UNIVERSITÄT OSNABRÜCK

Controlling Self-Driving Race Cars with Deep Neural Networks

UNIVERSITY OF OSNABRÜCK

DEPARTMENT OF NEUROINFORMATICS

BACHELOR'S THESIS

Author:
Christoph Stenkamp

Supervisors:
Prof. Dr. Gordon Pipa
Leon Sütfeld

Osnabrück,
August 26, 2017

Abstract

This Thesis will be written in the next two months, and I'm pretty scared about that.
TODO: sobald der komplette text steht bei den Formeln auf die nicht referenziert wird die nummern weg machen (equation*)

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, August 26, 2017

Acknowledgements

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

"There are no surprising facts, only models that are surprised by facts; and if a model is surprised by the facts, it is no credit to that model."

Eliezer Yudkowsky

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
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
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

List of Symbols

For my friends, family, and especially Marie.

Chapter 1

Introduction

"sollte etwa 10% der Gesamtarbeit ausmachen"

1.1 Motivation

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

1.1.1 Problem Domain

1.1.2 Goal of this thesis

1.2 Research Questions

1.3 Reading Guidelines

1.4 noclue

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**

[rrt* macht halt sowas von 3., end-to-end macht die kiste insgesamt nen bisschen anders]

Semi-autonomous vehicle components: Radar, Visible-light-camera, LIDAR, infrared-camera, stereo vision, GPS/IMU, CAN, Audio

Localization and Mapping: eg. $file : ///C : /Users/Marie/Downloads/VISAPP_2015_145.pdf$ Scene Understanding/Object Detection: Scene Segentation Network (SegNet) Movement Planning: Previously by stuff like RRT* (optimization-based control), however reinforcement learning!

End-to-End: NVIDIA Paper

dass Reinforcement Learning ja eigentlich nicht so der beste approach ist

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

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

(snow, rain, fog), pedestrians, other cars, reflections, merging into ongoing traffic, ... we make it easier here.

Or, according to the Torcs-paper: The racing problem could be split into a number of different components, including robust control of the vehicle, dynamic and static trajectory planning, car setup, inference and vision, tactical decisions (such as overtaking) and fi- nally overall racing strategy. With only a single car on the track, the overall problem can be formalised as a partially observable Markov decision processes.

–> Because we only consider the case of a single car on the track it definitely is a POMDP! seeee next chapter yay

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, I will in this chapter 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. What will be mainly considered in this thesis will be a certain kind of semi-supervised learning: *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* gets as input the problem definition 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:

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

As the future rewards depend on future states it can, as already mentioned, 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

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

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, it alone does not help an agent to perform optimally, as neither the successor function $P(s'|s,a)$, nor the reward function $R(r|s,a)$ are 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:

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

[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 [3]). 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 [14], 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 table, 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*[15]: 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

In contrast to SARSA stands the *Q-learning* algorithm [18]. 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 [10]. To express its dependence on the policy π, it will be superscripted by it:

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

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 [19] 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)[20], 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 actions $a \in \mathcal{A}$. 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[12], 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 distribution the whole underlying model. ANNs require independent and identically distributed samples, which is not given if the samples are generated sequentially. As shown by [7], 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[9] and 2015[10], 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[13], 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. The intuition behind why that is necessary is, that 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 [10]:

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

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 [8]. This improves the stability of the algorithm even more.

As a pseudocode for the DQN-architecture is already stated in the corresponding paper [10], 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 33 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 in every estimate of the value of a state s_j the maximum Q-value of state s_{j+1}, 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, [5] 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 [5] 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: Overerstimations 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 [5], 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 ending by the car hitting the wall, all actions will have roughly the same Q-value.

The idea of the *dueling architecture*[17] 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, such 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, $\frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \theta^A)$ 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 [17].

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 [17], 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 [5], as well as the computation graph of the network using precisely the DDDQN-architecture as described above and in [17]. The last page of it (page 33) consists of a comparison of the Pseudocode of the DDDQN (as found in [10], with the changes from [5] and [8] 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 was 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

The technique introduced in the previous sections is a direct adaption of the Q-learning algorithm [15][18], 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 algorithms 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 [14], 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 [14].

