actor-critic	visual input as provided by TORCS	unknown discretization of (throttle, brake, steer)	velocity along the track direction	betwen 75% and 90% of the score of a human tester after 12 hours of training

TABLE 3.2: RL approaches to learn autonomous driving

with a good model, [6] mentions that solely this reward can lead to the problem of the car accelerating too much – an issue that is very relevant to this thesis as well.

Note further, that all of the listed approaches incorporate a stochastic start state distribution: The agent ist initialized at random speeds at varying positions of the track. While this makes it easier for reinforcement learning agents to train the whole track, it is an impossible premise in real life.

4 Program Architecture

The aim of this thesis is to convert a given racing game into an environment that can be played by reinforcement learning agents and to analyze the performance of different agents. In this chapter, I will explain the implementation that was developed throughout this process. I will start by explaining the design decisions that shaped this implementation in section 4.1. In the following section (4.2) I will explain the particular source code, starting with the game as it was given as starting point, most parts of which are not implemented by the author of this thesis. Then the implemented extensions of the game are explained, before leading over to the agent.

4.1 Characteristics and design decisions

As stated in chapter 3.1, the usual framework for solving reinforcement learning tasks is fairly rigid and has to be adjusted for the task in this work. The differences causing this will be discussed in this section, alongside further design decisions and challenges with their respective solution. For that, I will explain general principles in the first section of this chapter, and go into detail about some of the specific details about the implementation in the subsequent section.

4.1.1 Characteristics of this project

A difference of this implementation to the agents presented in chapter 2 is, that those are developed as general-purpose problem solvers, with the intention to solve any arbitrary task given to them. In this approach, that is not the case – the goal is to solve the specific given game. This allows in principle to incorporate expert knowledge of the task domain, to for example forbid certain combinations of the output or to use particular types of exploration, which are specifically useful in this scenario.

This thesis' game is a racing game, which has implications in several domains, for example making the standard ϵ-greedy approach for exploration much worse as in many other domains. Next to that the game is, in contrast to games solved in OpenAI's gym-environment (3.1), live. While the game could in theory be manually paused for every RL step, it provides advantages to let the agent run "live", such that it runs fluently and its progress can manually be inspected. Another challenge was the fact, that the game is coded in a programming language for which no efficient deep learning architectures exist, which led to the necessity of a propietary communication between the environment and its agent.

The fact that there is a game that is supposed to be *solved* also extended the entire problem domain: While usually the focus of developing RL agents can lie purely on the agent, assuming that the environment and consequently its definition of state/observation and reward is fixed, for the focus of this work it is also necessary to test what a useful definition of observation or reward looks like. Many of the subsequent design decisions are made with the idea in mind, that it must be as easy as possible to compare different agents using a unique combination of *observation*-definition, *reward*-definition, *model* and *exploration technique*.

Furthermore, the question of how important supervised pretraining is will be addressed in this thesis. For that, the game must provide a way of recording manual actions and their

corresponding observation and to record that in a way, that a learner can read it and train on this data. Doing so allows to compare agents relying purely on supervised training to reinforcement learning. Further, the question arises if there is a way to combine pre-training with reinforcement training in racing tasks, as famously done in the board game *Go* (see [28]).

4.1.2 Characteristics of the game

The given game is a simple racing simulation with realistic slip-behaviour. As of now, it consists of one car driving one track. While it is possible to implement additional AI drivers or different tracks, no such thing is considered in the current implementation. As the physics of this game is more realistic than that of all known `Atari`-games, this game is much harder to predict and master than what the original DQN was tested on.

The track itself is a circular course with a combination of unique curves, requiring the car to steer left as well as right. Along every point of the course, there are three different surfaces providing different friction – the *track* (inside) itself provides the most grip, while the *off-track* (outside) surface is far more slippery. Between the track-surface and the off-track-surface there is the *curb*, which manifests as a small bump with separate friction properties. The track, curb and off-track each have a consistent width throughout the circuit. To the outside of the off-track there is a wall that cannot be traversed. On this track, there is a car which is controlled by the agent.

It is the task of an agent to provide commands for the values of throttle, brake and steering, which are continuous in their respective domains: $throttle \in [0, 1]$, $brake \in [0, 1]$ and $steer \in [-1, 1]$. The *throttle* increases the simulated motor-torque, which leads to faster rotation of the tires and thus applying forward force to the car. The *brake* simulates a continuous brake force slowing down the rotation of the tires. It is not possible to complete a lap while constantly accelerating as much as possible without braking. It is important to note that slip and spin is also simulated: While the rotation of the tires can be accelerated or decelerated fairly abrupt due to their small mass, the car itself has a higher mass and thus more inertia, with forbids abrupt changes in movement. As the tires are rigidly attached to the car, they lose grip on the street, which lessens the impact of consequent forces applied to the tires. The last command an agent must output is the *steering*, which turns the front tires of the car, leading the car applying force to the respective side the tires steered towards.

As is the general case in reinforcement learning problems, the agent does not know the implications of its actions in advance but must learn them via trial and error. Important to note is that the agent provides those actions *to the car*, which must thus be seen as part of the environment. Because of the described simulation of slip and spin, an action does not have a reliable impact on the car's behaviour. Consequences of the implemented physics are for instance smaller turning circles for slower speeds, or a reduced effect for simultaneous braking and steering at high velocities – if the inertia is high, the car's tires will slip, loose grip and the impact of the steering will vanish almost completely.

The game as a reinforcement learning problem

For a reinforcement learning agent to successfully learn how to operate in an unknown environment, it must correspond precisely to an MDP, which is a tuple of $\langle S, A, P, R, \gamma \rangle$ (details explained in section 2.1). Because in this simulation only the case of a single agent without any other cars on the track is considered, the racing problem can be formalized as a Markov Decision Processes with a similar reasoning than Wymann et al. [39, chapter 4] put forward for `TORCS`.

As is the general case in simulations, while every update-step of the physics aims to simulate a continuous process, those updates must be discretized in the temporal domain. As

however both agent and environment run live, the temporal discretization of the agent can not always correspond to that of the environment if an update-step in the agent takes longer than in the environment. To put that into numbers, the fixed timestep for the environment's physics is at 500 updates per second, while the agent discretizes to maximally 40 actions per second.

As mentioned in the previous section, the action space required by the agent is continuous with $\mathcal{A} \subset \mathbb{R}^3$.

The environment's state is a linear combination of different factors, as for example the car's speed, absolute position, current slip-values and much more. While certainly finite, it consists of many high-dimensional values: $\mathcal{S}_e \subset \mathbb{R}^{n \in \mathbb{N}}$. Because of certain factors in the implementation of the environment itself, the environment is indeterministic[1]. As it can in principle be argued that the environment's state corresponds to all information stored in the game's section of the computer's RAM, the game trivially fulfills the Markov property. As the environment is only a single-agent system, the transition function of the environments can be expressed as a stochastic function of state and action: $\mathcal{S}_e \times \mathcal{A} \to \mathcal{S}_e$. In contrast to many existing approaches (see section 3.2.2), it was decided that the start-state is no distribution over all possible states, but always set to the same position, with speed and inertia set to zero. While this makes the game harder to be learned, it is more realistic and comparable to real situations.

There is no formal definition of a reward returned by the environment in the game. While it could in theory be argued that the inverse of the time needed to complete a lap could be taken as the reward, it is obvious that this is infeasible due to many reasons: Finishing one lap takes several seconds, which means there are hundreds of iterations between start state and final state. Next to the obviously arising *credit assignment problem*, the chance of an agent even getting to the finish line without crashing is practically zero without rewarding intermediate progress. Instead, it makes sense to give more *dense* rewards. As mentioned in section 4.1.1, the reward is also subject of experiments in the scope of this thesis. For the game to correspond to a proper environment, this reward must be a scalar value depending on state and action.

Though it is in theory possible to implement an agent that takes the entire underlying state of the environment as its input, an approach like that is far from feasible, as for example this state also contains much information only necessary for rendering the game. Instead, in the chosen approach the agent only receives an *observation* of the environment's state.

Summarizing all those factors, it becomes obvious that the given game can clearly be defined in terms of a *Partially Observed Indeterministic Markov Decision Process*.

It is worth noting, that while the dimensionality of the observation is likely much smaller than the dimensionality of the state, any feasible observation will be high-dimensional or real-valued. The chance for any particular combination of parameters to appear multiple times is vanishingly low, which makes the use of function approximation necessary.

While the notion of *POMDPs* itself contains no final state definition, they are necessary to provice a hard limit on Q-value calculation. A design decision on this was made to let the environment only provide candidates of what could terminate an episode (e.g. a time limit or steering into a wall). As the current work wanted to test which definition of final states was most efficient, the agent decides in which states it resets the environment

On start-up, the game lets a user choose between three different game modes: In the `Drive` mode, users manually drive the car via the keyboard. The `Train AI supervisedly` mode records

---

[1]The game is programmed in Unity, which has non-predictable physics. As discussed for example in `https://forum.unity3d.com/threads/is-unity-physics-is-deterministic.429778/` [accessed on 1st August, 2017], this is because of the fact that any random-number-generator depends on the current system time, which is never fully equal in subsequent trials. Because the calculation of some states can be more complex than others, this effect can snowball even more – longer calculation in one step leads to later timing of a subsequent update step, which can lead to a whole other trajectory along the state space, even though the start state was equal.

information about the manual driving, which can be used as (pre-) training data for the agents. In the `Drive AI` mode the car is controlled by an agent (until the user actively interferes). A screenshot of the start menu can be found in figure B.1 in appendix B. Note that the background of the menu provides a bird-eye view of the track.

4.1.3 Characteristics of the agent

As stated above, it was a design decision to leave as many options open to the agent as possible. To summarize from the above chapters, the features unique to every agent are:

- The definition of the agent's *observation-function*, providing its internal state: $s = o(s_e)$

- Definition of the *reward-function*, returning a scalar from state, action and subsequent state: $\mathcal{S}_e \times \mathcal{A} \times \mathcal{S}_e \rightarrow \mathbb{R}$

- Definition of its internal *model*, basing on which it calculates its policy

- Under which conditions an episode is considered to be terminated

- Definition of the agent's *exploration technique*

- If and how the agent relies on pretraining with supervised data

> In the following sections, I will refer to the definitions of these functions/options as the *features* of the agent. In the implementation, there are many possible features, not all of which are actually used by an agent. I will refer to those as *possible features*. When I talk specifically about the possible observations influencing the observation-function, I refer to those as *(possible) vectors*. Furthermore, the specific implementation of the agent's memory will be considered a feature. This has however no further consequences as it only differs in some agents in order to save its replay memory more efficiently.

This design decision is why differences arise between this project and the structure put forward in algorithm 1. For example, the outer loop (line 3) becomes obsolete, as the reset conditions are decided by the agent. It additionally becomes possible to experiment with rewards changing over time, allowing e.g. first learning to move forward and only later to stay strictly on the track.

Because there are many features in which agents can differ, it makes sense to use the object-oriented programming concept of *inheritance* for the different agents. In such an implementation, the main methods common to every agent as well as default definitions of the features are implemented in a superclass. Particular agents inherit from this class, while overwriting the attributes/functions for the feature in which they differ.

In this chapter, only the general implementation of agent and environment are described. Because the definition of the `features` is replaceable, with the specific implementation being one of many possibilities, it is considered the *resulting implementation* and thus described in chapter 5

4.2 Implementation

The associated programs are, as will be explicitly stated, written by the author of this work or its first supervisor, and are licensed under the GNU General Public License (GNU GPLv3). Their source codes can be found digitally on the enclosed CD-ROM as well as online. Version control of this project relied on `GitHub`[2], and was split into three repositories: The source code of the actual game written with the game engine `Unity 3D` (*BA-rAIce*[3]), the source code of

---

[2] `https://github.com/`
[3] `https://github.com/cstenkamp/BA-rAIce`

the implementation, written in `Python` (*Ba-rAIce-ANN*[4]), as well as the present text, written in
LaTeX (*BAText*[5]). To ease connections between the following descriptions and their correspon-
dences in the actual source code, footnotes will refer as hyperlink to the files on GitHub (the
relative path on the enclosed CD is equal to those on GitHub). In order to ensure that no work
after the deadline is considered, it is referred to the signed commits **7afd0fc** and **1a94b2b**.

The game is programmed using Unity Personal with a free license for students[6]. It is tested
under version `2017.1.0f3`[7]. Scripts belonging to the game are coded using the programming
language `C#`. The agent was programmed with `Python 3.5`, relying on the open-source machine
learning library `TensorFlow`[1][8] in version `1.3`. For a listing of all further used python-packages
and their versions, it is referred to the BA-rAIce-ANN/requirements.txt-file.

While the original framework of the game was coded in advance to this thesis by the first
supervisor, *Leon Sütfeld*, most of this work was contributed by the author of this thesis to enable
learning of the game and making it playable by a machine. While it will be explicitly stated
what was already given later in this chapter, it is also referred to the respective branch on
Github (*Ba-rAIce – LeonsVersion*[9]). The implementation of the agent was not influenced by any
other people than the author of this work.

In the description of the sourcecode, code-excerpts will be merged with the text. To dis-
tinguish code from text, these excerpts will be shaded in gray. A further distinction is made
between `methods()` and `objects`, `variables` and `classes`. For the sake of brevity, the param-
eter list of `methods` will be left out if not explicitly needed.

The game, making up the environment, is completely independent of the agent and runs as a
separate process on the machine. The agent is written in another programming language and
must thus make up a distinct process as well. Because of that, it is necessary that agent and
environment communicate over a protocoll that allows inter-process-communication. In this
work, it was decided to use *sockets* as means of communication. While explained in following
section, it is for now important to know that sockets are best implemented as running in a
separate *thread*, where they can send textual information to another socket running in another
process.

The following two sections describe the game in the form of a manual, which is useful to
understand the project enough to continue working on it, but may be irrelevant for others. A
reader that only needs information about the agent can thus skip to section 4.2.3.

4.2.1 The game as given

The program flow of the game is encapsuled by the framework provided by `Unity 3D`. To ease
the implemention of games, it provides numerous `game objects` with pre-implemented prop-
erties like friction or gravity, as well as drag-and-drop functionality to add 3D components or
cameras to the Graphical User Interface (**GUI**).

---

[4] https://github.com/cstenkamp/BA-rAIce-ANN

[5] https://github.com/cstenkamp/BAText

[6] https://store.unity.com/products/unity-personal

[7] As of now, 14th September, 2017, there is a bug in Unity that causes it to crash due to a memory leak if UI
components are updated too often (which happens after a few hours of running). Because of that, in the current
release of this project, all updates of the Unity UI are disabled in AI-Mode. A bug report to Unity was filed (case
935432) on 27th July, 2017, and it was promised that this issue will soon be fixed. Once that is the case, the variable
`DEBUG_DISABLEGUI_AIMODE` in BA-rAIce/AiInterface.cs can be set to `false`.

[8] https://www.tensorflow.org/

[9] https://github.com/cstenkamp/BA-rAIce/tree/LeonsVersion

To implement additional behaviour or features not predefined by Unity, it allows for scripts, written in object-oriented *C#*. Such a file will only be instantiated during runtime if it is specified in Unity's `Object Hierarchy`. For that, the script has to provide a class that extends[10] Unity's class `MonoBehavior` or be used by such a class.

After staring the program, Unity will create all objects specified in the object hierarchy. To enable the possibility of instanciations `knowing` each other during runtime, they must provide public variables of the type of the respective subclass of `MonoBehaviour`, which can via drag-and-drop be assigned to the respective future instance specified in the hierachy.

`MonoBehaviour` provides a number of functions that are automatically called by Unity at different times during runtime. The most important ones are `void Start()`, called when first instanciating the respective object, as well as `void Update()` and `void FixedUpdate()`, called every Update-step or Physics-update-step, respectively[11]. If a subclass of `MonoBehaviour` is attached to a game object, it can provide specific additional functions that are called when specific events occur during runtime – an example would be `OnCollisionEnter(Collision)`.

In the following, I will describe the game in chunks corresponding roughly to implemented classes, which is referred to by a footnote in the headline. Note, that sometimes optional features will be mentioned. As those options are relevant not to a user of the game but to its developers, those options are specified in the code, more precisely in a static class called `Consts` in `AiInterface.cs`[12].

Game modes[13]

On the surface, there are three modes the user can choose from in the main menu: `Drive`, `Drive AI` and `train AI supervisedly`. The UI of the menu can be seen in figure B.1 in appendix B.

In the actual implementation however, the game mode is handled a bit different. `mode` is a public string-array of the globally known object `Game` (an instanciation of a `GameScript`, a subclass of `MonoBehaviour` specified in `GameScript.cs`[13]). `mode` contains one or more strings of the following group: `"driving"`, `"menu"`, `"train_AI"`, `"drive_AI"` and `"keyboarddriving"`. If the game's main menu is opened, `mode = ["menu"]`, and in all other cases it is a set consisting of `"driving"` as well as the respectively obvious elements. This implementation is advantagous, because some behaviour needs to be triggered in multiple modes – the car's movement is for instance calculated if `Game.mode.Contains("driving")`, which is the case in all three of the above mentioned modes. The current implementation also makes it easier to add further behaviour: If for example an AI-agent shall also function to generate supervised data, one can simply add the respective mode in the `Gamescript.cs` file.

The functionality for switching the game mode is specified in the `SwitchMode(newMode)` function, found in `GameScript.cs`. After setting the respective mode as described above, this function disconnects any connected sockets and activates the required cameras for the mode, updates the UI's display indicating the mode and calls some further initializing functions

---

[10]In the following sections, I will use the terms extends, implements, knows and has. When I use those, I mean them in the strict sense in the context of object-oriented programming languages (and the concepts inheritance, interfaces, references and part-of relations).

[11]Concerning the difference between Update and Fixedupdate: `Update` is called once per frame, in other words as often as possible. It is generally not used to update physics, as its call frequency depends on the current **FPS** – if calculations here take to long, the FPS of the game will decrease. `FixedUpdate` is called precisely in a fixed interval of game-internal time. If calculations in FixedUpdate are too slow, the progress of the game-internal time is delayed until FixedUpdate catches up. The update-interval of FixedUpdate can be freely chosen and is 0.002 seconds in the current implementation. If Unity's `Time.timeScale` is set to zero, `FixedUpdate()` will not be called at all.

[12] `https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/AiInterface.cs`

[13]`https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/GameScript.cs`

(`AiInt.StartedAIMode()` or `Rec-StartedSV_SaveMode()`), if applicable. Because particularly the initialization of the `drive_AI` -mode involves connecting with an external socket, it is done in part in a side thread. This means that the main thread does not wait for the initialization to be finished – because of that, the `StartedAIMode()` -function sets the variable `AiInt.AIMode` to `true` once it is done. Any behaviour that depends on a successful initialization of the mode can thus simply check for this variable instead.

The object `Game` is responsible for switching the `mode` to `menu` in its `Start()` -method or after the press of the Esc -button at any time during the runtime of the game. Besides this, its definition in GameScript.cs also contains the methods `QuickPause(string reason)` and `UnQuickPause(string reason)` . The purpose of QuickPause is to pause all physics processes of the game, such that other functionalities running in parallel get time to catch up with their calculations. For that, the `QuickPause` -function sets Unity's `Time.timeScale` to zero, which freezes all of Unity's internal physics, as well as stopping future `FixedUpdate()` -calls. Further, this function removes `driving` from `mode` , so that other driving-related functions in `Update()` are also stopped. Lastly, the QuickPause-mode changes the game's GUI to make the difference visible to the User.

While `QuickPause(string reason)` is a public function and can in principle be called from every method inside Unity, it contains non-thread-safe functionalities (due to design concepts of Unity), which only the main-thread can call safely. Asynchronous threads use the public variable `shouldQuickpauseReason` to request the activation of the QuickPause-mode, and `Game` checks every `Update()` -step if a side thread made such a request. To make sure that multiple requests are treated sequentially, `QuickPause` has to be called with a `string reason` , which is pushed on Game's list `FreezeReasons` . When one of the reasons to request QuickPause is resolved, the method `UnQuickPause(string reason)` must be called with the same `reason` . This method then removes this `reason` from the list, but the normal game process is only continued if this results in the list to be empty.

User Interface[14]

The job of the Game's `UI` (an instance of type `UIScript` [14]) is to update the user interface, which is overlayed over the current scene. In its `MenuOverLayHandling()` -Method, `UI` specifies the view of the game's menu-mode (as can be seen in figure B.1 in appendix B), as well as the key bindings to activate the required mode. In the method `DrivingOverlayHandling()` , it sets visibility, content, color or position of numerous UI components (defined in Unitys Object Hierachy), that are seen in non-menu-modes. Both `DrivingOverlayHandling()` and `MenuOverLayHandling()` are called every `Update()` , such that the view elements are always contemporary. Further, the `UI` specifies the `void onGUI()` -function, which Unity calls every time it re-renders the GUI. This function overlays Debug information on the screen and changes the view if the QuickPause-mode is activated.

The User Interface of the game while driving can be seen in the figures B.2 and B.3 in appendix B, which are screenshots for the `keyboarddriving` and `drive_AI` mode, respectively. The latter of those screenshots is annotated with labelled boxes around each UI component, with page 90 explaining each component a bit further[15].

---

[14]https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/UIScript.cs

[15]Note that the entire UI, besides the content behind labels **A**, **F**, **H** and **I**, was already implemented like this by the first supervisor .

Controls

If `Game.mode.Contains("keyboarddriving")`, the game is steered with the arrow keys ⬅ and ➡. The throttle is triggered via the Ⓐ-key, whereas the brake is called via Ⓨ. The Ⓡ - key flips the reverse gear. Note that as long as the `pedalType` in `CarController` is set to `"digital"`, throttle and brake are binary when controlled via keyboard.

If `Game.mode.Contains("drive_AI")` the car is usually controlled by the agent. It is however possible to re-gain control over it via pressing the Ⓗ-key. Once that occurs, the variable `AiInt.HumanTakingControl` is set to `true`, indicating the program that keyboard-inputs must be accepted. This is useful for example if one wants to check if rewards or Q-values of an agent are realistic. If human interference of the `drive_AI` mode is active, it is possible to simulate speeds to the agent (meaning that not the actual speed of the car, but a specified value is sent a connected agent). This can be done with the number keys, where the pretended speed is evenly spread between 0 km/h (Ⓞ) and 250 km/h (Ⓖ). The Ⓟ key is reserved to simulate a full throttle value. To hand control back to the agent, Ⓗ must be pressed again.

In the `drive_AI`-mode, a user can also manually disconnect or attempt a connection-trial with an agent. The keys to do that are Ⓓ and Ⓒ, respectively. During any `mode` containing `"driving"`, Ⓠ can be pressed to activate the `QuickPause`-mode, which allows to spectate the current screen. `QuickPause` is ended with another hit of Ⓠ.

The car[16]

The `car` itself is a `Rigidbody`, which is a Unity-gameObject with certain properties like spatial expanse, mass and gravity. Attached to the `car` is, next to no further mentioned visible components, a `BoxCollider` as well as four `WheelColliders` gameObjects. While the `BoxCollider`'s purpose is to trigger the call of particular functions in scripts attached to other gameObjects upon simulated physical contact, the `WheelColliders` are predefined with certain physical properties, allowing for precise simulation of the behaviour of actual tires. How the car moves is specified in an instance of the class `CarController`, which is attached to the respective `Rigidbody`.

All functions of the `CarController` [16] are only called in modes containing `"driving"`. In its `FixedUpdate()`-step the script adjusts the wheelCollider's friction according to the current surface the wheels are on, and checks if the car moved outside the street's surface. Furthermore the car's velocity as well as some other values are calculated. Finally, the torques for acceleration and braking are applied and the front wheels are turned according to the steering-value. The amount of those torques and angles depend on three values: $steeringValue \in [-1, 1]$, $throttlePedalValue \in [0, 1]$ and $brakePedalValue \in [0, 1]$.

If `Game.mode.Contains("keyboarddriving")` or `AiInt.HumanTakingControl == true`, those values depend on the User's keyboard input. Otherwise, if `AiInt.AIMode` is enabled, the values are defined as known values from the `AiInt`, namely `nn_steer`, `nn_throttle` and `nn_throttle`. `AiInt` is an instance of class AiInterface[12] that will be elaborated on later.

In `CarController.Update()`, the outer appearance of the car is updated, consisting of wheel height, wheel rotation and steering angle.

As explained in section 4.1.3, a connected agent must be able to reset the car at any time during runtime. To allow for that, the `CarController` provides a `ResetCar` method. Additionally, there is a `ResetToPosition`-function that resets the `car` to any specified position and rotation. To reset the car, it is necessary to completely reset its inertia without the respective values being overwritten in the next `FixedUpdate()` call. `CarController.FixedUpdate()` therefore checks

---

[16]https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/CarController.cs

the boolean variable `justrespawned` on every call, which is set to `true` by `ResetToPosition`, to reliably remove all of the car's inertia before setting the boolean to `false` again.

Position tracking[17]

To successfully learn capable driving policies, the agent must get precise knowledge of the car's position which goes beyond its mere coordinates. Additional useful information is for example the car's position in relation to the street or information about the course of the road ahead of it. To allow for that, the game incorporates a `TrackingSystem`, which is an instance of the class `PositionTracking` [17]. The `TrackingSystem` knows the gameObject `trackOutline` and converts it to an array of coordinates located regularly along the track, each one respectively located at the middle of the street – the `Vector3[] anchorVector`. Using this array, much high-level information about the track can be calculated. As almost all of the respective functionality was however not implemented by the author of this thesis, few examples of what can be done with it shall suffice.

The total length of the track can be calculated by summing up all distances between all `anchorVector` coordinates and their respective successor. By putting this in relation to the current advancement of the car (which is the sum of all distances up to the coordinates closest to the car, calculated in `GetClosestAnchor(Vector3 position)`), one can calculate the car's rough `progress` in percent.

As every successive coordinate in `anchorVector` is in the middle of the street, the direction of the street at position `p` can be determined by calculating the vector `anchorVector[GetClosestAnchor(p)+1] - anchorVector[GetClosestAnchor(p)]`. This can be used as basis for many further calculations: For instance, the car's traverse distance to the center of the street can be found by calculating the norm of the orthogonal projection from the car's coordinate onto this vector (from now on called the *perpendicular*). The direction of the car relative to the street can be found by calculating the angle between its `direction`-vector and the previosly explained vector.

Besides defining the `anchorVector`-array and other helper-arrays dependent on that in its `Start()`-method, the `TrackingSystem` calculates the car's current `progress` at every `Update()`-step and triggers the `UpdateList()`-method of the `RecordingSystem` [18] in regular progress intervals. Furthermore, the `TrackingSystem` provides certain public methods that provide information for agents, such as `getCarAngle()`, `GetSpeedInDir()` and `GetCenterDist()`, the precise content of which will be explained in a later section.

