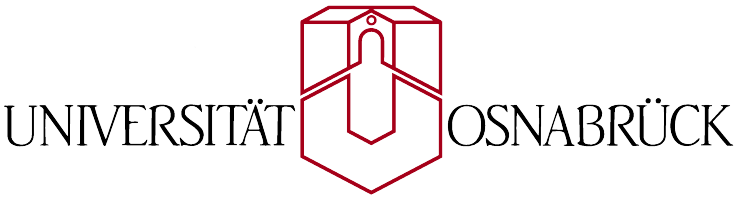

Controlling Self-Driving Race Cars with Deep Neural Networks

UNIVERSITY OF OSNABRÜCK

DEPARTMENT OF NEUROINFORMATICS

BACHELOR'S THESIS

Author:
Christoph Stenkamp

Supervisors:
Prof. Dr. Gordon Pipa
Leon Sütfeld

Osnabrück,
August 15, 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 15, 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

Abstract i

Preface ii

Acknowledgements iii

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Problem Domain . . . 1
  - 1.1.2 Goal of this thesis . . . 1
- 1.2 Research Questions . . . 1
- 1.3 Reading Guidelines . . . 1

2 Reinforcement Learning 2
- 2.1 Reinforcement Learning Problems . . . 2
  - Markov Decision Processes . . . 2
  - Value of a state . . . 3
  - Value of an action . . . 4
  - Quality of a policy . . . 5
- 2.2 Temporal difference Learning . . . 6
  - SARSA . . . 7
  - Q-learning . . . 7
- 2.3 Q-Learning with Neural Networks . . . 9
  - 2.3.1 Deep Q-learning . . . 9
  - Experience Replay . . . 10
  - Target Networks . . . 10
  - Double-Q-Learning . . . 11
  - Dueling Q-Learning . . . 11
  - Using Recurrent Networks . . . 11
- 2.4 Policy Gradient Techniques . . . 11
  - 2.4.1 Actor-Critic architectures . . . 11
  - Deterministic Policy Gradient . . . 12
  - 2.4.2 Exploration techniques . . . 15

3 Related work 16
- 3.1 Reinforcement Learning Frameworks . . . 16
- 3.2 Self-driving cars . . . 16

4 Program Architecture 17
- 4.1 Design choices . . . 17
  - 4.1.1 The game as a reinforcement learning problem . . . 17
  - 4.1.2 The vectors . . . 17

4.1.3 Exploration . . . 17
4.1.4 Reward . . . 17
4.1.5 Performance measure . . . 17
4.2 Implementation . . . 17
4.2.1 The game . . . 17
What Leon did already . . . 17
Communication . . . 17
4.2.2 The agent . . . 17
Challenges and Solutions . . . 17
Pretraining . . . 17
The different agents . . . 17
Network architecture . . . 18

5 Analysis, Results and open Questions **19**

6 Discussion **20**

7 Conclusion and future directions **21**

A Comparison Pseudocode & Python-code **22**
A.1 DQN . . . 22
A.2 DDPG . . . 25

Bibliography **28**

Declaration of Authorship **29**

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

List of Symbols

For my friends, family, and especially Marie.

Chapter 1

Introduction

"sollte etwa 10% der Gesamtarbeit ausmachen"

1.1 Motivation

1.1.1 Problem Domain

1.1.2 Goal of this thesis

1.2 Research Questions

1.3 Reading Guidelines

Chapter 2

Reinforcement Learning

As the task as 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, to give insights into why I did what I did. Also, I will try to keep this stuff as general as possible, getting into detail when speaking about the used algorithms. [The sense of this chapter is to give an intro of MDPs and RL. It shall also go into enough details on how to specify an MDP such that an RL agent can learn on it, because a big part of the work was exactly that. It's supposed to end with SARSA and Q-learning as the two Ideas on how to perform RL]

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 (i.e. it uses 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, by for example grouping objects by similarity of their properties.
What will be mainly considered in this thesis will be a certain kind of semi-supervised learning: *Reinforcement learning*. In Reinforcement Learning (**RL**), instead of labels for the data, there is a *weak teacher*, which provides feedback to the actions the agent took.

