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 Sütfeld
Prof. Dr. Gordon Pipa

Osnabrück,
11th September, 2017

Abstract

State-of-the-art self-driving cars rely on handcrafted algorithms to control the car's movement. As those systems rely on complete understanding of the environment, tactical decisions such as optimizing its speed are not considered as factor in their driving profile. In this thesis, a realistic 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. The racing simulation was given as a game implemented with the Unity game engine, to which this thesis adds the capabilities to be played and learned using artificial agents. After discussing the implementation of environment and agent, first experiments are conducted concerning the best reward-function, the effect of supervised pre-training and required inputs to the agent.

Preface

This document was written as the author's bachelor thesis at the department of neuroinformatics at the University of Osnabrück during summer 2017 and is an original and independent work by the author Christoph Stenkamp.

Christoph Stenkamp
Osnabrück, 11th September, 2017

Acknowledgements

Thanks to my parents, Marie, my supervisors, and my friends. . . .

Contents

List of Figures

List of Tables

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 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
DDPG	**D**eep **D**eterministic **P**olicy **G**radient - Network
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
POMDP	**P**artially **O**bserved **M**arkov **D**ecision **P**rocess
TORCS	**T**he **O**pen **R**acing **C**ar **S**imulator *(software)*

List of Symbols

For my friends, family, and especially Marie.

Chapter 1

Introduction

1.1 Motivation

Self-driving cars are becoming more and more a center of public attention.

Google's self-driving car, Tesla autopilot, die vision von Ubers autonomous taxis, ...

self-driving cars are the shit

Following https://arxiv.org/pdf/1704.03952.pdf, it is possible to transform virtual driving agents to real agents

https://qz.com/694520/tesla-has-780-million-miles-of-driving-data-and-adds-another-million-every-10-hours/

1.2 Goal of this thesis

"Leon, die ganze scheiße drumherum zu machen dauert lange"
"make good tactical race decisions."
"die karre schnell um kurs"

1.2.1 Driving

As put forward by *Lex Fridman* in his MIT lecture "*Deep Learning for Self-Driving Cars*"[1] 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*[33], 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 was 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 are made, offensive tactical profiles require good operational systems. In car racing, be

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

it virtual or real motorsports, extreme tactical profiles are used, as the goal is to perform at the limits of the possbile. 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 throughout 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 course 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 archieve 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 many environments openAI's *gym* or the *TORCS* platform), in this thesis a proprieatary software will be 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 in textual form 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 the source code 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.

1.2.3 Research Questions

Additionally to implementing platform and agent, some first research questions will be crystallized and answered. In doing so, different agents will be developed, and their performance compared. Some of the 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 presenteted. 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 of 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 self-driving cars, in real life as well as in simulations, will be given.
Chapter 4	details the program architecture 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.
Chapter 5	explains the result of the implementation the agents that crystallize out
Chapter 6	presents first results and answers of the research questions
Chapter 7	ends this thesis with discussion and conclusion.

Chapter 2

Reinforcement Learning

As the task at hand was not only to provide a reinforcement learning agent, but also to convert a game itself into something the agent can successfully play, In this chapter I will go into detail about reinforcement learning in general, giving insights on the specific approach chosen. The descriptions will be kept as general as possible at first, with detailed explanations following in the sections about specific algorithms.

2.1 Reinforcement Learning Problems

Machine Learning can mainly 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. In unsupervised learning, no such label exists, and the data must be clustered into meaningful parts without any knowledge, for example by grouping objects by similarity in their properties.
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, there is a *weak teacher*, which provides feedback on actions performed by the learner.

Markov Decision Processes

RL 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. In contrast to the classical ML approaches, in RL the agent is also responsible for exploration, as he has to acquire his knowledge actively. Thus, a reinforcement learning problem is given if the only way to collect information about the *underlying model* (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 training episode ends.

In the classical RL approach, the environment is divided into discrete time steps. If that is the case, the environment corresponds to a *Markov Decision Process* (**MDP**). Formally, a MDP is a 5-tuple $\langle S, A, P, R, \gamma \rangle$, consisting of the following:

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

In general, both the state transition function and the reward function may be indeterministic, meaning 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. Given both s and s' however, the reward is assumed to be deterministic. 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.

While an *offline learner* receives the problem definition as input in the form of a complete MDP, where the only task left is to classify 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, performing actions a. The future state s_{t+1} of the environment may be indeterministic, but depends on the history of previous states $s_0, .., s_t$ as well as the action of the agent a_t. It is assumed that the *Markov property* holds, which means that a state s_{t+1} depends only on the current state s_t and currenct action a_t: $p(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_0, a_0, .., s_t, a_t)$

Throughout interacting with the environment, the agent receives rewards r, depending on his action a as well as the state of the environment s. 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 observable 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 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 tries to perform as well as possible in the previously unknown environment. For that, it uses an action-policy π, depending on some parameters θ. The policy maps states to actions, which in the case of a *deterministic* policy leads to $\pi_\theta(s) = a$. 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 the ground truth, 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 archieve by continuing to follow its policy π from this

[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 $o(s_e)$

[2]It is obvious, that 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.

state on:

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

As the future rewards depend on future states, it can only be talked about the expected value depending on the actual state distribution. This distribution depends on the agents policy, but may still be indeterministic[3]. 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 archievable 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, then the optimal policy can be defined as the one archieving maximal value for every state of the MDP: $\pi^* := argmax_{\pi}V^{\pi}(s)\forall s \in \mathcal{S}$. Knowing what an optimal policy does, the definition of the value $V^{\pi}(s)$ 2.1 can be written recursively as

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

This relation is known as the *Bellman Equation*, which allowed for the birth of dynamic programming[4].

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, afterwards following the policy π:

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

[3]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 hopefully helps making the agent perform good actions as quickly as possible.

[4]Dynamic programming is another solution strategy for MDPs. In contrast to RL however, it requires the complete MDP as input to find an optimal policy, which cannot be given in many relevant situations.

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

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

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. This is because of the definition of Bellman's *Principle of Optimality*, which 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 [4]). Thanks to the principle of optimality, the value of our 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:

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

As the value of a state is defined as the maximally archievable 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 the actual Q-value for each action of each state is known, then the optimal policy can be defined as the one taking the optimal action in 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 (more on that later).

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 [23], be expressed by the expectation of the value of states following the distribution $s \sim \rho^\pi$:

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

We assume for now that once an agent knows Q^*, it can simply follow the policy that always takes the action yielding the highest value for every state (the *greedy* policy)[5].

[5]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 thesis.

Thus, the goal of a model-free RL agent is to get a maximally precise estimate of Q^*. 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 le, whereas for areas of interest, the function is estimated using a nonlinear function approximator. The agent's approximation of Q^π will be denoted $\hat{Q}^\pi$.

Throughout exploration of the environment, the agent collects more information about it, continually updating its estimate $\hat{Q}^\pi$. For that, it uses samples from its episodes of interacting with the environment.

2.2 Temporal difference learning

Throughout the process of reinforcement learning, the agent continually improves its estimates $\hat{Q}^\pi$ of Q^π. The loss of its current estimate could be seen as the squared difference $(\hat{Q}^\pi - Q^\pi)^2$, however, as the agent has no knowledge of Q^π, it needs some way of approximating it. For that, a Q-learning agent performs *iterative approximation*, using the information about the environment, to continually update its estimates of Q^π. Using the recursive definition of a Q-value given in the Bellman equation 2.4 allows for a technique called *temporal difference learning*[25]: At time $t+1$, the agent can compare its estimate of the Q-function of the last step, $\hat{Q}^\pi(s_t, a_t)$, 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:

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

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

Keeping in mind that $\hat{Q}^\pi$ is only an estimator of the Q^π-values of the underlying model, it becomes clear that equation 2.9 is closer to the actual Q^π, as it incorporates more information stemming from the model itself.

In temporal difference learning, the mean-squared error of the *temporal difference* from this Bellman equation, $r_t + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$, gets minimized via iterative approximation. Even though $r_t + \gamma * \hat{Q}^\pi(s_{t+1}, a_{t+1})$ also uses an estimate, it contains more information from the environment, and is thus a *more informed guess* than $\hat{Q}^\pi(s_s, a_s)$. That makes it reasonable to substitute the unknown $Q^\pi(s_{t+1}, a_{t+1})$ by $\hat{Q}^\pi(s_{t+1}, a_{t+1})$.
It is noteworthy, that each update of the Q-function using the temporal difference will affect not only the last prediction, but all previous predictions.

SARSA

The new knowledge about the environment can be incorporated in two different ways. For the first method, the agent 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 $TD := (r_t + \gamma * \hat{Q}^\pi(s_{t+1}, a_{t+1})) - \hat{Q}^\pi(s_t, a_t)$. This algorithm of calculating the temporal difference error is known as SARSA, and it 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. If the policy of the agent is not stochastic, this method can however lead to it getting stuck in local optima.

Q-learning

The *Q-learning* algorithm [29] stands in contrast to SARSA. Q-learning does not need to sample the action a_{t+1}, as it calculates the Q-update at iteration i using the best possible action in state s_{t+1}[6].

As the previous definition of Q-values was only correct in non-terminal states, a case differentiation must be introduced for 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, almost resulting in the formula found in [17]. To express its dependence on the policy π, it will be superscripted:

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

The temporal difference error for time t is accordingly defined as

$$TD_t := y_t^\pi - \hat{Q}^\pi(s_t, a_t) \tag{2.11}$$

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 if 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 (his *model*) with a lookup-table, initialized to all zeros. It is proven by [30] that for finite-state Markovian problems with nonnegative rewards the update-rule for the Q-table (with $0 \leq \alpha \leq 1$ as the learning rate)

$$\hat{Q}_{i+1}^\pi(s_t, a_t) \leftarrow \alpha * \left(r_t^{a_t} + \gamma * \hat{Q}_i^\pi(s_{t+1}^{a_t}, a_{t+1}) \right) + (1 - \alpha) * \hat{Q}_i^\pi(s_t, a_t) \tag{2.12}$$

converges to the optimal Q^*-function, making the greedy policy π^* optimal[7]. Note, that the same update rule as for the Q-function could be performed for the V-function.

In contrast to SARSA, Q-learning is an *off-policy* algorithm, meaning that the policy it uses in its evaluation of the Q-value is not necessarily the one it actually uses: When calculating the temporal difference error, the agent considers Q-values $\hat{Q}^{\pi_{greedy}}(s, a)$, based on $\pi_{greedy}(s) = argmax_{a'} \hat{Q}(s, a')$, as better approximation of the real action-value function $Q^\pi(s, a)$. Therefore, it learns about π_{greedy}, which is to always take the action promising maximum value. Because following the deterministic π_{greedy} does not allow for *exploration*, this is not the policy the agent actually pursues.

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,

[6]A slight deviation from this is*double-Q-learning*, an architecture I will go into detail about later on.
[7]Of course the agent will need some kind of exploration technique first, more on that later

as it must incorporate that while the value of a state is calculated using our deterministic π, the distribution of states follows from stoachastic policy β:

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

The process 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 considered here, 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 to be a *soft* version of the greedy policy $\pi_{i+1}(s) := argmax_{a'} Q^{\pi_i}(s,a') \forall s \in \mathcal{S}$, while keeping a mechanism allowing for exploration. Learning with this approach is however generally limited: $argmax_{a'} Q(\cdot, a')$ can only easily be found in settings where the action space $\mathcal{A}$ is finite and discrete, as it requires a global maximization over all possible actions. In a later section, I will go into detail about another architecture which does circumvents those problems by splitting up policy evaluation and policy improvement explicitly.

As there are also relevant situations in which discrete actions $\mathcal{A} \subseteq \mathbb{N}^n$ are sufficient, I will stick to those situations for now. Also in these circumstances, a Q-learner using 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*, which 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, which will be elaborated lateron in more detail. The L2-Loss for batch $batch$ with model-parameters θ_i, making up the policy π_{θ_i} is thus defined as the following:

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

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)[34], 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

[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`

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

The reason to use neural function approximators instead of a simple Q-table approach for reinforcement learning problems is easy to see: 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. 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 observe 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. It is no surprise that the success of *Deep-Q-Networks* started its journey shortly after the introduction of CNNs, which are able to learn abstract representations of similar images and by now used in countless Computer Vision Applications.

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. These weights correspond to the parameters of the $\hat{Q}$-function of the agent. As previously mentioned, this kind of Q-learning defines its policy straight-forward, depending on the $argmax_a$ of the Q-function. I will therefore replace the dependence of $\hat{Q}^{\pi}(s,a)$ on π by a dependence on its parameters: $\hat{Q}(s,a;\theta_i)$. 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: instead of giving both the state s and the action a as input to the network, in DQNs only the state is input to the network, with the network returning 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.

While there are attempts to use Artificial Neural Networks for Q-learning as early as 1994[20], some key components of modern Deep-Q-Networks were missing, leading to satisfactory performance only in very limited settings. In standard online RL tasks, the update step minimizing the loss specified in 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 [10], the update using gradient descent is prone to feedback loops and thus oscillation in its result, thus never converging to an optimal Q^{π}-function.

It was not until *Deepmind*'s famous papers in 2013[16] and 2015[17], that those issues were successfully adressed. 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 only from the most recent experiences biases the policy towards those situations, limiting convergence of the Q-function. To adress 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[21], 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 (eq. 2.14), necessary for the learning step of every iteration. Intuitively speakin, 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. To lessen the risk of such feedback loops, the DQN algorithm introduced the use of a second network for calculating the loss: the *target network*. This is only periodically updated with the weights of the online network used for the policy, which reduces 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, using the loss as put forward in [17]:

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

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 dependend on a 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 [12]. This improves the stability of the algorithm even more.

As a pseudocode for the DQN-architecture is already stated in the corresponding paper [17], listing it again here would be superflous. Instead, I try to compare the pseudocode with the code of my actual implementation using Python and Tensorflow in appendix A.1. In the first two pages of the appendix, the necessary definitions of agent and network structure are introduced, before in page 78 there is the actual comparison between the pseudocode and its correspondences in the actual code, namely the `__init__`, `inference` and `q_learn`-functions of the model-class. Note that the blue lines of the pseudocode correspond to difference from the original DQN in favor of later improvements.