Tracking time[18]

As visible in the game screenshots (more precisely annotations **B**, **C**, **P** and **Q** of Figure B.3), the game displays information about the current laptime, the last laptime as well as the time needed for the fastest lap. Furthermore the game provides visual feedback on the difference in time needed for a specific section of the street in the current lap versus the fastest lap (annotation **E**). This is possible because the game records current laptime multiple times throughout the course. As mentioned above, `RecordingSystem.UpdateList()` gets called regularly by the `TrackingSystem`. `RecordingSystem` is an instance of the type `Recorder`. It contains three lists of `PointInTime`s, for `thisLap`, `lastLap` and `fastestLap`. A `PointInTime` is a serializable object (also defined in `Recorder.cs`) that contains two floats, for a progress and a corresponding time.

---

[17] https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/PositionTracking.cs
[18] https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/Recorder.cs

An instance of a separate class, `TimingScript` (found in `TimingScript.cs`[19]) is attached to a permeable Collider right on the start/finish line that functions as trigger. As a subclass of `MonoBehaviour`, `TimingScript` has a `void OnTriggerExit(Collider other)`, that is invoked as soon as another Collider stops contact with it. As the only movable collider is the car's boxCollider, this method is called as soon as the car starts a lap. A lap is considered valid under two conditions: First, the car needs to pass a second collider (`confirmCollider`, with its attached `ConfirmColliderScript` [20]), which ensures that the car did in fact drive a complete lap instead of backing up right back on the start/finish line. The second condition for validity is, that at no time all four tires of the car left the street's surface.

If a lap is considered valid, the `TimingSystem`'s `onTriggerExit`-procedure calls `RecordingSystem`'s `Rec.FinishList()`-method. Afterwards and under no restrictions, it prepares the start of a new lap by calling `Rec.StartList()`. Once `StartList` is called, the `RecordingSystem` creates a new List of `PointInTime`s, to which the `TrackingSystem` then regularly adds new tuples of progress and corresponding time. Once `FinishList` is called, the `RecordingSystem` checks if the lap just now is a new record, and saves it on the computer's disk if so. In its methods `GetDelta()` and `GetFeedback()`, which are called every `Update()`-step of the `UI`, it can then compare the time of the currently latest progress with the corresponding time of `fastestLap`.

4.2.2   Game extensions to serve as environment

The code explained so far is sufficient for the game to work in the `"keyboarddriving"`-mode. The framework for this mode was working entirely when the author of this thesis received it, as it was implemented by the first supervisor. The major additions implemented in the scope of this thesis that were mentioned so far is the behaviour following game modes other than `"keyboarddriving"` or `"menu"`, the `QuickPause`-functionality, the mentioned additions to the User Interface, the means to reset the car as well as the functions `GetCarAngle()` and `GetSpeedInDir()` of the `TrackingSystem`, which will be more thoroughly explained later on.

The minimap cameras[21]

The content of the minimap cameras can be seen behind annotations **H** and **I** of figure B.3. They are implemented to serve as an exclusive or additional input to agents, in the hope of providing enough information to learn successful policies. As can be seen, the minimap cameras provide a bird-eye view of the track ahead of the car, by filming vertically downwards. In contrast to the foreshortened main-camera, the minimap cameras are orthogonal, which means that distances are true to scale, irrespectively of their position. Because the cameras are attached to the `car`'s Rigidbody, they are always in the same position relative to it. In the current implementation, up to two cameras can be used (it is however possible to disable one or both cameras by setting a corresponding value in the class `Consts` in `AiInterface.cs`). When both cameras are active, one of them is mounted closer to the car, such that one provides high accuracy whereas the other provides a greater field of view. If only one camera is enabled, its distance is set for tradeoff of accuracy and field of view. As both cameras must be handled separately, this happens in the `Start()`-method of the `Gamescript.cs`, which knows both cameras.

While a previous implementation of a similar functionality was provided by the first supervisor using a complex and inefficient ray-tracing, in this implementation the minimaps base

---

[19]`https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/TimingScript.cs`

[20]`https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/ConfirmColliderScript.cs`

[21]`https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/MiniMapScript.cs`

on Unity `Camera` -objects, which are efficiently calculated on the computer's GPU. Attached to each camera is a respective instance of `MiniMapScript` [21]. While ordinarily the content of Unity's cameras is directly rendered to the game's main screen, this script contains methods to convert the image of the camera to a format that can be sent to an agent. That is made possible by the usage of a `RenderTexture` as well as a `Texture2D` , which are created as private objects in `MiniMapScript` 's `PrepareVision(int xLen, int yLen)` -method. This method is called from outside and expects as parameter the dimensionality of the produced matrix, which is set in the class `Consts` in `AiInterface.cs`.

Both cameras provide the public function `GetVisionDisplay()` . When this function is called, it sets the above mentioned `RenderTexture` as the camera's `targetTexture` , forces the camera to `Render()` to this texture, and then reads the rendered contents into the specified `Texture2D` . After this process, it must reset the camera's `targetTexture` , such that it renders back to the game's main display, such that it can be inspected visually. The `Texture2D` however can be then be read pixel by pixel and thus converted to an array or string. As it was decided that the resulting display only differentiates between track, curb and off, the cameras use a `Culling Mask` that visually filter out all other gameObjects. In the current implementation, both minimaps have a resolution of 30×45 ternary pixels.

Recording training data[22]

For an agent to learn and perform in it, data of the environment must be recorded in regular intervals, to either be sent to the agent in the case of it playing the game or learning via re-inforcement learning, or to be exported to a file, which an agent can perform pretraining on. This section will deal only with how this data is saved and sent, as the vectors are explained in section 5.1. Because of that, I will refer to this data under the name vectors, which are in detail explained in section 5.1. Collecting the lastest vectors happens in the function `GetAllInfos()` of `AiInterface.cs`, which calls a number of functions and returns a string containing the combined result of those.

As calculating the data that needs to be exported can take relatively much time, this process cannot be performed every `FixedUpdate()` -step. Because of that, the following function is used to perform a function in regular time intervals:

```
long currtime = AiInterface.UnityTime();
if (currtime - lasttrack >= Consts.trackAllXMS) {
  lasttrack = lasttrack + Consts.trackAllXMS;
  SVLearnUpdateList ();
}
```

ALGORITHM 2: Executing a function in regular intervals

Where `AiInterface.UnityTime()` returns Unity's internal time by calling `Time.time * 1000` . Using this definition of time has the advantage that it is maximally precise inside Unity, as the time of calling `FixedUpdate()` is likewise dependent on `Time.time` . A disadvantage of this measurement of time is however, that it can only be used in the main thread and asynchronous methods must rely on the system's time, for which no conversion method exists.

Once user selects the `Train AI supervisedly` mode, Recorder's `void StartedSV_SaveMode()` gets called, which enables the minimap-cameras and sets `SV_SaveMode = true` . If that variable is `true` , the recorder will in its `StartList()` create a new `SVLearnLap = new List<TrackingPoint> ()` . `TrackingPoint` itself is a class defined in `Recorder.cs`, that contains certain values about the

---

[22]https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/Recorder.cs

state of the game, if provided at creation. While driving, the recorder checks every `FixedUpdate()` step with the mentioned method if it calls `SVLearnUpdateList()`. This method then collects the recent values for the currently performed actions, laptime, progress and speed as well as the respectively latest `Vectors`, creates a `TrackingPoint` from those and updates the `SVLearnLap` with it. When `RecordingSystem.FinishList` is called upon the next crossing of the start/finish line, the `SVLearnLap` is saved to a file. As the vectors can contain multiple pixel matrices from the minimap cameras, this may however take quite long. To prevent the game from freezing everytime the car passes the start/finish line, saving the actual file is performed in a seperate thread.

Because the agent using this exported data is written in another programming language than the environment, the data cannot be exported as binary file. In this implementation, it was decided to save the data in the XML-format. It is worth mentioning that not only the list of `TrackingPoint`s is exported, but also additional meta-information, stating among others the interval of how often a `TrackingPoint` was exported, which can be interpreted and used by an agent as well as manually inspected.

Communicating with an agent[23,24]

As already mentioned, the game is running live and is in general not stopped by the agent, as done for example when interfacing with the OpenAI gym (section 3.1). Because of that, the speed of communication between agent and environment is a bottleneck in how good an agent can perform, and needs to be as fast and efficiently implemented as possible. To ensure quick reaction times, it was also decided that agent and game must run on the same machine, as sending the data to another machine increases the needed time drastically[25].

In the scope of this thesis, it was experimented a lot with the flow of communication between agent and environment, with the current version as its most efficient one so far. While there are multiple possibilities of how two different programming languages can communicate with each other (for example *zero-mq*, *named pipes* or *shared memory*), it was decided to use *sockets* for the communication. When both sockets are on the same machine and communicate over *localhost* instead of online, all unnecessary protocol layers are skipped, making a connection over sockets very efficient[26]. Using the current implementation, reaction times with an average of $30ms$ were archived (including the network inference), which is fast enough for all purposes.

Communication over sockets generally asymmetric: There must always be a *Server* and one or more *Clients*, both of which contain instances of the class `socket`. Upon being started, the server registers a server-socket at a certain port of the machine, where it can be found by clients. It is only possible for clients to connect to it as long as the server keeps up this socket, which is why it is advisable to do so in a separate thread. When a client is started, it needs to know the IP-adress of the server (in this case localhost), as well as the number of the port the corresponding socket waits at. Once the client finds a waiting server-socket behind the port, it is common practice that the server creates a new socket at another port. While this new socket represents a stable connection between server and client, the original server-socket can keep on waiting for new clients to connect to in the future. Note, that socket-connections reqire

---

[23]https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/AiInterface.cs

[24]https://github.com/cstenkamp/BA-rAIce/blob/master/Assets/Scripts/AsyncClient.cs

[25]This is the reason this project was implemented entirely under Windows: There is no stable Unity Editor for Linux, and there is no contemporary GPU-supporting TensorFlow under Mac. The only common ground for which both are available ist therefore the Windows platform.

[26]As is explained by alleged former Microsoft networking developer Jeff Tucker in this Stackoverflow-thread: http://stackoverflow.com/questions/10872557/how-slow-are-tcp-sockets-compared-to-named-pipes-on-windows-for-localhost-ipc [accessed on 1st August, 2017]

the length of each message is transmitted with it, such that the receiver knows when to stop reading the input stream.

For this project, it was decided that the game functions as a client, whereas the server is written in Python. In contrast to the implementation of [16], this also means that main loop happens in Python, with the game only considered as an additional thread providing the agent's input (as can be seen in figure 4.1). While this makes it harder to connect multiple agents simultaneously to the same game engine, multiple engines could be used to train an agent.

If the `Drive AI` mode was selected in the game's `menu`, the `AiInt` calls its function `StartedAIMode()` which, next to calling the known `PrepareVision` of the cameras, creates a `SenderClient` and a `ReceiverClient`, both of which are instances of `AsynchronousClient`, a class defined in `AsyncClient.cs`[24]. Afterwards, it connects sender and receiver by calling their `StartClientSocket()` and starts the `ReceiverClient`'s receive-loop in an asynchronous thread via calling their respective method `StartReceiveLoop()`.

AsynchronousClient is a wrapper-class that contains a C# `System.Net.Sockets.Socket`. While `SenderClient` and `ReceiverClient` have different tasks, much of their functionality overlaps, which is why the same class is used for both. It is useful to use two different sockets for sending and receiving data to ensure minumum latency: As they are in different threads, data can be sent and received simultaneously. As also the used `sockets` work asynchronously internally, they require callback-methods and `ManualResetEvent`s in their methods to connect (`StartClientSocket()`) as well as to `Send` and `Receive` data[27].

As soon as `StartClientSocket()` is called, an `AsynchronousClient` tries to connect to a server for a specified number of times. If this number is exceeded, it assumes there is no server and prevents further trials to connect by setting `serverdown = true`. Both clients can manually be reconnected by pressing Ⓒ, which resets this value and tries to connect again. Note that connecting the server necessarily goes along with sending the message `"resetServer"` to the server and starting the receiver's receive-loop.

Whenever data must be transmitted to the server, `AiInt.SendToPython(string data)` is used. This method appends the current time to the data and calls `SenderClient.SendInAnyCase(data)` in an asynchronous thread. To be absolutely failsafe, this method tries to send the data over its socket, and if that fails, creates a new socket to send the data over.

`ReceiverClient.StartReceiveLoop()` runs constantly in another asynchronous thread to wait for messages sent by a server. As soon as it got a message, the method `ReceiveCallback` calls `response.update()` with this method.

Messages from the server are stored in an object of type `Response`, which is part of the `ReceiverClient`. When the `SenderClient` sends information to Python, it always appends the current system-time. When the server sends a result basing on this input, it also returns that time. Upon being updated, the `response` takes another timestamp and calculates the response-time of the server and maintains a running average of these response times in `lastRts`. If this average reaction time is too high (`lastRTs.getAverage()>2*Consts.MAX_PYTHON_RT`), the receiver-thread requests a `QuickPause` – this stops the `SenderClient` from sending further data, but not the asynchronous `ReceiverClient`, which can thus catch up with its receiving progress. As soon as the latest chunk of data was returned by Python (the time of which is known through `AiInt.lastpythonupdate`), the receiver requests `UnQuickPause`.

---

[27]A `ManualResetEvent` is a C#-object notifies waiting threads that an event occured. Before for example the client's `beginConnect`-method is called, `ManualResetEvent`'s value, which can be accessed from outside threads, is set to `false`. The method itself is called with `ConnectCallback` as argument. Once `beginConnect` is done, `ConnectCallback` sets the corresponding value to true again, such that other threads can wait for this event.

Main communication loop   The main communication loop is specified in the methods `Update()` and `FixedUpdate()` of the `AiInterface`. As the former of those also runs also when the game is in `QuickPause`-mode, this method is responsible to handle the `othercommand`s sent from another socket. These commands may be to reset the car, or to activate/deactivate the `QuickPause`-mode. To differentiate these commands from steering-commands, the `ReceiverClient.response` holds a respective field for `othercommand`s.

The main loop of sending and receiving data to and from an agent however happens in the `FixedUpdate()`-method. In this method, the data is both sent to a server and the response of a such is handled.

Sending   Every `FixedUpdate()`-step, the `AiInt` checks if enough time has passed to send the latest data to an agent (see alg. 2) in the method `LoadAndSendToPython`. If so, it calls its method `load_infos`. This method performs a quick check if the car's situation changed enough to require a reload of the data, simply passing the last result otherwise for efficiency-reasons. If the data is re-loaded, the method `GetAllInfos()` is called, which aggregates all getter-function for the various vectors as well as the minimap-cameras into a string containing all the data. Note that speed as well as throttle-value as it is sent to a server is already possibly overwritten at this point. After aggregating it, the data is subsequently sent to a server socket using `SenderClient.SendInAnyCase` in a separate thread. In the current implementation, the game updates its agent in an interval of 10 FPS, a faster communication is however easily possible.

Furthermore, the `AiInt` has the method `notify_wallhit`, which is called in the method `onCollisionEnter` of a `WallColliderScript` attached to the track's walls. This method notifies the server in form of a *special command* if the car crashed into the track's wall to possibly reset it. Similar behaviour is triggered by the `TimingScript` upon completion of a lap.

Receiving   Every `FixedUpdate()`-step, the `AiInt` checks the newest response. If Unity is not set for a `Consts.fixedresultusagetime`, it immediately sets the values `nn_throttle`, `nn_brake` and `nn_steer`, such that the `CarController` can use it in the next iteration. Otherwise, it appends a copy of the latest response to a buffer in the form of a fixed-size queue. Every `FixedUpdate()`-step, the `AiInt` checks if a new item is in the buffer that was sent exactly `Consts.MAX_PYTHON_RT` milliseconds and sets the respective values for the `CarController` if so. This buffer can also be used to interpolate between the respective controls to counter jittering of the car's controls. These values are however only used if the variable `HumanTakingControl` is `false`, which is toggled with a hit of [H]. The used setting is comparable to [37], in that there are two levels of control – The agent tells the car the desired position/speed of rotation of the cars actuators, and the lower-level must interpolate between the sent results.

4.2.3   The agent[28]

In this implementation, the environment cannot be specified in a separate class, on which the agent can perform a function like `step()` (as in line 9 of algorithm 1). As the closest equivalent to such a class, the file server.py[29] (from now on called *server*) contains the thread-safe class `InputValContainer` that provides explicit functions to read its state, corresponding to the data sent from the environment. Upon executing the server, an instance `inputval` of

---

[28]I will rely on UML sequence diagrams and class diagrams to explain program flow and classes of agents. In case a reader is unfamiliar with the standards of UML 2.0, it is referred to e.g. `https://www.ibm.com/developerworks/rational/library/3101.html` for sequence diagrams and `https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html` for class diagrams.

[29]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/server.py`

type `InputValContainer` is created. This instance is `known` to objects that read or write from it through `containers`, an instance of class `Containers`. This class contains references to several objects and settings that must be known to multiple objects and functions (even in between threads), which can thus for example easily access the mentioned input-value as `containers.inputval`.

It is further important to keep in mind, that the server receives *raw data* from the client, and not a tuple consisting of `observation, reward, done`. The raw data is forwarded to the agent, which calculates the respective `features` on its own.

Communicating with an environment

The behaviour that will be explained is represented graphically in the sequence diagram in figure 4.1.

The main loop of the agent is started by the `main`-method of the server. Upon execution of this file, it interprets any passed arguments (to show a listing of those, the server can be called with the argument `"-help"`) and starts the main-method. To allow a fast communication with the game, the main loop of the server is multi-threaded, such that a stable socket-connection to Unity can be preserved while the agent performs its actions in another thread. This may lead to some objects being accessed by different threads (like for example `outputval`), which requires a thread-safe implementation of them.

In its `main`-method, the server creates two server-sockets[30], one on the receiver-port (which is equal to Unity's sender-port), and another one on the sender-port (Unity's receiver-port). Afterwards, the server creates an agent of choice by initializing it and calling its `initForDriving`, passing command-line-parameters. Afterwards, the `InputValContainer` `inputval` as well as an `OutputValContainer` termed `outputval` are created, and their references are added to `containers`. Once those are created, the server starts two other `threads`, more precisely a `ReceiverListenerThread` and a `SenderListenerThread`. These threads use their respective server-socket they know through `containers`, and wait on their port by calling the socket's `accept()`-method over and over.

Once a client connects to one of those listener-sockets, a new connected socket is created. Using this socket, a `receiver_thread` or `sender_thread` is created respectively, representing a stable connection to Unity. After creating this thread, the listener thread lets `containers` know about its reference (by appending it to `containers.receiverthreads` or `containers.senderthreads`) and starts its listening-loop over again. This means that during runtime, the two listener threads are always active. If for some reason the old `receiver_thread` or `sender_thread` loses its connection, the listener threads will simply establish a new one and create a new thread. Both `receiver_thread` and `sender_thread` contain a method to destroy themselves if needed, such that at any time there is exactly one thread responsible for the connection with Unity.

It was experimented with an explicit learning-thread, which runs in parallel to all others. As however the very same TensorFlow-model is needed for learning and inference, doing so in parallel emerged to increase the reaction time of the server from about $30ms$ to about $300ms$. While forcing the inference to run on the CPU and the learning to run on GPU decreased the delay a bit, the performance was still far from satisfactory and the idea was dropped.

While the main thread is constantly running, it does not contribute to the agent's process after initializing all threads and objects. Instead it can, if the agent provides methods to fill

---

[30]More precisely, the server uses a wrapper-class called `MySocket` (defined in `server.py`), which provides the additional behaviour of prepending the length of the future message to it, such that the socket on the other end knows how much it needs to read.

it, run the mainloop of the GUI (defined in `infoscreen.py`[31]), which is programmed with the package `tkinter` and must be in the main thread. To allow for side threads to write content to the GUI, its respective widgets (`Text` and `Canvas`) are replaced by thread-safe wrapper-classes using `queue` s. When a side thread wants it to print information, it appends the new content to the queue, whereas the main thread always pops the latest element from that queue to print it. Side threads know those wrappers over the dictionary `containers.screenwidgets` and can call the `write` or `updateCol` -method of the respective element.

The main-thread listens for a termination of the user with $\boxed{\text{CTRL}}+\boxed{\text{C}}$. Once that occurs, it notifies all other threads to shut down by setting `containers.KeepRunning = False` and waits for their termination by `join` ing them, such that the main thread is guaranteed to be the last to finish.

The main loop

In the `receiver_thread` , the socket is constantly waiting for data. When it receives data, it first checks if it was the usual raw data used for the agent's observation/reward or a *special command*. If it was the latter, the `receiver_thread` calls the agent's function `handle_commands` which will, according to its preferences, terminate the episode (for which it resets `inputval` and `outputval` and notifies the environment to reset itself).

If the data was no such command, the `receiver_thread` updates the `inputval` and starts an inference of the agent by calling its method `performAction` , passing the latest content of the `inputval` . This method is where the agent calculates its observation/reward, adds this to its replay memory, calculates the action on basis of its model and performs learning steps as appropriate. Immediately after the action is calculated, the agent updates the `outputval` with the the result. Upon being updated, the `outputval` informs the `sender_thread` of this update. This thread can then read the latest content of `outputval` and send it back to the client using its socket, which may be simultaneous to the agent performing its learning-step[32].

As mentioned above, the `inputval` is the server's closest correspondance of the environment. Every time new raw data from the game is sent to the server, `inputval.update` is called, which then incorporates the new information as well as timestamps of when they were send (used to calculate how long an inference took). Because the information that is sent over the sockets is textual, the method `cutoutandreturnvectors` from the file `read_supervised.py`[33] is called to convert this text back to python-arrays. Note, that there will consistently be made a difference between the content of the minimap-cameras (also termed *vision-vector*, see 4.2.2) and the other `vectors`, as they are stored in a different type of object. While the vision-vectors are each stored as a two-dimensional `numpy` -array, the other vectors are bundled to a class called `Otherinputs` , a subclass of `namedtuple` [34], which provides easily readable dot-access to its members instead of solely by index. This allows to add new vectors easily, as only a name of the new feature must be added in the definition of `Otherinputs` in the file `read_supervised.py`. Further, it increases the overseeability of functions using its components a lot.

The `inputval` contains arrays to store the history of vision-vectors (`vvec_hist`), otherin-puts (`otherinput_hist`) as well as an array for the history of performed actions (`action_hist`).

---

[31] `https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/infoscreen.py`

[32] Note that all of the agent's inference runs thus in the receiver-thread. It was experimented to use an additional agent-thread that in regular intervals looks if the `inputval` was filled with new data, such that the inference of the agent and a waiting to receive new data can occur simultenously. However, as the agent's inference runs fast enough if it does not perform a learning-step after an inference and too slow in any case if it does, this concept was abandoned.

[33] `https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/read_supervised.py`

[34] `https://docs.python.org/2/library/collections.html#collections.namedtuple` [accessed on 2nd September, 2017]
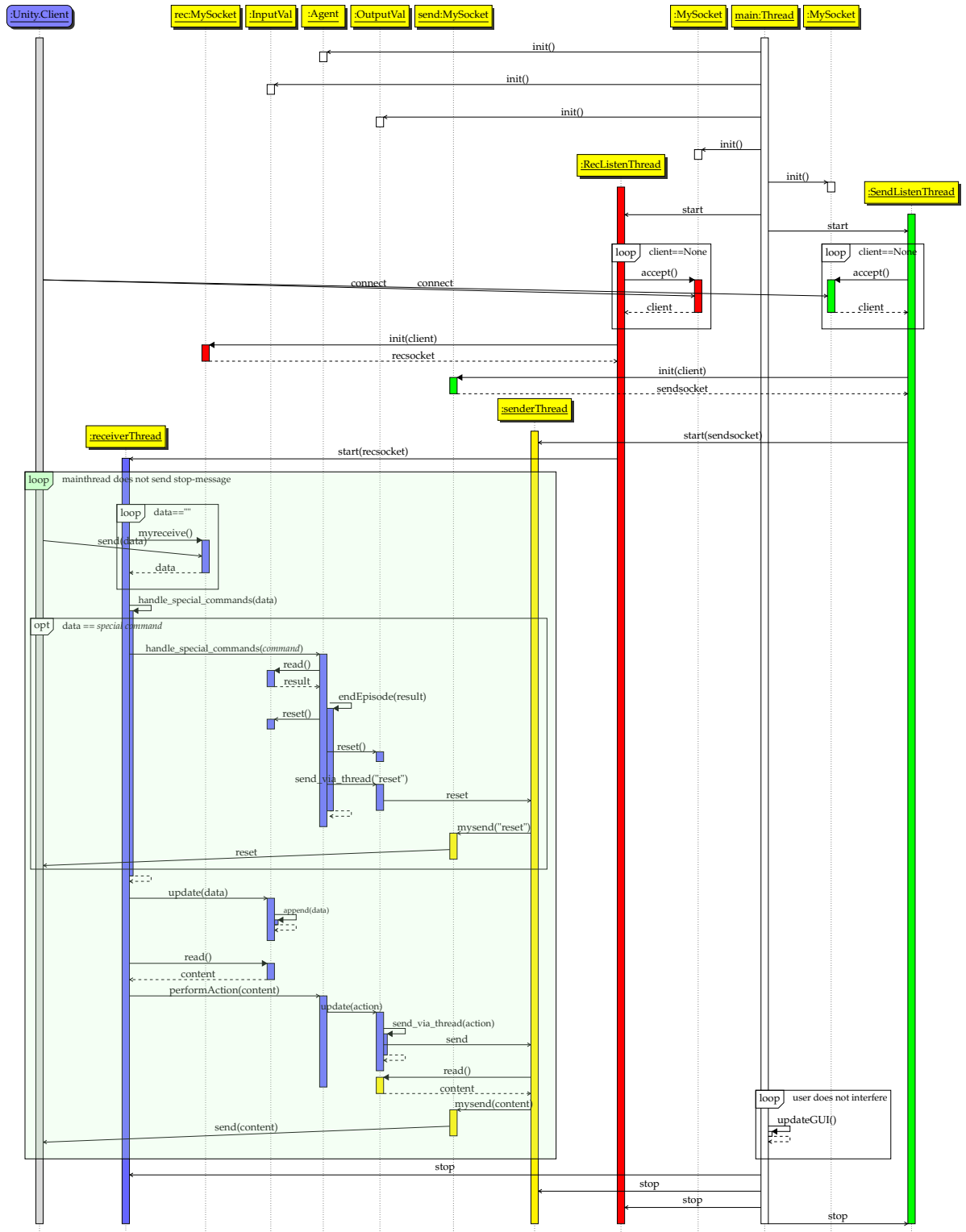
FIGURE 4.1: Sequence Diagram of the Server

Reading instructions: Y-axis is time. Columns with a full colored bar are threads, other columns are objects. The color of a bar represents which thread a method runs in. An arrow from A to B means "A calls a method from B".

In its `update` -method, it normalizes the `otherinputs` and appends it as well as the latest vision-vectors, where it merges both into the `vvec_hist` . To reduce the working memory load it causes, the `inputval` only saves the latest information send from the game, which corresponds

to the maximum of information an agent *could know* about the last or second to last state. After appending the newest vectors to the value, the `inputval` also checks if the car faced into the wrong direction for a certain amount of time and notifies the agent if so, such that the agent can optionally reset the environment.