Markov Decision Processes

The metaphor behind RL is that of a decision maker (*agent*) and 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:

$\mathcal{S}$ – set of states $s \in \mathcal{S}$

$\mathcal{A}$ – set of actions $a \in \mathcal{A}$

$P(s'|s,a)$ – transition probability function from state s to state s' under action $a : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$

$R(r|s,a)$ – reward probability function for action a performed in state $s : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$

γ – discount factor for future rewards $0 \leq \gamma \leq 1$

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, it can always only be talked about the expected value of anything, depending on the random distribution of states. I will come back to that later. 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. An episode thus consists for the agent of a sequence 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

[1] From now on, when I mean the state of the environment, I will explicitly refer to it as s_e, while reserving s for the agent's obvervation of the enviroment $o(s_e)$

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 for what 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 his policy π from this state on:

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

As the future rewards depend on future states one can, as already mentioned, only talk 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 for a 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, using which those high-value states are actually reached. 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, we can re-write or definition of the value $V^\pi(s)$ 2.1 recursively as

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

This relation is known as the *Bellman Equation*, which allowed for the birth of dynamic programming. Dynamic programming however needs to the complete MDP, which is not given in most of the relevant situations.

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

[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 πs dependence on θ.

[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.

$R(r|s,a)$ 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:

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

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

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

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[4]. Thus, we can rewrite the value of our decision problem at time t 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 agents goal is to find a policy that can follow the trajectory of that state distribution with the highest expected reward. If the actual Q-value for each action of each state was 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. It integrates over the whole state space $\mathcal{S}$ with each state s weighted by its distribution due to policy π. If we also considered stochastic policies, one would also have to integrate over the whole action space $\mathcal{A}$, however we will only consider deterministic policies here. The integral can, as shown by [10], be expressed by the

[4]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 [2])

expectation of the Value of states following the distribution $s \sim \rho^\pi$:

$$J(\pi) = \int_{\mathcal{S}} \rho^\pi(s) V^\pi(s) ds \tag{2.7}$$
$$= \mathbb{E}_{s\sim\rho^\pi}\left[V^\pi(s)\right] \tag{2.8}$$
$$= \mathbb{E}_{s\sim\rho^\pi}\left[Q^\pi(s, \pi(s))\right] \tag{2.9}$$

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].

Thus, the goal of a model-free RL agent is to get a maximally precise estimate of Q^*. For 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 kind of nonlinear function approximator. The agents approximation of Q^π will be denoted $\hat{Q}^\pi$.

Throughout exploration of the environment, the agent collects more information of 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*[11]: 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:

$$\hat{Q}^\pi(s_t, a_t) = r_t + \mathbb{E}_{s\sim\rho^\pi}\left[\gamma * max_{a_{t+1}} \hat{Q}^\pi(s_{t+1}, a_{t+1})\right] \tag{2.10}$$
$$\approx r_t + \gamma * r_{t+1} + \mathbb{E}_{s\sim\rho^\pi}\left[\gamma^2 * max_{a_{t+2}} \hat{Q}^\pi(s_{t+2}, a_{t+2})\right] \tag{2.11}$$

Keeping in mind that $\hat{Q}^\pi$ is only an estimator of the Q^π-values of the underyling model, it becomes clear that equation 2.11 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*

[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.

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 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. That can, if the policy of the agent is not stochastic, lead to the agent getting stuck in local optima.

Q-learning

In contrast to SARSA stands the *Q-learning* algorithm [12]. 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 the environment, almost resulting in the formula found in [7]. To express its dependence on the policy π, it will be superscripted by it:

$$y_t^\pi = \begin{cases} r_t & \text{if } t = t_t \\ r_t + \gamma * max_{a_{t+1}} \hat{Q}^\pi(s_{t+1}, a_{t+1}) & \text{otherwise} \end{cases} \tag{2.12}$$

The temporal difference error for time t is accordingly defined as

$$TD_t := y_t^\pi - \hat{Q}^\pi(s_t, a_t) \tag{2.13}$$

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 latter is the information if state s_{t+1} was a terminal state). That information is then used to calculate the temporal difference error.