[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 [16], 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.

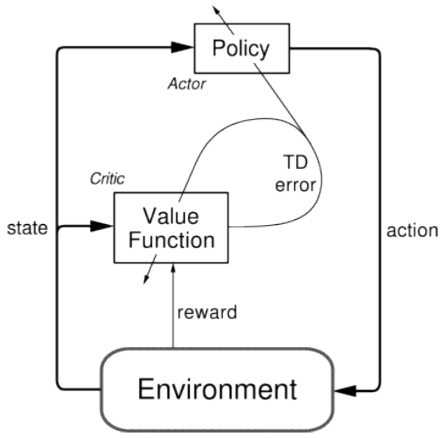


FIGURE 2.1: The actor-critic architecture. Reprinted from [11].

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 [14])

updates its parameters in the direction of the critic's action-value gradient:

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

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

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

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

As stated by [14], 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**)[8] 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 are 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. [8] 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 [8]:

Target-value of the critic:

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

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 36, there is again a comparison of the pseudocode (as provided in [8]) and the correspondences in actual-python code, where each line of the 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, which I 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 sure to be 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 were 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.

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

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

[hier zuerst noch ne kurze zusammenfassung, und das bild mit der environment-agent metapher!] Gym/Universe Torcs schreiben dass die Arcade Learning Environment (Bellemare et al., 2013 aus dem Dueling) zu Gym wurde (I guess) torcs: im DDPG-paper steht "Torcs has previously been used as a testbed in other policy learning approaches (Koutnik et al., 2014b). "!!!!!!!!!!!!!!! fußnote dass ich auch im code nen evaluator für ddpg hab der gym und pendulum swingup nutzt

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.

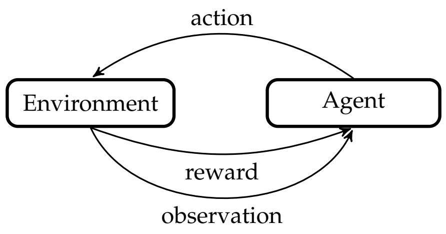


FIGURE 3.1: Interaction between agent and environment in RL

When developing RL agents, it is not enough to create algorithm, but also to have a specification of agent environment that allows for a dataflow as described above. The original Deep-Q-Network [9] as well as its follow-ups [5][17] trained their agents on several ATARI games using the *Arcade Learning Environment* (**ALE**) [2]. The ALE 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. The ALE consists of an Atari-Simulator as well as a game-handling layer, which transformed all of the included Atari-games to a standard reinforcement learning problem. 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. The possible input to the simulation consists of 18 discrete actions.

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* environment [4]. The OpenAI gym[1] is created as a toolkit, helping reinforcement learning research by including a collection of benchmark problems with a common interface. The idea is to provide consistent, standardized environments, such that algorithms are easy to benchmark. In that respect, their goal was to provide the reinforcement-learning-pendant of a labeled dataset, such as ImageNet. 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. These tasks are varied in their complexity, input and output: some of them use low-dimensional state representations consisting of only a four-dimensional vector of scalars, while others use a 2D-pixel image of RGB colors.