Furthermore, the `inputval` defines a `read(pastState)` -method, such that an agent can access the information needed for its observation of the latest two states. This method unpacks and returns the respective vision-vector history of both minimap-cameras, the `otherinput_hist` as well as the `action_hist` . It is necessary that also the information about the penultimate state can be obtained, as the agent must add a full tuple of $\langle s_t, a_t, r_t, s_{t+1}, t+1 == t_t \rangle$ to its replay memory (see chapter 2.3.1) to perform Q-learning.

The `outputval` is less complex than the inputval, and only contains the latest result of the inference. When its `update` -method is called, it adds the action to the `inputval` (because the corresponding action to the environment could otherwise not be known) and goes on notifying a `sender_thread` that new information can be sent to Unity using its method `send_via_senderthread` . This method is also called when the client needs to be paused (by sending `"pleaseFreeze"`), needs to continue (`"pleaseUnFreeze"`) or needs to be reset (by sending `"pleasereset"`). The function `resetUnityAndServer` combines sending the reset-command to Unity and resetting `inputval` and `outputval` .

Agent-independent features

All of the above explained functionality is necessary for the agent to be quick enough for the given game, and has turned out this way due to certain design choices (like the agent deciding itself when to reset the environment) and peculiarities of the necessary proprietary communication with the game. As the game is its own thread, it is constantly running and does not wait while the agent calculates the action (the space between *send(data)* and *send(content)* in the leftmost column of figure 4.1). While this makes it necessary for the agent to be quick, it provides the advantage that a user inspect the live environment and inspect it at will. If inside Unity the H-key is pressed, the actions the agent suggested are overwritten, however the agent still gets input from the environment. If an agent features a GUI, the Q-values and rewards of the states a user drives will be printed to screen, including the agent's suggested action.

When talking about the `inputval` , it was mentioned that it contains as much information as an agent *could* know. In this implementation, there are some features that are clearly specific to an agent, while others are general settings indepent of the current agent. Those are stored in an instance of the class `Config` , defined in `config.py`[35], while the settings of the agent are saved in the respective file of the agent. Note that the value `msperframe` , which specifies the interval in which the agent performs actions, must be always equal to its correspondance in Unity, `Consts.updatepythonintervalms` . The value for the maximum number of history frames, `history_frame_nr` , is currently set to four. When running the `server`, an instance `conf` of type `Config` is created. The reference to this object is passed to all threads and objects, such that they can access all their settings clearly.

Agents

Until it arrives at the agent, neither the observation nor the reward is set. Thus, the agent needs to specify its own functions for its observation (given environment-state) and reward (given state and action). Because of this, some methods of an agent receive the `gameState` instead of the `agentState` . Furthermore, the server notifies the agent of specific events that happened

---

[35]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/config.py`

in the game, to which the agent could react by resetting the environment and starting a new episode. The agent must therefore implement this functionality additionally to the regular action performance or learning-step conductance, while the inner workings of the server are hidden to the agent.

Inheritance relation   To ease the implementation of agents, they all inherit from a superclass called `AbstractAgent` as well as a class termed `AbstractRLAgent`, both of which are defined in `agent.py`[36]. The key idea is, that all functions and features mutual to all agents are set in these classes, such that any specific agent must only implement/overwrite the functions in which it differs from its superclasses. Because not all agents use reinforcement learning, it was decided to use two superclasses instead of only one – the class `AbstractAgent` does for example not specify a memory, as purely pre-trained agents do not need it.

In the scope of this thesis, multiple agents where developed, using different models, `features` and approaches. In the final version of the implementation there are five agents, which can all be found in the directory `agents`[37].

Figure 4.2 depicts the inheritance-relation of an agent and its super-classes using an UML-diagram.
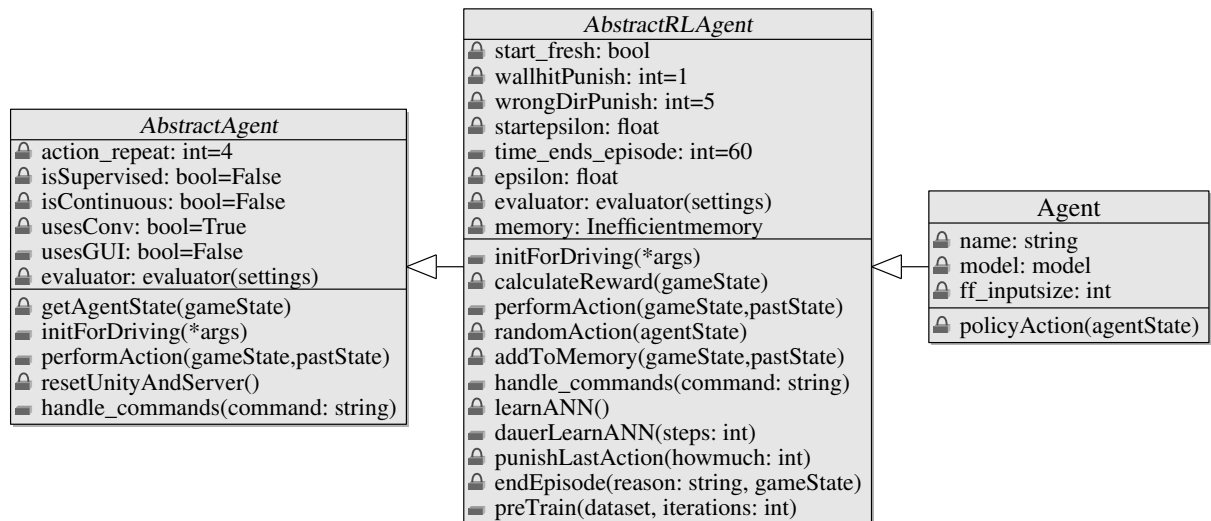
FIGURE 4.2: (incomplete) UML-diagram of an agent and its super-classes.

To add a new RL-agent, a class extending `AbstractRLAgent` that specifies at least `name`, `ff_inputsize`, an initialized `model` and the function `policyAction(agentState)` must be added to the `agents`-directory, an exemplary agent that does so is listed in appendix C.1. The agent's `name` is as directory-name to save its data to. A non-RL agent must only provide the method `policyAction`, next to any kind of model and, if needed, a `preTrain`-method.

To use any agent, its filename (without extension) must be supplied as argument when running the server, such that it gets automatically imported and initialized:
`python server.py --agent ddpg_rl_agent`. To add a model, it must be stated in the `models`-directory and used by an agent.

Agent-state and game-state   Most of the functions an agent specifies are only for internal use – the only ones which are called from outside (by the server) are the ones for initializing

---

[36] `https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agent.py`
[37] `https://github.com/cstenkamp/BA-rAIce-ANN/tree/master/agents`

the agent, performing an action, reacting to (reset-) commands and performing pretraining (marked as public in diagram 4.2)

As can be seen in diagram 4.2, some methods require a `gameState` (or `pastState`) as argument, whereas others require an `agentState` . Inside this implemention, those are both clearly defined. A `gameState` is what is provided by the server, namely a tuple of ⟨ `visionvectorHistory` , `visionvector2History` , `otherinputHistory` , `actionHistory` ⟩. This tuple is used to calculate the agent's observation in the function `getAgentState(gameState)` . An agent-state is defined as a tuple consisting of ⟨ `convolutionalInputs` , `otherInputs` , `standsInput` ⟩. If used by an agent, `convolutionalInputs` consists of stacked 2D-matrices originally stemming from the minimaps, whereas all non2D-input must be given in `otherInputs` . It was decided to seperate the `convolutionalInputs` from the `otherInputs` such that hybrid ANNs, having both a convolutional as well as a feed-forward component, have a distinct input for either. A distinctiveness of this implementation to others is the additional usage of `standsInput` , which is a boolean that is supposed to be `true` if the car currently stands. When having such an input, models can easily be hard-coded to prevent situations in which the car stands undesirably still. The internals of the function to create the agent's observation (`agentState`) from the `gameState` depend on the agent's `features` and is a result of experimentation, as such described in chapter 5.

Next to `getAgentState(gameState)` , there are other helper-functions specified in `AbstractAgent` , like for example `getAction(gameState)` . This function returns the agent's last action in a way that can be used by the agent's model. This function is necessary, because agents may use a non-continuous model (like DQN), for which the action must be discretized at first. As the `AbstractAgent` was implemented with DQN-based agents in mind, it provides the function to discretize and dediscretize the actions, which wraps such a function from the file `read_supervised.py`. The function `makeInferenceUsable(state)` combines those functions for certain kinds of states, such that it is the only one needing to be called whenever the agent accesses its `model` .

Methods called by the server   In their `__init__` , called when server creates the agent, the classes `AbstractAgent` and `AbstractRLAgent` define some standard settings for the configuration of their `features`, which can be overwritten if an agent's actual configurations differs. When initializing its memory or its model, the agent also passes a reference to itself, such that these configuration-variables are accessible from inside those objects.

The server calls not only the `__init__` , but also the `initForDriving` . In this function, the agent's memory is (according to its `features`) initialized, as well as an `evaluator`. The evaluator is an independent object, specified in `evaluator.py`[38]. It provides functions to save information about the agent's driving performance in the form of running averages of the agent's rewards, q-values, progress per episode or others to an XML-file. The result of the respective file can be inspected by running `python show_plot.py --agent FILENAME` , which was also used to produce the plots from chapter 6. If the application is not run with the argument `-noplot` , the evaluator also plots these running averages live, using the library `matplotlib` (note that the number of iterations as stated by the live-plotter restarts every new run). The agent uses the evaluator by calling its method `eval_episodeVals` , passing respective parameters. The performance of a human can be evaluated by using any kind of agent that uses the evalutor and overwriting the agent's actions by pressing Ⓗ. An exemplary live-plot can be inspected in figure B.4.

During runtime, the agent's `performAction(gameState, pastState)` is called. It is provided by `AbstractAgent` (and overwritten by `AbstractRLAgent`), and does not need to be changed

---

[38]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/evaluator.py`

by agents in the general case[39]. The function checks if an action should be taken, converts the given `gameState` into an `agentState` and provides the action. Where this action is taken from is decided in this function – it can use the previous action (a variable `action_repeat` is specified in it with similar reasoning than that of the DQN[20]), a random action (by calling `randomAction(agentState)`) or an action according to its model (`policyAction(agentState)`).

The implementation given in the `AbstractRLAgent` assumes a simple ϵ-greey approach, such that `randomAction` is called with a probability of ϵ, where epsilon decreases over time. In agents where it is desired to overwrite this exploration method, either the `performAction`-function can be overwritten, or (as done in the provided DDPG-based agents) the `randomAction` simply returns `policyAction`, and the actual exploration-technique is integrated into the function `policyAction`. The method `performAction(gameState, pastState)` takes care for updating the server's `outputval`, such that all issues of communication (next to eg. having to work with the `gameState`) are abstracted away from `policyAction` and `randomAction` (and thus from everything a rudimentary agent implements).

As mentioned, the server does not only call the agent's `performAction`, but also its function `handle_commands(command)` if it got a *special command* from the client (environment). The `handle_commands`-function gets as input the string providing information about what this command was. In the current implementation, the existing commands are `"wallhit"`, `"lapdone"`, `"timeover"` and `"turnedaround"`. The default reaction, specified in `AbstractRLAgent` is to end the training episode (with `endEpisode(reason, gameState)`) for either of those commands – after possibly giving explicit negative reward for the last seen state using `punishLastAction(howmuch)`. The necessity of these two functions is an artifact of the design decision to let the agent decide on when an episode ends. In this implementation, the main loop is not nested in an outer loop that recognizes when an episode ends, but instead the agent is only notified of the end of an episode after it stored the last state already in its replay memory. Using those two functions, the last addition to that can be changed in retrospect, such that the flag $t == t_t$ can be set to `true`, indicating a final state. `endEpisode` also calls the evaluator to save and plot the respective running averages.

All `features` that are not needed for supervised agents are only specified in `AbstractRLAgent`. This includes the methods to calculate reward, performing random actions, managing its replay memory and reinforced learning.

Learning   It is possible to use the `"parallel"` learing nmode or the `"between"` learning mode (as specified in `conf`). If set to the latter, `performAction` starts the execution of a specified number of learning steps in a specified interval, according to `conf`'s configuration of `ForEveryInf` and `ComesALearn`. This corresponds basically to the `update frequency` of the DQN[20]. Learning is specified in the method `learnANN()`, itself executed by `dauerlearnANN(steps)`. To perform the actual learning, the former extracts a batch of a specified number of transitions from its replay memory using `create_QLearnInputs_from_MemoryBatch` and performs the model's method `q_train_step` on the extracted batch.

Performing a q-train-step takes significantly longer than performing an inference. While the given agents are fast enough to perform an inference-step in between two updates with new information about the game (everything that happens in the main loop between the Unity-client sending data and doing it again in figure 4.1), they loose a significant amount of time when also having perform a train-step. Because of this, additionally to Unity's client automatically pausing the game if Python's response is delayed too much (see section 4.2.2), the class `AbstractRLAgent` includes methods ask the client to freeze itself. In doing so, it uses another string-array `freezeInfReasons` to which a reason is appended for every call of

---

[39]A (minimally abstracted) sourcecode of this function is found in lines 14-35 of appendix A.

`freezeInf(reason)` , and popped for every call of `unFreezeInf(reason)` , with similar reasoning than described for the `QuickPause`-functionality of the game. If the respective conditions apply, a message `"pleaseFreeze"` is sent to the client, such that the agent has time to perform its learning-steps. Once it is done, it notifies the client by sending `"pleaseUnFreeze"` . As it was a design decision that the game runs as smoothly as possible, it was decided to not rigidly perform one learning-step every `update frequency` steps, but to increase the respective batchsizes (for example 100 learn steps every 400 steps), such that the game is not constantly interrupted due to learning steps.

Because the agent was also developed with the learn mode `"between"` in mind, respective functions can also be found in the agent and server. If this mode is set, then the method `dauerlearnANN` is run by the server in a separate thread. To keep the ratio of learning steps and inference roughly constant at all times, the methods `checkIfAction` and `dauerLearnANN` provide functionality to freeze either learning or inference, if the respective thread was faster than the other. For that, a `freezeLearn` -method is necessary, which sets a respective boolean value to `false` , stopping the learning. A ratio of $4 : 1$, as already done in the original DQN, showed to decrease the amount of time any thread must be frozen the most.

GUI   As mentioned before, the implementation also features a basic GUI, to which current information about the agent's latest values can be sent to. The typical view of this GUI is shown in figure B.5 in appendix B. For a respective agent to use this GUI, some functions must be overwritten to also include those printing-functions. To do so, an agent must import the infoscreen, such that its functions can simply call their respective counterpart of the agent's super-class and print its result to the infoscreen using `infoscreen.print` . The functions that are overwritten to show a GUI as in figure B.5 are `eval_episodeVals` , `punishLastAction` , `addToMemory` , `learnANN` , `policyAction` and `calulateReward` . Most agents of the given implementations do so, the reader is for example referred to agents/ddpg_rl_agent.py[40]. Note that an exception is the added code of `calculateReward` , which does not update a text-element of the GUI, but a color. This color is a gray-scale value whose saturation depends on a scalar value between zero and one. This feature is useful to inspect if the agent's reward-function is useful – the lighter the color-area, the higher the current reward.

Memory

During its learning process, the agent stores every transition it encounters in its replay memory, such that its ANN can learn from uncorellated samples. Following the tradition of the DQN[20], it is a buffer of limited size, consisting of tuples $\langle s_t, a_t, r_t, s_{t+1}, t == t_t \rangle$. This buffer is wrapped in a class of type `Memory` , specified in either in the file inefficientmemory.py[41] or efficientmemory.py[42]. Note that while the agent's `addToMemory(gameState, pastState)` is called with the `gameState` , this function converts both `gameState` and `pastState` to the respective `agentState` to save on its memory consumption. It contains methods to change the last stored transition's reward in retrospect as well as to update if the episode ended in the last step – the values for reward and if the state was terminal in `addToMemory` are thus possibly only provisional.

When performing a q-training-step, a random sample of size `conf.batch_size` is sampled from the replay memory. It is established that `Prioritized Experience Replay`[26] increases the agent's speed of learning, but while the implementation allows for such an addition, it is not incoporated yet.

---

[40]https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/ddpg_rl_agent.py

[41]https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/inefficientmemory.py

[42]https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/efficientmemory.py

The memory-classes of this implementation can be saved to a file using Python's `pickle` - library. The memory is generally saved together with the agent's model, using its function `save_memory()`. Because the memory can become very large (consisting of hundreds of thousands of transitions), both inference and learning are frozen while the memory is saved. To be more failsafe, the agent is always stored twice (with both files containing the exact same memory). If the computer runs out of storage while one of these files is being created, the respective file is corrupted – however one backup-copy will always be save.

In the given project, there are two memory-classes provided which both contain the same methods, where the one (`efficientmemory.py`) is used by certain agents, as it is more efficient in those ones, namely the ones using `history_frames`. The reason for this is the following: Let's assume an agent's observation consists of the current gameState as well as the three ones before that, corresponding to four history-frames: $agentstate_t = (state_t, state_{t-1}, state_{t-2}, state_{t-3})$. A tuple added to the memory contains its current state and the state before that: $agentstate_t$ and $agentstate_{t-1}$, which corresponds under the previous definition of an agent's state to the following environment-states: $(state_t, state_{t-1}, state_{t-2}, state_{t-3})$ and $(state_{t-1}, state_{t-2}, state_{t-3}, state_{t-4})$. The eight saved states contain only five different states. In fact, states are even more redundantly saved, as the next inference of this hypothetital agent saves the following states: $state_{t+1}, state_t, state_{t-1}, state_{t-2}$ and $state_t, state_{t-1}, state_{t-2}, state_{t-3}$, where only one state is actually new. It is easy to see, that in this scenario an implementation that always adds $agentstate_t$ and $agentstate_{t-1}$ is very inefficient, using roughly 8 times as much storage as needed. The class specified in `efficientmemory.py` works around those problems by always adding only the latest frame to the replay-memory. While this file is internally far more efficient than the inefficient memory for agents using many history-frames, it can be interfaced just like `inefficientmemory.py`, such that they easily be exchanged, providing the exact same values in all situations[43]. If an agent uses many history-frames (especially relevant when using minimaps as input), it can load the efficient memory in its `initForDriving` -function. If no such is given, the inefficient memory is loaded by default.

Pretraining

As stated in section 4.2.2, additionally to reinforcement learning functionality, pretraining is enabled in this implementation by functions to record and export manual driving data to XML. By running the file `pretrain.py`[44] with the respective agent as command-line parameter, all exported data which has the file-ending `.svlap` and is stored in the directory `conf.LapFolderName` is used to pre-train the agent.

The for pretraining exported data consists of all `possible vectors` (corresponding to the game's state) for every point in time it recorded – because the data is independent of individual `agentState` s, it can be used by all agents. Additionally, the data in the XML saves the time of recording (system's time as well as Unity-Time), speed, progress, and explicit user-input (`throttlePedalValue` , `brakePedalValue` , `steeringValue`). The file `read_supervised.py`[33] contains (among others) the necessary classes and methods such that this XML-data can be converted into a format an agent can use for pretraining. For that, it contains a definition of a `TrackingPoint` -class. This class is a counterpart of the game's class of the same name, defined in `AiInterface.cs`. Furthermore, `read_supervised.py` contains a class `TPList` , which corresponds to a complete usable dataset of driving-data.

---

[43]It could be argued that the memory is different after a reset if it only adds the latest part of the transition, however a solution for that problem is simply overwriting the latest transitions in that case. A proof that both memorys behave exactly equal is given in an older branch of the repository: `https://github.com/cstenkamp/BA-rAIce-ANN/tree/newmemory`

[44]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/pretrain.py`

Upon execution of `pretrain.py`, the dataset (an instance of type `TPList`), is created. In its `__init__`, it reads all `.svlap`-files and creates a `TrackingPoint` for every point in time saved in either of those files.

There are two specific characteristics in this implementation. First of all, it seems natural to provide a `TrackingPoint` with state and corresponding action from the same point in time. However, all actions from the server will be delayed due to the time needed for sending the state, deciding on an action and executing it. Because of that, the function `condider_delay` remaps an action to a gameState a few milliseconds before the respective action. The second characteristic is, that supervised data is generally exported in a higher frequency than what the agent performs at. The millisecond-time between two datapoints is saved in the XML, such that only every n^{th} datapoint will be used using the function `extract_appropriate`. This is performed separately for each file, such that the dataset consists of samples with equal distance of time. To iterate over its contents, the class `TPList` provides the methods `reset_batch` and `next_batch`.

In its main method, the application `pretrain.py` calls the method `preTrain` of an agent, passing a `TPList`-dataset. This function, as specified in `AbstractRLAgent`, loops over the entire dataset for a specified number of iterations. Every iteration, it takes minibatches of size `conf.pretrain_batch_size` with the method `next_batch`, to be used as batch for the model to learn on. While it can be specified with the command-line-argument `"-supervised"` to use the model's `sv_train_step`-function instead of the `q_train_step`-method, only the implemented DQN-model includes this method. To convert a batch of `gameState`s (as provided by `next_batch`) into a batch of `agentState`s, the agent uses the dataset's static function `create_QLearnInputs_fromBatch`, passing a reference to itself. Passing its reference is necessary because this method uses the agent's observation-functions on the batches from the dataset.

In the actual implementation, the `AbstractRLAgent`'s `preTrain`-method is implemented a bit differently, the agent cannot learn useful policies with q-training on an existing dataset created by a completely different policy, as will be handled by section 6.3.

Models

The class `AbstractAgent` does not access any functions of the agent's model. Thus, an agent that only extends this class can use any kind of model in its `policyAction(agentState)`-function. The `random_agent.py`[45] for example simply assigns random values as its actions.

An agent however that extends `AbstractRLAgent` must provide a model that implements the interface[46] specified in figure 4.3. In the functions specified in that diagram, the abbreviations [s], [a], [r] and [t] stand for arrays of `agentState`, `action`, `reward` and the boolean flag $t = t_t$ (`terminal`). The methods provided by the interface are all the ones necessary to perform adequate Q-learning (or DDPG).

It is the responsibility of a model to save the information about how many inferences it already conducted as well as its number of learning steps for pretraining as well as reinforced training. To differentiate the former from the latter, both models contain a flag `isPretrain`, which is passed upon initialization and specifies if the pretrain-episodes and steps are assigned to the respective counters for pre-training or actual training, and forbids inferences if set to `True`.

When an agent initializes its model, it must pass a reference to itself to it. The model thus knows the agent and can access some of its variables representing the configuration of its features.

---

[45] `https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/random_agent.py`

[46] Note that no such interface is actually implemented, as doing so in Python is not necessary because of its *duck typing* design pattern.

```
                    ┌─────────────────────────────────────────┐
                    │              <<interface>>               │
                    │                 Model                    │
                    ├─────────────────────────────────────────┤
                    │ ⬤ isPretrain: bool                       │
                    ├─────────────────────────────────────────┤
                    │ ⬤ __init__()                             │
                    │ ⬤ initNet(load: bool/string)             │
                    │ ⬤ save()                                 │
                    │ ⬤ inference(oldstates: [s]): [a], _      │
                    │ ⬤ getAccuracy(batch: [s],[a],[r],[s],[t]>, _): float │
                    │ ⬤ statevalue(oldstates: [s]): float      │
                    │ ⬤ qvalue(oldstates: [s], actions: [a]): float │
                    │ ⬤ q_train_step(batch: <[s],[a],[r],[s],[t]>): float │
                    │ ⬤ run_inferences(): int                  │
                    │ ⬤ step(): int                            │
                    │ ⬤ pretrain_episode(): int                │
                    │ ⬤ inc_episode()                          │
                    └─────────────────────────────────────────┘
```
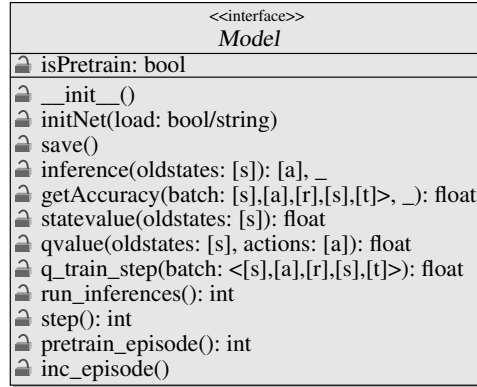
FIGURE 4.3: UML-diagram of the interface a model must implement

Functionality provided by AbstractRLAgent

As can be seen in figure 4.2, an agent that extends the class `AbstractRLAgent` only needs to specify a model (implementing the interface 4.3), a name, the feed-forward inputsize for its model and the method `policyAction(agentState)`. This is because a default version all other methods, even though they are features of the agent, is already implemented in the `AbstractAgent`.

The following lists describes situations in which some of this functionality must be changed.

- An agent that uses any kind of model is likely to overwrite the `__init__` -function to initialize its model.
- An agent that uses a GUI must overwrite the functions described in the paragraph **GUI** on page 50.
- If the agent features a different observation-function than provided, it must overwrite the function `getAgentState(gameState)` and possibly `makeNetUsableOtherinputs`.
- If the agent's model does not discretize, an agent must overwrite `makeNetUsableAction`.
- If the agent uses another exploration-technique than ϵ-greedy, it must either overwrite `randomAction(agentState)` or `performAction(gameState, pastState)`. This may include changing `startepsilon`, `minepsilon` and `finalepsilonframe`.
- If the agent uses another method to calculate random actions than the one provided, it must overwrite `randomAction(agentState)`.
- If the agent calculates the reward in a different manner than provided, it must overwrite the method `calculateReward(gameState)`.
- If the agent performs pretraining differently than specified, it must overwrite the method `preTrain(dataset, iterations, supervised)`.
- If the agent uses another memory, it must assign `self.memory` in `initForDriving`.
- If the agent is supposed to reset the environment due to different events, the method `handle_commands(command)` must be overwritten. This may include changing `wallhitPunish`, `wrongDirPunish` or `time_ends_episode`.

5 Results – Implementation

This chapter provides the implementation of the `features` of the implemented agents. It will start with the `possible vectors` that are sent to any agent and can be used. Afterwards, the agents that where implemented in the scope of this thesis are explained, including all their features and why they were selected this way. These features include the used `model`s, the exploration-functions, their specific observation-functions (using the `possible features`), their methods of incorporating pretraining as well as their reward-functions.

It is worth to note, that some of these functions (as for example the `calculateReward(gameState)` function) are implemented not in the individual agent, but in `AbstractRLAgent`, as they would otherwise have to be redundantly implemented multiple times.

5.1 Possible vectors

`Possible Vectors` refers to all information the game streams over its socket to the agent. While most of this information will be used to make up an agent's state, the vectors also provide information about if the game must be reset as well as information to calculate the reward from. This information is collected in Unity in the function `GetAllInfos()`, and converted into a namedtuple-wrapper called `Otherinputs`, specified in `read_supervised.py`. In this section, I provide an overview of those vectors and their meaning in the game. I refer to the individual possible vectors by the name used in `Otherinputs`.

ProgressVec This vector contains information about the current progress of the car on the track, which consists of the actual progress in percent, the time the car needed for the current lap so far, the number of the current lap, as well as the flag if the round is still *valid*.