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_j it includes a maximization step over estimated action values $Q(s_j, a)$, 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 (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) and many other issues. Because Q-learning uses the maximum Q-value of state s_{j+1} in every estimate of the value of state s_j, only those estimates are propagated where the noise of the estimation is in a positive direction. Because of that, state s_{j-1} will have the accumulated upward noise from both state s_j and s_{j+1}, and so forth. This leads to unrealistically high action-values. While this would not constitute a problem if all Q-values would be uniformly overestimated, [7] showed that its very likely that the Q-value $Q(s_j, a)$ for only some actions a is overestimated – which changes the result of the $argmax_a$ operation and thus leads to biased policies. They also show that the drop in DQN performance correlates with this overestimation of actions.

The solution suggested by [7] is called *Double-Q-learning*. In its original definition without using Neural Networks as function approximators, a double-Q-learner learns two value functions in parallel, by letting each experience update only one of the two value functions at random. For each update then, one function is used to determine the greedy policy, while the other is used to determine the value of this policy[9]. The authors proved that the lower bound for the overestimation of action-value, $max_{a'} Q^\pi(s, a') - V^*(s)$, is > 0 for the standard Q-learning update rule, whereas it is 0 in the case of DoubleQ. Additionally they showed that these overestimations are indeed harmful by showing the superiority of a DoubleQ-learner in comparison with a normal Q-learner.

Deep-Double-Q-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

[9]*"In DoubleQ, we still use the greedy policy to select actions, however we evaluate how good it is with another set of weights"*. The intuition behind that is, that the probability of both value-functions always over-estimating the same actions is basically zero.

of the second value function. Although online network and target network are not fully decoupled, experiments showed that it provides a good candidate for independent action evaluation, without the need of additional functions. The technique of DDQN is thus to still use the online network to choose an action (evaluate the greedy policy according to the online network), but the target network to generate the target Q-value for that action (to estimate its value, instead of the max-operation in DQN). As shown by [7], 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 from equation 2.10 must thus be re-defined as

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

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 (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 can't stop the car from driving into the wall. As all actions will end in the episode will end by the car hitting the wall, all actions will have roughly the same Q-value.

The idea of the *dueling architecture*[28] for DQN is to learn the action-state-value 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 $\hat{Q}(s, a)$-function, the actions are not input to the network, but instead it outputs $|\mathcal{A}|$ Q-values, one for each action. A *Dueling Double Deep-Q-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 ($\hat{V_{s_t}}$) of a state s_t, irrespective of the action that can be taken in this state. The other stream is the advantage stream, which has as output $|\mathcal{A}|$ 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 can't 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 in 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, than 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):

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

Actually, during testing it turned out that normalizing the advantage not with respect to the best action, but with respect to the average of all actions, $\frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\theta^A)$ lead to better stability. By splitting up into separate advantage- and valuestream and combining the two streams in a fully-connected layer using the formula stated above, the authors archieved far better performance then their predecessor, DDQN, on the same dataset [28].

The reason to split into two streams is obvious: 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 [28], the difference is especially drastic in situations where many actions have similar outcomes.

In appendix A.1, I listed an exemplary source code of the implementation of an agent using python and tensorflow. The agent stands alone without an environment, and some crucial parts of it, like an implementation of its memory, are missing. However, all aspects mentioned in the previous sections are found in it, which is the algorithm of the actual Q-learning as described above and in [7], as well as the computation graph of the network using precisely the DDDQN-architecture as described above and in [28]. The last page of it (page 78) consists of a comparison of the Pseudocode of the DDDQN (as found in [17], with the changes from [7] and [12] incorporated in and marked blue), where 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 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. There is no need to approximate the value-function of states or actions explicitly. The quality of the policy is again measured by its performance as in equation 2.13.

Before continuing to explain PG techniques, it makes sense to define another measure of the quality of an action: The *advantage*. The advantage will be what PG methods optimize for, and it can be defined analogously to the value of a state in equation 2.1, with the difference that this time, as we do not explicitly model the value of following states, we can only calculate it 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 then 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, and 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 may sound unintuitive at first, because there is no specific loss function the ANN can optimize for. However in fact, using the gradient with respect to the advantage 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

"AC-algorithm is essentially a hybrid to combine the pg method and the value function together."

"technically lernt der critic mit sarsa"

The technique introduced in the previous sections is a direct adaption of the Q-learning algorithm [25][29], adapted for higher-level function approximators. Standard Q-learning is a kind of *generalized policy iteration*, where the policy evaluation and policy improvement happen in the same step. The algorithm learns via temporal differences the state-action value $Q^{\pi_{greedy}}(s, a)$ for the states it encountered with its current policy π. It then updates π to a soft version of that greedy policy: $\pi_{i+1}(s) := soft(argmax_a Q^{\pi_{greedy}}(s, a) \forall s \in \mathcal{S})$, where the $soft$-function ensures appropriate exploration. Learning the Q-function and the policy simultaneously is however generally limited: $argmax_a Q(\cdot, a)$ can only easily be found in settings where the action space $\mathcal{A}$ is finite and discrete, as it requires a global maximization over all possible actions. While discretizing the action space is possible, 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.

However in a lot of scenarios, the action space is not discrete, but continuous: $A \subseteq \mathbb{R}^n$. In such situations, the alternative is to move 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 again to a Bellman-function-approximator, using temporal differences to estimate the action-value $Q^\pi(s,a)$[10]. In contrast to the previous approach 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 necessesity 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 accordingly 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[11]. 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.

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 existance 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 *off-policy deterministic policy gradient theorem*, put forward in [23], 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.

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

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 position of the optima are however preserved, the term can be left out, making the approximation no less useful. Both claims are shown in [23].

[10]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.

[11]The original policy gradient, introduced in [26], 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.

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 (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[12]. Note that for the update of the policy parameters, only the gradient w.r.t. the weight of the actions as input of the Q-functions are relevant, not the trained weights of the approximator Q^w itself. Figure 2.1 visualizes the idea behind the algorithm graphically.

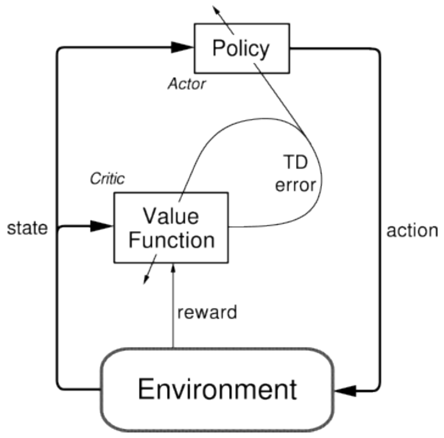


FIGURE 2.1: The actor-critic architecture. Reprinted from [19].

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

[12] *"The critic estimates the action-value function while the actor ascends the gradient of the action-value-function"* (quote [23])

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 [23], 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 continous action-spaces. The basic idea behind *Deep DPG* (**DDPG**)[12] is to combine 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_theta, 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 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 targetvalue, 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 – in fact, Lillicrap et. al. [12] 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 bases 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 snipplet of the agent's learning step, adopted from [12]:

Target-value of the critic:

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

Loss the critic minimizes:

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

Sampled policy gradient the actor maximizes for:

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

The correspondences of this ANN implementation and the correspondences of the general 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, I refer 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 missing. On page 81, there is again a comparison of the pseudocode (as provided in [12]) 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).

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. In supervised learning it is no problem to shuffle the dataset beforehand, giving the learner i.i.d. samples, which are certainly representatitive about the dataset. In RL however, this is not possible: Consider the situation where the agent drives a car around a track – as long as the agent doesn't drive the car well enough, it will probably not reach high speeds, and will thus learn nothing about situations in which it drives at high speeds. Another problem which was also mentioned before is that so far, only deterministic policies $\pi(s) = a$ were considered. It is obvious that in practice, purely using deterministic policies leads to a complete lack of *exploration* of the state space $\mathcal{S}$ of the MDP: Once the agent found one path 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.

The two algorithms considered here are both variants of Q-learning, which is an off-policy algorithm. In practice that means that the agent learned about some greedy, deterministic policy while following another policy, which I said was a *noisy* version of that greedy policy. A noisy version helps ensuring adequate exploration. The big advantage of off-policy algorithms is thus, that the problem of exploration can be treated independently of the learning algorithm – whatever the mechanism for exploration will be, it is easy to incorporate it in both of the explained algorithms by letting it implement the previously mentioned $soft(\pi)$ method, which takes as input a greedy, deterministic policy and yields a version of it that accounts for adequate state space exploration.

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 traning samples. Because of that, most exploration techniques use an exploration parameter ϵ, that decreases over time. This is called *simulated annealing*.

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

2.5.1 Exploration for discrete action-spaces

The easiest way 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 lateron exploit the greedy policy.

While epsilon-greedy is the most popular approach (also used in the original DQN), it has some obvious shortcomings: a purely random action has no way to incorporate any semantics, manchmal its ne folge die relevant ist,

temporal correlation against jerking movement situations in which all but one action is fatal takes into account either verybest or completelyrandom das an states, die man schon kennt, exploited werden soll und nur an unbekannten/grundsätzlich schlechten exploration schon den speziellen fall von autonomous driving, dass plötzlich volle möhre rechts sehr scheiße ist

–> boltzmann exploration legt ne wahrscheinlichkeitsverteilung über die eh schon ge-softmax-ten einzelnen actions -> incorporates also value of other actions. to anneal, a temperature tau (spread of the softmax) wird annealed over time. Damit das sinnvoll ist müssten die values confidence sein

In tabular settings, count-based exploration techniques are also 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 [3], 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 fulfil certain criteria, as being linearly increasing, generalizing across states and being roughly zero for novel events. 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 [14], is to compute 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 bellmann approximator for the psudocount – states that have less frequently observed features are deemed more uncertain. -darauf dann halt some exploration-bonus-algorithm (die... was macht? explored what suprises? -key problem is to compute appropirate similiarity measure efficiently -less computationally expensive than [3] -can easily be added to existing stuff (to value-based RL, can principally aded in any case as exploration bonus

– lower count = higher bonus) -...but needs feature maps used for liniar function approximation...use the same feature dimension to astimate value and uncertainty, near state-of-the-art performance -dass die aber die values der einzelnen componenten des vektors zählen... while it can be high-dimensional, it must be countable leider -their model assigns higher proability to state feature vectors that share features with visited states -man könnte also versuchen den zu diskretisieren, ... -in any case, density model über feature space: higher probab to states that share more features iwth more frequenclty observed states... this feature-visit density is calculated as the product of independent factor distributions over the vector bomponents –> count-based estimator if U is countable. Then they compute the pseudocount for from this density model accorting to [3].

Das ich das versuche, aber noch nicht so weit bin.

2.5.2 Exploration for continuous action-spaces

Switching from discrete to continuous action-spaces allows for new exploration techniques. In the 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:

$$\pi^{\theta}_{noisy}(s_t) = \pi^{\theta}(s_t) + \mathcal{N}$$

Selecting noise from this distribution is however still not optimal for self-driving cars or robotics: consecutive actions are selected independently on 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 [31], which selects the noise basing 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 [31] is, to provide a noise function ξ_t that has the same distribution for each t, with ξ_{t+1} dependant 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 dependant on the temporal discretization – the coefficient needs to be adjusted for different levels of temporal discretization. [31] provides extensive explanation how to select parameters appropriately. In general, a control policy basing 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 lead to the exploration function used in [12]. To incorporate autocorrelated noise, they used an Ornstein-Uhlenbeck process [27], which was originally created to model the velocity of particles with friction, which makes it very suitable to for physical processes with intertia. 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 variable reverts towards the desired mean μ. Additionally, it keeps a parameter σ, controlloling the volitility. Using this process, an action is calculated like the following:

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

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.

Chapter 3

Related work

[as mentioned before, there are two levels in the thesis - on the abstract level, the aim is to build a good agent for self-driving cars. On the practical level, it is still one specific simulation of a game, and one at first has to make the given game a RL problem, before you can eben talk about trying to solve that. To show how that is normally done, there is the first section. In the second section I will then talk about successful approaches of self-driving cars in reality. That is at first subdivided into real-life and games. Our game is (like so many others) supposed to work as a bridge between reallife and games. As for that, I will also talk about what data is normally available, ...

3.1 Reinforcement Learning Frameworks

In the previous chapter, I outlined the general structure of a reinforcement learning problem. In summary, it consists of agent and environment, where the environment is discretized into timesteps and only partially observed by the agent (**POMDP**). Each timestep, the agent gets an observation of the environment's state (making up its *internal state* and a scalar reward, and chooses an action to return to the environment. A graphical description of this interaction is depicted in figure 3.1.

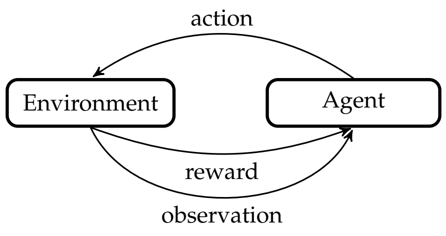


FIGURE 3.1: Interaction between agent and environment in RL

3.1.1 openAI gym

When developing RL agents, it is not enough to create algorithms, but also to have a specification of agent and environment that allows for a dataflow as described above. The original Deep-Q-Network [16] as well as its follow-ups [7][28] trained their agents on several ATARI games using the *Arcade Learning Environment* (**ALE**) [2], which was developed with the intention to evaluate the generality of several AI techniques, especially general game playing, reinforcement learning and planning. The important contribution of [2] was to provide a testbed for any kind of agent, by providing a simple common interface to more than a hundred different tasks – converting games into reinforcement learning problems. 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 severly limited, and most of the interesting real-world applications, as for example autonomous driving, require however 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 above mentioned environments are by now, among many others, merged into the *OpenAI gym*[1] environment [6], a toolkit helping reinforcement learning research by including a collection of benchmark problems with a common interface. Aside the previously mentioned ALE as well as a number of physics tasks using the MuJoCo physics-engine, the openAI gym contains the boardgame Go, two-dimensional continous control tasks (*Box2D games*), a 3D-shooter by the name of *Doom*, and several other tasks, varying in complexity, input and output.

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.