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

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

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

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
```

3.2 Self-driving cars

3.2.1 real-life

Nvidias deep-drive RRT* Tesla

available data in real-life

Lidar

3.2.2 games

Tensorkart hier in den fußnoten die ganzen non-scientific quellen wie tensorkart undso

Das Aufteilen von autonomous driving into the subcomponents, wie bei der vorlesung von Julian... oder, as quoted by TORCS-Paper: "The racing problem could be split into a number of different components, including robust control of the vehicle, dynamic and static trajectory planning, car setup, inference and vision, tactical decisions (such as overtaking) and finally overall racing strategy"

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

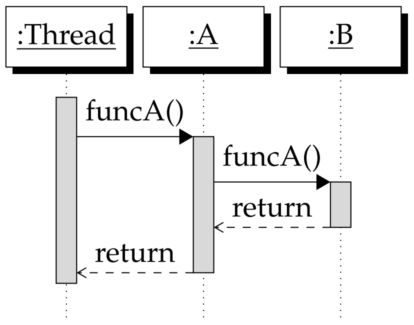
Chapter 4

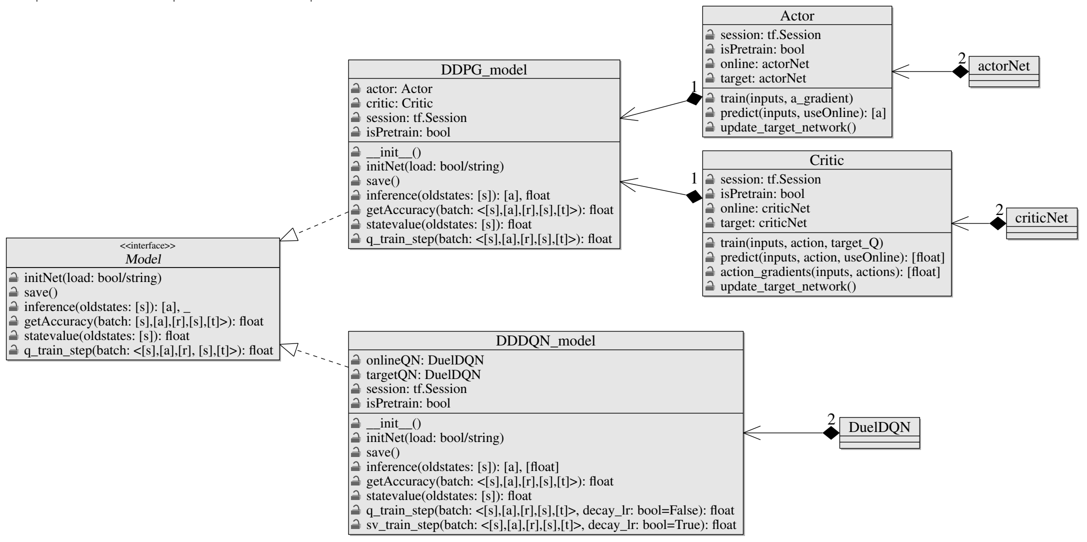
Program Architecture

The program was written by the author of this work and is licensed under the GNU General Public License (GNU GPLv3). Its source code is attached in the appendix of this work and additionally can be found digitally on the enclosed CD-ROM. The machine learning part was written in PYTHON, using the TENSORFLOW-library [1].
[1]

[tf gibt automatic differentiation, multi-gpu support, python interface]

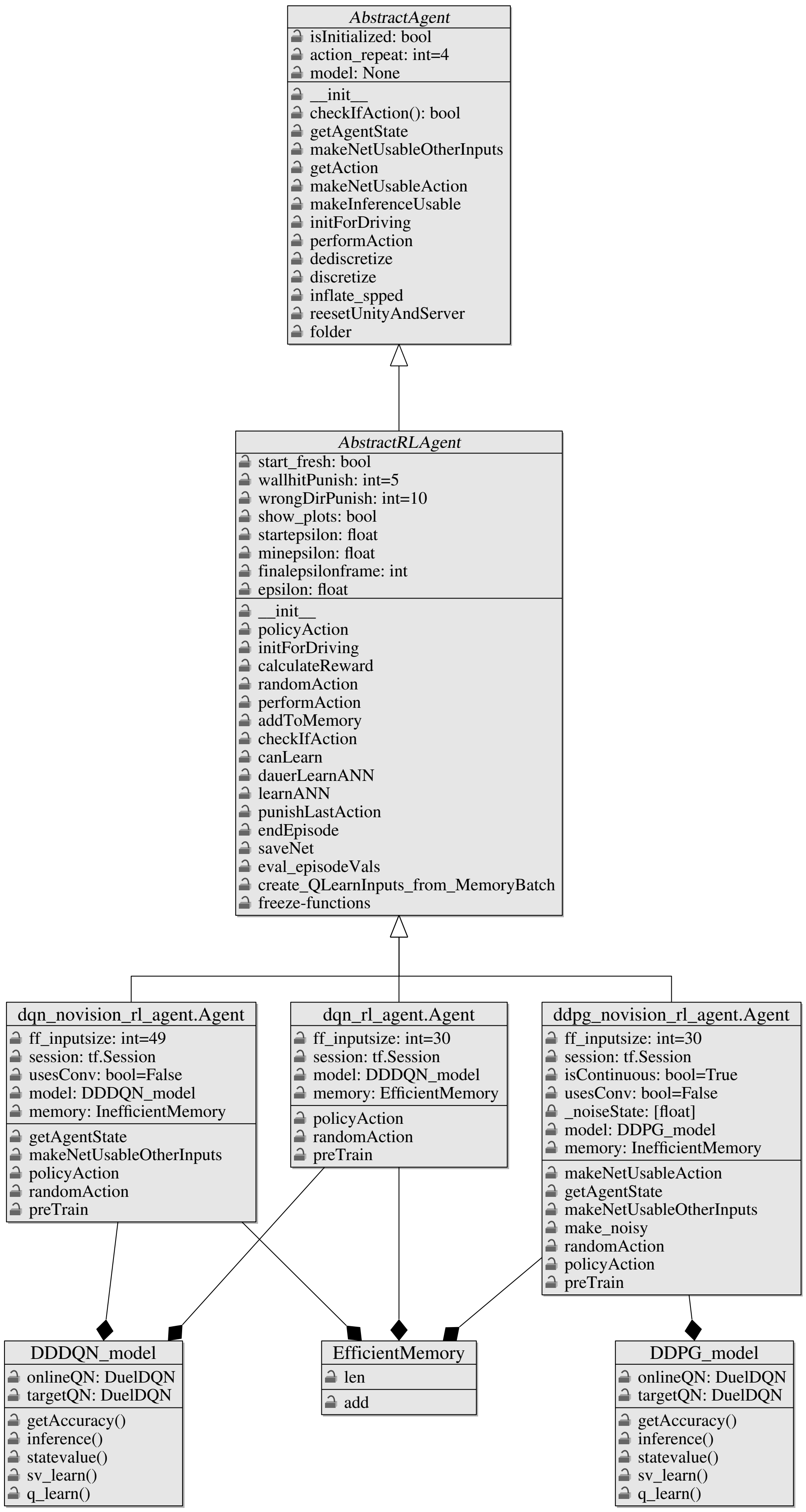
4.1 Design choices





Actor
session: tf.Session
isPretrain: bool
online: actorNet
target: actorNet
train(inputs, a_gradient)
predict(inputs, useOnline): [a]
update_target_network()
2
actorNet
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
q_train_step(batch: <[s],[a],[r],[s],[t]>): float
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()
2
criticNet
<<interface>>
Model
initNet(load: bool/string)
save()
inference(oldstates: [s]): [a], _
getAccuracy(batch: [s],[a],[r],[s],[t]>): float
statevalue(oldstates: [s]): float
q_train_step(batch: <[s],[a],[r], [s],[t]>): float
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]>): float
statevalue(oldstates: [s]): 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
2
DuelDQN