SpeedSteer In this vector, information about the car's velocity and its steer angle is encoded. It consists of the following values:

RLTorque	The motor torque applied to the left back tire
RRTorque	The motor torque applied to the right back tire
FLSteer	The steering angle of the front left tire
FRSteer	The steering angle of the front right tire
velocity	The velocity of the car as scalar independent of directions
rightDirection	A boolean value if the car moves into the intended direction
velocityOfPerpendicular	The velocity of the orthogonal projection of the car onto the center of the street
carAngle	The car's angle (in degrees) in relation to the street's direction
speedInStreetDir	The car's velocity into the street's direction (calculated using the dot-product between the car's velocity-vectors and the direction-vector of the street at the car's current position)
speedInTraverDir	The car's velocity into the orthogonal of the street's direction
CurvinessBeforeCar	A measure of the curvature of the street immediately ahead of the car ($CurvinessBeforeCar \in [0, 1]$, where a value of zero corresponds to a straight street)

StatusVector   This vector contains eight values, corresponding to the forward slip and sideways slip of each of the car's tires, using a function provided by Unity's `WheelCollider` -object. The more the car slips, the smaller the impact of movement commands. The current slip-values are presented in the GUI of the game (and can be seen behind annotation **R** in figure B.3).

CenterDist and CenterDistVec   The *CenterDist* corresponds to the car's orthogonal distance to the street's center in the form of the *perpendicular* (also visually represented in the car's GUI, behind annotation **N** in figure B.3). The *CenterDistVec* contains the same information presented in another way: It is a vector of length 15, where the middle element corresponds to the car's longitutional position. The other elemnts correspond to points with regular distances to the car's left and right. The value of each respective element is calculated using the reversed distance between this position and the longitudional center of the street. The content of this vector is visually represented behind annotation **M** in figure B.3.

WallDistVec   This vector contains seven values, corresponding to the car's distance to the wall along a certain ray. It does not contain the closest distance to the wall – because this value is already represented by the CenterDist. As the wall has always a fixed distance from the street's center (with an absolute value of five), the distance to the closest wall can be calculated as $5 - abs(CenterDist)$. For the calculation of the *WallDistVec*, several rays are casted from the car's (or the perpendicular's) position into a particular direction. Returned is then the distance from their respective origin and their first intersection with a wall. The vector contains seven values, using rays with different origins and different directions. In the following table, I provide an explanation of each of those values, while figure B.6 in appendix B visually represents these rays. The respective color is mentioned in the table.

#	color in B.6	origin	direction
1	black	car's center	direction the car faces
2	magenta	car's center	direction the car steers
3	red	car's center	direction the car moves
4	white	car's center	short-sighted direction of the street (calculated as the vector between the closest `anchorVector` behind the car and the closest one before the car)
5	yellow	perpendicular	short-sighted direction of the street (calculated as the vector between the closest `anchorVector` behind the car and the closest one before the car)
6	blue	car's center	long-sighted direction of the street (calculated as the vector between the closest `anchorVector` to the car and the one 5 in advance)
7	gray	perpendicular	long-sighted direction of the street (calculated as the vector between the closest `anchorVector` to the car and the one 5 in advance)

LookAheadVec This vector corresponds to the course of the street ahead of the car. It is a 30-dimensional vector, corresponding to regularly spaced `anchorVector` s, starting at the position of the car, following the direction of the street. The value at each position i of this vector corresponds to the angle between the direction of the street at position i and the direction at position $i + 1$. In other words, if the street makes a shart turn 4 units ahead of the car, then element 3 will contain a high value. The angle is measured in degrees. A graphical representation of this vector can be seen behind annotation **N** in figure B.3.

FBDelta This vector consists of two values, namely *Feedback* and *Delta*. The Feedback-value is the temporal difference of how long the car needed for a specific section of the course (constrained via the a `anchorVector` close to the car) in the current process in comparison to the time needed in the fastest lap so far. The Delta-value is the absolute difference in time needed for the entire lap so far.

As both of these values are only useful in relation to the time needed for the fastest lap, it must be ensured that this does not change during training. For that, The file `AiInterface.cs` contains in its class `Consts` a flag `lock_fastestlap_in_AIMode`. Note however, that even if this flag is set, the values of *FBDelta* could at most be used to calculate the agent's reward and not as part of its state.

Action This is a three-dimensional vector corresponding to the actual action the environment recorded. While the agent knows what action it provided, it could be manually overwritten (after the press of H) – which is why the agent stores this vector in its replay memory instead of the action it would have performed otherwise.

minimaps While the minimaps are not contained in the namedtuple `Otherinputs`, their value is transmitted from the game just like the other vectors. In the current implementation, both cameras have a resolution of 30×45 pixels, where one camera is 15 units away of the car, and the other one 75 units, which means that the former provides a smaller but more detailed view of the car. The closer camera is mostly referred to as $vvec2$, whereas the other

one is denoted *visionvec* or *vvec*.

As a working game was already provided by the first supervisor of this thesis, so were some of these vectors, namely the calculation of FBDelta, LookAheadVec, CenterDist and CenterDistVec.

It is worth noting that most `possible vectors` are normalized after loading, according to (in part estimated) minimal- and maximal values, such that all their values in [0,1]. The corresponding `MINVALS` and `MAXVALS` are defined in `read_supervised.py`

5.2   Implemented models

Within the context of this thesis, two different models were developed that can be used for agents to learn and play the given game: a *Double Dueling Deep-Q-Learning* model, specified in the class `DDDQN_model` in `models/dddqn.py`[1] as well as a *Deep Deterministic Policy Gradient* model, specified in `DDPG_model` in `models/ddpg.py`[2] The theory of the model was explained in chapters 2.3.1ff and 2.4.1, respectively.

Both models can take two-dimensional as well as one-dimensional input and return outputs corresponding to the action that needs to be taken, which are three real numbers. One has to be aware though that while a DDPG-model can naturally return such, a DQN-model only works for discrete actions. Because it was developed with DQN-models in mind, the `AbstractAgent` contains functions to `discretize` the action-value returned by the game to a one-hot vector to be used by the model, as well as functions to `dediscretize` a one-hot vector from the model that can be used by the environment. It is obvious, that discretizing actions has severe disadvantages: If the discretization is very fine-grained, the action-space becomes intractably big due to a combinatorial explosion (the *curse of dimensionality*), whereas if a discretization is coarse, much of the information is lost and precise steering becomes impossible. A further downside of discretizing actions into one-hot vectors is, that it limits the design space of exploration strategies, as information about similiarity of actions is lost and only the uninformed ϵ-greedy mechanism remains possible.

While the learning technique differs in both implemented models, both implement the interface provided in figure 4.3, such that the functions relevant for the agent are accessible in the same way. An UML-diagram of most of the agent's functions as well as the interface is given in figure 5.1. If an agent does not need to discretize actions, it must overwrite the method `getAction(gameState)`.

Both implemented models provide the possibility to load and save a model from/to the harddisk using TensorFlow's saver-class. If a model is loaded in the `isPretrain`-mode, it can be saved to file and loaded again such that it is usable. If the model is saved, the information about its already performed numbers of inferences and learning-steps are saved within Tensor-Flow as well. When a model is saved, it saves into the directory `conf.pretrain_checkpoint_dir` if the respective flag is set, and `conf.checkpoint_dir` otherwise. When loading a model (in `initNet(load)`), the `load`-parameter specifies if the model should be loaded from the pretrain-directory, the non-pretrain-directory or none at all.

While both models are implemented to be used by an agent as described above and in the described situation, they are general models for reinforcement learning – and are as such usable to learn reasonable policies in any task described as Markov decision process. To show the generality of the implemented models, a file `gym_test.py`[3] is provided. In this file, the implemented `model`s are used to solve arbitrary `OpenAI-gym`-tasks. As the program flow of the

---

[1]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/models/dddqn.py`

[2]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/models/ddpg.py`

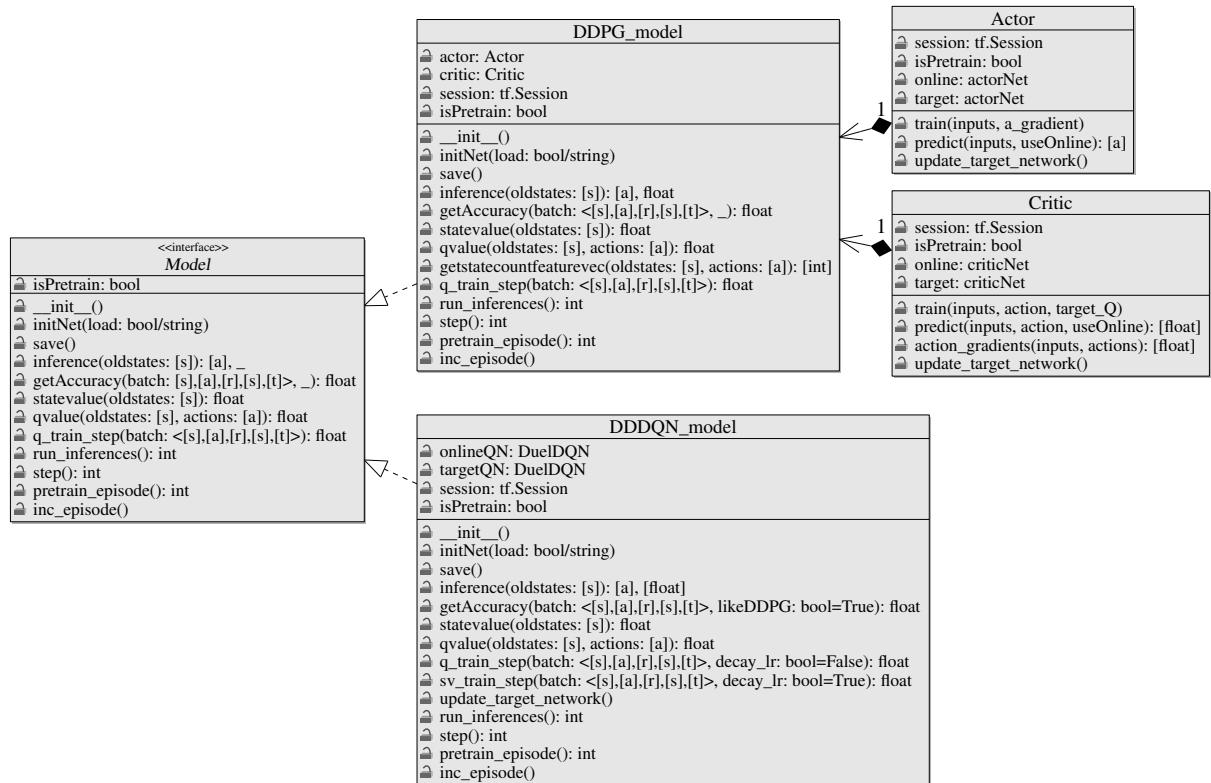[3]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/gym_test.py`

FIGURE 5.1: UML-diagram of the two given models as well as the interface both of them implement

interaction with a gym-environment differs to that one of the implemented program, the file contains specialized definitions of `agent`, `config` and `memory` to work with the API-schema of `gym`. As gym-environments explicitly define `terminal` and `reward`, the `agent` must for example not work with the `gameState`, but contains only methods using the immediate `agentState`. The way how an agent accesses its model is however equal to the agents for the given game. The `gym_test.py` can be used to test if a model is functional – if a model works on several gym-tasks, it can be assumed that it is implemented correctly. Having such a method at hand is very handy during the implementation of reinforcement-learning agents, as it provides a reliable method to check where the reason for any errors may lie. Both implemented models were successfully tested on several tasks each[4].

In the following two sections, I will at first explain the DQN-model and afterwards the DDPG-model. Note that both models are used in multiple agents each. The amount of input-values may differ from agent to agent, and the presented network architecture is specific to one of the implemented agents as stated.

5.2.1 DQN-model

In the `Drive`-mode of the game, throttle and steering are both binary, but it is still possible for Users to drive the course well. Because of that, it was decided to keep steering and throttle binary for this model as well to reduce the dimensionality of discretization. Further, simultaneous activation of throttle and brake was forbidden. Because of these design choices, an action

---

[4]Note however, that using other gym-environments than the ones on which was tested (`Pendulum-v0`, `FrozenLake-v0`, `MountainCar-v0` and `CarRacing-v0`) may require implementing specific exploration or preprocessing techniques. Note further, that some environments can only be solved with models that allow for continuous action-spaces.

for a DQN-model is a one-hot vector of the size `3*conf.steering_steps`, with steering_steps set to 7 in the current version.

The class `DDDQN_model` has two `DuelDQN`s, which are Deep Convolutional Neural Networks with Dueling Architectures, specified as computational graph using TensorFlow – one of them being its online network, and the other the target network. The `DDDQN_model` combines them for the learning method of Double-Q-Learning (as detailed in section 2.3.1ff as well as appendix A) in its method `q_train_step`.

The provided DQN-model can however not only learn via Q-learning, but can also be trained supervisedly. For that, it specifies the function `sv_train_step`. This method can only be called if `isPretrain` and learns directly on its target network. For that, the used `DuelDQN`s must provide a TensorFlow-placeholder for the target-action. An agent that pretrained supervisedly however should not subsequently be used for reinforcement learning (an explanation for that will follow in section 6.1.1).

It was mentioned before that the model incorporates hard-coded domain-specific knowledge. Next to the ban of simultaneous braking and steering as action, the model explicitly forbids actions that do not accelerate the car if it is standing. As already mentioned, part of the `agentState` is a `stands_input`, which the method `getAgentState` sets to `true` if the car's velocity is below a certain value. The model consistently has a `stands_input` expecting such a boolean value. Note that the model's `make_inputs` also actively sets this value to `true` if the car stood up to a specified number of inferences before the current one. If this value and the respective option `conf.use_settozero` are both `true`, then the model sets in its function `settozero` all Q-values of actions that do not include accelerating the car to zero. As a DQN-based model selects its actions using a *greedy* strategy, it always selects the action with the highest Q-value – which is now the maximum of all Q-values that include the action of accelerating the car. Note however, that the model's `stands_input` is only set to `true` in inferences, and not in learning-steps.

The network architecture of the two `DuelDQN`s that the agent uses is described in the next section. Note that a `DuelDQN` is a particular class known to the agent. All TensorFlow-Layers, inputs, outputs as well as a `saver` and certain collections can be accessed from the `DDDQN_model`. As explained in chapter 2.3.1, the use of both online-network to train on as well as target-network to sample experiences from is necessary for adequate performance. As suggested by [15], the implementation of this DQN uses *soft target-updates*, meaning that after every `q_train_step` the relevant weights of the target-network are moved by a factor $0 < \tau \ll 1$ into the direction of the online-network. To do so, both networks maintain a collection of their trainable variables, `trainables`. The respective assignment operations are added to the computational graph as `smoothTargetNetUpdate`, specified in the model's `__init__`. After initializing a model or loading it from a file, a similar assignment operation is performed to ensure that both networks are equal.

Network architecture

An overview of the general network architecture, as used by the `dqn_rl_agent`, is provided in figure 5.2. Note however that the actual architecture differs in other agents, as it depends on the agent's configuration of `features`. As some agents do not use the minimaps as input, their implementation for those does not use any of the convolutional layers from the upper line of the figure. When initializing a `DuelDQN`, a reference to the agent is passed, such that the network knows how many input and output-neurons to specifiy.

If convolutional input is specified, four convolutional layers layers process the input to 256 feature-maps of size 1×1, which can subsequently be flattened to a one-dimensional layer. Note that there are no pooling layers in between convolutional layers. As suggested by [30],
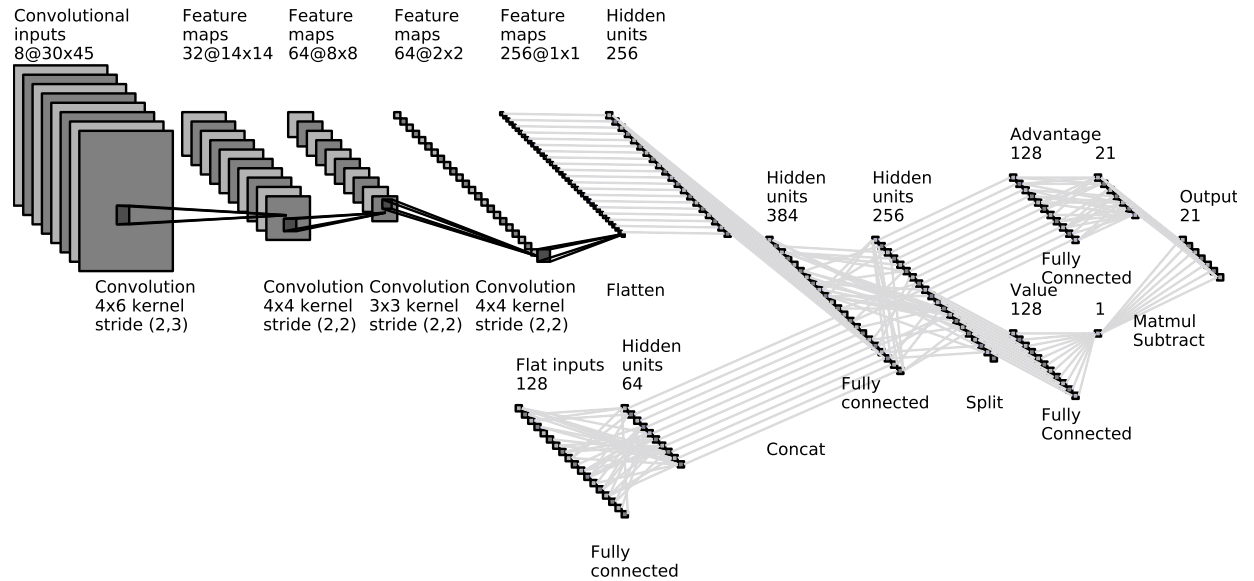
FIGURE 5.2: The used convolutional DQN with a dueling architecture

pooling leads to a loss of localization information and can be discarded in favor of a larger `stride` in the convolutional layers.

While the two-dimensional `conv_inputs` are processed with convolutional layers, `ff_inputs` are fully connected to a hidden layer of variable size. This layer is concatenated to the flattened output to the convolutional layer, the result of which is densely connected to another hidden layer of 256 neurons. This layer is then, following the dueling architecture from [34], split into a separate *advantage stream* and *value stream*. While the value stream is fully connected to one hidden layer (corresponding to the state-value), the advantage ends in one output neuron for each action, which is 21 in the current implementation. The advantage stream and value stream are finally combined (as described in section 2.3.3) to yield 21 output neurons, corresponding to one Q-value for every action in the provided state.

Informal search led to the selection of the rectified non-linearity (`tf.nn.relu`) as activation function as well as Adam[14] as optimizer. All weights of this network are initialized around zero (or slightly higher, to prevent *dead neurons* in combination with the relu activation-function) with only a very small standard deviation (10^{-20} up to 10^{-3}), as doing otherwise showed to impair the agent's performance after a fresh start.

In the current implementation, the DQN-model does not perform *batch normalization*[11], as it showed to impair the agent's performance. The functions `convolutional_layer` and `dense`, which are used to initialize the layers, wrap respective TensorFlow-functions and can be found in `utils.py`[5].

The final structure of this network was subject of much experimentation. Any used hyper-parameter that does not correspond to its counterpart in [20] or [34] is the result of informal search, showing the best performance so far. This does by any means not mean that the parameters are optimal. Using this structure, the network performed reasonably well on given OpenAI gym-tasks, as explained earlier in this chapter.

5.2.2 DDPG-model

A DDPG-model must incorporate four ANNs to work correctly: an online- and a target version of both `actor` and `critic`, as found by [15]. While the online networks will be used for

---
[5]https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/utils.py

online predictions and will be updated at every timestep, the target networks will be used for determining the directions into which the online networks should be updated. The given implementation of the `DDPG_model` thus `has` an actor and a critic, which in turn have two `actorNet`s or `criticNet`s, respectively. If the `save()`-method of `DDPG_model` is called, it calls the respective functions of actor and critic, which save their individual variables individually. Because of that, the same critic could be used for different actors or vice versa. The information about the numbers of current inferences and others is saved in and loaded from the actor.

As the `DDPG_model` `has` `actor` and `critic`, it can access all their values and methods. If it accesses any of these, it specifies as argument if it wants them to internally use their online- or target network. Both actor and critic provide respective methods to update their target network.

It is not possible to use the same network structure as used in the `DQN_model` in any network of the `DDPG_model`. The actor returns continuous values which correspond directly to the action instead of a `softmax`-distribution over possible discrete actions. The critic needs, in contrast to the DQN-architecture, the actions as additional input to return one single Q-value. It is obvious that the `dueling architecture` cannot be adopted for the DDPG-critic.

In its `q_train_step`-method, the `DDPG_model` trains both its actor and its critic by calling their respective methods – the theoretical basis of this learning step is explained in section 2.4.1, while appendix A.2 compares the source-code to the pseudocode given in [15]. While normally TensorFlow minimizes losses via an optimizer (and also does so in the critic), it also allows the possibility to `apply_gradients`, which optimizes all elements of its respective computation graph into the direction of the supplied values.

As the actor only needs the critic's gradients once, these can simply be passed into its `feed_dict`. Besides this interaction, actor and critic can be implemented completely independent of each other. In the following section, The network structure of both of them will be described individually. It is worth noting, that this model supports the usage of *TensorBoard*, and saves a summary of all its variables in a preset interval of update-steps.

Network architecture

Critic   The actual computational graph of the network the `critic` uses is specified in an extra class. In contrast to the `DQN_model`, it was decided to use a different class for agents that use convolutional input than for agents which do not, as informal testing showed that the `DDPG_model` appears to be less forgiving to changes of its network structure. This means, that the `critic` uses a `conv_criticNet` if `agent.usesConv` and a `lowdim_criticNet` otherwise.

In contrast to the model of a DQN, the critic gets a state and an action as input, returning a single estimate of the respective state-action value `Q`. Like a DQN however, it is trained using temporal differences, requiring a better estimate of each Q-value to update its parameters. As however only the online network is updated this way (whereas the target-network receives smooth updates), the `placeholder` to hold these is specified in the class `Critic`, not in one of its used networks. The `Critic`-class further specifies a function to return the `action_gradiens(inputs, actions)` for the `actor`.

The network-architecture of the convolutional critic is specified in figure 5.3.

The number of input-maps is variable, however the kernel size and stride of the convolutional layers are adjusted for an input-size of 30×45 pixels to create 32 2×2 feature maps in three convolutional layers (again, pooling layers were dropped in favor of higher stride). Afterwards, the feature maps are flattened into a layer of 256 hidden units, to which the 3 action-inputs are concatenated. This layer is densely connected to two other hidden layers of 200 units, which ultimately leads to one output-neuron: the Q-value.
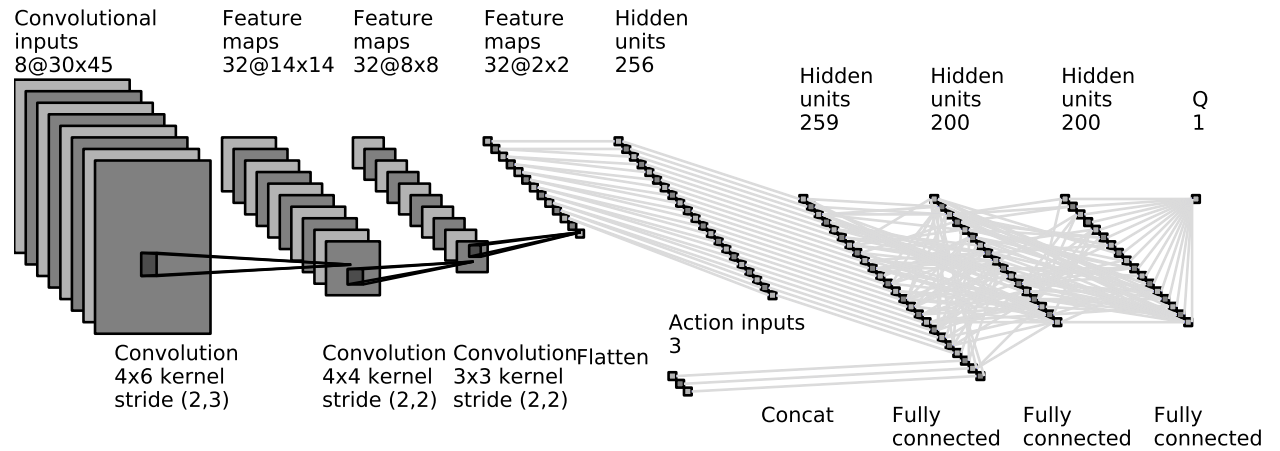
FIGURE 5.3: Convolutional critic

The network-architecture of the low-dimensional critic (`lowdim_criticNet`) is depicted in figure 5.4.
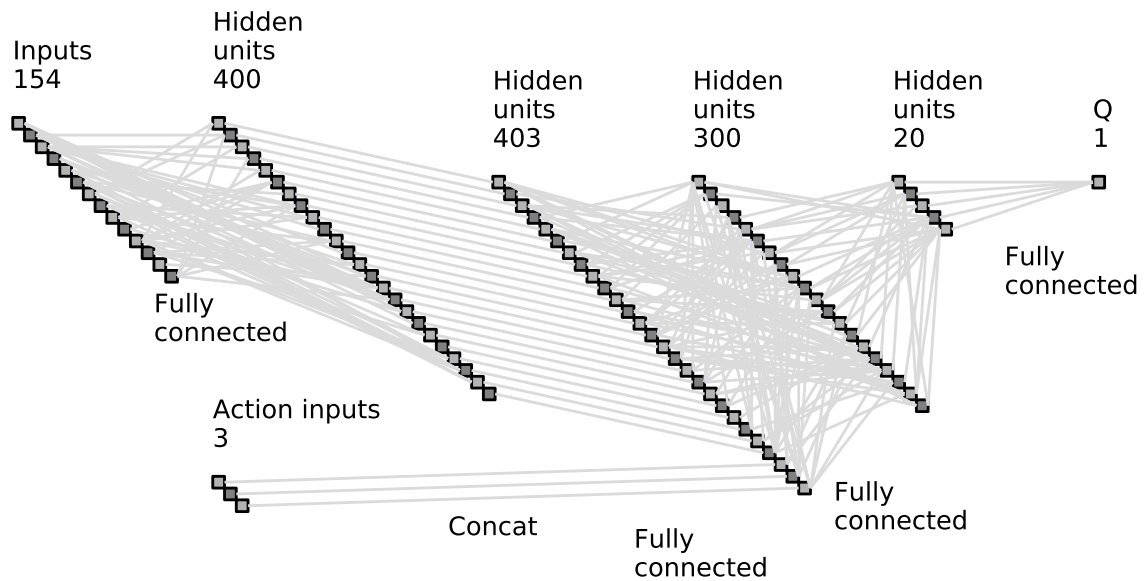
FIGURE 5.4: Low-dimensional critic

The number of input-neurons the low-dimensional critic uses is variable depending on the agent – for the `ddpg_novison_rl_agent` it consisted of 154 inputs. These are densely connected to a layer of 400 hidden units, to which the 3 action-inputs are concatenated. This layer of 403 units is fully connected to another hidden layer of 300 units, which is connected to a layer of 20 hidden units. This final hidden layer is connected to the output-layer specifying the Q-value.