Algorithm 1 Interaction with the openAI gym environment

```
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
```

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 easy example, the action is performed by usage of the observation. Once an agent has calculated the action and provided it to the environment, it can perform another simulation step, returning $\langle observation, reward, done, info \rangle$, which is a tuple consisting of another observation of the environment's state, a scalar reward, the information if the episode is finished and it is time to reset the environment, as well as some information for debugging, typically not used in the final description of an agent. In the remainder of this work, I will refer to this dataflow as

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

a baseline on how the interaction of environment and agent should look like.

It is worth noting, that the openAI gym is not even

3.1.2 TORCS

TORCS is short for *The Open Source Race Car Simulator*[32, 33]. 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.

Ontop 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 provides a challenge on its own[2].

Another approach is given for the Simulated Car Racing Championship Competition, as presented in their manual[13]. In that framework, agents communicate over a UDP-server with the game-environment. To do so, the game functions as a server, that in an interval of $20ms$ sends an observation of the current game-state to connected agents. 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 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. A notable difference to the presented approach is however, that their implementation saw agents as clients, such that multiple agents can simultanously connect to the same game engine. In the developed approach here however, the intention is to use possibly multiple copies of an environment to train one agent, such that the focus lies on the agent, which will thus make up the server.

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 [13] contains a table providing detailed overview of the vectors sent to an agent (*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 divers state is necessary in semi-autonomous vehicles, the used approach does not consider a driver.

[2]This GitHub-repository (`https://github.com/ahoereth/ddpg`), implemented by a fellow student from the author of this work, provides an instruction how to install and use the TORCS-environment in python using openAI gym.

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 are unkown. 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 [9], 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 mehtod 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. In that paper, they give a clear definition of the motion-planning problem, evaluting the algorithm with the cost function of minimal time to complete the maneuver, showing that aggressive skidding maneuvers (*trail-braking*) as performed by humans are actually optimal in situations of loose surface with low friction.

3.2.1 Supervised learning

Knowing the full underlying physics is however nearly impossible, and even if it was known, optimal control is computationally very complex and unlikely to be incorporated in acutal 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 hand-crafting 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 however, no clear differentiation between feature extraction and controlling can be made as the semantics of the individual network layers remain largely unkown.

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[18]. Published as early as 1989, it used a three-layer neural network to directly learn steering commands. The input to the network consisted of a front-facing 30×32 pixel grayscale camera as well as a matrix of 8×32 values form a laser range finder. The steering-output it produces (it did not learn acceleration or brakes) is discretized into a smoothed one-hot vector of 45 units. The data used as input was artificially generated, it learned with a (self-defined) accuracy of 90% in simulations. In a real testing, it drove a real car for 400 meters at a speed of $1.8kph$.

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*[5]. In this approach, they used a convolutional neural network producing direct steering commands from a single front-facing camera. 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) on, producing the steering command as a single output neuron (hence continuous, but again no steering or brake). The network was trained supervisedly, minimizing the mean-squared error between the output and the command of the human driver saved in the dataset. To remove bias towards driving straight, the training data included a higher proprotion 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. If the simulated car drove too far off-road, an intervention was simulated setting the virtual car back to the ground truth of the corresponding frame of the original dataset. In this testing, 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]

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 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 log-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.

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-live. 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 [35]. In this paper, the authors propose a neural network that translates the image generated by a race gar simulation (specifically, they use the introduced TORCS engine) into a a realistic scene with similar structure, using a network architecture known as *SegNet* https://arxiv.org/pdf/1511.00561.pdf. Furthermore, they provide a self-driving agent which uses a discrete version of the *A3C*[15]-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 deemed as ground truh), a successful driving policy was learned that can adapt to real world driving data.

[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/

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.

However, first of all, 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 compoents include, but are not limited to: Radar, Visible-light-camera, LIDAR, infrared-camera, stereo vision, GPS or Audio.

Interesting is for example the *LIDAR* sensor, ultrakurzbeschreibung that can produce very high-level information of the surroundings of the car[4]. 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 https://arxiv.org/pdf/1511.00561.pdf 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. A lot of them use the metioned TORCS as their environment, while also some other, more or less realistic driving simulations are incorporated. The following table will summarize some of the known implementation. Note that this table also contains non-reinforcement training agents.

dass der torcs-keras die ddpg-reward funktion verbessert hat die O-U-werte vom torcs-keras-guy

wie wichtig der ornstein-uhlenbeck ist

dass man auch im video vom keras-torcs-guy sieht wie krass der shaked!

DDPG auf torc hat übrigens im pixel-case nen sehrsehr ählnichen punktestand wie im low-dimensional case (1840 vs 1876 im best case, -393 vs -401 im average case).

Project	trained on	model	input	output	optimization function
Nvidia Autopilot[5]	Annotated real-world data	TensorFlow-implementation of [5]	vision of front-facing camera	continuous steering-command	MSE to actual steering
TensorKart by Kevin Hughes[6]	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

[4]A video visualizing the data is available under https://www.youtube.com/watch?v=nXlqv_k4P8Q

[5]https://github.com/SullyChen/Autopilot-TensorFlow

[6]https://kevinhughes.ca/blog/tensor-kart

[7]https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html

[8]see https://www.youtube.com/watch?time_continue=4&v=4hoLGtnK_3U

Project	trained on	model	input	output	reward	performance
DDPG [12]	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 re able to complete a circuit around the track.* [12]
DDPG [12]	TORCS	DDPG	low-dimensional, similar to [13]	(throttle, brake, steer) $\subset \mathbb{R}^{n\in\mathbb{N}}$	velocity along the track direction, penalty of -1 for collisions	
DDPG-Keras-Torcs[7]	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 [13])	(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[8]
A3C [15]	TODO					

TABLE 3.2: RL approaches to learn autonomous driving

Chapter 4

Program Architecture

As explained in the introduction (1), 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. I will start 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. In the last section of this chapter (section **??**), I will explain what kind of data the game provides that can be used by an agent.

4.1 Characteristics and design decisions

As stated in chapter 3.1, the usual framework for solving reinforcement learning tasks is fairly rigid. However, the task of this work differed in some respects from the usual implementation of a reinforcement learning agent, which led to the necessity of differing from the usual framework in numerous situations. In the following sections, I will provide an overview of the difference between this project and other work in the reinforcement learning domain. Furthermore, I will discuss some of the design decisions that found their way into the final version of the program as well as challenges that occured during its implementation and 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

There are several differences from this project to others in the reinforcement learning community. The most important difference 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. The game which is supposed to be played 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 which is supposed to be played 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 is worth trying 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, the focus of this work was to learn this one game as well as possible – in other words 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 state-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. The natural questions arising are how good an agent relying purely on supervised training performs in comparison to others. Another important question is if there is a way to combine pre-training with reinforcement training in racing tasks, as famously done in the board game Go[22].

4.1.2 Characteristics of the game

The given game is a simple but realistic racing simulation. As of now, it consists of one car driving one track. While it possible to implement additional AI drivers or different tracks, no such thing is considered in the current implementation. The main focus of the given simulation lies on realistics physics, which make the game far harder to predict and masterthan many of those implemented for the `Atari`-console, on which the original `DQN` was tested.

The input expected from an agent consists of three continuous actions: The *throttle* increases the simulated motor-torque, which leads 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 intertia, 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.

It is the task of an agent to provide commands for the values of throttle, brake and steering, which are continous in their respective domains: $throttle \in [0, 1]$, $brake \in [0, 1]$ and $steer \in [-1, 1]$. Of course, 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. It should however also be kept in mind that the agent provides those actions *to the car*, and that the car must be seen as part of the environment as well, as the actions do not have a reliable impact on its behaviour. For instance, if the car is moving at fast paces, hitting the brake will not lead to an abrupt stop, as the car's inertia still applies a forward force. Also, the impact of the steering changes with the movement of the car. For example, the turning circle of the car has a much slower radius in smaller speeds. Another consequence of the realism of the physics implemented in this simulation is the fact that simultaneous braking and steering at high velocities has almost no effect: Because of the high speed, the car has a strong forward force. That leads in combination with the braking to the front tires slipping a lot, which reduces the impact on forces applied to them on the car – it will not follow the direction of the steering but continue its forward pace determined by the stronger force, namely the inertia.

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.

The game as a reinforcement learning problem

As detailed in chapter 2, reinforcement learning agents are solvers for (possibly only partially observed) Markov Decision Processes with unknown transition relations. In other words, for the agent to successfully learn how to operate in an unknown environment, this environment must correspond precisely to 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. [33, chapter 4] put forward for `TORCS`. As is the general case in simulations, while every update-step of the physics aims to simulate a continous 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 20 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], while the start-state is always equal. As 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: $\mathcal{S}_e \times \mathcal{A} \rightarrow \mathcal{S}_e$.

[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/`, 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.

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 also is 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 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 *POMDP*s itself contains no definition of final states, it is necessary to have use those, as they provide hard limit of the horizon in the calculation of Q-values. Another design decision in the course of the implementation of this project was, that the agent itself can define what it wants to see as the end of an episode. It is a matter of testing after how many seconds the agent shall give up on a trial, or if steering into a wall should be viewed as the end of a training episode. Because of that, the environment provides only candidates of what could terminate an episode to the agent, such that the agent can decide to reset the environment or not.

The game itself has three different game modes the user can choose between at the start of the game: In the `Drive` mode, users can manually drive the car with the computer's keyboard. If the `Train AI supervisedly` mode is activated, the car must still be driven manually, while the game exports information about the driving episode that can be used as (pre-) training data for artificial driving agents. The last mode is the `Drive AI` mode, where car can be controlled by an agent, as long as the User doesn't actively interfere. 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, such that several agents can each have their own respective definitions of certain features. 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.

It is due to this design decision that the program flow of this project cannot follow the exact same structure than the one put forward in algorithm 1. One big structural difference is for example that the outer loop (line 3) becomes obsolete, as the agent decides under what conditions the environment must be reset. A further advantage of an implementation like this is also, that it can be experimented with rewards that change over time, such that an agent can for example first learn how to move forward, and only later to always stay on 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 basic 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.

I will go into detail about the implemented features and vectors in a later section after describing the implementation of agent and environment, as the data relies on functions and data specific to this implementation.

4.2 Implementation

The associated programs are, as will be explicitly stated, written by the author of this work as well as 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 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 correspondenced 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 THE, SIGNED, COMMITS.

[2]`https://github.com/`

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

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

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

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 author of this work contributed most of the work to change the given game such that it can be learned and played by a machine, the original version of the game was given in advance, coded by the first supervisor of this work. 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 however not influenced by any other people than the author of this work.

As already hinted at, the program flow of the implementation differs from that of other implementations. 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.

4.2.1 The game as given

The program flow of the game is encapsuled by the framework that the `Unity 3D` provides: To ease the implemention of games, Unity provides numerous `game objects` with pre-implemented properties like friction or gravity, as well as drag-and-drop functionality to add 3D components or cameras to the Graphical User Interface (**GUI**). Another advantage of Unity is, that it targets many graphics APIs, which takes a lot of work from the programmer to implement an efficient graphics pipeline.

To implement additional behaviour or features not predefined by Unity, it allows for scripts, written in object-oriented *C#*. Scripts that are supposed to be used by Unity must provide a class that extends[10] its class `MonoBehaviour` or be used by such a one. For the file to be instanciated during runtime, it must be specified in Unity's `Object Hierarchy`. After staring the program, Unity will create all objects specified in the hierarchy. To enable the possibility of instanciations `knowing` each other during runtime, they must provide public variables of the type of the respective subclass

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

[7]As of now, 11th 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

[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, and references).

of `MonoBehaviour`, which can via drag-and-drop be assigned to the respective future instance specified in the Object 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 an implemented class, to which the footnote in the headline refers. In appendix C, I also provide an informal table of each class and its important methods. Keep in mind that the structure of this files as well as most of their content is not created by the author of this thesis. Note also, that I will sometimes mention optinal features. 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, the game has three different game modes that can be active: `Drive`, `Drive AI` and `train AI supervisedly`. These are the three modes the User can choose from in the game's main menu (the UI of which 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 (`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

[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`

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 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 to it get time to catch up with their calculations. For that, the `QuickPause(string reason)`-function sets the Game'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 the public function `QuickPause()` could in principle be called from every method of the game, it contains functionalities that are (due to design concepts of Unity) not thread safe, meaning that only the main-thread can call them safely. To enable the possibility of asynchronous threads activating the QuickPause-mode, `Game` contains a public variable `shouldQuickpauseReason`. This variable can be set to a value from asynchronous threads, and the `Game` checks every `Update()`-step (guaranteed to be main-thread and to also be called if QuickPause is active) if another side thread requested to invoke this method. Once the method that originally invoked QuickPause wants the game to continue its normal process, it must call `UnQuickPause(string reason)` (or request for it to be called). However, QuickPause could have been enabled by another method as well, that is not ready for the normal game process yet. To prevent such a scenario, the methods to start and to end QuickPause must both be called with a string `reason`. In every call of `QuickPause(string reason)`, the function pushes `reason` on `Game`'s List `FreezeReasons`, and in every call of `UnQuickPause(string reason)`, it removes the respective reason from the list and only continues the normal game process if the list becomes empty.

User Interface[14]

The job of the Game's `UI` (an instance of type UIScript) is to update the user interface. This UI is overlayed over the current scene, such that its components can be seen simultaneously to the image of an active camera. 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,

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

respectively. The latter of those screenshots is annotated with labelled boxes around each UI component, with page 83 explaining each component a bit further[15].

Controls

If `Game.mode.Contains("keyboarddriving")`, the game is steered with the arrow keys ← and →. The throttle is triggered via the A-key, whereas the brake is called via Y. The R - 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 H-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 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 to the agent). This can be done with the number keys, where the pretended speed is evenly spread between $0\ kph$ (0) and $250\ kph$ (9). The P key is reserved to simulate a full throttle value. To hand control back to the agent, H 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 D and C, respectively. During any `mode` containing `driving`, Q can be pressed to activate the `QuickPause`-mode, which allows to spectate the current screen. QuickPause is ended with another hit of Q.

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 Unity `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` 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 it is checked 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`.

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

[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 .

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

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. Because the car must stand still after a reset, it is necessary to completely kill its inertia. To do so, it is not enough to apply zero motorTorque and infinite brakeTorque to the car and call `car.ResetInertiaTensor()` inside `ResetToPosition`, because those values would simply be overwritten in the next call of `FixedUpdate()`. This is why in the given implementation the function `ResetToPosition` sets a boolean variable `justrespawned` to `true`. In every call of `CarController.FixedUpdate()`, it is checked if the car did just reset, and performs the necessary forces to remove all of the car's inertia right there, before setting `justrespawned = false`.