[1] I will rely heavily on UML to explain my code and stuff. Version UML 2.0. to get an overview of how that look like, I refer the reader to https://www.ibm.com/developerworks/rational/library/3101.html and https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html for sequence diagrams and class diagrams, respectively.


AbstractAgent
isInitialized: bool
action_repeat: int=4
model: None
__init__
checkIfAction(): bool
getAgentState
makeNetUsableOtherInputs
getAction
makeNetUsableAction
makeInferenceUsable
initForDriving
performAction
dediscretize
discretize
inflate_spped
reesetUnityAndServer
folder
AbstractRLAgent
start_fresh: bool
wallhitPunish: int=5
wrongDirPunish: int=10
show_plots: bool
startepsilon: float
minepsilon: float
finalepsilonframe: int
epsilon: float
__init__
policyAction
initForDriving
calculateReward
randomAction
performAction
addToMemory
checkIfAction
canLearn
dauerLearnANN
learnANN
punishLastAction
endEpisode
saveNet
eval_episodeVals
create_QLearnInputs_from_MemoryBatch
freeze-functions
dqn_novision_rl_agent.Agent
ff_inputsize: int=49
session: tf.Session
usesConv: bool=False
model: DDDQN_model
memory: InefficientMemory
getAgentState
makeNetUsableOtherInputs
policyAction
randomAction
preTrain
dqn_rl_agent.Agent
ff_inputsize: int=30
session: tf.Session
model: DDDQN_model
memory: EfficientMemory
policyAction
randomAction
preTrain
ddpg_novision_rl_agent.Agent
ff_inputsize: int=30
session: tf.Session
isContinuous: bool=True
usesConv: bool=False
_noiseState: [float]
model: DDPG_model
memory: InefficientMemory
makeNetUsableAction
getAgentState
makeNetUsableOtherInputs
make_noisy
randomAction
policyAction
preTrain
DDDQN_model
onlineQN: DuelDQN
targetQN: DuelDQN
getAccuracy()
inference()
statevalue()
sv_learn()
q_learn()
EfficientMemory
len
add
DDPG_model
onlineQN: DuelDQN
targetQN: DuelDQN
getAccuracy()
inference()
statevalue()
sv_learn()
q_learn()

4.1.1 The game as a reinforcement learning problem

-ungefähres UML-diagramm -das spiel itself ist ein partially observable MDP, und es ist (as tested) surprisingly indeterministic

[Ah. Should have googled first before posting. Differences in FPU calculations on different processors. Differences in timing affecting things like the number of times a random number generator is called. These can cause the physics to get out of sync] https://forum.unity3d.com/threads/is-unity-physics-is-deterministic.429778/