As suggested by [15], both variants of the critic use *L*2 weight decay for all their layers. Furthermore, the implementation allows to use *batch normalization*[11], which can be toggled for each layer individually with a respective argument. In the original DDPG-algorithm, the

authors used this technique in order to use the same network hyperparameters for differently scaled input-values. In the learning step when using minibatches, batch normalization normalizes each dimension across the samples in a batch to have unit mean and variance, whilst keeping a running mean and variance to normalize in non-learning steps. In Tensorflow, batchnorm can be added with respective additional layers and one additional input that specifies the phase (learning step/non-learning step)[6]. Though [15] report success on using batch normalization, in practice it often leads to instability. After much informal it turned out that using batch normalization in the critic only harms the performance by giving a very small and similar Q-value for all states, which is why it is deactivated in the final implementation. All layers are summarized with the function `variable_summary` from `utils.py`, writing their summary to a file that can be inspected by `TensorBoard` every `conf.summarize_tensorboard_allstep` steps.

Actor Just like the `critic`, the `actor` uses a `conv_actorNet` if `agent.usesConv` and a `lowdim_actorNet` otherwise.

The network architecture of the high-dimensional critic can be inspected in figure 5.5. Its structure is similar to that of the high-dimensional critic, with the difference that the actor does not take the actions as input, but connects the last hidden layer to three output neurons, corresponding to the values for the three actions. This *Outs*-layer is scaled afterwards to yield the *ScaledOuts*-layer. All hidden units but the last use a `tf.nn.relu` activation function, whereas the last uses a `tanh`-activation-function. While *tanh* has a range of $[-1, 1]$, the actions may have a different range (e.g. *acceleration* $\in [0, 1]$). The final scaling layer adjusts the range correspondingly.
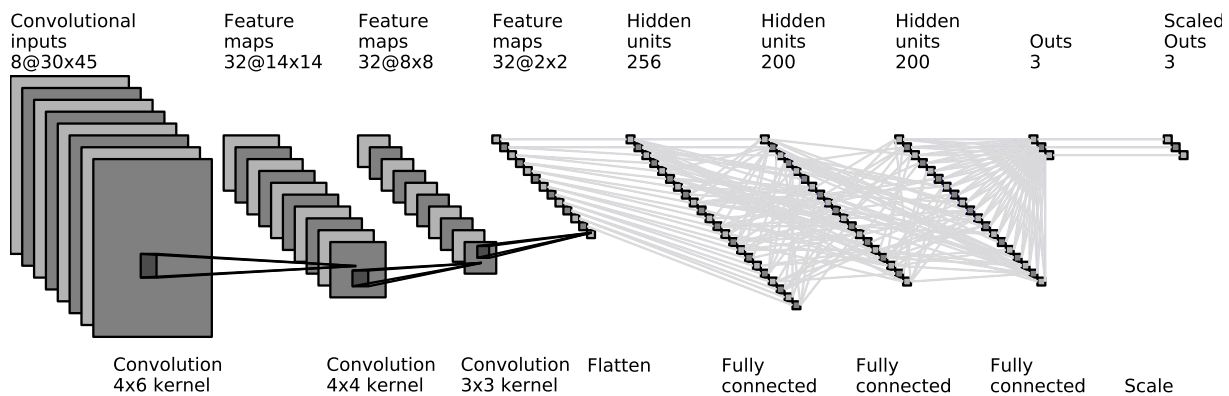

FIGURE 5.5: Convolutional actor

The structure of the low-dimensional actor (figure 5.6) is fairly simple. The inputs are fully connected to 300 hidden units, which are fully connected to another 200 hidden units, which is in turn connected to the output-units, which are scaled again as in the convolutional case.

In contrast to the critic, the actor does not use $L2$ normalization, as suggested by [15]. Informal testing showed, that batch normalization indeed helps convergence in the actor, which is why it is enabled for all fully-connected layers. Besides the critic's preterminal layer, all hyperparameters correspond roughly to [15], with all differences being the result of informal testing. As in the DQN-model, the rectified non-linearity (`tf.nn.relu`) serves as activation function and Adam[14] as optimizer. The convolutional layers are implemented using `conv2d`

---

[6]for further information, see `https://www.tensorflow.org/api_docs/python/tf/contrib/layers/batch_norm` [accessed on 5th September, 2017]
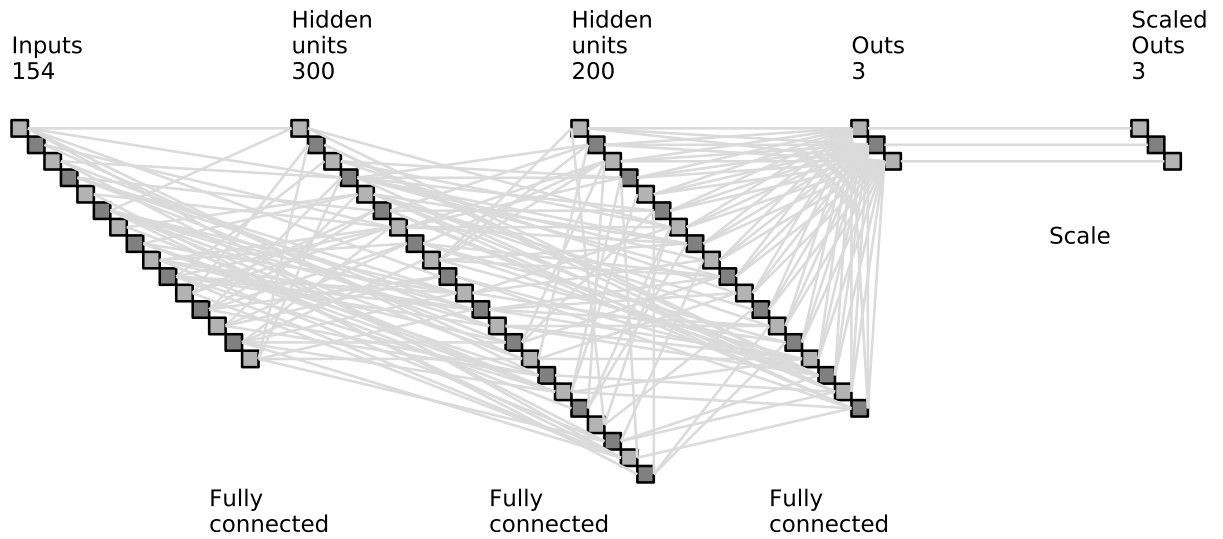
FIGURE 5.6: Low-dimensional actor

from TensorFlow's `slim` package. All layers that are recorded for TensorBoard are added to the collection `summaryOps`.

Like the DQN-model, this model also incorporates a function that forces the model to accelerate if the `stands_input` is set to `true`. This is however implemented differently, as it is not necessary to change the Q-value of actions because the output of the actor can be changed instead. If the corresponding variables `conf.use_settozero` and `stands_input` are `true`, the value for the *brake* is simply set to zero, whereas the value for *throttle* is set to a random value of at least 0.5.

Further, the DDPG-model also manually forbids simultaneous braking and accelerating. In contrast to the DQN-model however, no combined action is returned, but an individual value for *throttle* and *break*. In the method `apply_constraints`, the actor checks if both values are simultaneously above 0.5, and randomly sets one of those to a random value below 0.5 if so.

Agents that use the DDPG-model must overwrite the functions `makeNetUsableAction` to account for the fact the DDPG-model works with un-discretized action and needs these as input to perform its learning step. It was further mentioned in section 2.5.2 that an agent using a continuous model works better with noisy actions instead of completely random actions as exploration function. To do so, a method like `make_noisy`, and the change the implemented `randomAction(gameState)` are required.

5.3 Implemented agents

In the course of this thesis, five different agents were implemented to both demonstrate how to use the given framework and answer the research questions stated in section 1.2.3. The following table shows their names and distinctive properties:

filename	uses model	uses visionvector	performs RL
ddpg_novision_rl_agent.py[7]	DDPG	no	yes
ddpg_rl_agent.py[8]	DDPG	yes	yes
dqn_novision_rl_agent.py[9]	DQN	no	yes
dqn_rl_agent.py[10]	DQN	yes	yes
dqn_sv_agent.py[11]	supervised network with DQN-structure	yes	no

The agents differ in the used *model* (and through that their *exploration-function*), their *observation function*, and if and how they incorporate *pretraining*. The UML-diagram in figure 5.7 shows the attributes and methods that the respective actions overwrite (note however that it is incomplete and for example does not show the functions that are overwritten to incorporate a GUI). All agents end a training episode after crashing into a wall, completing a lap, turning around or after 60 seconds driving-time without either of those.

In the following sections, these functions will be described. Further, the used reward function will be elucidated, even though it is the same for all agents. To specify which agent a `feature` belongs to, captions will indicated the respective agents. If no such caption is given, it is assumed that all agents use the same feature.

5.3.1 Pretraining

The exported `.svlap`s from Unity are used to train an agent. Note however, that only complete, `valid` laps are exported. This guarantees that no laps where the car steeres into the wall or drives too slow will be in the sample. It can thus also be safely assumed that all datapoints in the set are *good*, in the sense of contributing to relatively fast lap.

To assess the quality of a pre-training, the model's method `getAccuracy` is used. The *accuracy* as defined by this method is the percentage of all calculated actions that result in the same discretization than their counterpart from the dataset. While for the DDPG-model, another way of assessing the accuracy using gaussian distances could be used, it will not be considered in the remainder of this thesis when talking about *accuracy*.

dqn_sv_agent

Using only good datapoints is perfectly suited for supervised training, where the agent learns to do the same actions as provided in the dataset. The `dqn_sv_agent` simply iterates over the dataset for a specified number of iterations, splitting the dataset into minibatches according to its settings. To do so, this agent specifies the `preTrain`-function itself, as it is not specified in its only superclass, `AbstractAgent`.

The original definition of an agent's observation-function included, following [20], the last actions an agent performed. Interestingly however, agents that included the action learned an action solely in dependance of this input. This can be explained due to a huge portion of consecutive actions being equal (for example when driving a straight track). Because of this, the previous actions were removed as part of the observation for all agents.
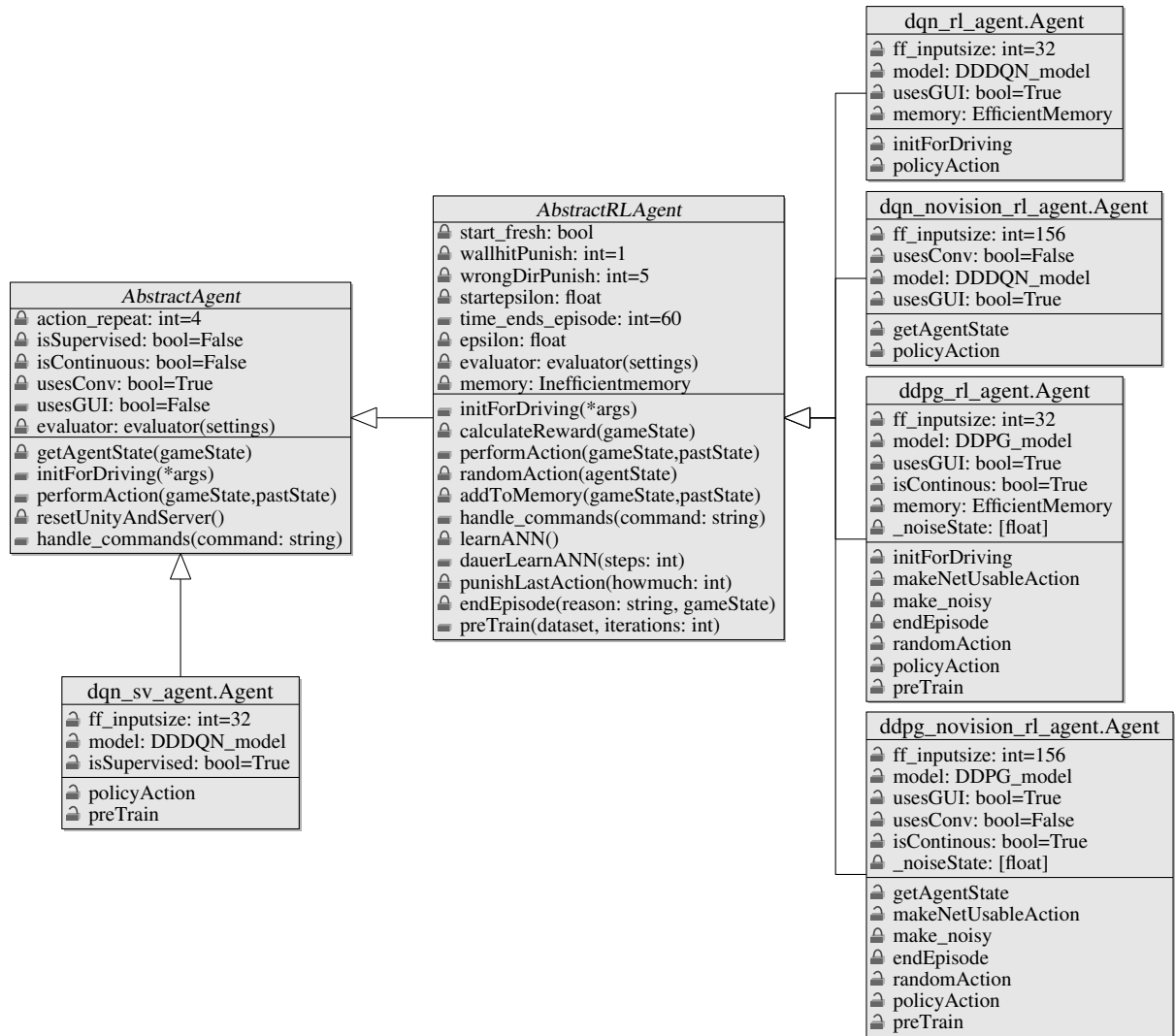

FIGURE 5.7: (incomplete) UML-Diagram of the five implemented agents and their abstract superclasses

The implementation forbids to use an agent that performed supervised pre-training to subsequently perform reinforcement learning, as it does not make sense to re-use an agent that trained supervisedly for q-learning. While a supervisedly trained agent may select the same actions than a q-learning correspondance, it does not ascribe any Q-values to these actions. As long as the $argmax$ is the same, it does not matter for a supervisedly trained agent what the respective value of all actions are. As the $argmax$-operation is for example invariant to equal scaling of all actions, it is obvious that the chance that the respective activations correspond to actual q-values is extremely slight.

Other agents

While it is good for a supervisedly trained agent to learn only from good actions, the same cannot be said for q-learning agents. As mentioned before, q-training is normally used when the state dynamics of the underlying environment is unknown. To learn accurately, a q-learning agent must thus get *representative* samples of the environment. The actions in the dataset used for pretraining are all parts of *good* rounds, where the car mostly drives at high speeds and does not crash into a wall. It is easy to see, that using only those samples is not representative

about the environment. Almost all values in the dataset are likely to have high rewards. If a q-learning agent is trained purely on samples with a high reward, it will assume that high rewards are representative for the entire environment and assign a high reward to all those state-action pairs. Testing showed that this was actually the case, and that an agent trained this way did not learn at all. Because of this, in the `preTrain` -function of `AbstractRLAgent` , the method `make_trainbatch` (printed in algorithm 3) is used.

```
1  def make_trainbatch(self,dataset,batchsize,epsilon):
2    trainBatch = dataset.create_QLearnInputs_fromBatch(*dataset.next_batch(self.conf, self,
         ➥ batchsize), self)
3    s,a,r,s2,t = trainBatch
4    if np.random.random() < epsilon:
5      r = np.zeros_like(r)
6      a = np.array([random_unlike(i,self) for i in a])
7      t = np.array([True]*len(t))
8      trainBatch = s,a,r,s2,t
9    return trainBatch
```

ALGORITHM 3: The `make_trainbatch` - function

This function inflates the dataset, by adding state-action combinations that were not originally in the dataset with a respective reward of zero. Testing showed that training worked best if 80% percent of those fake-actions were included.

In the case of discrete actions, a is saved as the $argmax$ of the one-hot vector. In this case, the method `random_unlike` returns an index dissimilar from that argmax. In the case of continuouous actions, the action is discretized first, and the result afterwards dediscretized. While it was tried to instead add gaussian noise to the action while diminishing the reward only according to the gaussian distance of these two actions, testing showed no success for this approach.

This method simulates an environment that rewards the actions from the pretraining as demanded and provides a reward of zero for any other action. Using this method makes it possible to archieve reasonable accuracies with q-pretraining, where at least the performed action gets ascribed a correct Q-value.

Unfortunately, reinforcement learning agents that used pre-training in this fashion were still not able to correctly make use of it, as figure 6.3 will show.

5.3.2 Exploration

An agent uses an exploration function to learn about different states. As a supervised agent does not learn about the environment at all, it does not incorporate an exploration function.

dqn_rl_agent and dqn_novision_rl_agent

As the model of DQN-based agent discretizes the action into one unique value, so does its exploration function. Both agents use the `randomAction` -method as provided in `AbstractRLAgent` . As the `randomAction` -method is called with a probability of ϵ and otherwise the greedy result of `policyAction` , their exploration algorithm corresponds to the standard ϵ-greedy approach.

The `randomAction` itself selects an action purely by chance, while forbidding simultaneous activation of throttle and brake and forcing a throttle if the car stands.

Even though it was not done for these agents, it would also be possible to incorporate higher-level information into the `randomAction` -function. As the values required by the environment are continuouos, the method could for example use the result of `policyAction` and apply noise to it or select actions basing on the Boltzmann-exploration function. It is an open question how this would manifest in the performance of these agents.

ddpg_rl_agent and ddpg_novision_rl_agent

As for DDPG-based actions it holds that $action \subset \mathbb{R}^{n \in \mathbb{N}}$, which makes it natural that their `randomAction` also returns continuous values. For exploration the straightforward approach is to add some noise to their `policyAction`. As explained in section 2.5.2 and [37], pure random noise leads to very jerky behaviour, which is unrealistic for the given simulation. Because of that, temporal correlation is incorporated with an Ornstein-Uhlenbeck process.

Modelling noise with an Ornstein-Uhlenbeck process changes the standard framework for ϵ-greedy, as no completely random actions are taken anymore. Because noise is added to the result of a networks inference in all inferences, the method `randomAction` is overwritten to return the value of `policyAction`. The parameter `epsilon` is incorporated nevertheless, to control how much noise influences the action of `policyAction`. This method calculates every action according to its model, and then uses a `make_noisy` -function to add the temporally correlated noise to its result, as defined in section 2.5.2. The method `make_noisy` is further specified in algorithm 4, inclusive the values for the Ornstein-Uhlenbeck as they resulted from testing.

In this method, the values for μ, Θ and σ are different for all three actions. A relatively high mean value μ for the throttle together with a low μ for the brake ensures that the car will have a high prior probability to accelerate the car, instead of standing a lot which would cripple exploration. A lower Θ for the steering ensures that the temporal correlation is more relevant than reverting to the mean.

Note that an agent that uses an Ornstein-Uhlenbeck exploration function must specify a `_noiseState` variable, which is reset as soon as an episode ends.

```
1  self.OU_mu = [0.5, 0.05, 0.0]
2  self.OU_theta = [0.85, 0.85, 0.6]
3  self.OU_sigma = [0.1, 0.05, 0.3]

5  def make_noisy(self, action):
6    def Ornstein(x,mu,theta,sigma):
7      return theta * (mu - x) + sigma * np.random.randn(1)
8    self._noiseState[0] = Ornstein(self._noiseState[0],  self.OU_mu[0] , self.OU_theta[0],
          ➥ self.OU_sigma[0])
9    self._noiseState[1] = Ornstein(self._noiseState[1],  brakemu,       self.OU_theta[1],
          ➥ self.OU_sigma[1])
10   self._noiseState[2] = Ornstein(self._noiseState[2],  self.OU_mu[2] , self.OU_theta[2],
          ➥ self.OU_sigma[2])
11   action[0] += self.epsilon * self._noiseState[0]
12   action[1] += self.epsilon * self._noiseState[1]
13   action[2] += self.epsilon * self._noiseState[2]
14   action = np.array([clip(action[i],self.conf.action_bounds[i]) for i in range(len(action))
          ➥ ])
15   return action
```

ALGORITHM 4: Ornstein-Uhlenbeck process to generate noisy actions

5.3.3   Reward

The success of a reinforcement learning agent hinges critically on its reward function. Especially in the domain of virtual racing simulations, there is much debate about what serves as a good reward. As listed in section 3.2.2, the original DDPG-paper [15] used the car's velocity in direction of the street as reward. Applying this reward-function to the given progress did not lead to success (as figure 6.5 shows). If only speed is rewarded, the car accelerates as much as possible and simply crashes into the wall at the first turn. In this implementation – as well as in many others – accelerating as much as possible is not the correct behaviour. In theory, a reinforcement learning agent should learn to accept a smaller reward in the current state in favor of a more desirable follow-up state, however no agent seemed to have explored enough to learn that other possibilities besides the unevitable crash into the wall are possible by braking for a consecutive number of frames.

To create an agent that learns successfully, a reward-function must thus be implemented that rewards the *correct* behaviour at any time, including braking. Consequently, it was decided to implement a reward function that rewards high speeds only if no sharp turn is ahead, and punishes them otherwise. The resulting definition uses `OtherInputs.WallDistVec[6]` (corresponding to the grey line in figure B.6) to measure the distance to the closest wall ahead of the car. The employed function to calculate the reward is specified in algorithm 5. A visualization of its value in dependence of speed and wall-distance can further be found in plot **D** of figure 5.8.

```
1  vvec1_hist, vvec2_hist, otherinput_hist, action_hist = gameState
2  reward = otherinput_hist[0].WallDistVec[6]-otherinput_hist[0].SpeedSteer.speedInStreetDir
        ➥ +(80/250)
3  reward = 1-(abs(reward)*3) if reward < 0 else (1-reward)+0.33
4  reward = min(1,reward)
5  reward += 0.3*otherinput_hist[0].SpeedSteer.speedInStreetDir
```

ALGORITHM 5: Rewarding speed in relation to the wall-distance

While an agent using purely this reward definition performed much better than one using only the speed, it turned out that adding other components to the reward optimized the performance and tempo of learning even more. Figure 5.8 visualizes some other components that were additionally used, with the source code to caluclate those in appendix C.1

Plot **A** shows the sub-component of rewarding the car's *traverse position*. As it is reasonable for the car to stay on the street (less slip and better control), doing so is rewarded. As long as the car stays on the street, it receives the full reward, whereas this reward decreases steeply from the *curb* on.

Another component is the *Minimum-Speed-Reward* (plot **C**): It is easily possible to keep a constant speed at all times. To ensure that driving to slow is punished, this function subtracts a value for speeds lower than 20% of the maximally possible speed. As the function from algorithm 5 sometimes rewards very low speed, it is useful to combine it with this component to discourage too slow speeds.

If the *carAngle* is too big, it is also rewarded to steer back into the direction that decreases this angle. As the purpose of this component is to make the car steer back to the street if it is too close to a wall, this reward is only relevant if the traverse position of the car indicates that it is close to a wall. Plot **E** shows the preliminary steering-reward in dependence of car-angle and steering-command, and plot **F** shows the actually resulting reward, depending on the result from **E** and the car's traverse position. This steer-bonus is stated in lines 19-26 of the source code in appendix C.1.
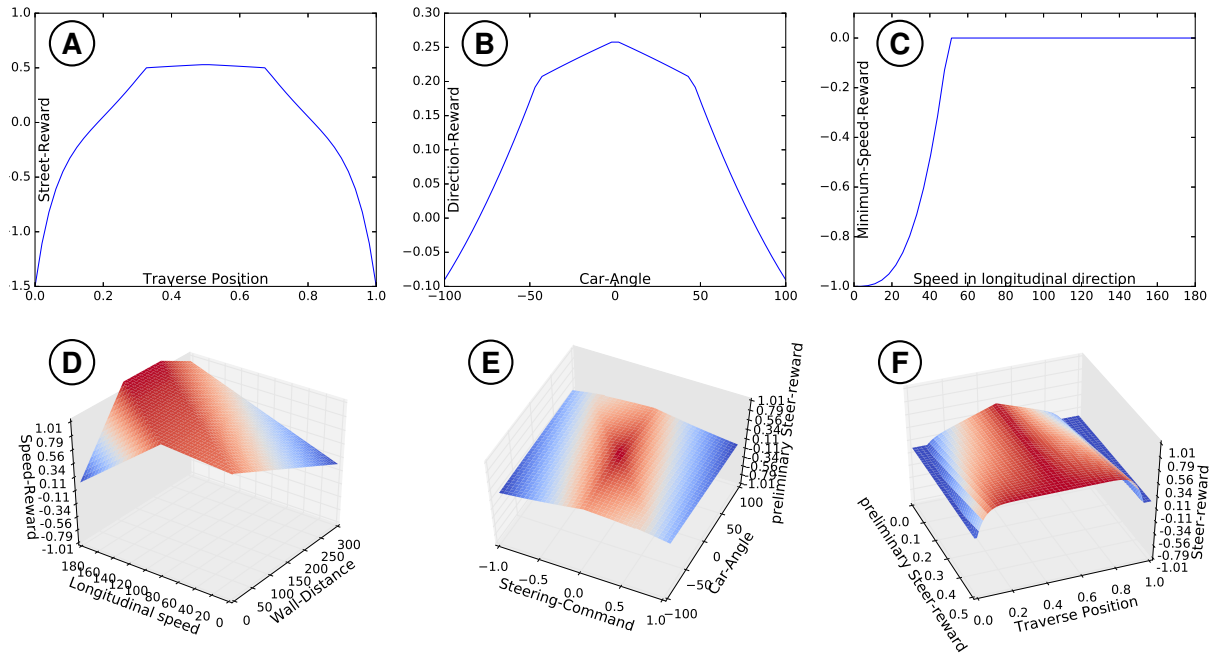
FIGURE 5.8: Reward-components

Note that a reward function must not provide more reward to get out of an undesired situation as it gives for not getting into such a situation in the first place, as that would lead to an agent intentionally generating the undesired situation.

These components are not all that are used in the final agent, but listing all of them would be beyond the scope of this thesis. The interested reader is therefore referred to appendix C.1, where the source-code of this function is given.

The actual reward is a weighted sum of all these components, with the respective weights being a result of informal testing. Note that the final result is clipped as `reward = max(0, reward)`. It is a well-known fact that negative rewards impair the behaviour of an agent more than they help. If for example the reward for being too far off-center is all negative, then the car would get a higher cumulative when it ends the episode by driving into a wall. Testing confirmed this scenario. To prevent such scenarios, all rewards but the one that ends the episode must be positive.

5.3.4 Observation

An agent's observation-function is specified in its method `getAgentState(gameState)`, which creates the `agentState` used for its model and its memory from the `gameState`. `AbstractAgent` specifies an observation-function that can be used by sub-classes. This function is only overwritten by the agents termed *novision*-agents, specifically the `ddpg_novison_rl_agent` and the `dqn_novision_rl_agent`.