Position tracking[17]

To successfully learn useful 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 the it. To allow for that, the game incorporates a `TrackingSystem`, which is an instance of the class `PositionTracking`). 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 was not implemented by the author of this thesis, a short scetch of how it can be used shall suffice.

To calulate the progress of the car in percent, one needs the total length of the track as well as the distance the car advanced so far. The first of those can be calculated by summing up the distances between a coordinate and its successor. To get the approximate distance the car advanced, one needs iterate through the `anchorVector`-array to find the one closest to the car (which happens in `GetClosestAnchor(Vector3 position)`). The cumulated distance of successive vectors until the one closest to the car corresponds roughly to its progress in percent.

As every successive coordinate in `anchorVector` is in the middle of the street, one can calculate the direction of the street at position `p` 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 distance to the center of the street can be found via by calculating the norm of the orthogonal projection from the car's coordinate onto this vector. 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 depending on it in its `Start()`-method, the `TrackingSystem` calculates the car's current `progress` at every `Update()`-step and triggers the `UpdateList()`-method of the `RecordingSystem` in regular progress-intervals. Furthermore, the `TrackingSystem` provides certain public methods that can be used by an agent, namely `getCarAngle()`, `GetSpeedInDir()` and `GetCenterDist()`, the precise content of which will be explained in a later section.

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

Tracking time[18]

As visible in the game's 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. Futhermore the game provides a visual feedback basing on the time needed for a specific section of the street in comparison to the time that was needed for this section in the fastest lap so far (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.

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 hit 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 The 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 lateron.

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

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 sucessful 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 the car. 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 further away from 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 on Unity `Camera`-objects, which are efficiently calculated on the computer's GPU. Attached to each camera is a respective instance of `MiniMapScript`. 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 form 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.

Recording training data[22]

For the game to be played by an AI agent, 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 reinforcement learning, or to be exported to a file, which an agent can perform pretraining on. In the following sections, I will not talk about what those data exactly looks like, but only how it is saved and sent. 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.