Using the above error straight away allows for the update-rule of an agent in a very limited setting: Consider an agent, specifying his approximation of the Q-function (his *model*) with a lookup-table, initialized to all zeros. It is proven by [13] 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)

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

converges to the optimal Q^*-function, making the greedy policy π^* optimal[7]. Note,

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

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 $\hat{Q}^{\pi_{greedy}}(s,a)$, basing 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 $J(\pi)$ must change, as it must incorporate that while the value of a state is calculated using π, the distribution of states follows from policy β:

$$J_{\beta}(\pi) = \mathbb{E}_{s\sim\rho^{\beta}}\left[Q^{\pi}(s,\pi(s))\right] \tag{2.14}$$

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, basing on that, *policy improvement*. In the standard Q-learning considered here, those steps are interleaved, leading to a form of generalised policy iteration: the Q-learner learns 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 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 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 run into those problems, by splitting up policy evaluation and policy improvement explicitly.

Sticking for a while with discrete actions $\mathcal{A} \subseteq \mathbb{N}^n$ however, a Q-learner using tables as Q-function-approximator also 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 2.2 becomes irrelevant quickly. Instead, a better idea is to use this definition of the temporal difference error for 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. If one does that, it may also be useful to calculate the loss of a batch of temporal differences simultaneously (more on that later). The L2-Loss for batch $batch$ with model-parameters θ_i, making up the policy π_{θ_i} is thus defined as the following:

$$L_{batch}(\theta_i) := \mathbb{E}_{s,a,r\sim batch}\left[\left(y_{batch}^{\pi_{\theta_i}} - \hat{Q}_{batch}^{\pi_{\theta_i}}(s,a)\right)^2\right] \tag{2.15}$$

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. A special focus must also lie on *Convolutional Neural Networks* (**CNN**s) [14], mainly used in image processing. 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[8], it is known that ANNs are an example of such. The defining feature of ANNs in comparison to other Machine Learning techniques is their ability to store complex, abstract representations of their input when using a *deep* enough architecture.

2.3.1 Deep Q-learning

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, 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 using Deep Neural Networks. When using ANNs as function approximators for the model of the environment, it will result in 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, $\nabla_{\theta_i} L(\theta_i)$.

While there are attempts to use Artificial Neural Networks for Q-learning as early as 1994[8], 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 2.15 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 consequtive steps of MPDs tend to be correllated: It is very likely, that the maximizing action of time t is similar to the one at $t+1$. Thus, when using consecutive steps of the MDP is not representatitive for the distribution of the whole underlying model. ANNs require independent and identically distributed samples, which is not given if the samples are generated sequentially. As shown by [4], 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.

[8] http://neuralnetworksanddeeplearning.com/chap4.html, I need a better source on this!

It was not until *Deepmind*'s famous papers in 2013[6] and 2015[7], 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 $TD_0, .., TD_{t_t}$ (taking far too long with 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 (similar to that one of 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 hindered 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[9], 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, used to compute the loss (2.15), necessary for the learning step of every iteration. The idea behind that 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 a similar 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 2.12).

The use of this two techniques leads to the Q-learning update rule, using the loss as used in [7]:

$$L_i(\theta_i) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r + \gamma * max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}; \theta_i^-) - \hat{Q}(s_t, a_t; \theta_i) \right)^2 \right] \tag{2.16}$$

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 by now - modern algorithms don't perform a hard update of the target network every C steps, but instead perform *soft target network update*, 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 [5]. This improves the stability of the algorithm even more.

Double-Q-Learning

In DoubleQ, we still use the greedy policy to select actions, however we evaluate how good it is with another set of weights DDQN: instead of taking the max over Q-values when computing the target-Q value for our training step, we use our primary network to chose an action, and our target network to generate the target Q-value for that action.

Dueling Q-Learning

Using Recurrent Networks

2.4 Policy Gradient Techniques

2.4.1 Actor-Critic architectures

The previously introduced technique, a direct adaptation of the Q-learning algorithm [11][12], is a kind of *generalised 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 simulatenously is however generally limited: $argmax_a Q(\cdot, a)$ can only easily be found in settings where the action space 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* uses temporal differences to estimate the Q-value of states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$. If the critic were perfect, it would return the true action-value function of the policy π, $Q^\pi(s, a)$. As that is not the case however, it is in fact similar to the Bellman-function-approximator from previous sections. In contrast to those 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 of our policy. 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 2.14, which averages over the state distribution of our behaviour policy $\beta \neq \pi$.

To learn 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 the performance function 2.14. The idea behind 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.