`ddpg_rl_agent, dqn_rl_agent` and `dqn_sv_agent`

These agents use the result of both minimaps (see section 5.1) in their definition of agent-state. As suggested by Mnih et al. [20], an agent uses the `M` most recent states as its observation, with `M` (`config.history_frame_nr`) set to four. This ensures that the agent has a way of inferring trajectories from the static images. While this should in theory be enough to calculate the current velocity, testing showed that additionally passing the current velocity improved the

accuracy on a supervised test set. Note that this velocity is passed to the network in the form of a vector instead of a single number.

The actual two-dimensional input to a network of this form does not consist of four feature maps, but of eight, as both mini-maps are stacked before handing it in as input. Both maps are provided in the hope that one of them provides a coarse overview, whereas the other helps precise navigation. Note that the mini-maps record mostly the track in front of the car, not the surroundings of it.

While in the original DQN as put forward by Mnih et al. [20] its last actions are also input to the agent, this is removed in the current version of the implementation. Supervised testing showed, that an agent only learns to repeat its last action – which is the correct behaviour in a high percentage of cases, but does not generalize at all when applied to new scenarios. As further testing showed that there seemed to be no benefit of providing the action altogether, it was decided to remove this input from the agent's observation in all scenarios. It is however possible to change that back by uncommenting the respective line in `AbstractRLAgent.getAgentState`.

Note that as mentioned before, an agent's state treats the convolutional input (the minimaps) separately from the other inputs, which in this case consists only of the speed. The third information of an agent's state is the information whether a car stands to forbid braking – according to this observation-function, this input is `true` if the car drives slower than 4% of its maximum-speed, corresponding to roughly $6km/h$.

`ddpg_novison_rl_agent` and `dqn_novision_rl_agent`

The observation of the *novision*-agents does not use the minimaps, to save on memory required for the replay buffer. Because of that, the respective convolutional input of the `agentState` is `None`. To make up for that loss, the agent instead uses a low-level information. As the provided information is already explained in section 5.1, a numeration of the used vectors must suffice here:

- CenterDistVec
- values $5 - 11$ from SpeedSteer
- StatusVector
- WallDistVec
- LookAheadVec

All of these input-values are scaled to a range of $[0, 1]$, such that values on a naturally high scale do not influence the agent's policy out of proportion.

Note, that these agents also do not provide the past action, with similar reasoning than above. Testing even showed that providing the motor torque and steer information (values 1-4 from *SpeedSteer*) is just as destructive: If this vector was provided, an agent learned to hit the throttle if the motor torque was already high, and to simply steer into the same direction as the frame before, which lead to the car driving right into the wall as fast as possible. Removing these elements from an agent's observation was a necessary condition for a capable policy.

While *WallDistVec* and *LookAheadVec* provide information of the course of the track in front of the car, it does (just like the mini-maps) not provide information about the course of the track to the side of the car. Incorporating fixed range-finder sensors, as suggested by [16], may be a worthy addition in future implementations.

6 Results – Performance and Analysis

As stated in the introduction, in the course of this thesis not only platform and agents were implemented, but the agents were also tested in an attempt to answer the research questions from section 1.2.3. The testing took place on a *Windows 10 Pro* machine, using GPU-accelerated Tensorflow with an *NVIDIA GeForce GTX 970* graphics card. In this section, the performance of several agents is visualized and compared to assess their quality.

Unfortunately, the analysis of the implemented agents was restricted a lot by the time limit of this thesis, which also incorporated implementing the platform and writing this text. Testing models and agents while implementing them takes a significant amount of time, as mistakes become apparent only after hours of training. Because training the agent also takes a significant amount of time, the plots given here are the result of much less training episodes than in for example [20]. The reliably found trends in these plots can nevertheless serve as a basis to answer the posed research questions.

If not explicitly mentioned otherwise, all agents use the same hyperparameters, as specified in their respective file[1] or in the general config-file[2].

6.1 General performance

The general performance of agents can only be assessed in comparison to a baseline. In figures D.1 and D.2, exemplary laps as driven by a human and a random policy are depicted. In both plots, an episode is either ended by hitting a wall, finishing a round or after 60 seconds time. Note that negative progresses are possible because all agents start at a certain point around 8% in front of the start/finish line. The laptime counter does also not start until this line is passed.

As can be seen in figure D.2, the baseline performance of a random agent is negligible. While it sometimes advances some distance and does not bump into walls for several seconds, it can unmistakenly be seen that the baseline progress as provided by a random policy is on avarage not much above zero.

Figure D.1 shows the performance of a human tester with several hours of driving experience for this game. Many of the episodes driven by a human also end with the car hitting the wall, which can be explained by a human driver trying to minimize the laptime and accelerating too much. However, there are also many episodes in which a full lap was driven. The time taken for an episode by human testers lays between 32 and 60 seconds (typically around 32-35 seconds), with no lap driven faster than that.

As can be seen in the progress-plots of the agents (for example in figure 6.3), the game appears to have some very obvious local optima. These local optima are at about 16%, 40%, 60% and 75% and correspond to sharp curves of the track.

Due to severe memory constraints of the provided machine and the fact that agents need a replay memory of hundreds of thousands of states, testing of RL-agents using the minimaps as observation was very constrained, allowing only a small replay memory in comparison

---

[1]All agent's path is `https://github.com/cstenkamp/BA-rAIce-ANN/tree/master/agents`. The title of the respective file is stated in text and figure.

[2]`https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/config.py`

to others. However, as the supervised agent that also relied on this observation achieved an outstanding accuracy, it can be assumed that all findings for the *novision*-agents also hold for their respective counterpart.

The first research question, formalized as *how agents relying purely on pretraining perform in comparison to reinforcement learning agents*, will be answered in the following section.

6.1.1 Supervised agents

It is hard to assess the performance of purely supervisedly trained agents. Most agents achieved an accuracy of more than 95% on a testing set[3], but failed completely when tested as agent on the game. The reason for this is easy to see: In this game, one false action at the start can already be fatal, and a supervised agent cannot learn to recover from this mistake, thus repeating it over and over.

Testing showed that agents that rely purely on supervised pre-training are not able to drive around the track when tested. Figure 6.1 shows this exemplary in the form of an agent that was trained for around 50.000 steps on a dataset of 46 exported laps, achieving a testing set accuracy of 93%.
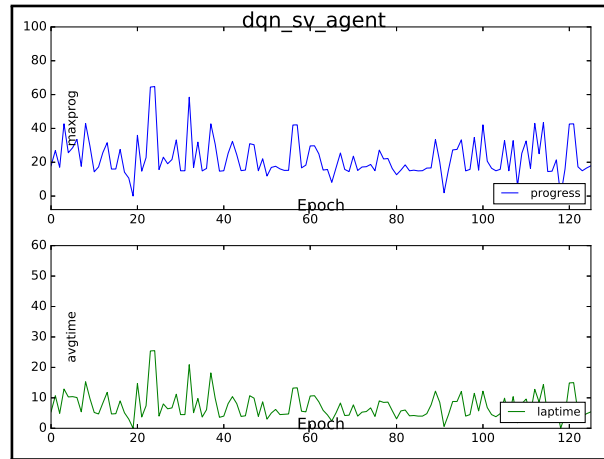

FIGURE 6.1: Exemplary performance of the `dqn_sv_agent`.

As can be seen in this plot, high testing set accuracy does not mean that an agent achieves a useful policy. An interesting finding was, that if a maximum-speed of 80 kph (33% of `Consts.maxSpeed`) was given for both generating the dataset as well as for the agent that learns on it, some supervised agents were able to learn successful policies, completing a lap almost every time. Most supervisedly trained agents learned to accelerate as much as possible, driving straight into the first turn. A likely explanation for that is that due to their temporal discretization the brake is hit only in a fraction of the states than the dataset that was trained on.

6.1.2 RL agents

The first and foremost result is, that some agents were able to learn successful policies that are able to drive complete laps in a reasonable time. Figures 6.2 and 6.3 show exemplary performances of the `dqn_novision_rl_agent` and the `ddpg_novision_rl_agent`. Note that the training for both agents was terminated as soon as they learned a policy that completes the

---

[3]Supervised agents are tested on the `.svlap`-files found in `https://github.com/cstenkamp/BA-rAIce-ANN/tree/master/SavedLaps`. To generate a testing set, one of the files was removed from the training set.

circuit ten times in a row. Note further, that in all plots that smooth over a specified number of episodes, the maximum reward, Q-value and progress of these episodes is taken.

Testing of the generated Q-values showed internal stat representation learned by the agents has many desired properties, for example 1) the state-value of a state immediately in front of a wall is much lower than everywhere else, 2) the q-value of braking is lower than the one of accelerating in a straight street, but higher in close proximities to a turn or wall, 3) steering away from a wall has a higher Q-value than doing nothing or driving towards it, and 4) driving fast leads generally to higher state-values than driving slow.

It is also very obvious that some agents seem to learn only *turn by turn*, in that the straight track between sharp turns can be driven easily, whereas a large number of episodes is needed to overcome every new turn.

6.2 Discretizing actions

This section serves to answer the posed research question of *how different models perform in comparison, and specifically if discretizing the action-space impairs performance*. To do so, the difference between performances of the `dqn_novision_rl_agent` and the `ddpg_novision_rl_agent` will be elaborated. Both agents use the same reward function as specified in section 5.3.3, as well as the `novision`-observation function from section 5.3.4. They only differ in their model and, due to that, their exploration-function.

An exemplary performance of the `dqn_novision_rl_agent` can be seen in figure 6.2. The performance of the `ddpg_novision_rl_agent` is depicted in figure 6.3.

FIGURE 6.2: Exemplary performance of the `dqn_novison_rl_agent`. Plots are smoothed by averaging over 10 episodes.

In both cases, Q-value, average reward and progress increase throughout training. Both agents are further able to complete a whole lap. To do so, a DQN-agent required around 3000 training episodes (over 400.000 minibatch-trainingsteps), whereas the DDPG-agent only needed around 1400 episodes (corresponding to less than 300.000 inferences) in the exemplary run. It is further interesting, that the DQN-agent seems to learn sequentially, where each turn

requires hundreds of training-iterations until it is mastered. In contrast to that, the DDPG-agent seems to generalize better from the first part of the track towards the whole track, as can be seen in the very steep learning curve towards the end.
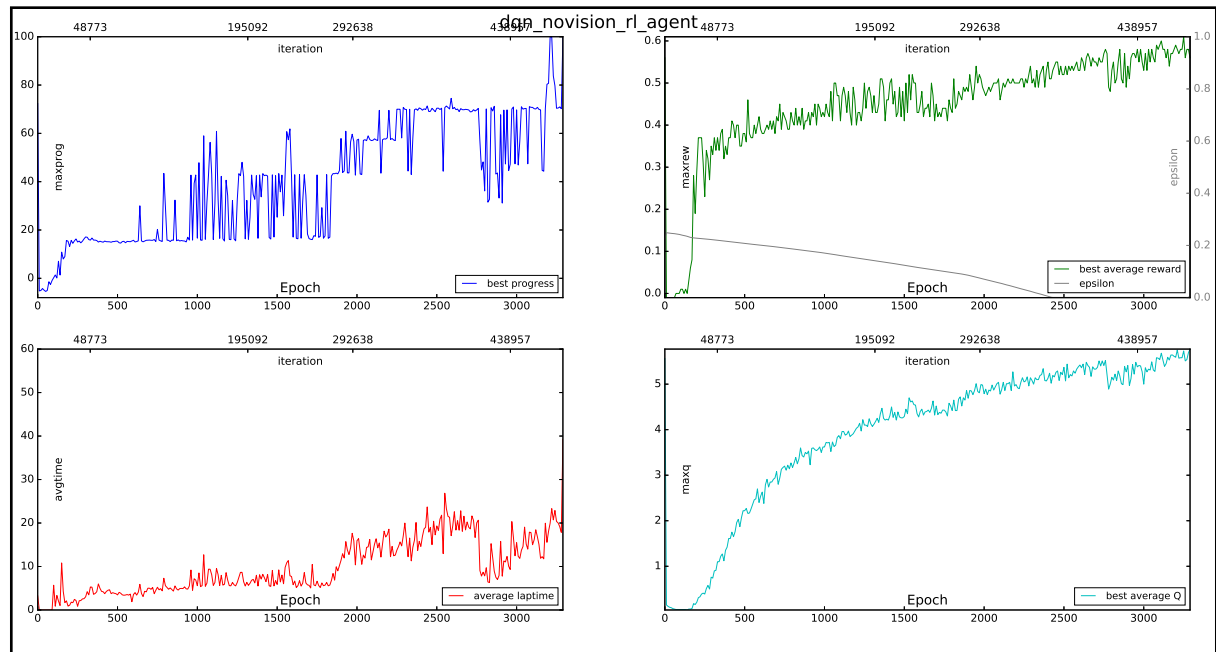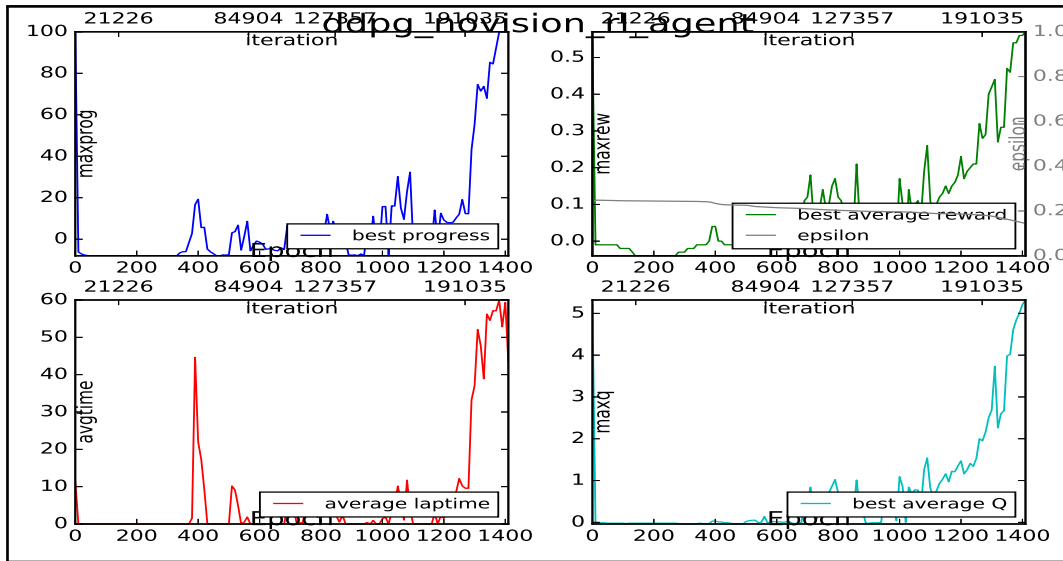

FIGURE 6.3: Exemplary performance of the `ddpg_novison_rl_agent`. Plots are smoothed by averaging over 10 episodes.

This result shows that while an agent discretizing the action-space can certainly learn the track, an agent that does so seems to learn slower than its continuous counterpart. As the continuous agent however also used a better exploration function, it remains a question to further investigation how much of this performance gain must be accredited to that.

Another question that remains open is, how much faster the laps driven by a continuous agent will ultimately be. No full lap driven by either of the agents finished in less than 50 seconds time, which is a lot more than the human average. As the action-space of continuous agents includes that of the discretized agents, it is certain that the upper limit of the former's maximal performance is at least as high, and probably much higher, than that of the latter.

Interestingly, both algorithms oversteer a lot even in straight track sections, which leads to *jittering* movement. This seems to be a general problem of the employed techniques, as it is seen throughout many other implementations[4].

6.3   Incorporating pretraining

One question this thesis aimed to answer is *how to incorporate pretraining into reinforcedly learning agents*. As explained in section 5.3.1, an agent that trained supervisedly does not adequately *transfer* this knowledge when applied to a reinforcement learning paradigm. In this thesis, it was tried to find a solution for that by setting correct Q-values for the respective actions, while setting a Q-value of zero for all others.

To test if pretraining an agent increases the learning pace, an agent that performed q-pretraining as specified in section 5.3.1 subsequently underwent normal reinforcement training. Specifically, a `ddpg_novision_rl_agent` was pre-trained for 40.000 pretraining steps on a dataset consisting of 46 laps (around 14.000 individual datapoints), such that it had a testing set performance of 96%. While the testing set accuracy is high, this agent rarely got further than the first turn of the track.

---

[4]See for example this video `https://www.youtube.com/watch?v=4hoLGtnK_3U` [accessed on 11th September, 2017], which shows the an agent's policy of the implementation of the project *DDPG-Keras-Torcs* as listed in table 3.2.

Figure 6.4 shows the performance of this agent in actual reinforcement learning. As can be seen, while the rewards and q-values are high at the beginning, it seems impossible for the agent to use that knowledge. In fact, the graph rather suggests the opposite, as: 1) the reward drops very fast to zero, and stays close to zero for longer time than a non-pretrained agent, 2) the progress-milestone of 16% is reached at around the same time than in an agent that did not perform pretraining (epoch 1300), and 3) The q-value is decreasing until this episode, rising only with the rise of the reward.
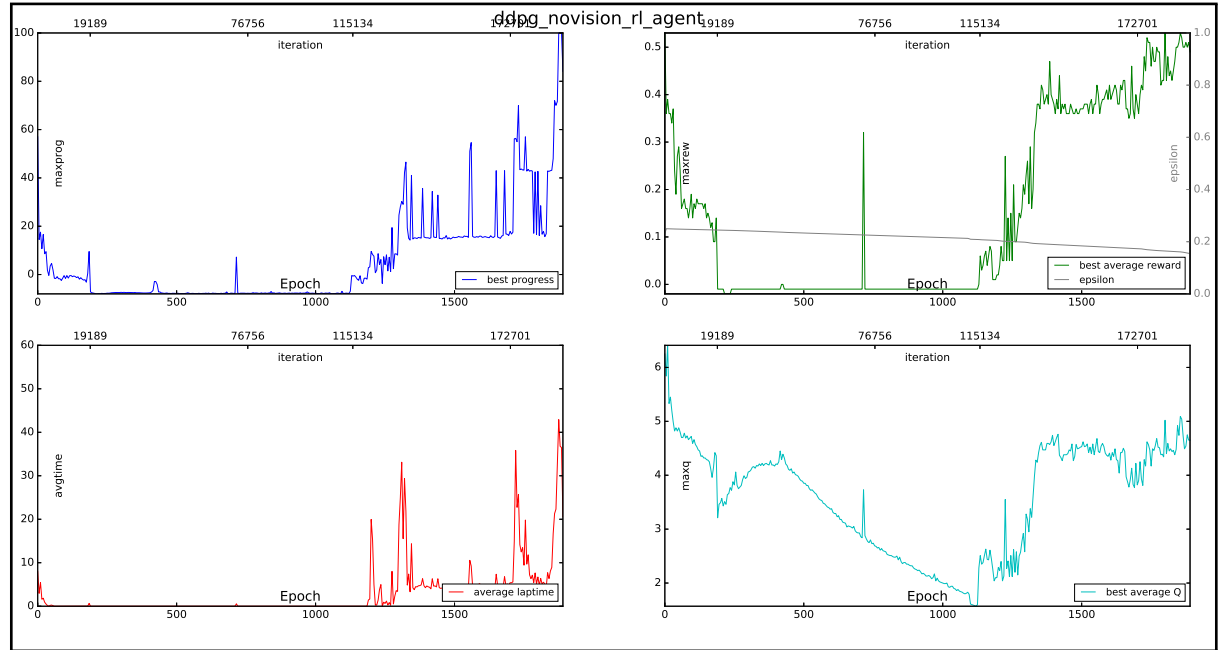

FIGURE 6.4: Exemplary performance of the `ddpg_novison_rl_agent` after 40000 pretraining steps. Plots are smoothed by averaging over 5 episodes.

All in all, this agent completed its first full circuit after around 1900 episodes, more than 500 epochs later than an agent that did not perform pretraining.

The presented run is by far not the only one that showed this behaviour, and while not printed in this thesis, the plot of a pretrained `dqn_novision_rl_agent` also showed the same properties than the one described.

In conclusion it has to be said, that this thesis did not find a successful way to incorporate a pretraining based on manually driven rounds. This can be due to three main reasons: 1) the dataset was too small and must be extended, 2) it is simply not possible to learn from only *good* rounds, or 3) the employed method is not the correct approach. Further resarch must be taken, especially trying to find a better method than the one used.

6.4 Reward function

The last research question asked *what a good reward function looks like, that rewards the correct behaviour at all times (including braking).* All agents of the previously mentioned figures used the reward function from section 5.3.3. The fact that these agents succeeded is evidence that incorporating this function contributes to successful driving policies. For this section, this method is compared with two other reward functions.

In the original DDPG-Paper, the authors used as reward only *"the velocity of the car projected along the track direction and a penalty of -1 for collisions."*(quote [15]). In the given simulation, this corresponds to the feature `SpeedSteer.SpeedInStreetDir`, with the collision punished in
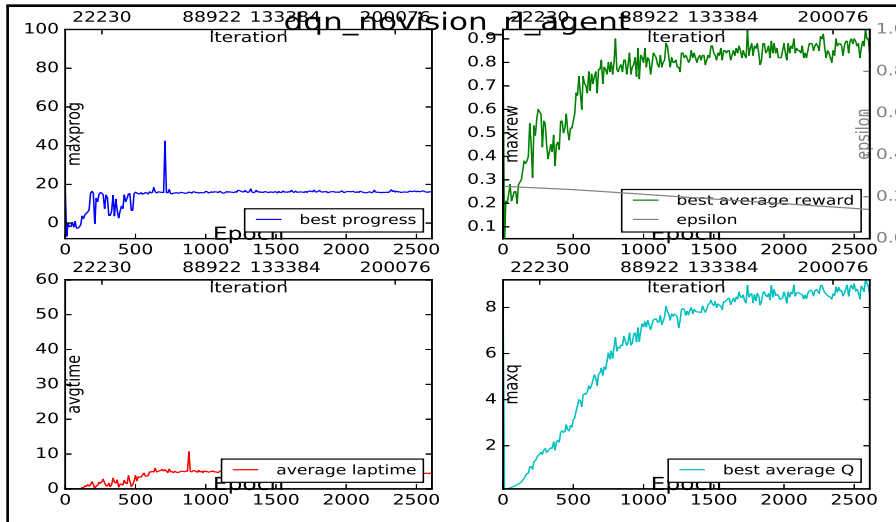
FIGURE 6.5: Exemplary performance of the `dqn_novison_rl_agent` with the reward function from [15]. Plots are smoothed by averaging over 10 episodes.

`handle_commands` . Figure 6.5 shows the performance of an agent that uses this reward function. As can be seen in the respective plot, this reward does not lead to successful results after 2500 episodes, whereas its average reward is almost maximal. As this reward function does not reward braking at all, the agent does not learn to do so and almost every episode ends with the car skidding and crashing into the wall at the first turn.
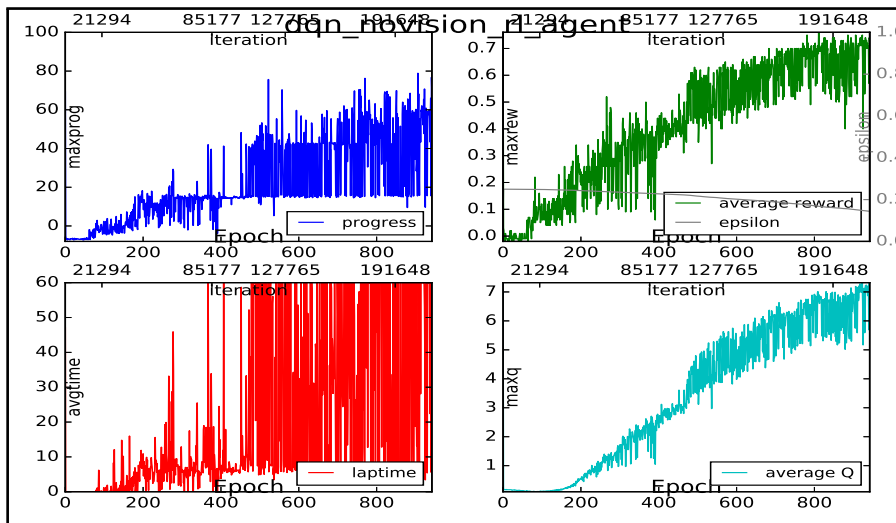
FIGURE 6.6: Exemplary performance of the `dqn_novison_rl_agent` with Speed-InRelationToWallDist as only reward. Plots are not smoothed.

To demonstrate the contributions of the other reward-components, figure 6.6 shows the performance of an agent with the other reward-components removed (thus corresponding to lines 1-4 from algorithm 5). While the progress makes it appear as if the agent learns a useful policy, the plot of the lap-time shows that almost every episode ended because of the time limit of 60 seconds – which means that the agent will likely not get any further. As this reward-function does always reward driving fast, it is no reasonable reward-function on its own.

7 Discussion and future directions

In retrospect, all of the objectives of this thesis were met. The game was sucessfully transformed into an efficiently working platform for research on reinforcement learning, and several agents were implemented that learn reasonable policies. The resulting framework also makes it easy to add new agents without deep understanding of the specific implementation.

7.1 Platform

It was a long way until the setup of environment, client and server was at the state it is in the final version. The socket-connection alone has gone through countless changes that could each fill its own thesis to come to the final version it is at right now. It is surprising how efficient this version is – an average response time of $20 - 30ms$ including the TensorFlow-inference is on par with for example the delay in *openAI Universe*[1], and it allows for a connection of up to 40 FPS, which should be enough time for any agents to learn (the DQN ran for example with only 20 FPS). In future work it would be possible to additionally increase Unity's internal `time.TimeScale` to up to twice its current value – for that, however, the millisecond-intervals have to be adjusted as well.

In a prior version, learning and inference were performed simultaneously in different threads. This, however, decreased the runtime by a factor of ten if both must access the same TensorFlow model. This could change if the usage of TensorFlow `feed_dict` was replaced by thread-safe `queue` s instead[2]. A multi-agent solution is imaginable, in which a threaded TensorFlow graph is fed via different threads. While it is an open question if the non-asynchronous models as presented in this thesis benefit from multithreaded feeding, it was not subsequently performed due to the increased runtime.

In implementations where target- and online-network are updated non-continuously (like the original DQN) it is especially easy to outsource the learning thread to a remote machine, where information exchange is limited to updates of the replay memory and occasional updates of the computation graph. A further possibility to speed up the learning process could be to allow multiple Unity-clients that all aggregate samples for the same agent, stemming from the same policy. It is an open question how much delay has to be expected if server and environment run on different machines. As a major bottleneck of this implementation is samples generation, it is expected that this would increse the speed of learning drastically.

An important distinction of this platform from others is the fact that it is live, and that a user can interfere in anything an agent does. This is especially convenient to see if an agent's Q-values are reasonable. Further, a human's performance can be evaluated just like the performance of agents, also with respect to reward and Q-values as an agent would ascribe them. Having a live visual feedback of the reward (`if config.showColorArea`) has also shown to be useful functionality to assess reward quality.