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

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

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 ();
}
```

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 dependant 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 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 the RecordingSystem's `FinishList` function 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 minimapcameras, this may however take quite long. To prevent the game from freezing everytime the car passes the start/finish line, the saving of 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` 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].

[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.

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. In most use cases, sockets are used to communicate over the internet, which is why they normally abstract data over several layers, before being sent to another machine. When both sockets are on the same machine and communicate over localhost however, the unnecessary 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. One characteristic when using sockets as means of communication is, that the communicating sockets need to know the length of a message, to know when to stop reading the input stream. To do that, the length of each message will be transmitted with it.

In this project it was decided that the game functions as a client, whereas the server is written in python. This fits the scheme of the implementation, where the 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).

If the `Drive AI` mode was selected in the game's `menu`, the `AiInt` (an instance of the class AiInterface[12]) calls its function `StartedAIMode()`, which, next to calling the known `PrepareVision` of the cameras, creates two instances of `AsynchronousClient`, a class defined in `AsyncClient.cs`[27]

the function `LoadAndSendToPython`.

dass ich dass wie das wryman-paper mit robotern mache, dass unity dafür zuständig ist zu interpolier

The used setting is comparable to [31]. 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 (->unity) tries to make the joints follow them

-warum ich senden und empfangen mit nem DIFFERENT socket mache (dann ists in 2 threads und kann asyncrhon sein aka nichta ufgehalten werden) -dass es entweder raw data oder special commands schickt

-nochmal drauf eingehen dass das spiel live ist -das von leon gemalte ablaufdiagramm dass der server alle 0.1 sekunden was von unity bekommt! dass der client mitbekommt wenn ne

dass das auto selbst environment ist ist auch so in [31] (bei robots)

[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

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

4.2.3 The agent[28]

As already explained, the implementation at hand differs in its program flow from other implementations, as 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. Upon executing the server, an instance `inputval` of 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. The next section will detail how the server initiates threads needed for the communication with the environment, before leading over to the main loop of the learning process.

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 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 `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.

[28] I will rely heavily 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. In the main text, only small class-diagrams accompany the explanation of the code, with respective complete versions in appendix ??.

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

[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.

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 ListenerThread 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 ListenerThreads are always active. If for some reason the old `receiver_thread` or `sender_thread` loses its connection, the ListenerThreads 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 other threads. 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 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 CTRL+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, more on that later).

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 his 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

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

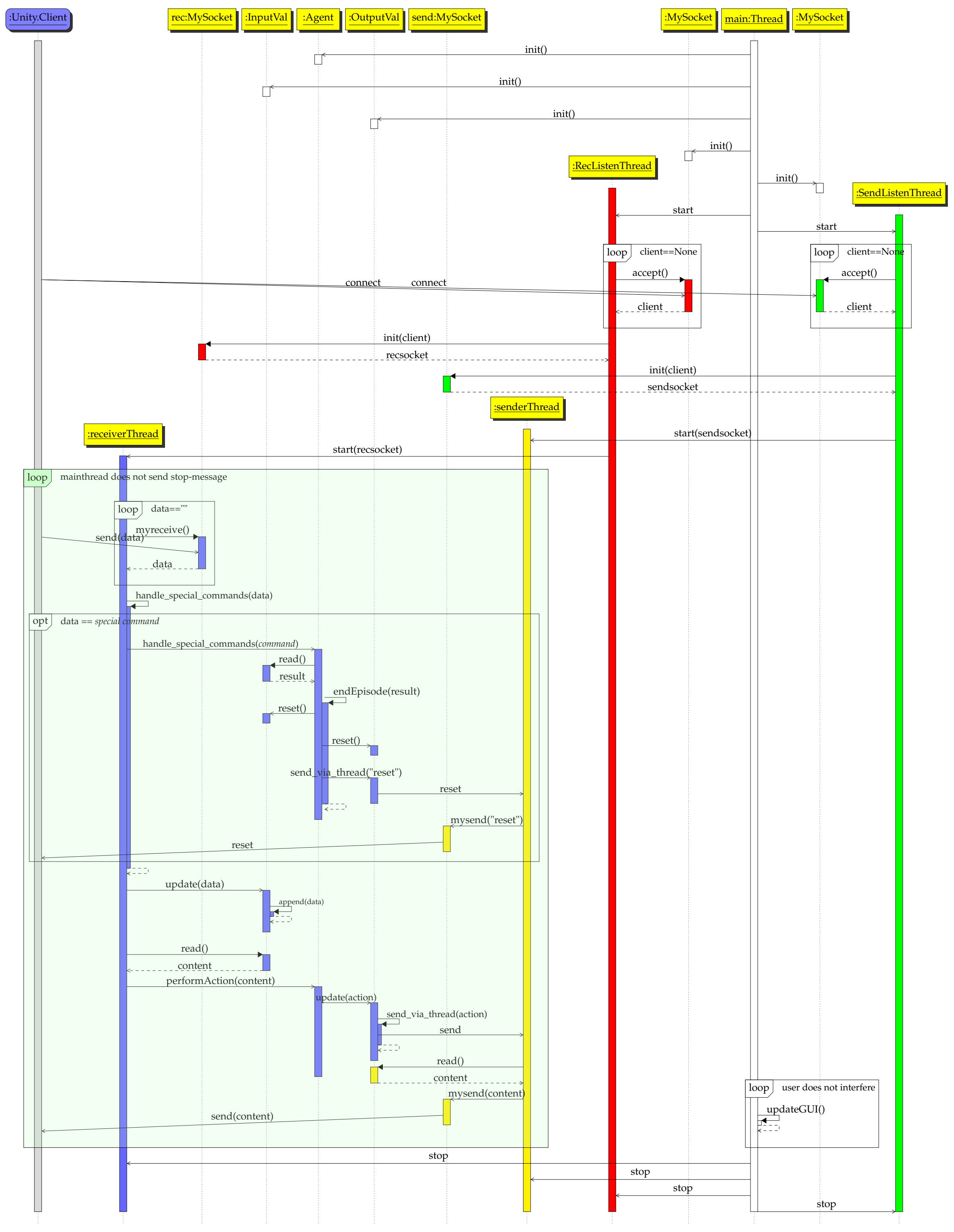


FIGURE 4.1: Sequence Diagram of the Server

Reading instructions: Y-axis is time. Columns with a full colored bar are threads, other objects 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".

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].

[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

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 (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`), otherinputs (`otherinput_hist`) as well as an array for the history of performed actions (`action_hist`). 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 computational load, the `inputval` does not contain the entire history of the game. As already explained, it contains a superset of the observations of the actual agent, which likely contains much superflous information. 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=False)`-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 in 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 reset (by sending `"pleasereset"`). The function `resetUnityAndServer` combines sending the reset-command to Unity and resetting `inputval` and `outputval`.

data, such that the inference of the agent and a waiting to receive new data can occur simulatenously. 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

Agent-independent features

All of the above explained functionality is necessary for the agent to learn 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 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 `Consts`, 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 in what interval the agent performs actions, must 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 basically all threads and objects, such that they can access all their settings clearly.

Agents

dass der AbstractAgent noch containers KENNT und darauf zugreifen kann, das aber bis zu den specific

Until it gets to 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`, in contrast to other approaches. Furthermore, the server notifies the agent of specific events that happened in the game, to which the agent could react by resetting the environment and starting a new episode. These methods, next to the obvious ones of performing an action or conducting a learning-step, must thus be specified by an agent. When describing the agent in this section, only those functions are relevant, as 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 mutural 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 a purely pre-trained agent does not need it.

In the scope of this thesis, multiple agents where developed, using different models, vectors and approaches. In the final version of the implementation there are five

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

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

exemplary agents left, which can all be found in the folder `agents/`[37]. These agents differ in the model they use, if they use the vision-vector and if they rely on supervised pretraining, as will be detailed in chapter 6. 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. A more complete version can be seen in appendix ??

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 D. 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`.

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 the agent, performing an action, reacting to (reset-) commands and performing pretraining.

As can be seen in the diagram, some methods require a `gameState` (or `pastState`) as argument, whereas others require an `agentState`. Inside the implemention, they are both clearly defined. A `gameState` is what is provided by the server, 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 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 only 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. I will not provide the internals of the function to create the agent's observation (`agentState`) from the `gameState`, as it depends on the agent's `features` and is a result of experimentation, as such described in chapter 5.

Next to `getAgentState(gameState)`, there are several 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 action, which wraps such a function from the file `read_supervised.py`. The function `makeInferenceUsable(state)` combines those functions for certain kinds of states,

[37]https://github.com/cstenkamp/BA-rAIce-ANN/tree/master/agents

such that it is the only one needing to be called whenever the agent accesses its `model`.

Methods called by the server In its `__init__`, called when `server` creates the `agent`, the classes `AbstractAgent` and `AbstractRLAgent` define some assumptions about about the configuration of its `features`. These variables can be overwritten by its subclasses if their actual configeration differs, such that these variables are reliable information about the agent's configuration of `features`. When initializing its memory or its model, the agent also passes a reference to itself, such that the 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. This can be inspected by running the file FILENAME123, whichproduces the plots If the application is not run with the argument `-noplot`, the evaluator also plots these running averages live, using the library `matplotlib`. 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 $\boxed{H}$. An exemplary 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 by agents in the general case. 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[17]), 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` with a probability ϵ, where epsilon decreases over time. In agents where it is desired to overwrite the 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 `policyAction`-function. `performAction(gameState, pastState)` cares 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`.

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 explicitly giving negative reward for the last seen state using its function `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

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

not nested in an outer loop that recognizes when an epsiode end, 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.

As especially `calculateReward(gameState)`, `randomAction(agentState)`, `handle_commands(command)` and `policyAction(agentState)` are considered `features` of the agent, they are explained in chapter 5.

An agent using a DDPG- or DQN-model requires a replay memory, such that its ANN can learn from uncorellated samples. The function `addToMemory(gameState, pastState)` adds an agent's state to its memory, and the function `learnANN` accesses its values, converted into a usable shape by `create_QLearnInputs_from_MemoryBatch`.

Learning It is possible to use the `"parallel"` learn-mode or the `"between"` learn-mode (as specified in `conf`). If the learning mode is 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[17]. 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 does it??, 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 `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 (started in its main-method). 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 itsfunctions 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 to so, the reader is for example referred to `agents/ddpg_rl_agent.py`[39]. 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. Following the tradition of the DQN[17], 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`[40] or `efficientmemory.py`[41]. 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. Adding to its memory is the only situation where the agent's function `calculateReward(gameState)` is called. The memory 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`[21] increases the agent's speed of learning, but while the implementation allows for such an addition, nothing comparable is done in the actual implementation.

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 containing 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

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

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

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

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 in fact 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 in fact new. It is easy to see, that 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[42]. 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 inefficientmemory is loaded by default.

Pretraining

As stated in section 4.2.2, not only the possibility for reinforcement learning is given in this implementation, but the game also contains functionalities to record manual driving data and export those via XML. By running the file `pretrain.py`[43] with the respective agent as command-line parameter (see server), 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 data the game exports for pretraining consists of the `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` 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.

Upon execution of `pretrain.py`, the dataset `trackinpoints`, 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 an corresponding action from the same point in time. However, all actions from the server will be delayed due to the time of sending the state, deciding on an action and executing it. Because of that, the function `condider_delay` remaps

[42]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

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

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 function. 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.

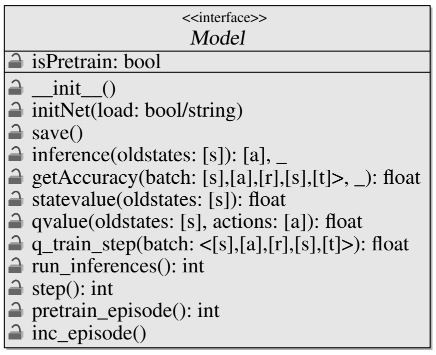
In the actual implementation, the `AbstractRLAgent`'s `preTrain`-method is implemented a bit differently, the agent cannot learn useful policies with q-training on an existing dataset created by a completely different policy.

dass pretrian.py auch ne fake_real (aka vom memory) methode hat

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`[44] for example simply assigns random values as its actions.

An agent however that extends `AbstractRLAgent` must provide a model that implements the (because of python's duck typing only hypothetical) interface 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.



<<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

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

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. In the `isPretrain`-mode, the model cannot perform inferences and must therefore be re-created with out the `isPretrain`-flag. If the model is initialized by the server, the flag is set to false, whereas if the model is initialized by the application `pretrain.py`, it is set to true.

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

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 basic 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 respective section.
- 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 discretized, an agent must overwrite the method `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 oferwrite 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`.

In the next chapter, I will provide the implementation of the agent's `features`. For that, I will start with `possible vectors` that are sent to any agent and can be used. Afterwards, I will explain the agents that where implemented in the scope of this thesis. I will explain the `features` of all agents, including why they where selected that 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`. This is however only done for convenience, such that these methods do not have to be implemented the same way several times.

nochmal auf den program flow im vergleich mit dem DQN-pseudocode im anahng eingehen; UNBED

Chapter 5

Results - Implementation

blablabla dass die das ergebnis von –langem– rumprobieren sind

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 (which is the case if the car did not leave the street yet).

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
velocityOfPerpendiculars	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 for 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, as calculated in where (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 vector consisting of 30 elements, 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 closest `anchorVector`s of 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.

Mini-maps While the content of the mini-maps is not contained in the namedtuple `Otherinput`, 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, as well as many elements necessary to compute many of the other ones. 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] Both models learn via temporal differences and are in detail 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 uninformed ϵ-greedy remains possible.

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 in this mode as well, to reduce the dimensionality of discretization. Further, simultaneous activation of throttle and brake was forbidden. Because of these design choices, an action 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. If an agent does not need to discretize actions, it must overwrite the method `getAction(gameState)`.

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

[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

accessible in the same way. An UML-diagram of most of their functions as well as the interface is given in figure 5.1.

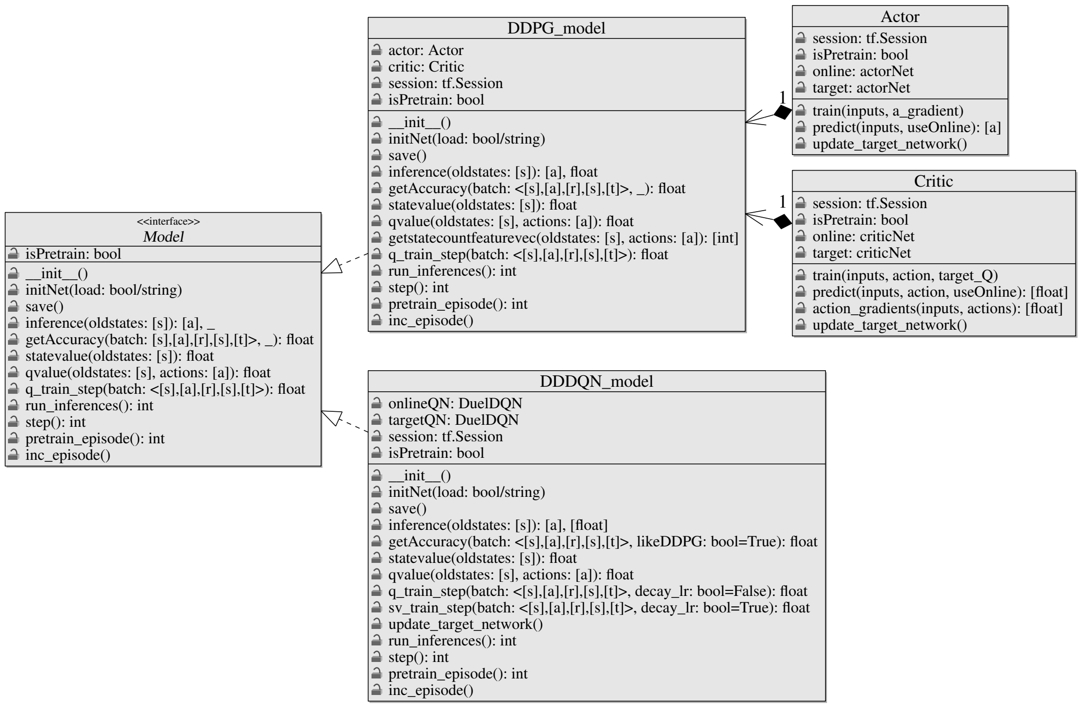

DDPG_model
actor: Actor
critic: Critic
session: tf.Session
isPretrain: bool
__init__()
initNet(load: bool/string)
save()
inference(oldstates: [s]): [a], float
getAccuracy(batch: <[s],[a],[r],[s],[t]>, _): float
statevalue(oldstates: [s]): float
qvalue(oldstates: [s], actions: [a]): float
getstatecountfeaturevec(oldstates: [s], actions: [a]): [int]
q_train_step(batch: <[s],[a],[r],[s],[t]>): float
run_inferences(): int
step(): int
pretrain_episode(): int
inc_episode()
Actor
session: tf.Session
isPretrain: bool
online: actorNet
target: actorNet
train(inputs, a_gradient)
predict(inputs, useOnline): [a]
update_target_network()
1
Critic
session: tf.Session
isPretrain: bool
online: criticNet
target: criticNet
train(inputs, action, target_Q)
predict(inputs, action, useOnline): [float]
action_gradients(inputs, actions): [float]
update_target_network()
1
<<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()
DDDQN_model
onlineQN: DuelDQN
targetQN: DuelDQN
session: tf.Session
isPretrain: bool
__init__()
initNet(load: bool/string)
save()
inference(oldstates: [s]): [a], [float]
getAccuracy(batch: <[s],[a],[r],[s],[t]>, likeDDPG: bool=True): float
statevalue(oldstates: [s]): float
qvalue(oldstates: [s], actions: [a]): float
q_train_step(batch: <[s],[a],[r],[s],[t]>, decay_lr: bool=False): float
sv_train_step(batch: <[s],[a],[r],[s],[t]>, decay_lr: bool=True): float
update_target_network()
run_inferences(): int
step(): int
pretrain_episode(): int
inc_episode()

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