Deterministic Policy Gradient

The idea in the *Deterministic Policy Gradient* (**DPG**) technique is to use a relation between the gradient of the (deterministic) policy (estimated by the actor), and the gradient of the action-value function Q. The critic estimates the Q-function using a differentiable function approximator, and then the actor updates the *policy* parameters in the direction of the gradient of Q, rather than to maximize it globally.[9]

The deterministic policy gradient theorem, put forward in [10], uses the chain rule to state a relation between the gradient of the performance objective of a policy $J(\pi)$ (see equation 2.14) and the gradients of the policy-function π and the Q-function Q.

$$\begin{aligned}\nabla_\theta J_\beta(\pi_\theta) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) ds \\ &= \mathbb{E}_{s\sim\rho^\beta}\Big[\nabla_\theta \pi_\theta Q^\pi\big(s, \pi_\theta(s)\big)\Big] \\ &= \mathbb{E}_{s\sim\rho^\beta}\Big[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)\Big|_{a=\pi_\theta(s)}\Big]\end{aligned} \tag{2.17}$$

This shows, that the gradient of the performance-objective of the policy with respect to its parameters (this is what we want to maximize) corresponds oth the gradient of the policy with respect to its weights times the gradient of the Q-function w.r.t. the actions.

In practice, the true function $Q^\pi(s,a)$ is unkown and must be estimated. For that, the critic uses Q-learning as explained in the sections above. It is however important, that the the approximation $\hat{Q}\pi(s,a)$ is *compatible*, preserving the true gradient $\nabla_a Q^\pi(s,a) \approx \nabla_a \hat{Q}^\pi(s,a)$. This is the case when its gradient w.r.t. its weights is orthogonal to the gradient w.r.t. its actions, and the critic minimizes the mean-squared error between the gradient of Q^π and $\hat{Q}^\pi$. Both conditions are however approximately fulfilled when using a critic that finds $Q^\pi(s,a) \approx \hat{Q}^\pi(s,a)$.

What this results in, is a *compatible off-policy deterministic actor critic* algorithm. In the first step of this algorithm, the critic calculates the temporal difference error to update its own parameters like in previous sections, and then the actor updates its

[9] *"The critic estimates the action-value function while the actor ascends the gradient of the action-value-function"* (quote [10])

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

$$TD_t = r_t + \gamma Q^w(s_{t+1}, \pi_\theta(s_{t+1})) - Q^w(s_t, a_t) \tag{2.18}$$
$$w_{t+1} = w_t + \alpha_w * TD_t * \nabla_w Q^w(s_t, a_t) \tag{2.19}$$
$$\theta_{i+1} = \theta_t + \alpha_\theta * \nabla_\theta \pi_\theta(s_t) \nabla_a Q^w(s_t, a_t)\big|_{a=\pi_\theta(s)} \tag{2.20}$$

When updating the policy, its parameters θ_{i+1} are updated in proportion to the gradient $\nabla_\theta J(\pi_\theta)$. Because of the deterministic policy gradient theorem 2.17, this

As however ach state suggests a different direciton, one must average by taking the expectation w.r.t. the state distribution $\rho^\pi(s)$:

$$\theta_{i+1} = \theta_i + \alpha * \mathbb{E}_{s\sim\rho^{\pi_i}}\left[\nabla_\theta J(\pi_\theta)\right] \tag{2.21}$$

Using the chain rule, this can be decomposed into the gradient of the action-value w.r.t. the actions and the gradient of the policy w.r.t. the policy parameters:

$$\theta_{i+1} = \theta_i + \alpha * \mathbb{E}_{s\sim\rho^{\pi_i}}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^{\pi_i}(s, a)\Big|_{a=\pi_\theta(s)}\right] \tag{2.22}$$

Silver et al [10] proved that it is not necessary to compute the gradient of the state distribution, and that this [line above] update follows precisely the gradient of the performance objective:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim\rho^{\pi_i}}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^{\pi}(s, a)\Big|_{a=\pi_\theta(s)}\right] \tag{2.23}$$

They say themselves that these algorithms may have convergence issues in practice, due both to bias introduced by the fucntion approximator, as well as instablilities caused by off-policy learning.