---

[1] https://github.com/openai/universe

[2] see for example https://github.com/tensorflow/tensorflow/issues/2919 [accessed on 1st September, 2017]

A major focus point of this work was to make it as easy as possible to add further agents, models and *vectors*. A user implementing a new agent or model only needs to create a respective class that implements the interface stated in figure 4.2 or figure 4.3 and put it in the respective folder. Training for a new agent can be started by executing `python server.py --agent NAME`. All details of communicating with the environment or its memory are abstracted away by abstract classes. Adding further vectors is also easily done, as it just needs to be implemented in Unity and added to the namedtuple `OtherInputs`. Sending all possible vectors to an agent, such that the agent decides what to make of it allows for a various amount of agents, and adding new components that rely on the environment's state is far easier than in other known environments like TORCS.

Future research will show if the agents' used observations can be improved further, for example by adding different vectors. Additionally, the agents' capacity to generalize should be tested, which could be done by adding race tracks to the game.

7.2   Agents

Most of the implemented agents were able to learn successful driving policies, even though their time to complete a track is worse than that of a human tester. This may suggest that individual components or hyperparameters are not optimized, however one has to keep in mind that no agent trained for longer than 500.000 steps, which stands in comparison to $50.000.000$ training steps in the original DQN-paper and $2.500.000$ steps in the DDPG-paper. Due to the nature of all current deep reinforcement learning algorithms, the change of hyperparameters manifest as qualitative changes in the policy only after several hours. While temporal constraints did not allow for much further testing beyond what is depicted in this thesis, there is reason to be optimistic about the capacity of the used agents.

For many components, there are however ways to test them more efficiently. The included `gym_test.py` script is for example a very good way to test if an implemented model works. The file expects the same interface of an agent, and as many of the gym-environments are much easier than the given game, testing can be done much faster. To test a new definition of the observation-function, it is useful to first create a supervised agent that uses this observation function, as such an agent can test much faster if the dataset is even *seperable* under a certain definition of observation.

Racing games are generally an exceptionally hard case for reinforcement learning because of their highly correlated states. To encounter for example the situation of *being fast in a turn*, an agent first needs to drive stable enough to speed up, which is impossible purely by chance. While it would help to simulate such situations as an agent's initial state with some probability, this approach was not considered as it does not reflect real behaviour.

Testing seemed to show that this game is in fact harder than TORCS, as agents that successfully perform in TORCS show no success in this environment. The track is especially dificult, as it starts with a long straight stretch and a very sharp tun afterwards. Many agents learned in the first epsiodes that accelerating as much as possible provides maximum reward, which leads to them driving into the wall over and over, as for example the agent using the reward function from [15]. Analysis of this reward-function also showed that iterations of high reward do not correspond to iterations of much progress, a further argument against such a function. The car always starting at the same track position is a likely reason for why agents are prone to repeatedly crash in the first turn. While it was demanded to not use a stochastic start state distribution, it would be interesting to see if a simpler definition of reward works better if that would be done. It has to be kept in mind however that while the current definition of reward is very complex, it appears to be somewhat reasonable, as it is the only tested reward-function that helped an agent learn sucessful policies.

There are many open ends that would probably improve the performance of agents a lot, that could not be implemented yet due to temporal constraints. A very interesting approach is for example to incorporate a pseudocount-based exploration function as suggested by [22] or [17]. While it was tried to incorporate such a density-model into the current implementation (this is why the low-dimensional critic from figure 5.4 has 20 final hidden units), it did not succeed yet. It is however assumed that doing so would improve the speed of learning of agents a lot, as it can be used to force an agent to only explore in unkown situations. This would reduce the amount of training episodes where a car spontaneously steers into a wall while at full speed. Optimal would be an agent that largely follows its policy for the known part of the track, exploring only in later sections. Besides that, OpenAI recently suggested that incorporating *parameter noise* as a means of providing additional exploration greatly improves performance of both implemented algorithms[3].

Another approach that speeds up learning is *prioritized experience replay*, as introduced by [26]. Next to that, there are architectures that can wrap the implemented approaches in a multi-threaded fashion such as A3C[18] or GORILA[21], which would probably also greatly improve the speed of learning.

The current pace of improvements in the field of deep reinforcement learning is very quick. In the midst of writing this thesis a new technique, termed *Proximal Policy Optimization* [27] surpassed the performance of the used algorithms in many MuJoCo physics-tasks. A future direction would be to implement this algorithm and compare its performance to that of the two used algorithms.

Unfortunately, this thesis did not find a way to properly incorporate pretraining into a q-learning agent, neither for DQN-based agents nor for DDPG-based agents. The implemented algorithm (3) enabled that q-pretraining achieves reasonable testing set accuracy, but did not speed up the learning process of actual training. It is assumed that this is because the method sets zero reward for any other action. However, another version of the `make_trainbatch`-function that also rewards noisy actions basing on their gaussian distance (the code for which is still in `ddpg_novision_rl_agent.py`) did not solve this problem either. The solution to this problem remains unknown and a matter of further research.

Due to memory constraints of the provided working station, the agents that require a *visionvector* were not tested extensively. Further research may go into this direction to test how good the performance of agents that rely purely on these vision-vectors is in comparison to agents that receive low-level information. It is also interesting to figure out what the minimal observation is that an agent requires to infer a capable driving policy. Many other approaches, like the original DDPG [15] or *NVIDIA*s approach [7] use the same visual information that a human receives as input. It is worth following the same approach in this implementation.

Furthermore, there is no good way to assess an agent's performance so far. In pre-training, the accuracy is simply taken as the percentage of actions that have the same discretized action than the supervised label or as the mean gaussian distance of those. This is similar to other approaches (e.g. [7]) in that it assumes that the human performance is the ground truth of optimal performance. This approach has however several disadvantages – for example, an agent simply driving forward already has a reasonable accuracy according to this measure, or the fact that tiny differences in steering can make the difference between driving optimally and crashing into a wall. As section 6.1.1 shows, this measure of accuracy does not correlate with actual driving performance. The only useful measure found so far therefore is the agent's progress and laptime. An optimal method to measure performance that works in both use cases would be to compare every point of the policy's trajectory with that of an optimal algorithm, like for example the introduced RRT^* algorithm.

---

[3] `https://blog.openai.com/better-exploration-with-parameter-noise/` [accessed on 10th September, 2017]

Another future direction could be to introduce rewards that change throughout the training process. At start, the agent could be rewarded for making some kind of progress, whereas only lateron its speed and laptime become more relevant. So far, it did also not matter if an agent drives a *valid* round in which it stays on the track-surface at all times. A more strict punishment of driving off-track or even resetting the car if it does so are imaginable to counter this behaviour. This could also be introduced only after an agent learned a useful driving policy.

This thesis has shown that existing agents from the literature (see 3.2.2) cannot be straightforward applied to the implemented simulated racing platform. Further testing must show if the reason of that is the deterministic start-state as used here, or if the TORCS environment is more condoning for short-sighted action generation than this implementation. If the latter is the case, it questions if the TORCS environment is a good testbed for long-sighted action planning. Another interesting task would also be to transfer the resulted agents from this implementation, including their observation and reward-definition, to the TORCS environment to see how they perform in comparison to agents from the literature.

Normally when dealing with self-driving cars, there are countless additional issues, each making up a whole new challenge on their own. Examples for those are wheather conditions (snow, rain, fog), pedestrians, other cars, reflections, merging into ongoing traffic, and many more situations. The situation considered here is a small sample of all situations a real car must handle. A further plan for this project is to approach more of these situations. An interesting idea is for example to introduce semantic parameters to the neural network, telling the agent information about friction-properties of the road, such that one and the same agent can transfer its knowledge to new situations. This may lead to a similar approach to *InfoGAN*[9] in the realm of reinforcement learning.

7.3   Conclusion

In conclusion, the developed framework has proven to provide a reasonable reinforcement learning environment and enables quick creation and inclusion of new agents with distinct observation- and reward functions. It is therefore suited well to serve as a research platform for reinforcement learning in self-driving cars and even showed some possible weaknesses in other such platforms, such as TORCS.

While the optimization of in this thesis proposed functions and parameters will have to be continued by future research, they already enabled learning of reasonable driving-policies. This success was achieved even though the proposed framework seemed more difficult than average.

Due to a large amount of possible improvements that did not fit under the scope of this thesis, the author is optimistic that further optimization of the agents and their performance is possible and that the developed framework will be a useful tool for further development.

Generally, the field of reinforcement learning is making quick progress, and the possibilities of virtual training to be applied in physical self-driving cars are endless. It will be interesting to see, in what direction this research will go and how long it will take for reinforcement learning applications to become a norm in our everyday lives.

A Comparison pseudocode & Python-code

A.1 DQN

The following section describes the structure of an actual reinforcement learning agent, using a **Dueling Deep-Q-Network** as its model (as described in [34]), performing **Double Q-learning** (as described in [10]). The last page consists of a comparison between the pseudocode of the general program flow of a DDQN-network (taken from [20], with changes from [10] and [15] in blue) to the left and its corresponding python-code to the right, where each line of the pseudocode corresponds exactly to the respective line of the python-code. For information on which python- and tensorflow version are used, please see chapter 4. This code is extracted from the actual implementation within the scope of this thesis, with some changes abstracting away irrelevant details.

```python
 1  class Agent():
 2   def __init__(self, inputsize):
 3    self.inputsize = inputsize
 4    self.model = DDDQN_model #or DDPG_model
 5    self.memory = Memory(500000, self)  #for definition see code
 6    self.action_repeat = 4
 7    self.update_frequency = 4
 8    self.batch_size = 32
 9    self.replaystartsize = 1000
10    self.epsilon = 0.05
11    self.last_action = None
12    self.repeated_action_for = self.action_repeat

14   def performAction(self, gameState, pastState):
15    self.numsteps += 1
16    self.repeated_action_for += 1
17    self.addToMemory(gameState, pastState)
18    if self.stepsAfterStart <= self.conf.headstart_num:
19     toUse, toSave = self.headstartAction()
20    elif self.repeated_action_for < self.action_repeat:
21     toUse, toSave = self.last_action
22    else:
23     agentState = self.getAgentState(*gameState) #may be overridden
24     if len(self.memory) >= self.conf.replaystartsize:
25      self.epsilon = decrease(epsilon)
26      if np.random.random() < self.epsilon:
27       toUse, toSave = self.randomAction(agentState)
28      else:
29       toUse, toSave = self.policyAction(agentState)
30     else:
31      toUse, toSave = self.randomAction(agentState)
32     self.last_action = toUse, toSave
33    self.containers.outputval.update(toUse, toSave)
34    if self.numsteps % self.conf.ForEveryInf == 0:
35     self.learnStep(self.conf.ComesALearn)

37   def learnStep(self, iterations):
38    for i in range(iterations):
```

```python
39      QLearnInputs = self.memory.sample(self.batch_size)
40      self.model.q_train_step(QLearnInputs)

42    def addToMemory(self, gameState, pastState):
43      s = self.getAgentState(*pastState)  #for definition see code
44      a = self.getAction(*pastState)      #for definition see code
45      r = self.calculateReward(*gameState)#for definition see code
46      s2= self.getAgentState(*gameState)  #for definition see code
47      t = False #will be updated if episode did end
48      self.memory.append([s,a,r,s2,t])

50  class DuelDQN():
51    def __init__(self, name, inputsize, num_actions):
52     with tf.variable_scope(name, initializer = tf.random_normal_initializer(0, 1e-3)):
53      #for the inference
54      self.inputs = tf.placeholder(tf.float32, shape=[None, inputsize], name="inputs")
55      self.fc1 = tf.layers.dense(self.inputs, 400, activation=tf.nn.relu)
56      #modifications from the Dueling DQN architecture
57      self.streamA, self.streamV = tf.split(self.fc1,2,1)
58      xavier_init = tf.contrib.layers.xavier_initializer()
59      neutral_init = tf.random_normal_initializer(0, 1e-50)
60      self.AW = tf.Variable(xavier_init([200,self.num_actions]))
61      self.VW = tf.Variable(neutral_init([200,1]))
62      self.Advantage = tf.matmul(self.streamA,self.AW)
63      self.Value = tf.matmul(self.streamV,self.VW)
64      self.Qout = self.Value + tf.subtract(self.Advantage,tf.reduce_mean(self.Advantage,axis=1,keep_dims=
            ➡ True))
65      self.Qmax = tf.reduce_max(self.Qout, axis=1)
66      self.predict = tf.argmax(self.Qout,1)
67      #for the learning
68      self.targetQ = tf.placeholder(shape=[None],dtype=tf.float32)
69      self.targetA = tf.placeholder(shape=[None],dtype=tf.int32)
70      self.targetA_OH = tf.one_hot(self.targetA, self.num_actions, dtype=tf.float32)
71      self.compareQ = tf.reduce_sum(tf.multiply(self.Qout, self.targetA_OH), axis=1)
72      self.td_error = tf.square(self.targetQ - self.compareQ)
73      self.q_loss = tf.reduce_mean(self.td_error)
74      q_trainer = tf.train.AdamOptimizer(learning_rate=0.00025)
75      q_OP = q_trainer.minimize(self.q_loss)
76     self.trainables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=name)

78  def netCopyOps(fromNet, toNet, tau = 1):
79   op_holder = []
80   for idx,var in enumerate(fromNet.trainables[:]):
81    op_holder.append(toNet.trainables[idx].assign((var.value()*tau) + ((1-tau)*toNet.trainables[idx].
          ➡ value())))
82   return op_holder
```

```
1   #see agent
2   class DDDQN_model():
3     def __init__(self, sess, inputsize, num_action):
4       self.sess = sess
5       self.onlineQN = DuelDQN("onlineNet", inputsize, num_action)
6       self.targetQN = DuelDQN("targetNet", inputsize, num_action)
7       self.sess.run(tf.global_variables_initializer())
8       self.sess.run(netCopyOps(self.targetQN, self.onlineQN))

10    #see agent
11    def inference(self, statesBatch): #called for every step t
12      #model is not accessed if random action
13      return self.sess.run([self.targetQN.predict,self.targetQN.Qout],feed_dict={
          ↪ self.targetQN.inputs:statesBatch,self.targetQN.stands_input:False})
14    #see agent
15    #see agent
16    #see agent
17    def q_train_step(self, batch): #also called for every step t
18      oldstates, actions, rewards, newstates, terminals = batch
19      act = self.sess.run(self.onlineQN.predict, {self.onlineQN.inputs:newstates})
20      folgeQ = self.sess.run(self.targetQN.Qout, {self.targetQN.inputs:newstates})
21      doubleQ = folgeQ[range(len(terminals)),act]
22      consider_stateval = -(terminals - 1)
23      targetQ = rewards + (0.99 * doubleQ * consider_stateval)
24      self.sess.run(self.onlineQN.q_OP, feed_dict={self.onlineQN.inputs:oldstates,
          ↪ self.onlineQN.targetQ:targetQ, self.onlineQN.targetA:actions})
25      self.sess.run(netCopyOps(self.onlineQN, self.targetQN, 0.001))
26      return
```

1   Initialize replay memory D to capacity N

5   Initialize action-value function $Q(s,a;\theta)$ with random weights θ

8   Initialize target action-value function $Q(s,a;\theta^-)$ with weights $\theta^- = \theta$

9   **For** episode = 1,M **do**

10   Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

11   **For** 1 = 1,T **do**

12   With probability ϵ select random action a_t

13   otherwise select $a_t = argmax_a Q(\phi(s_t),a;\theta)$

15   Execute action a_t in emulator and observe reward r_t and image x_{t+1}

16   Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

17   Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

19   Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

20   Define $a^{max}(\phi_{j+1};\theta) = argmax_{a'} Q(\phi_{j+1},a';\theta)$

21   Define $Q^{j+1} = Q(\phi_{j+1},a^{max}(\phi_{t+1};\theta);\theta^-)$

23   If episode terminates at step $j+1$ then set $y_j = r_j$,
     ↪ Otherwise set $y_j = r_j + \gamma * Q^{j+1}$

24   Perform a gradient descent step on $\left(y_j - Q(\phi_j,a_j;\theta)\right)^2$ with respect
     ↪ to the network parameters θ

25   Update target network: $\theta^- \leftarrow \tau*\theta + (1-\tau)\theta^-$

26   **End For**

27   **End For**

A.2   DDPG

The following section describes the structure of a reinforcement learning agent, using an **actor-critic architecture** as its model, basing on the Deep Deterministic Policy gradient, as described in [29] and [15]. The last page consists of a comparison between the pseudocode of the general program flow of a DDPG-agent (taken from [15]) to the left and its corresponding python-code to the right, where each line of the pseudocode corresponds exactly to the respective line of the python-code. For information on which python- and tensorflow version are used, please see chapter 4. This code is extracted from the actual implementation within the scope of this thesis, with some changes abstracting away irrelevant details. Note that this listing does not contain a definition of a class *agent*, as it is the same defined as in lines 1-48 of appendix A.1.

```
1   class Actor(object):
2    def __init__(self, inputsize, num_actions, actionbounds, session):
3     with tf.variable_scope("actor"):
4      self.online = lowdim_actorNet(inputsize, num_actions, actionbounds)
5      self.target = lowdim_actorNet(inputsize, num_actions, actionbounds, name="target")
6      # provided by the critic network
7      self.action_gradient = tf.placeholder(tf.float32, [None, num_actions], name="actiongradient")
8      self.actor_gradients = tf.gradients(self.online.scaled_out, self.online.trainables, -self.
              ➥ action_gradient)
9      self.optimize = tf.train.AdamOptimizer(1e-4).apply_gradients(zip(self.actor_gradients, self.online.
              ➥ trainables))
10    def train(self, inputs, a_gradient):
11     self.session.run(self.optimize, feed_dict={self.online.ff_inputs:inputs, self.action_gradient:
              ➥ a_gradient})
12    def predict(self, inputs, which="online"):
13     net = self.online if which == "online" else self.target
14     return self.session.run(net.scaled_out, feed_dict={net.ff_inputs:inputs})
15    def update_target_network(self):
16     self.session.run(netCopyOps(self.online, self.target, 0.001))

18   class Critic(object):
19    def __init__(self, inputsize, num_actions, session):
20     with tf.variable_scope("critic"):
21      self.online = lowdim_criticNet(inputsize, num_actions)
22      self.target = lowdim_criticNet(inputsize, num_actions, name="target")
23      self.target_Q = tf.placeholder(tf.float32, [None, 1], name="target_Q")
24      self.loss = tf.losses.mean_squared_error(self.target_Q, self.online.Q)
25      self.optimize = tf.train.AdamOptimizer(1e-3).minimize(self.loss)
26      self.action_grads = tf.gradients(self.online.Q, self.online.actions)
27    def train(self, inputs, action, target_Q):
28     return self.session.run([self.optimize, self.loss], feed_dict={self.online.ff_inputs:inputs, self.
              ➥ online.actions: action, self.target_Q: target_Q})
29    def predict(self, inputs, action, which="online"):
30     net = self.online if which == "online" else self.target
31     return self.session.run(net.Q, feed_dict={net.ff_inputs:inputs, net.actions: action})
32    def action_gradients(self, inputs, actions):
33     return self.session.run(self.action_grads, feed_dict={self.online.ff_inputs:inputs, self.online.
              ➥ actions: actions})
34    def update_target_network(self):
35     self.session.run(netCopyOps(self.online, self.target, 0.001))

37   def netCopyOps(fromNet, toNet, tau = 1):
38    op_holder = []
```

```
39    for idx,var in enumerate(fromNet.trainables[:]):
40     op_holder.append(toNet.trainables[idx].assign((var.value()*tau) + ((1-tau)*toNet.trainables[idx].
            ➥ value())))
41    return op_holder


43  def dense(x, units, activation=tf.identity, decay=None, minmax = float(x.shape[1].value) ** -.5):
44    return tf.layers.dense(x, units,activation=activation, kernel_initializer=tf.
          ➥ random_uniform_initializer(-minmax, minmax), kernel_regularizer=decay and tf.contrib.layers.
          ➥ l2_regularizer(1e-2))


46  class lowdim_actorNet():
47    def __init__(self, inputsize, num_actions, actionbounds, outerscope="actor", name="online"):
48      tanh_min_bounds,tanh_max_bounds = np.array([-1]), np.array([1])
49      min_bounds, max_bounds = np.array(list(zip(*actionbounds)))
50      self.name = name
51      with tf.variable_scope(name):
52        self.ff_inputs =   tf.placeholder(tf.float32, shape=[None, inputsize], name="ff_inputs")
53        self.fc1 = dense(self.ff_inputs, 400, tf.nn.relu, decay=decay)
54        self.fc2 = dense(self.fc1, 300, tf.nn.relu, decay=decay)
55        self.outs = dense(self.fc2, num_actions, tf.nn.tanh, decay=decay, minmax = 3e-4)
56        self.scaled_out = (((self.outs - tanh_min_bounds)/ (tanh_max_bounds - tanh_min_bounds)) * (
            ➥ max_bounds - min_bounds) + min_bounds)
57        self.trainables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=outerscope+"/"+self.
            ➥ name)


59  class lowdim_criticNet():
60    def __init__(self, inputsize, num_actions, outerscope="critic", name="online"):
61      self.name = name
62      with tf.variable_scope(name):
63        self.ff_inputs =   tf.placeholder(tf.float32, shape=[None, inputsize], name="ff_inputs")
64        self.actions = tf.placeholder(tf.float32, shape=[None, num_actions], name="action_inputs")
65        self.fc1 = dense(self.ff_inputs, 400, tf.nn.relu, decay=True)
66        self.fc1 =  tf.concat([self.fc1, self.actions], 1)
67        self.fc2 = dense(self.fc1, 300, tf.nn.relu, decay=True)
68        self.Q = dense(self.fc2, 1, decay=True, minmax=3e-4)
69        self.trainables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=outerscope+"/"+self.
            ➥ name)
```

Python-code:

```python
1   class DDPG_model():
2     def __init__(self, session):
3       self.session = session
4       self.critic = Critic(self.session)
5       self.actor = Actor(self.session)
6       self.session.run(tf.global_variables_initializer())
7       self.session.run(netCopyOps(self.actor.target, self.actor.online))
8       self.session.run(netCopyOps(self.critic.target, self.critic.online))
9       #replay buffer defined by the agent

11      #exploration noise added by the agent
12      #agent samples all observations
13    def inference(self, oldstates): #called for every step t
14      return self.actor.predict(oldstates, "target", is_training=False)
15      #agent adds exploration noise afterwards
16      #done by the agent
17      #done by the agent
18    def q_train_step(self, batch): #also called for every step t
19      oldstates, actions, rewards, newstates, terminals = batch
20      targetActorAction = self.actor.predict(newstates, "target")
21      targetCriticQ = self.critic.predict(newstates, targetActorAction, "target")
22      cumrewards = np.reshape([rewards[i] if terminals[i] else rewards[i]+0.99*
          ➥ targetCriticQ[i] for i in range(len(rewards))], (len(rewards),1))
23      -, loss = self.critic.train(oldstates, actions, cumrewards)

25      onlineActorActions = self.actor.predict(oldstates)
26      grads = self.critic.action_gradients(oldstates, onlineActorActions)
27      self.actor.train(oldstates, grads[0])
28      #updating the targetnets
29      self.critic.update_target_network()
30      self.actor.update_target_network()
31      return
```

Pseudocode:

4   Randomly initialize critic network $Q(s,a|\theta^Q)$ with weights θ^Q

5   and actor $\pi(s|\theta^\pi)$ with weights θ^π.

7   Initialize target network Q' weights $\theta^{Q'} \leftarrow \theta^Q$

8   and π' with weights $\theta^{\pi'} \leftarrow \theta^\pi$

9   Initialize replay buffer R

10   **for** episode = 1, M **do**

11   Initialize a random process $\mathcal{N}$ **for** action exploration

12   Receive initial observation state s_1

13   **for** t = 1, T **do**

14   Select action $a_t = \pi(s_t|\theta^\pi) + \mathcal{N}_t$ according to the current policy and ➥ exploration noise

15   Execute action a_t and observe reward r_t and observe new state s_{t+1}

16   Store transition (s_t, a_t, r_t, s_{t+1}) in R

18   Sample a random minibatch of N transitions (s_t, a_t, r_t, s_{t+1}) from R

19   targetActorAction = $\pi'(s_{i+1}|\theta^{\pi'})$

20   targetCriticQ = $Q'(s_{i+1}, \text{targetActorAction}|\theta^{Q'})$

21   Set $y_i = r_i + \gamma*\text{targetCriticQ}$ #only in nonterminal states

23   Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

24   Find the sampled policy gradient:

25   a_i = $\pi(s_i|\theta^\pi)$

26   $\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_a Q(s_i, a_i|\theta^Q) \nabla_{\theta^\pi} \pi(s_i|\theta^\pi)$

27   Update the actor policy using the sampled policy gradient

28   Update the target networks:

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

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

31   **end for**

32   **end for**
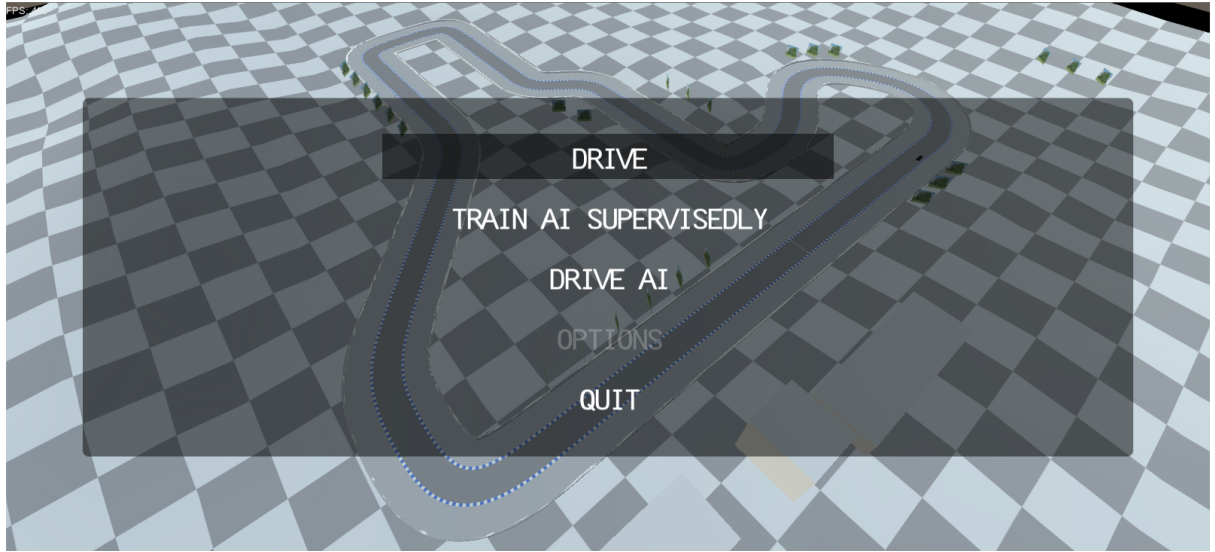
B Screenshots

B.1 Game

FIGURE B.1: Start screen / menu of the game, also showing a bird-eye view of the track

FIGURE B.2: **Drive** mode. For a description of the UI components, it is referred to section 4.2.1
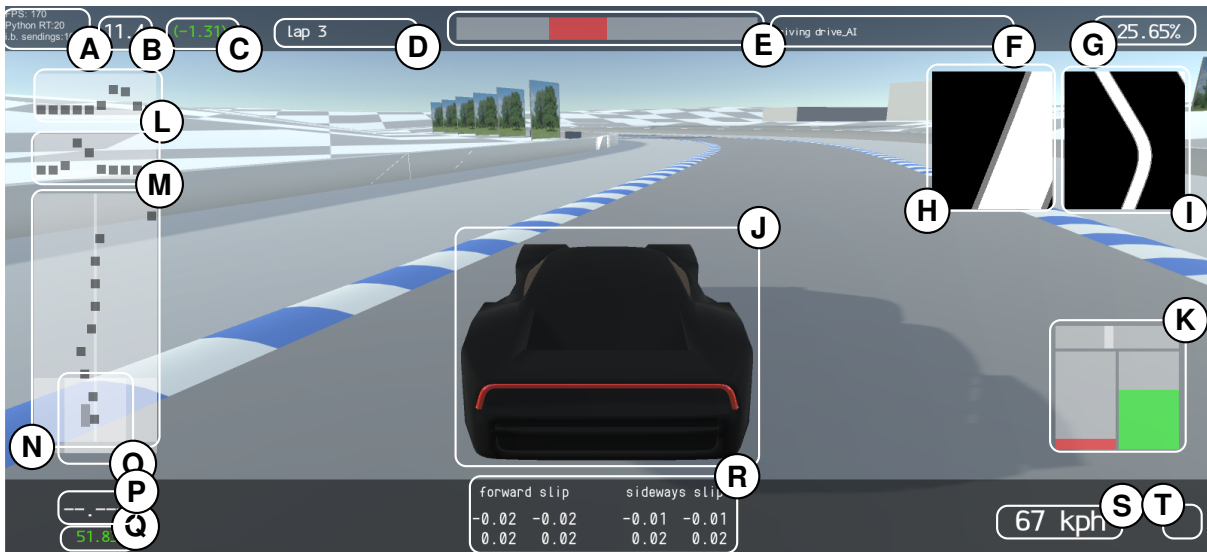
FIGURE B.3: **Drive AI** mode, showing many additional information directly on
the screen. For a description of those, it is referred to sections 4.2.1 and 5.1

- **A**: Debug information. Shows FPS, the agent's response time and the time in between two sendings to the agent (the latter two only visible in `drive_AI` mode).