Both implemented models provide the possibility to load and save a model from/to the harddisk unsing 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 TensorFlow 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` else. 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 the described agent 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 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

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

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]. https://gym.openai.com/envs/CarRacing-v0

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 just differ from agent to agent, and the presented network architecture is specific to one of the implemented agents, which will be explained in a following section.

5.2.1 DQN-model

decaying LR is implemented but should only be used in svtrain

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 then 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 also 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 the next chapter.

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 has consistently as `stands_input`, which expects 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 is `true` and the respective option `conf.use_settozero` is set to `true`, then the model sets in the function `settozero` all q-values of actions that do not include accelerating the car to a large negative value. 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 [12], the implementation of

[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.

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 store their collection of 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 therefore 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.

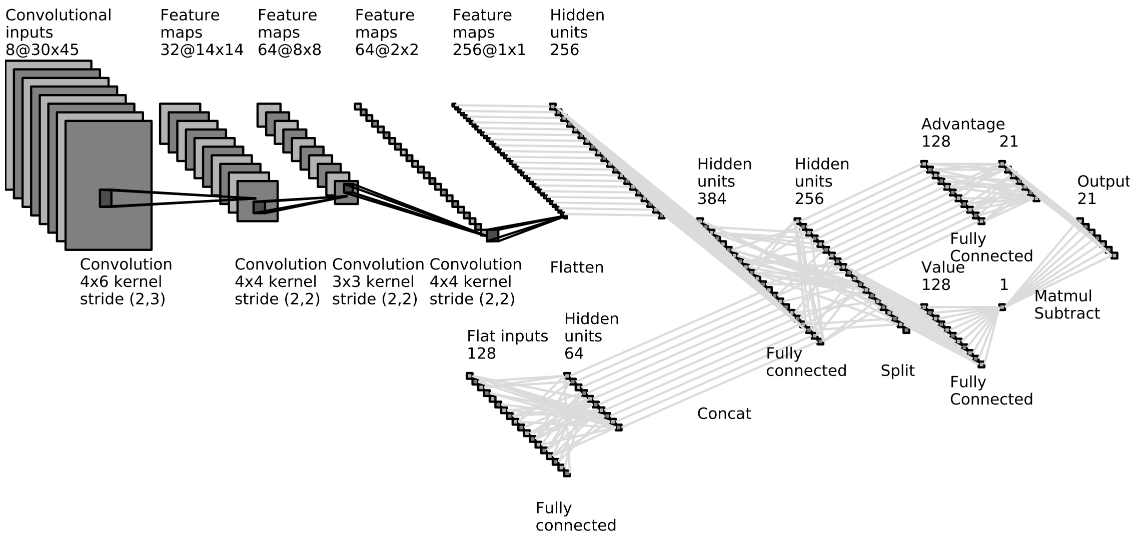


FIGURE 5.2: The used convolutional DQN with a dueling architecture

If convolutional input is specified, four convolutional layers layers processes 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 layer in between convolutional layers. As suggested by [24], 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, the `ff_inputs` are connected with a dense layer 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 [28], 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 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[11] 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 at start.

In the current implementation, the DQN-model does not perform `batch normalization`[8], as it showed to decresed 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 the subject of much experimentation. Any used hyperparameter that does not correspond to its counterpart in [17] or [28] 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 incorportate four ANNs to work correctly: an online- and a target version of both `actor` and `critic` , as found by [12]. While the online networks will be used for 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 sue their online- or target network. To update the target network, both actor and critic provide respective methods.

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 a 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 and appendix A.2 compares the source-code to the Pseudocode given in [12]. It is possible in TensorFlow to extract the `gradients` of a complete graph of computations, namely the critic. 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.

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

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, I will describe the network structure of both of them 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 it 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` else.

The critic gets as input a state and an action, and returns an estimate of the respective state-action value `Q`. In contrast to the model of a DQN, it gets both of these as input and returns a single value. 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, as explained above), 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.

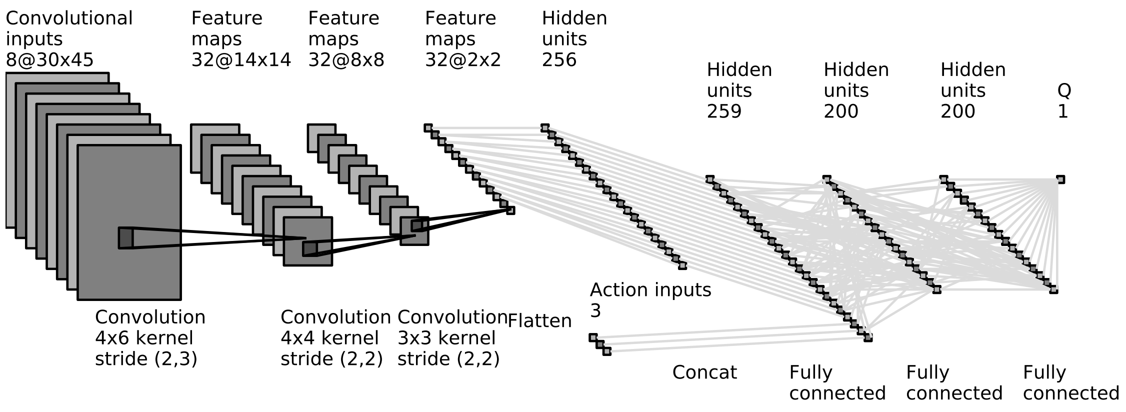


FIGURE 5.3: Convolutional critic

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.

The network-architecture of the low-dimensional critic (`lowdim_criticNet`) can be seen in figure 5.4.

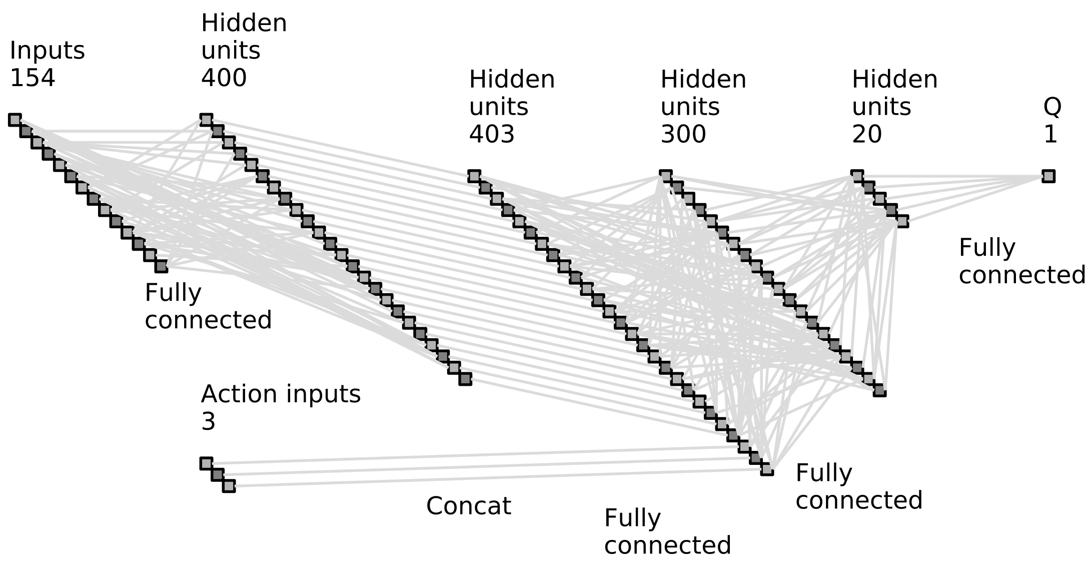


FIGURE 5.4: Low-dimensional critic

The number of input-neurons for the low-dimensional critic is variable depending on the agent – for the `ddpg_novison_rl_agent` it is 154 input neurons. 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 specifiying the Q-value.

As suggested by [12], both variants of the critic use $l2$ weight decay for all their layers. Furthermore, the implementation allows to use *batch normalization*[8], 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 an additional layer and an additional input that specifies the phase (learning step/non-learning step)[6]. Though [12] report success on using batch normalization , in practice it often leads to instability. After much informal testing 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_all` steps.

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

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 take the actions as input, but connects the last hidden layer to

[6]cf. https://www.tensorflow.org/api_docs/python/tf/contrib/layers/batch_norm

three output neurons, corresponding to the values for the three actions. This *Outs*-layer is afterwards scaled, 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.

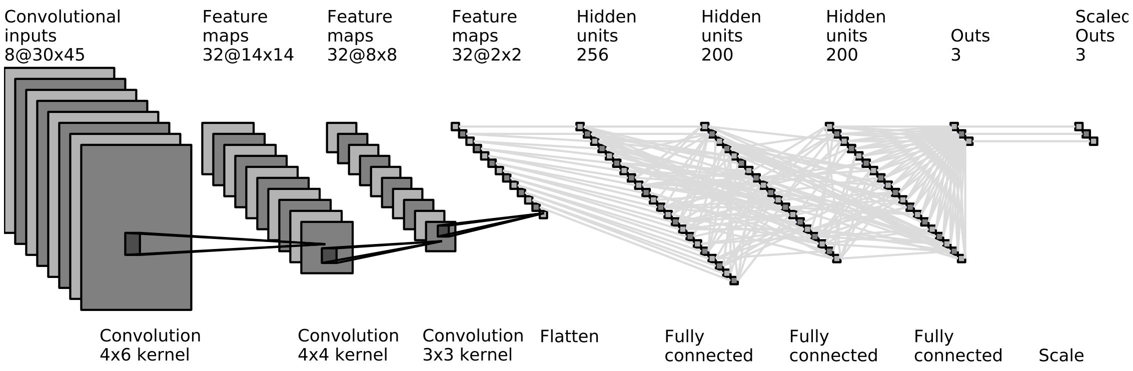


FIGURE 5.5: Convolutional actor

The structure of the low-dimensional critic (figure 5.6) is fairly easy. 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.

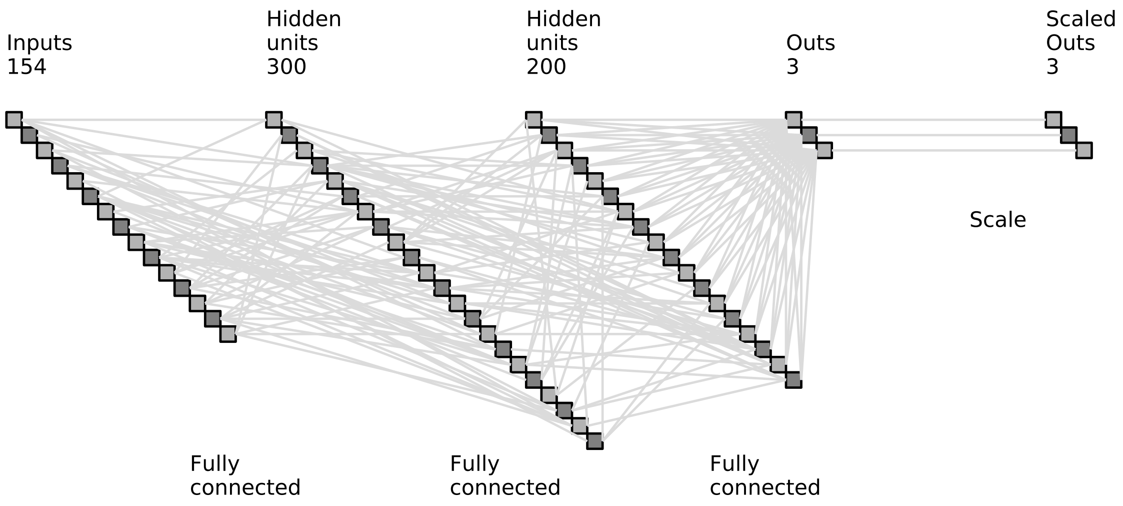


FIGURE 5.6: Low-dimensional actor

In contrast to the critic, the actor does not use $l2$ normalization, as suggested by [12]. Informal testing showed, that batch normalization indeed helps convergence in the actor, which is why it is enabled for all fully-connected layers. The reason why the critic has pre-terminal layer of 20 hidden units will be elaboarated on in the next section. Besides this layer, all hyperparameters correspond roughly to [12], with all difference being the result of informal testing. As in the DQN-model, the rectified non-linearity (`tf.nn.relu`) serves as activation function and Adam[11] as optimizer. The convolutional layers are implemented using `conv2d` from the Tensor-Flow's `slim`. 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 than in that approach, as it is not necessary to change the Q-value of actions, but one can instead change the output of the actor. If the corresponding variables `conf.use_settozero` and `stands_input` are given, 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 sets a random of those to a random value below 0.5.

Agents hat 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 q-learning. 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 change the implemented `randomAction(gameState)`.

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 (not however that it is incomplete and for example does not show the functions that are overwritten to incorporate a GUI).

In the following sections, these functions will be described. Further, the used reward function will be elucidated, even though it was the same for all agents.

[7] https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/ddpg_novision_rl_agent.py
[8] https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/ddpg_rl_agent.py
[9] https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/dqn_novision_rl_agent.py
[10] https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/dqn_rl_agent.py
[11] https://github.com/cstenkamp/BA-rAIce-ANN/blob/master/agents/dqn_sv_agent.py

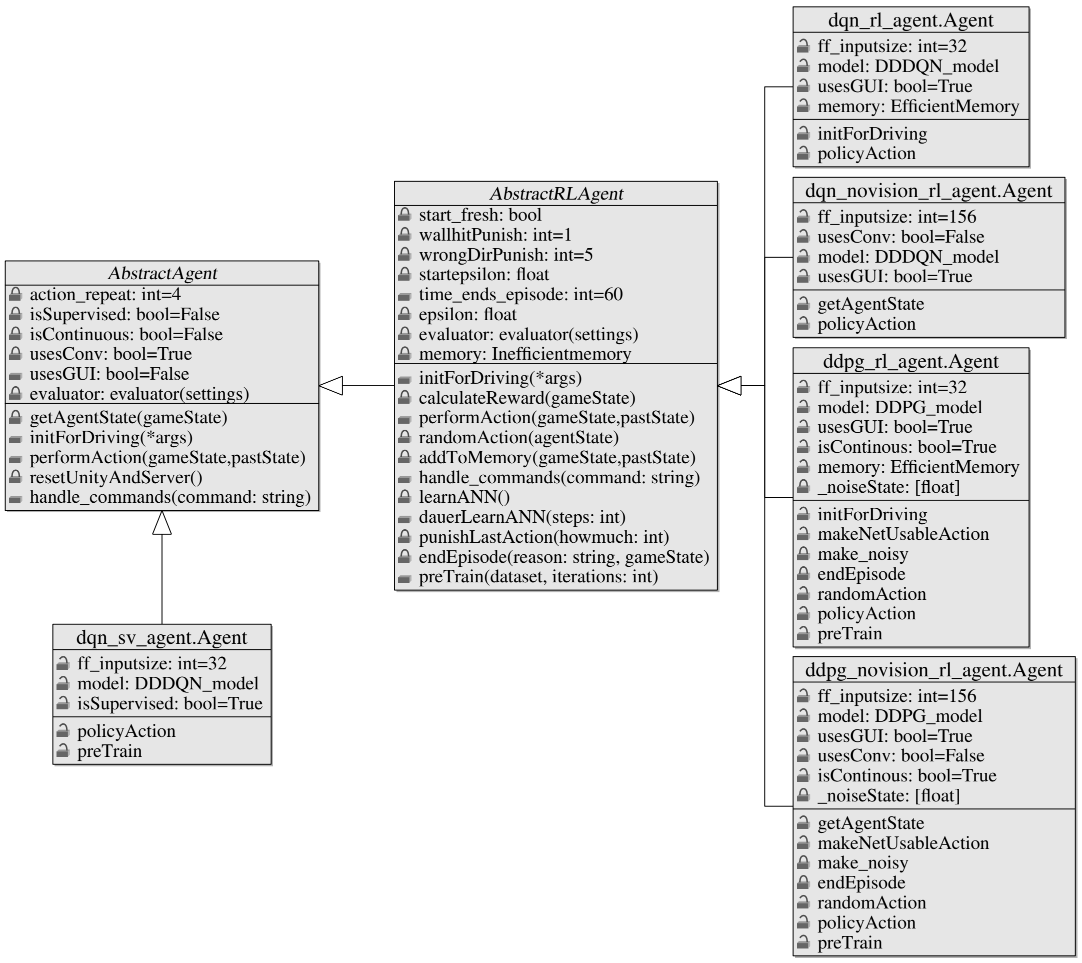


FIGURE 5.7: (incomplete) UML-Diagram of the five implemented agents and their abstract superclasses

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 means, that it is guaranteed 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.

Using only good datapoints is perfectly suited for supervised training, where the agent lerns 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, `AbstractRLAgent`.

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 is unknown. It is easy to see that 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 for the environment.

dass getAcurracy das flag likeDDPG hat

5.3.2 Exploration

das mit high-level information in der randomAction-funktion eigentlich ne wahrscheinlichkeitsmeasure über policyAction gemacht werden kann -dass man ähnliche actions mit ner höheren wahrscheinlichkeit nehmen könnte, boltzmann, oder einfach eine normale discretisieren und noise drauf packen könnte

-die expliziten values für den ornstein-uhlenbeck

das experimentelle getstatecountfeaturevec, das aber noch nix tun will

Only very basic exploration algorithms are provided in this thesis. There are two different methods used in the DQN-based agents compared to the agents with a DDPG-model, following their definition of an action. While in DDPG-based agents it holds that $action \subset \mathbb{R}^{n \in \mathbb{N}}$, in DQN-based agent the action is discretized into one unique value.

ALEX: With Deep Q-Networks the most common policy during training is ϵ-greedy. ϵ-greedy means, that for an ϵ portion of all actions a random action is chosen, otherwise the action with the highest Q value is chosen greedily. With continuous actions thats not an option. The DDPG actor at each timestep outputs an action vector of continuous values. For exploration the straight forward approach is to add some noise to those actions. Because we model some physical environment just adding noise would make the agents behavior very jumpy – therefore Lillicrap et al. instead propose to sample from a continuous Ornstein-Uhlenbeck process. Such a process models the continuous behavior of a physical particle, with some friction θ and some diffusion σ.

While the action-value

-epsilon-greedy als standard und make$_{n}oisyvomDDPG$

5.3.3 Reward

-meine theorien: continuous, non-negative, aus ner shclehcten darf nicht besser sein als in einem guten eh sein) -dass immer speed belohnen negativ ist, siehe ->problem bei ddpg, approach von dem kerastorcsguy, siehe kapitel Daskapitel -die funktion wo man sich den durchgängingen reward in ner farbe plotten kann (auch bei H)

5.3.4 Observation

-dass ne runde nach 60 sekunden/wallhit/lapdone endet

Chapter 6

Analysis, Results and open Questions

dazushcreiben dass das mit skript xy gemacht wurde, und die daten dazu online unterxyz zu finden sind

FIGURE 6.1: Performance of the dqn_novison_rl_agent

-dass alle cars useful driving policies haben: q-werte vor wänden gering, steeren von wänden weg, fahren widely geradeaus... nur lernen sie halt das vollgas voll super ist

-mit maxspeed klappts -gym_test klasse existiert, models klappen damit -dataset is separable as can be tested by dqn$_{s}v_{a}gent - dasheadstartscheinthelfen - vonsv -$ $trainingkannmannichtq-trainingweitermachen-meinereward-funktiondiebremsenbelohnt-$ $dertwistimpretraining$

-dass SV-pretraining nicht als grundlage für Q-training genutzt werden kann -dass auch Q-training sucks wenn nicht die make$_{t}rainbatchfunktion-dass, wennmanactionaslinputgibt, er$ $o-wieschnellhumansfahren-wieshcnellnenranodmagentfhrt, haha-diebeidenoffensichtlichenlocalma$ $dassnensupervisedagentnurmitmaxspeedklappt$

ok, also wenn du wirklich gar nichts hast zum vergleichen dann schreib auf was du hast, und was genau die nächsten schritte (also vergleiche etc.) wären, evtl. was du ungefähr erwarten würdest das rauskommt und warum. mach dabei klar was eine relativ gesicherte erkenntnis ist und was eher ne vermutung ist

es ist kein problem, dass du nicht fertig geworden bist. geh also mit dem fakt selbstbewusst um -dass sich herausgestellt hat das simultaneous inference and learning a lot slower ist!

-it is obvious, that DQN cannot be as good as DDPG - The wheel must be turned 50 (or more) to avoid the obstacle. A discrete action space such as Left/Right might not be enough of a turn to survive. Discretizing the wheel into 5 increments might work, but leads to a combinatorial explosion, as well as not allowing all the degrees of turn in between. -dass dqn zwar super nachfahren kann, aber von da hart -dass dqn-models auch leider nicht als base für ddpg taugen, weil sie halt nen output für jede action haben

die performance vom server

-left to do: no feed$_{d}ict, multipleagents, game2malsochnell, multiplegames, -dieneuestenopenai-$ $algorithms, ...$

-dank sockets können agent und env easypeasy auf verschiedenen maschinen sein

-da qualitative änderungen sich literally immer erst nach tagen bemerkbar machen war paramter-tuning EXTREM langwierig und ist noch nicht done

-wie schlecht die anderen eigentlcih funktioniert haben so "fleigt aus der ersten kurve, klappt bei konstant 1.8kmh speed"... meins fährt gut bei max-speed, und fährt hin und wieder ganze runden I mean thats good!!

testing took place on a win10 machine, ... Answer all of the research questions explicitly!!!

-wie praktisch das mit dem info senden ist, dass das mit torcs-env-accessen nicht so einfach wäre

-result: mein way of incorporating pretraining -dass ein supervised agent nicht lernen kann da er immer in die wand brettert -parameter selection gehört hier rein!!

-dass mit maximalgeschwindigkeit funtioniert -wie viel arbeit die socketkommunikation war und wie überraschend effizient die ist -mit dem aktuellem reward lernt er das vor der kurve bremsen besser ist als fahren, wie man sieht wenn man davor steht

-dass man mit dem SV-agent testen kann ob dataset separable, praktischerweise kann das supervisednet ja die gleiche struktur haben wie das dqn (gucken kann ob dieseundjene state-definition sinnvoll ist)

-dass er definitiv lernt vor ner wand zu bremsen, dank der reward function, war nen langer weg bis dahin.

-dass actions als input (beosnder bei pretraining) sehr schlecht ist da der dann lernt immer die vorherige zu machen

[wennde testen wilslt ob dataset seperable ist, guckste in sv_agent, wennde wissen willst ob das model funktioniert, guckste in gym_text]

Hab jetzt auch zig verschiedene sachen probiert wie ich q-werte initialisieren (alle negativ, alle positiv, alle ungefähr null

dass mit dem gleichem network mit dem man supervisedly getrained hat jetzt q-training zu machen nicht geht

The different agents

DQN vs DDPG, sehend vs nicht-sehend, ...

getstatecountfeaturevec macht für das was ich mache mit meiner netzwerk-struktur keinen sinn (256 neurons direkt bevor es in die actions geht)

Chapter 7

Discussion

-dass feed$_dictslowisthttps : //github.com/tensorflow/tensorflow/issues/2919$

-wie gut dieses framework mittlerweile funktioniert, und was für ein langer weg das dahin war (dass das definitiv was war das ne ganze arbeit füllt, aber zukünftige arbeiten jetzt super gut darauf gehen, weil wegen bam framework.) -das das framework definitiv comparable zu dem given in den beiden torcs-papern (BEIDE NOCHMAL ZITIEREN!!) ist. -man kann a3c nutzen statt dqn -dass ich kein anderes paper gesehen habe dass sinvolle sachen macht... das virutal to real macht 9 discrete actions, das ddpg-torcs-ding macht "sometimes useful pocicies", ...

Normally when dealing with self-driving cars, there are countless additional issues, each making up a whole new challenge on their own, like wheather conditions (snow, rain, fog), pedestrians, other cars, reflections, merging into ongoing traffic, ... we make it easier here.

-dass es zur beschleuniung möglich wäre mehrere aggregatoren zu haben, die immer die recht aktuellste policy haben und mit unity interagieren und das nem learn-thread schicken, auf verschiedenen Maschinen

"Fragestellung aus der Einleitung wird erneut aufgegriffen und die Arbeitsschritte werden resümiert" Zusammen mit der Conclusion 10% der Gesamtlänge

Chapter 8

Conclusion and future directions

Die paper die bei openAI erwähnt waren die das ganze in viel kreasser sind, die Dota spielen können etc! dass ich gerne hätte dass das am anfang der policy folgt und am ende weiter exploriert, damit es auchdie letzte kurve lernt... genau dafür ware halt intrinsic motivation und prioritized experience replay

Appendix 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 [28]), performing **Double Q-learning** (as described in [7]). The last page consists of a comparison between the pseudocode of the general program flow of a DDQN-network (taken from [17], with changes from [7] and [12] 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
class Agent():
 def __init__(self, inputsize):
  self.inputsize = inputsize
  self.model = DDDQN_model
  self.memory = Memory(10000, self)  #for definition see code
  self.action_repeat = 4
  self.update_frequency = 4
  self.batch_size = 32
  self.replaystartsize = 1000
  self.epsilon = 0.05
  self.last_action = None
  self.repeated_action_for = self.action_repeat

 def runInference(self, gameState, pastState):
  self.addToMemory(gameState, pastState)
  inputs = self.getAgentState(*gameState)
  self.repeated_action_for += 1
  if self.repeated_action_for < self.action_repeat:
   toUse, toSave = self.last_action
  else:
   self.repeated_action_for = 0
   if self.canLearn() and np.random.random() > self.epsilon:
    action, _ = self.model.inference(inputs)
    toSave = self.dediscretize(action[0])
    toUse = "["+str(throttle)+", "+str(brake)+", "+str(steer)+"]"
   else:
    toUse, toSave = self.randomAction() #for definition see code
   self.last_action = toUse, toSave
  self.containers.outputval.update(toUse, toSave)
  self.numsteps += 1
  if self.numsteps % self.update_frequency == 0 and len(self.memory) > self.replaystartsize):
   self.learnStep()

 def learnStep(self):
  QLearnInputs = self.memory.sample(self.batch_size)
  self.model.q_learn(QLearnInputs)

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


class DuelDQN():
 def __init__(self, name, inputsize, num_actions):
  with tf.variable_scope(name, initializer = tf.random_normal_initializer(0, 1e-3)):
   #for the inference
   self.inputs = tf.placeholder(tf.float32, shape=[None, inputsize], name="inputs")
   self.fc1 = tf.layers.dense(self.inputs, 400, activation=tf.nn.relu)
   #modifications from the Dueling DQN architecture
   self.streamA, self.streamV = tf.split(self.fc1,2,1)
   xavier_init = tf.contrib.layers.xavier_initializer()
   neutral_init = tf.random_normal_initializer(0, 1e-50)
   self.AW = tf.Variable(xavier_init([200,self.num_actions]))
   self.VW = tf.Variable(neutral_init([200,1]))
   self.Advantage = tf.matmul(self.streamA,self.AW)
   self.Value = tf.matmul(self.streamV,self.VW)
   self.Qout = self.Value + tf.subtract(self.Advantage,tf.reduce_mean(self.Advantage,axis=1,keep_dims=
       ➥ True))
   self.Qmax = tf.reduce_max(self.Qout, axis=1)
   self.predict = tf.argmax(self.Qout,1)
   #for the learning
   self.targetQ = tf.placeholder(shape=[None],dtype=tf.float32)
   self.targetA = tf.placeholder(shape=[None],dtype=tf.int32)
   self.targetA_OH = tf.one_hot(self.targetA, self.num_actions, dtype=tf.float32)
   self.compareQ = tf.reduce_sum(tf.multiply(self.Qout, self.targetA_OH), axis=1)
   self.td_error = tf.square(self.targetQ - self.compareQ)
   self.q_loss = tf.reduce_mean(self.td_error)
   q_trainer = tf.train.AdamOptimizer(learning_rate=0.00025)
   q_OP = q_trainer.minimize(self.q_loss)
  self.trainables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=name)


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