Off-policy deterministic actor-critic now learns a deterministic target policy $\pi(s)$ from trajectories generated by an arbitrary stochastic policy, $\beta(a|s) = \mathbb{P}[a|s]$. Now, the performance objective is the value function of the target policy Now we re-define the performance objective again by the value function of the target policy, averaged over the state distribution of the behaviour policy:

$$J_\beta(\pi_\theta) = \mathbb{E}_{s\sim\rho^\beta}\left[V^\pi_(s\right) \tag{2.24}$$
$$= \mathbb{E}_{s\sim\rho^\beta}\left[Q^\pi(s, \pi(s))\right] \tag{2.25}$$

OOOKAY; lets try this again. As before, we want to find a policy maximizing some measure of discounted future rewards, for which we will again consider the Value of a state, as defined in 2.1. A performance objective of our policy is now the value of the state, according to our policy. This performance objective depends however on the actual state distribution (which is, as defined, indeterministic), as well as the action distribution of our policy, which may well be stochastic. In the following, we will consider only deterministic policies. While that takes the necessity of finding the exptected value of that distribution, it only allows for off-policy algorithms, where the distribution of the states depends on another, stochastic, policy than the one we actually learn. To get the actual performance objective, we must thus average

the known value function over the state distribution of our behaviour policy $\beta \neq \pi$:

$$J_\beta(\pi_\theta) = \int_S \rho^\beta(s) V^\pi(s) ds \tag{2.26}$$
$$= \mathbb{E}_{s\sim\rho^\beta}\left[V^\pi(s)\right] \tag{2.27}$$
$$= \mathbb{E}_{s\sim\rho^\beta}\left[Q^\pi(s,\pi(s))\right] \tag{2.28}$$

The gradient of that is now

$$\nabla_\theta J_\beta(\pi_\theta) \approx \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)\big|_{a=\pi_\theta(s)}\right] \tag{2.29}$$

because of the deterministic policy gradient theorem put forward in [10]. What they then do is to develop an actor-critic algorithm that updates the policy in the direction of that. Again, they replace $Q^\pi(s,a)$ by $Q^w(s,a)$. A critic estimates this value function, off-policy from trajectroeies generated by $\beta(a|s)$. Then, in the update, they calculate....

$$TD_i = r_t + \gamma Q^w(s_{t+1}, \pi_\theta(s_{t+1})) - Q^w(s_t, a_t) \tag{2.30}$$
$$w_{i+1} = w_t + \alpha_w TD_i \nabla_w Q^w(s_t, a_t) \tag{2.31}$$
$$\theta_{i+1} = \theta_i + \alpha_\theta \nabla_\theta \pi_\theta(s_t) \nabla_a Q^w(s_t, a_t)|_{a=\pi_\theta(s)} \tag{2.32}$$

However, it must hold that $\nabla_a Q^\pi(s,a)$ can be replaced by $\nabla_a Q^w(s,a)$ without affecting the policy gradient 2.29. This is the case when the gradients are orthogonal, and w minimizes $MSE(\theta, w)$. Silver et al [10] suggest to do that via linear regression, however of course we will use ANNs (that then minimize the Mean-squared projected Bellman error by stochatic gradient descent)

What that results to is COPDAC-Q (compatible off-policy deterministic actor critic q learner)

DPG is more data efficient than normal PG, as it doesnt need to take the integral of the actionspace of the policy

WANN BAU ICH DAS BILD EIN????

[analogous, in a policy gradient context, to Q-learnign: it learns a determinsitic greedy policy, off-policy, while exectuging a noisy version of that]

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**) is to combine the ideas of the DQN (section 2.3.1) with the architecture put forward by Silver et al. [10] revolving around the deterministic policy gradient. As for that, they also use parameterized deterministic actor function $\pi(s) = a$ as well as critic function $Q(s,a)$.

The update of the critic is performed analogous to the Q-value approximator in the Deep-Q-Network architecture. $\langle s_t, a_t, r_t, s_{t+1}\rangle$-samples are sampled from a replay memory of limited size, to then perform Q-learnig via temporal differences (see 2.15 and 2.16). For that, it is also required to use target networks with the previously mentioned soft updates. Target networks are necessary for both the policy approximator as well as the Q-value approximator [SEE ALGORITHM BELOW].

in combination, that leads to the following update step: The critic minimizes the loss

$$Loss := \frac{1}{N} \sum_i \Big(\big(r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{Q'} \big) - Q(s_i, a_i|\theta^Q) \Big)^2 \tag{2.33}$$

And the actor updates its plicy using the sampled policy gradient:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_i \nabla_a Q(s_i, \pi(s_i)|\theta^Q) \nabla_{\theta^\pi} \pi(s_i|\theta^\pi) \tag{2.34}$$

How that looks in an actual Tensorflow implementation can be seen later in this thesis, the corresponding pseudocode is to be found in [5].