- **B**: The current lap time in seconds.

- **C**: The time difference of the current lap in comparison to the fastest lap so far.

- **D**: Indicator of the current lap. Also shows if a lap is `invalid`.

- **E**: Feedback bar, graphically visualizing the time difference of only the current course section in comparison to the fastest lap so far.

- **F**: Indicator for the current game mode. Also indicates if QuickPause is active or if a human interferes in the `drive_AI` mode.

- **G**: Current track progress in percent.

- **H**: Field of view of the first minimap-camera.

- **I**: Field of view of the second minimap-camera, if enabled.

- **J**: The car. As the main camera is fixed behind it, it will always be in this precise position.

- **K**: Visual representation of the values for steering (top), brake-pedal (bottom left) and throttle pedal (bottom right).

- **L**: Visual representation of the game's *progress-vector*. Only visible in `drive_AI` and `train_AI`.

- **M**: Visual representation of the game's *CenterDist-vector*. Only visible in `drive_AI` and `train_AI`.

- **N**: Visual representation of the game's *lookahead-vector*. Only visible in `drive_AI` and `train_AI`.

- **O**: Alternative representation of car's distance to the lane center. Only visible in `drive_AI` and `train_AI`. Overlapping with N due to low screen resolution on the machine used for the screenshot.

- **P**: Time needed for the last valid lap in seconds.

- **Q**: Time needed for the fastest valid lap (throughout different sessions) in seconds.

- **R**: Information about slip-behavious of the car's tires.

- **S**: Speed of the car in kilometers per hour.

- **T**: Indicates a "P" if the car is in reverse gear.
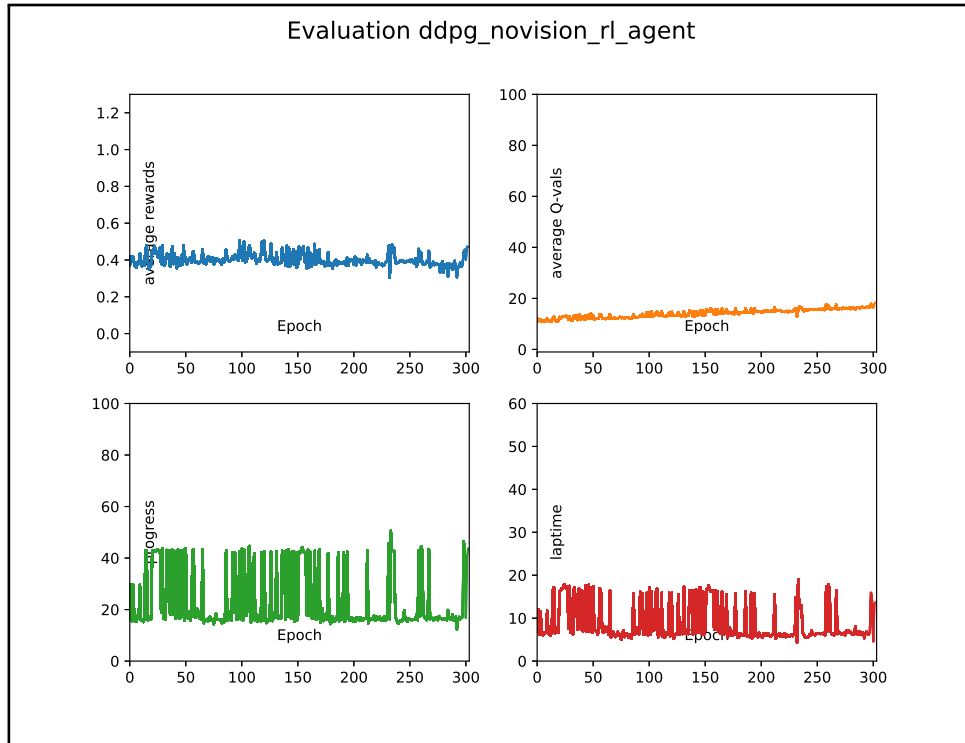
B.2 Agent

FIGURE B.4: A plot as the agent continually shows and updates during runtime
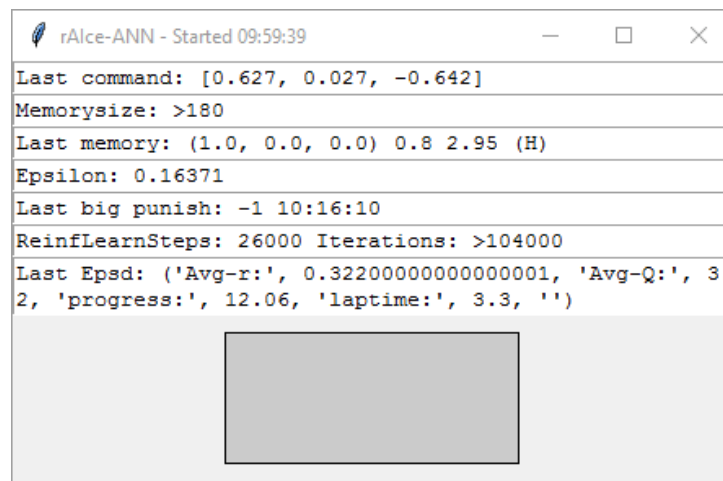

FIGURE B.5: Screenshot of the GUI in a typical run. The colored area at the bottom encodes the current reward via its intensity.
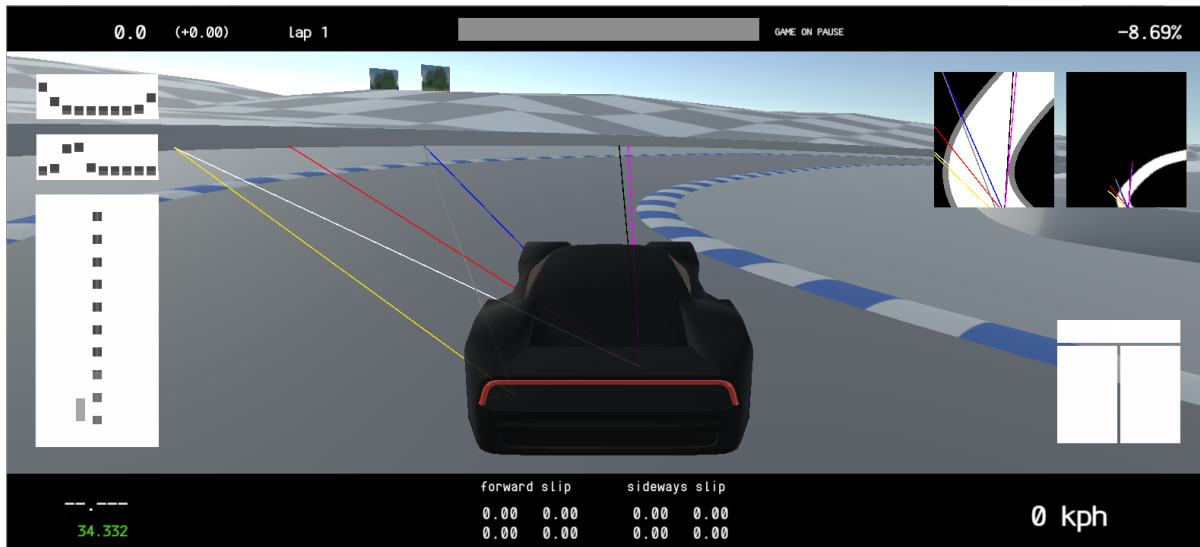
B.3 Vectors

FIGURE B.6: Graphical representation of the rays whose lenghts correspond to the `WallDistVec`. Note that the white and yellow ray as well as the blue and grey ray are actually parallel, as can be seen in the orthogonal minimap-cameras. To see these rays, `Gizmos` must be turned on in Unity.

C Code-excerpts

C.1 A minimal viable agent

```python
1   import tensorflow as tf
2   #====own classes====
3   from agent import AbstractRLAgent
4   from dddqn import DDDQN_model



8   class Agent(AbstractRLAgent):
9   def init(self, conf, containers, isPretrain=False, start_fresh=False, *args, **kwargs):
10    self.name = "dqn_rl_agent"
11    super().init(conf, containers, isPretrain, start_fresh, *args, **kwargs)
12    self.ff_inputsize = conf.speed_neurons + conf.num_actions * conf.ff_stacksize #32
13    self.model = DDDQN_model(self.conf, self, tf.Session(), isPretrain=isPretrain)
14    self.model.initNet(load=("preTrain" if (self.isPretrain and not start_fresh) else (not
          ➡ start_fresh)))



17  def policyAction(self, agentState):
18    action, _ = self.model.inference(self.makeInferenceUsable(agentState))
19    throttle, brake, steer = self.dediscretize(action[0])
20    toUse = "["+str(throttle)+", "+str(brake)+", "+str(steer)+"]"
21    return toUse, (throttle, brake, steer)
```

C.2 The used calculateReward-function

```python
1   def calculateReward(self, *gameState):
2     vvec1_hist, vvec2_hist, otherinput_hist, action_hist = gameState
3     self.steeraverage.append(action_hist[1][2])
4     dist = otherinput_hist[0].CenterDist[0]-0.5   #abs is 0 for center, 0.15 for curb, 0.5 wall
5     angle = otherinput_hist[0].SpeedSteer.carAngle - 0.5
6     badspeed = abs(2*otherinput_hist[0].SpeedSteer.speedInTraverDir-1)*5


8     stay_on_street = ((0.5-abs(dist))*2)+0.35
9     stay_on_street = stay_on_street**0.1 if stay_on_street > 1 else stay_on_street**2
10    #flat on-street, steep off-street
11    stay_on_street = ((1-((0.5-abs(dist))*2))**10) * -self.wallhitPunish + (1-(1-((0.5-abs(
          ➡ dist))*2))**10) * stay_on_street
```

```
12   #the influence of wallhitpunish is exponentially more relevant closer to the wall
13   stay_on_street -= 0.5 #in range [0.5,-1.5] for wallhitpunish=1

15   direction_bonus = abs((0.5-(abs(angle)))*2/0.75)
16   direction_bonus = ((direction_bonus**0.4 if direction_bonus > 1 else direction_bonus**2) /
         ➥   1.1 / 2) - 0.25 #no big difference until 45degrees, then BIG diff.
17   #maximally 0.25, minimally -0.25

19   tmp = (np.mean(self.steeraverage))
20   steer_bonus1 = tmp/5 + angle #rewards sterering into street-direction if the cars angle is
         ➥   off
21   steer_bonus1 = 0 if np.sign(steer_bonus1) != np.sign(angle) and abs(angle) > 0.15 else
         ➥   steer_bonus1
22   steer_bonus1 = (abs(dist*2)) * ((0.5-abs(angle)) * (1-abs(steer_bonus1))) + (1-abs(dist*2)
         ➥   )*0.5
23   #more relevant the further off you are.
24   steer_bonus2 = (1-((0.5-abs(dist))*2))**10 * -abs(((tmp+np.sign(dist))*np.sign(dist)))/1.5
25   #more relevant further off, steering away from wall is as valuable as doing nothing in
         ➥   center
26   steer = steer_bonus1+steer_bonus2 #maximally 0.5

28   #in front of curves, some values become less relevant
29   curveMultiplier = 1-abs(otherinput_hist[0].SpeedSteer.CurvinessBeforeCar-0.5)
30   direction_bonus *= curveMultiplier
31   badspeed *= curveMultiplier

33   speedInRelationToWallDist = otherinput_hist[0].WallDistVec[6]-otherinput_hist[0].
         ➥   SpeedSteer.speedInStreetDir+(80/250)
34   speedInRelationToWallDist = 1-(abs(speedInRelationToWallDist)*3) if
         ➥   speedInRelationToWallDist < 0 else (1-speedInRelationToWallDist)+0.33
35   speedInRelationToWallDist = min(1,speedInRelationToWallDist)
36   speedInRelationToWallDist += -badspeed + 0.3*otherinput_hist[0].SpeedSteer.
         ➥   speedInStreetDir
37   #rewards driving slow if close to wall

39   rew = (2*speedInRelationToWallDist + stay_on_street + 0.5*direction_bonus + 0.5*steer)/ 4

41   slidingToWall = (min(0.05, otherinput\_hist[0].WallDistVec[2]) / 0.05)**3
42   toWallSpeed =  (1-slidingToWall) * ((min(0.1, otherinput_hist[0].SpeedSteer.velocity) /
         ➥   0.1))
43   #if the car is sliding sliding to and almost at the wall, subtract a lot
44   tooslow = 1- ((min(0.2, otherinput_hist[0].SpeedSteer.speedInStreetDir) / 0.2) ** 3)
45   #drive faster at 0.2 at all times, its easily possible to keep this at 0 at all times
46   rew -= toWallSpeed
47   rew -= 0.5*tooslow

49   rew = max(rew, 0)
50   return rew
```
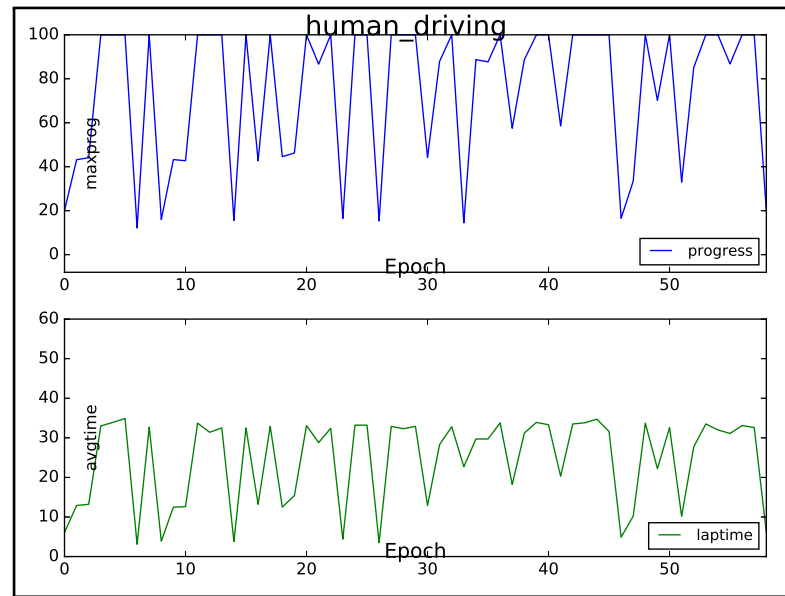
D Further performance-plots
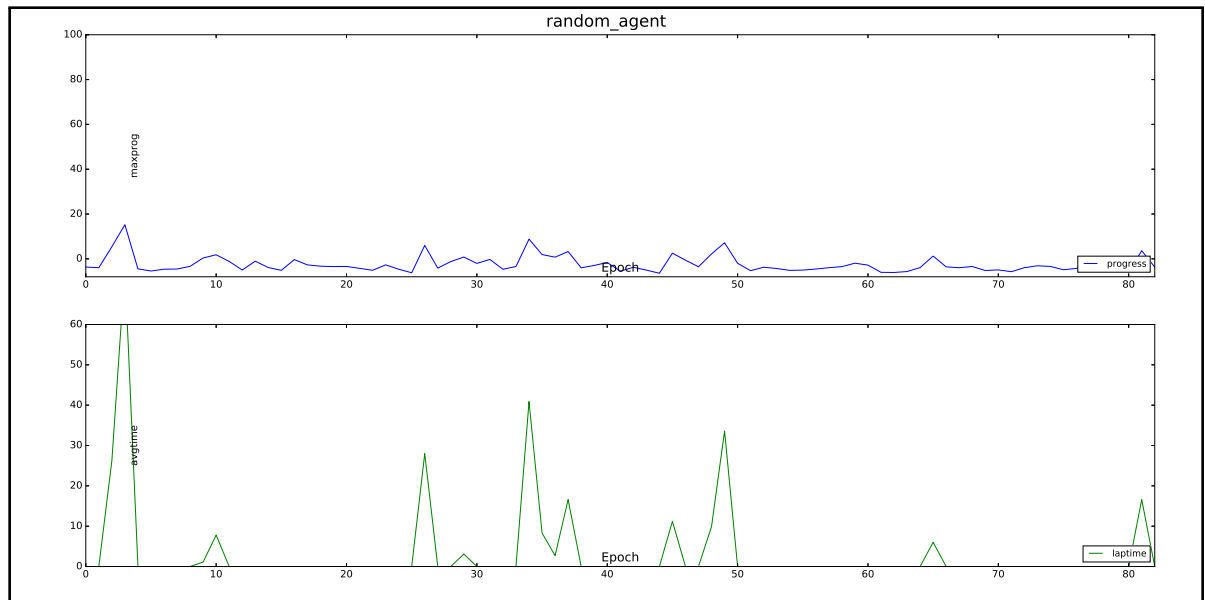

FIGURE D.1: Exemplary laps driven by a human

FIGURE D.2: Exemplary laps driven by a random agent

Bibliography

[1] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. *TensorFlow: Large-scale machine learning on heterogeneous systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[2] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *arXiv:1511.00561 [cs]* (Nov. 2015). arXiv: 1511.00561. URL: http://arxiv.org/abs/1511.00561 (visited on 09/11/2017).

[3] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *arXiv:1207.4708 [cs]* (July 2012). arXiv: 1207.4708. DOI: 10.1613/jair.3912. URL: http://arxiv.org/abs/1207.4708 (visited on 08/16/2017).

[4] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. "Unifying Count-Based Exploration and Intrinsic Motivation". In: *arXiv:1606.01868 [cs]* (June 2016). arXiv: 1606.01868. URL: http://arxiv.org/abs/1606.01868 (visited on 08/12/2017).

[5] Richard Bellman. *Dynamic Programming*. Princeton University Press. ISBN: 978-0-691-14668-3. URL: http://press.princeton.edu/titles/9234.html.

[6] Ben Lau. *Using Keras and Deep Deterministic Policy Gradient to play TORCS*. Oct. 2016. URL: https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html (visited on 09/14/2017).

[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, et al. "End to End Learning for Self-Driving Cars". In: *arXiv:1604.07316 [cs]* (Apr. 2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316 (visited on 08/16/2017).

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "OpenAI Gym". In: *arXiv:1606.01540 [cs]* (June 2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540 (visited on 08/16/2017).

[9] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. "InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets". In: *arXiv:1606.03657 [cs, stat]* (June 2016). arXiv: 1606.03657. URL: http://arxiv.org/abs/1606.03657 (visited on 09/14/2017).

[10] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *arXiv:1509.06461 [cs]* (Sept. 2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461 (visited on 08/12/2017).

[11] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv:1502.03167 [cs]* (Feb. 2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (visited on 08/12/2017).

[12] Jeong hwan Jeon, Sertac Karaman, and Emilio Frazzoli. "Anytime computation of time-optimal off-road vehicle maneuvers using the RRT". In: *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*. IEEE, 2011, pp. 3276–3282. URL: http://ieeexplore.ieee.org/abstract/document/6161521/ (visited on 09/09/2017).

[13] John N. Tsitsiklis and Benjamin Van Roy. "An Analysis of Temporal-Difference Learning with Function Approximation". In: *IEEE TRANSACTIONS ON AUTOMATIC CONTROL* 42.5 (May 1997). URL: http://web.mit.edu/jnt/www/Papers/J063-97-bvr-td.pdf (visited on 08/14/2017).

[14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Dec. 2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 08/12/2017).

[15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning". In: *arXiv:1509.02971 [cs, stat]* (Sept. 2015). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971 (visited on 08/12/2017).

[16] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. "Simulated Car Racing Championship: Competition Software Manual". In: *arXiv:1304.1672 [cs]* (Apr. 2013). arXiv: 1304.1672. URL: http://arxiv.org/abs/1304.1672 (visited on 09/08/2017).

[17] Jarryd Martin, Suraj Narayanan Sasikumar, Tom Everitt, and Marcus Hutter. "Count-Based Exploration in Feature Space for Reinforcement Learning". In: *arXiv:1706.08090 [cs]* (June 2017). arXiv: 1706.08090. URL: http://arxiv.org/abs/1706.08090 (visited on 09/09/2017).

[18] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv:1602.01783 [cs]* (Feb. 2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783 (visited on 09/09/2017).

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013). URL: https://arxiv.org/abs/1312.5602 (visited on 08/12/2017).

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, et al. "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836. DOI: 10.1038/nature14236. URL: http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html?foxtrotcallback=true.

[21] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, et al. "Massively Parallel Methods for Deep Reinforcement Learning". In: *arXiv:1507.04296 [cs]* (July 2015). arXiv: 1507.04296. URL: http://arxiv.org/abs/1507.04296 (visited on 09/13/2017).

[22] Georg Ostrovski, Marc G. Bellemare, Aaron van den Oord, and Remi Munos. "Count-Based Exploration with Neural Density Models". In: *arXiv:1703.01310 [cs]* (Mar. 2017). arXiv: 1703.01310. URL: http://arxiv.org/abs/1703.01310 (visited on 09/13/2017).

[23] Dean A. Pomerleau. "Alvinn: An autonomous land vehicle in a neural network". In: *Advances in neural information processing systems*. 1989, pp. 305–313. URL: http://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf (visited on 09/09/2017).

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 1st ed. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 1998. ISBN: 978-0-262-19398-6. URL: http://incompleteideas.net/sutton/book/ebook/the-book.html (visited on 08/17/2017).

[25] G. A. Rummery and M. Niranjan. *On-Line Q-Learning Using Connectionist Systems*. Tech. rep. 1994.

[26] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized Experience Replay". In: *arXiv:1511.05952 [cs]* (Nov. 2015). arXiv: 1511.05952. URL: `http://arxiv.org/abs/1511.05952` (visited on 08/12/2017).

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *arXiv:1707.06347 [cs]* (July 2017). arXiv: 1707.06347. URL: `http://arxiv.org/abs/1707.06347` (visited on 09/14/2017).

[28] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/nature16961`. URL: `http://www.nature.com/doifinder/10.1038/nature16961` (visited on 09/02/2017).

[29] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. "Deterministic policy gradient algorithms". In: *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 2014, pp. 387–395. URL: `http://www.jmlr.org/proceedings/papers/v32/silver14.pdf` (visited on 08/12/2017).

[30] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. "Striving for Simplicity: The All Convolutional Net". In: *arXiv:1412.6806 [cs]* (Dec. 2014). arXiv: 1412.6806. URL: `http://arxiv.org/abs/1412.6806` (visited on 09/08/2017).

[31] Richard S. Sutton. "Learning to predict by the methods of temporal differences". en. In: *Machine Learning* 3.1 (Aug. 1988), pp. 9–44. ISSN: 0885-6125, 1573-0565. DOI: `10.1007/BF00115009`. URL: `https://link.springer.com/article/10.1007/BF00115009`.

[32] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*. 2000, pp. 1057–1063. URL: `http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf` (visited on 08/18/2017).

[33] G. E. Uhlenbeck and L. S. Ornstein. "On the Theory of the Brownian Motion". In: *Physical Review* 36.5 (Sept. 1930), pp. 823–841. DOI: `10.1103/PhysRev.36.823`. URL: `https://link.aps.org/doi/10.1103/PhysRev.36.823` (visited on 08/12/2017).

[34] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. "Dueling Network Architectures for Deep Reinforcement Learning". In: *arXiv:1511.06581 [cs]* (Nov. 2015). arXiv: 1511.06581. URL: `http://arxiv.org/abs/1511.06581` (visited on 08/12/2017).

[35] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards". PhD thesis. King's College, May 1989. URL: `http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf` (visited on 08/10/2017).

[36] Christopher John Cornish Hellaby Watkins and Peter Dayan. "Technical Note - Q-Learning". In: *Machine Learning* 8 (1992), pp. 279–292. URL: `http://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf` (visited on 08/12/2017).

[37] Paweł Wawrzyński. "Control Policy with Autocorrelated Noise in Reinforcement Learning for Robotics". In: *International Journal of Machine Learning and Computing* 5.2 (Apr. 2015), pp. 91–95. ISSN: 20103700. DOI: `10.7763/IJMLC.2015.V5.489`. URL: `http://www.ijmlc.org/index.php?m=content&c=index&a=show&catid=56&id=551` (visited on 08/12/2017).

[38] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. *TORCS, the open racing car simulator*. 2013. URL: `http://www.torcs.org`.

[39] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. "Torcs, the open racing car simulator". In: *Software available at http://torcs. sourceforge. net* (2015). URL: `https://pdfs.semanticscholar.org/b9c4/d931665e c87c16fcd44cae8fdaec1215e81e.pdf` (visited on 08/16/2017).

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. URL: `http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf` (visited on 08/12/2017).

[41] Yurong You, Xinlei Pan, Ziyan Wang, and Cewu Lu. "Virtual to Real Reinforcement Learning for Autonomous Driving". In: *arXiv:1704.03952 [cs]* (Apr. 2017). arXiv: 1704.03952. URL: `http://arxiv.org/abs/1704.03952` (visited on 09/09/2017).

Declaration of Authorship

I, Christoph Stenkamp, hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

_____

signature

_____

city, date