4.1.2 The vectors

4.1.3 Exploration

4.1.4 Reward

4.1.5 Performance measure

4.2 Implementation

4.2.1 The game

What Leon did already

Communication

-den sockets post -das von leon gemalte ablaufdiagramm

4.2.2 The agent

Unbedingt auf jeden Fall UML-diagramm

Challenges and Solutions

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

Pretraining

The different agents

sehend vs nicht sehend, ...

Network architecture

1. dqn-algorithm - anzahl layer, Batchnorm, doubles dueling - clipping wieder rein, reference auf das dueling - grundsätzlich gegen batchnorm entschieden, siehe reddit post - MIT GRAFIK - Adam und tensorflow quoten, siehe zotero 2. ddpg - anzahl layer, Batchnorm - MIT GRAFIK

-schöne grafik. -auf meien DQN-config eingehen und(!!!) ne DDPG-config machen, using the "experiment details" vom ddpg paper

In the original DDPG-algorithm [8], the authors used *batch normalization* [6] to have the possibility of using 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, specifying the phase (learning step/non-learning step)[2]. Though Lillicrap et al. seemed to have success on using batch normalization , in practice it lead to unstability, even on simple physics tasks in openAI's gym. As I am not the only one having this issue [3], I left out batch normalization for good.

[2]cf. `https://www.tensorflow.org/api_docs/python/tf/contrib/layers/batch_norm`
[3]redditlink

Chapter 5

Analysis, Results and open Questions

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

Chapter 6

Discussion

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

Chapter 7

Conclusion and future directions

Die paper die bei openAI erwähnt waren die das ganze in viel kreasser sind, die Dota spielen können etc!

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 [17]), performing **Double Q-learning** (as described in [5]). The last page consists of a comparison between the pseudocode of the general program flow of a DDQN-network (taken from [10], with changes from [5] and [8] 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.