```
1  Initialize replay memory $D$ to capacity N
5  Initialize action-value function $Q(s,a;\theta)$ with random weights $\theta$
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 $\epsilon$ 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 $\theta$
25   Update target network: $\theta^- \leftarrow \tau * \theta + (1-\tau)\theta^-$
26  End For
27 End For
```

```
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

13   return self.sess.run([self.onlineQN.predict, self.onlineQN.Qout], feed_dict={
         ➥ self.onlineQN.inputs: statesBatch})
14  #see agent
15  #see agent
16  #see agent
17  def q_learn(self, batch): #also called for every step t
18   oldstates, actions, rewards, newstates, terminals = batch
19   action = 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)),action]
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
```

A.2 DDPG

The following section describes the structure of an actual reinforcement learning agent, using an **actor-critic architecture** as its model, basing on the Deep Deterministic Policy gradient, as described in [23] and [12]. The last page consists of a comparison between the pseudocode of the general program flow of a DDPG-agent (taken from [12]) 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
class Actor(object):
 def __init__(self, inputsize, num_actions, actionbounds, session):
  with tf.variable_scope("actor"):
   self.online = lowdim_actorNet(inputsize, num_actions, actionbounds)
   self.target = lowdim_actorNet(inputsize, num_actions, actionbounds, name="target")
   self.smoothTargetUpdate = _netCopyOps(self.online, self.target, 0.001)
   # provided by the critic network
   self.action_gradient = tf.placeholder(tf.float32, [None, num_actions], name="actiongradient")
   self.actor_gradients = tf.gradients(self.online.scaled_out, self.online.trainables, -self.action_gradient)
   self.optimize = tf.train.AdamOptimizer(1e-4).apply_gradients(zip(self.actor_gradients, self.online.trainables))
def train(self, inputs, a_gradient):
 self.session.run(self.optimize, feed_dict={self.online.ff_inputs:inputs, self.action_gradient:a_gradient})
def predict(self, inputs, which="online"):
 net = self.online if which == "online" else self.target
 return self.session.run(net.scaled_out, feed_dict={net.ff_inputs:inputs})
def update_target_network(self):
 self.session.run(self.smoothTargetUpdate)

class Critic(object):
 def __init__(self, inputsize, num_actions, session):
  with tf.variable_scope("critic"):
   self.online = lowdim_criticNet(inputsize, num_actions)
   self.target = lowdim_criticNet(inputsize, num_actions, name="target")
   self.smoothTargetUpdate = _netCopyOps(self.online, self.target, 0.001)
   self.target_Q = tf.placeholder(tf.float32, [None, 1], name="target_Q")
   self.loss = tf.losses.mean_squared_error(self.target_Q, self.online.Q)
   self.optimize = tf.train.AdamOptimizer(1e-3).minimize(self.loss)
   self.action_grads = tf.gradients(self.online.Q, self.online.actions)
 def train(self, inputs, action, target_Q):
  return self.session.run([self.optimize, self.loss], feed_dict={self.online.ff_inputs:inputs, self.online.actions: action, self.target_Q: target_Q})
 def predict(self, inputs, action, which="online"):
  net = self.online if which == "online" else self.target
  return self.session.run(net.Q, feed_dict={net.ff_inputs:inputs, net.actions: action})
 def action_gradients(self, inputs, actions):
  return self.session.run(self.action_grads, feed_dict={self.online.ff_inputs:inputs, self.online.actions: actions})
 def update_target_network(self):
  self.session.run(self.smoothTargetUpdate)

def _netCopyOps(fromNet, toNet, tau = 1):
 op_holder = []
 for idx,var in enumerate(fromNet.trainables[:]):
  op_holder.append(toNet.trainables[idx].assign((var.value()*tau) + ((1-tau)*toNet.trainables[idx].value())))
 return op_holder

def dense(x, units, activation=tf.identity, decay=None, minmax = float(x.shape[1].value) ** -.5):
 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))

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

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

```
Randomly initialize critic network Q(s,a|$\theta^Q$) with weights $\theta^Q$
 and actor $\pi$(s|$\theta^\pi$) with weights $\theta^\pi$.

Initialize target network Q' weights $\theta^{Q'} \leftarrow \theta^Q$
 and $\pi$' with weights $\theta^{\pi'} \leftarrow \theta^\pi$
Initialize replay buffer R
for episode = 1, M do
 Initialize a random process $\mathcal{N}$ for action exploration
 Receive initial observation state s$_1$
 for t = 1, T do
  Select action $a_t = \pi(s_t|\theta^\pi) + \mathcal{N}_t$ according to the current policy and
      ➥ exploration noise
  Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
  Store transition $(s_t, a_t, r_t, s_{t+1})$ in R

  Sample a random minibatch of N transitions $(s_t, a_t, r_t, s_{t+1})$ from R
  targetActorAction = $\pi'(s_{i+1}|\theta^{\pi'})$
  targetCriticQ = $Q'(s_{i+1}$,targetActorAction$|\theta^{Q'})$
  Set $y_i = r_i + \gamma*$targetCriticQ  #only in nonterminal states

  Update critic by minimizing the loss: L = $\frac{1}{N}\sum_i(y_i - Q(s_i, a_i|\theta^Q))^2$
  Find the sampled policy gradient:
    a_i = $\pi(s_i|\theta^\pi)$
    $\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_a Q(s_i, a_i|\theta^Q)\nabla_{\theta^\pi}\pi(s_i|\theta^\pi)$
  Update the actor policy using the sampled policy gradient
  Update the target networks:
    $\theta^{Q'} \leftarrow \tau * \theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\pi'} \leftarrow \tau * \theta^Q + (1-\tau)\theta^{\pi'}$
 end for
end for
```

```python
class DDPG_model():
 def __init__(self,  session):
  self.session = session
  self.critic = Critic(self.session)
  self.actor = Actor(self.session)
  self.session.run(tf.global_variables_initializer())
  self.session.run(_netCopyOps(self.actor.target, self.actor.online))
  self.session.run(_netCopyOps(self.critic.target, self.critic.online))
  #replay buffer defined by the agent

 #exploration noise added by the agent
 #agent samples all observations
 def inference(self, oldstates): #called for every step t
  return self.actor.predict(oldstates, "target", is_training=False)
  #agent adds exploration noise afterwards
  #done by the agent
  #done by the agent
 def train_step(self, batch): #also called for every step t
  oldstates, actions, rewards, newstates, terminals = batch
  targetActorAction = self.actor.predict(newstates, "target")
  targetCriticQ = self.critic.predict(newstates, targetActorAction, "target")
  cumrewards = np.reshape([rewards[i] if terminals[i] else rewards[i]+0.99*
      ➥ targetCriticQ[i] for i in range(len(rewards))], (len(rewards),1))
  _, loss = self.critic.train(oldstates, actions, cumrewards)