```
1  class Agent():
2   def __init__(self, inputsize):
3    self.inputsize = inputsize
4    self.model = DDDQN_model
5    self.memory = Memory(10000, self)  #for definition see code
6    self.action_repeat = 4
7    self.update_frequency = 4
8    self.batch_size = 32
9    self.replaystartsize = 1000
10   self.epsilon = 0.05
11   self.last_action = None
12   self.repeated_action_for = self.action_repeat

14  def runInference(self, gameState, pastState):
15   self.addToMemory(gameState, pastState)
16   inputs = self.getAgentState(*gameState)
17   self.repeated_action_for += 1
18   if self.repeated_action_for < self.action_repeat:
19    toUse, toSave = self.last_action
20   else:
21    self.repeated_action_for = 0
22    if self.canLearn() and np.random.random() > self.epsilon:
23     action, _ = self.model.inference(inputs)
24     toSave = self.dediscretize(action[0])
25     toUse = "["+str(throttle)+", "+str(brake)+", "+str(steer)+"]"
26    else:
27     toUse, toSave = self.randomAction() #for definition see code
28    self.last_action = toUse, toSave
29   self.containers.outputval.update(toUse, toSave)
30   self.numsteps += 1
31   if self.numsteps % self.update_frequency == 0 and len(self.memory) > self.replaystartsize):
32    self.learnStep()

34  def learnStep(self):
35   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
```

```
Initialize replay memory $D$ to capacity N
Initialize action-value function $Q(s,a;\theta)$ with random weights $\theta$
Initialize target action-value function $Q(s,a;\theta^-)$ with weights $\theta^- = \theta$
For episode = 1,M do
 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
 For 1 = 1,T do
  With probability $\epsilon$ select random action $a_t$
  otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$

  Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
  Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
  Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$

  Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
  Define $a^{max}(\phi_{j+1};\theta) = argmax_{a'} Q(\phi_{j+1}, a'; \theta)$
  Define $Q^{j+1} = Q(\phi_{j+1}, a^{max}(\phi_{t+1};\theta); \theta^-)$

  If episode terminates at step $j+1$ then set $y_j = r_j$,
      Otherwise set $y_j = r_j + \gamma * Q^{j+1}$
  Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect
      to the network parameters $\theta$
  Update target network: $\theta^- \leftarrow \tau * \theta + (1-\tau)\theta^-$
 End For
End For
```

```python
#see agent
class DDDQN_model():
 def __init__(self, sess, inputsize, num_action):
  self.sess = sess
  self.onlineQN = DuelDQN("onlineNet", inputsize, num_action)
  self.targetQN = DuelDQN("targetNet", inputsize, num_action)
  self.sess.run(tf.global_variables_initializer())
  self.sess.run(_netCopyOps(self.targetQN, self.onlineQN))

 #see agent
 def inference(self, statesBatch): #called for every step t

  return self.sess.run([self.onlineQN.predict, self.onlineQN.Qout], feed_dict={
      self.onlineQN.inputs: statesBatch})
 #see agent
 #see agent
 #see agent
 def q_learn(self, batch): #also called for every step t
  oldstates, actions, rewards, newstates, terminals = batch
  action = self.sess.run(self.onlineQN.predict, {self.onlineQN.inputs:newstates
      })
  folgeQ = self.sess.run(self.targetQN.Qout, {self.targetQN.inputs:newstates})
  doubleQ = folgeQ[range(len(terminals)),action]
  consider_stateval = -(terminals - 1)
  targetQ = rewards + (0.99 * doubleQ * consider_stateval)
  self.sess.run(self.onlineQN.q_OP, feed_dict={self.onlineQN.inputs:oldstates,
      self.onlineQN.targetQ:targetQ, self.onlineQN.targetA:actions})
  self.sess.run(_netCopyOps(self.onlineQN, self.targetQN, 0.001))
  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 [14] and [8]. The last page consists of a comparison between the pseudocode of the general program flow of a DDPG-agent (taken from [8]) 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.

```
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):
```

```python
 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
```

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

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] 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.

[11] 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).

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

[13] 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).

[14] 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).

[15] 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.

[16] 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).

[17] 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).

[18] 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).

[19] 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).

[20] 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).

Declaration of Authorship