  onlineActorActions = self.actor.predict(oldstates)
  grads = self.critic.action_gradients(oldstates, onlineActorActions)
  self.actor.train(oldstates, grads[0])
  #updating the targetnets
  self.critic.update_target_network()
  self.actor.update_target_network()
  return
```

Appendix B

Screenshots

B.1 Game

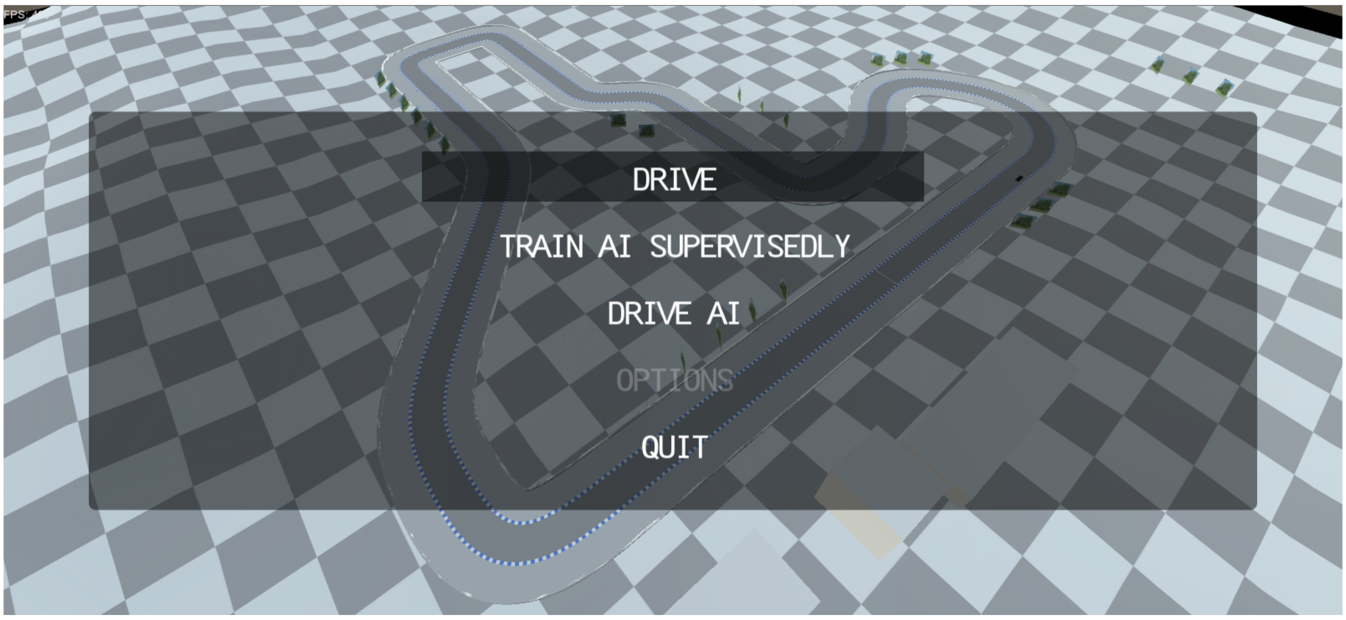


FIGURE B.1: Start screen / menu of the game, also showing a bird-eye view of the track

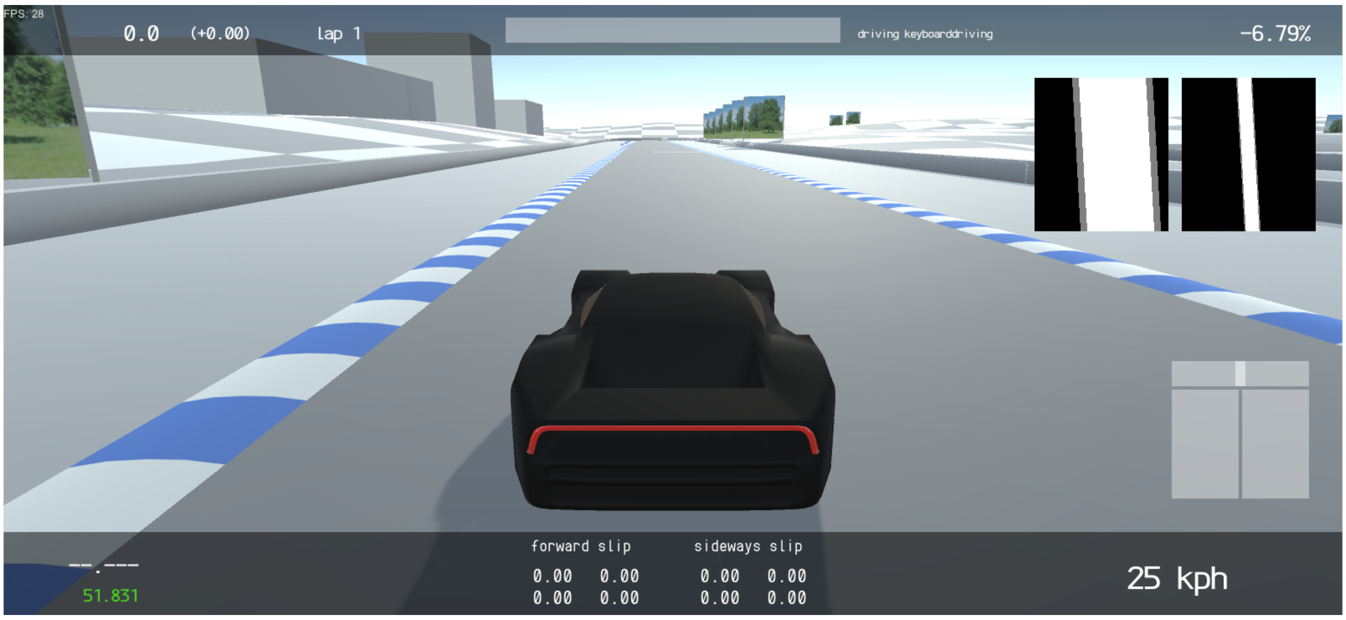
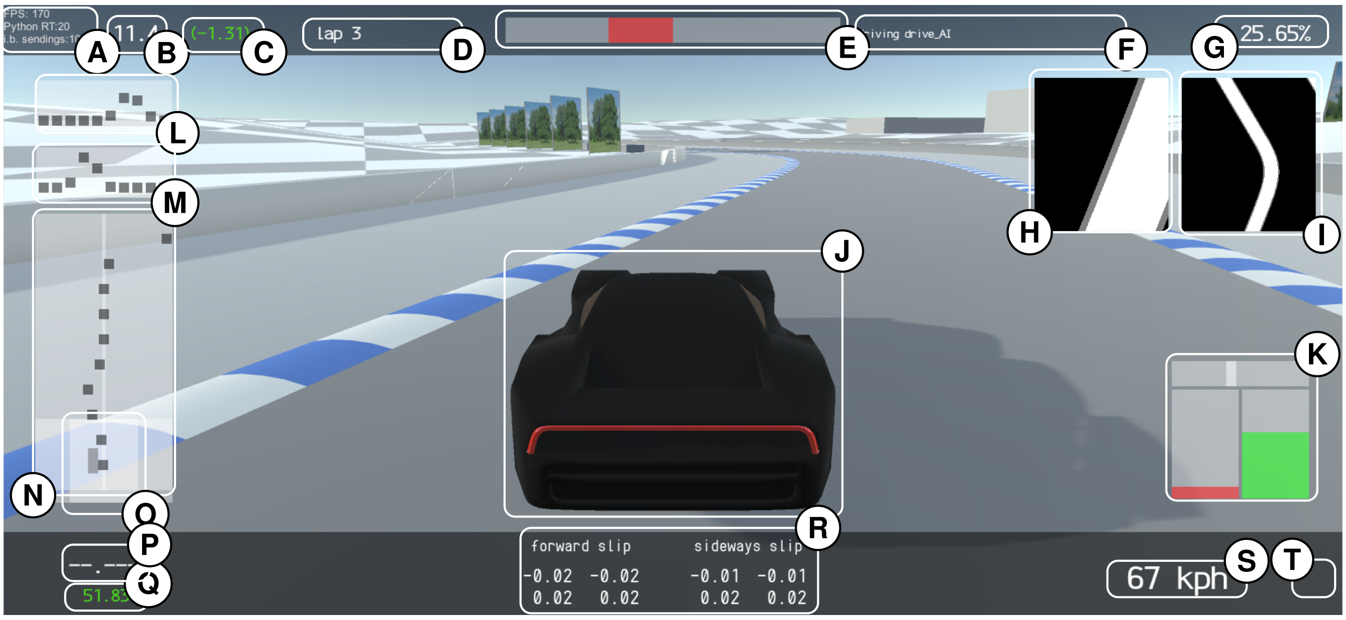


FIGURE B.2: **Drive** mode. For a description of the UI components, it is referred to section 4.2.1



```
forward slip        sideways slip
-0.02  -0.02        -0.01  -0.01
 0.02   0.02         0.02   0.02
```

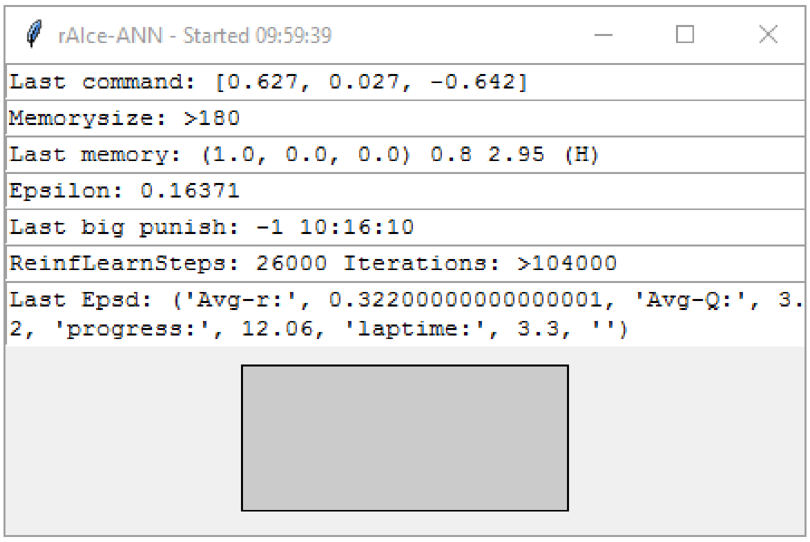
FIGURE B.3: **Drive AI** mode, showing many additional information directly on the screen. For a description of those, it is referred to sections 4.2.1 and 5.1

- **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

INCLUDE IMAGE FROM ABOVE IN FINAL VERSIONS

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



rAIce-ANN - Started 09:59:39

```
Last command: [0.627, 0.027, -0.642]
Memorysize: >180
Last memory: (1.0, 0.0, 0.0) 0.8 2.95 (H)
Epsilon: 0.16371
Last big punish: -1 10:16:10
ReinfLearnSteps: 26000 Iterations: >104000
Last Epsd: ('Avg-r:', 0.32200000000000001, 'Avg-Q:', 3.
2, 'progress:', 12.06, 'laptime:', 3.3, '')
```

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

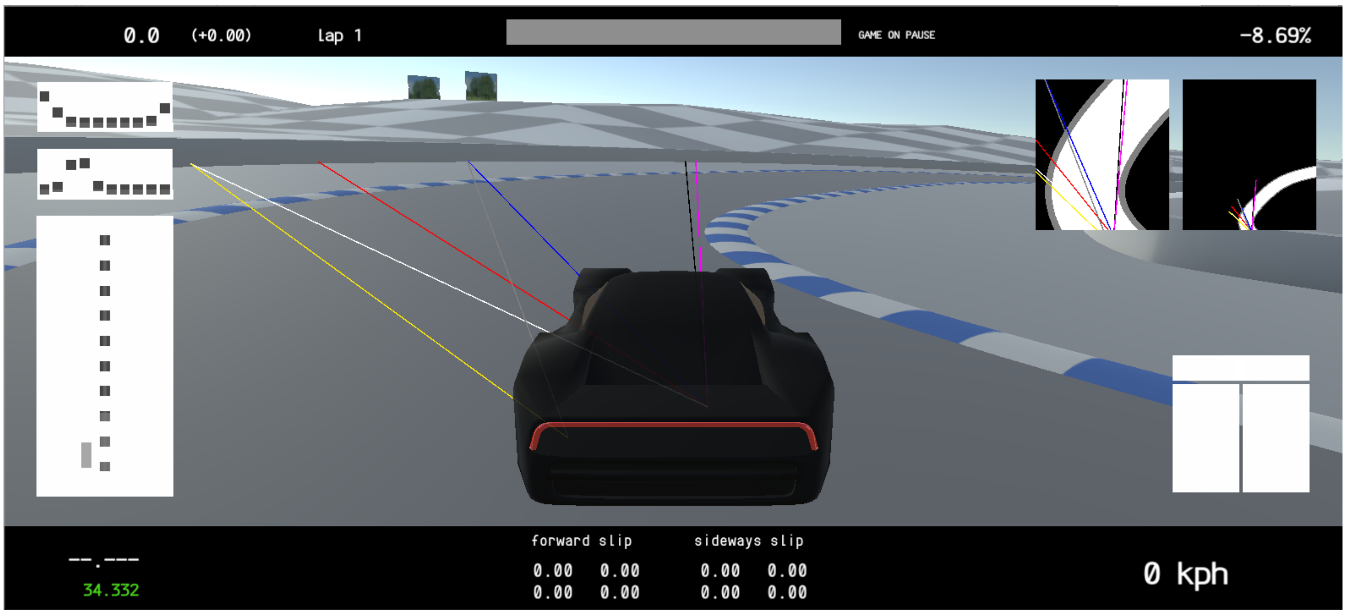


FIGURE B.6: Graphical representation of the rays whose lenghts correspond to the `WallDistVec`. Note that the white and yellow ray as well as the blue and grey ray are actually parallel, as can be seen in the orthogonal minimap-cameras. To see these rays, `Gizmos` must be turned on in Unity.

Appendix C

Informal description of the files belonging to the game

Filename	attached to	knows	Start()	Update()	FixedUpdate()	Other behaviour	other functions
UIScript.cs	Object Hierachy	all UI components	enable drivingOverlayActive	• DrivingOverlayHandling: update GUI while `driving` • MenuOverlayHandling: update GUI and listen for input while `menu`		onGUI(): Print Debug Information on screen	
Gamescript.cs	Object Hierachy		• Sets MiniMapCamera distances • Sets `mode` to `menu`	• enable/disable `QuickPause` if other threads demand it • switch `mode` to `menu` on Esc			• enabling and disabling QuickPause • SwitchMode inclusive the respective calls to initialize the mode
CarController.cs	Car		Sets Car's position	• Rotates and adjust height of visible wheels • Enable reverse gear on R	• Sets Car's Forces to zero after a reset • handles friction, checks if lap invalid, applices forces to Car		• ResetCar and ResetToPosition • AdjustFriction • GetSurface
TimingScript.cs	TimingSystem's Box Collider					OnTriggerExit(): • sets fast- and lastlaptime • calls Recfinishlist and, if applicable, AiInt.EndRound • resets activeLap, increases lapcoutn, … • Starts new lap for Rec	Other timing-related functions
ConfirmColliderScript.cs	TimingSystem's Confirm Colider					OnTriggerExit(): Calls Timing.flipCCPassed()	
Recorder							

Appendix D

A minimally viable agent

```
import tensorflow as tf
#====own classes====
from agent import AbstractRLAgent
from dddqn import DDDQN_model



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


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

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] 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).

[3] 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).

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

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] 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.

[18] 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).

[19] 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).

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

[21] 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).

[22] 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).

[23] 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).

[24] 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).

[25] 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.

[26] 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).

[27] 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).

[28] 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).

[29] 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).

[30] 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).

[31] 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).

[32] 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.

[33] 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/d931665ec87c16fcd44cae8fdaec1215e81e.pdf (visited on 08/16/2017).

[34] 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).

[35] 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