Dabieschreiben dass das halt so macht wegen den minibatches? arrg

Using normal Q-learning for the actor leads to the same overfitting problems as it does in DQN, if not using DoubleQ. [Q-learning is prone to overestimatng values] -> it did so in ddpg, however they say it still performed well

[hier schon auf wawrzynksi sowie wawrzynksi&tanwani2013 eingehen]

[WO im tatsächlichem Netz die actions bei actor und critic reingehen]

2.4.2 Exploration techniques

As mentioned in the beginning of this chapter, I only considered deterministic policies so far: $\pi(s) = a$. In practice however, using purely deterministic policies leads to a complete lack of *exploration* of the state space S of the MDP. Once the agent found a path to a terminal state, it will continue *exploiting* this path. In order to explore the full state space, in fact a stochastic policy is necessary.

Blablabla, dass das der Grund ist warum alle unsere algorithmen bisher off-policy waren - in DQN steht "wir lernen über die greedy argmax policy while performing epsilon-greedy", und in DPG steht auch "the basic idea is to choose actions according to a stochastic behaviour policy (to ensure adequate exploration), but to learn about a deterministic target policy"

von DDPG: "An advantage of off-policy algorithms such das DDPG is that we can trat the problem of exploration independently from the learning algorithm" [VIELLEICHT AM ANFANG SCHON???]

Chapter 3

Related work

3.1 Reinforcement Learning Frameworks

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

3.2 Self-driving cars

Nvidias deep-drive RRT* Tensorkart Tesla Lidar hier in den fußnoten die ganzen non-scientific quellen wie tensorkart undso

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].

4.1 Design choices

4.1.1 The game as a reinforcement learning problem

-ungefähres UML-diagramm

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

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

In the original DDPG-algorithm [5], the authors used *batch normalization* [3] 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)[1]. 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 [2], I left out batch normalization for good.

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

Appendix A

Comparison Pseudocode & Python-code

A.1 DQN

```
class Agent():
 def __init__(self, inputsize):
  self.inputsize = inputsize
  self.model = DDDQN_model
  self.memory = Memory(10000, self)  #for definition see code
  self.action_repeat = 4
  self.update_frequency = 4
  self.batch_size = 32
  self.replaystartsize = 1000
  self.epsilon = 0.05
  self.last_action = None
  self.repeated_action_for = self.action_repeat

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

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

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

```
    self.memory.append([s,a,r,s2,t])



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



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

```
Initialize replay memory D to capacity N

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

Initialize target action-value function Q(s,a;θ⁻) with weights θ⁻ = θ
For episode = 1,M do
 Initialize sequence s_1 = {x_1} and preprocessed sequence φ_1 = φ(s_1)
 For 1 = 1,T do
  With probability ϵ select random action a_t
  otherwise select a_t = argmax_a Q(φ(s_t),a;θ)

  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 φ_{t+1} = φ(s_{t+1})
  Store transition (φ_t,a_t,r_t,φ_{t+1}) in D

  Sample random minibatch of transitions (φ_j,a_j,r_j,φ_{j+1}) from D
  Define a^max(φ_{j+1};θ) = argmax_{a'} Q(φ_{j+1},a';θ)
  Define Q^{j+1} = Q(φ_{j+1},a^max(φ_{t+1};θ);θ⁻)

  If episode terminates at step j+1 then set y_j = r_j,
      ➥ Otherwise set y_j = r_j + γ * Q^{j+1}
  Perform a gradient descent step on (y_j - Q(φ_j,a_j;θ))^2 with respect
      ➥ to the network parameters θ
  Update target network: θ⁻ ← τ * θ + (1 - τ)θ⁻
 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

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

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

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

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

```
 return tf.layers.dense(x, units,activation=activation, kernel_initializer=tf.
     ➥ random_uniform_initializer(-minmax, minmax), kernel_regularizer=decay and tf.contrib.layers.
     ➥ l2_regularizer(1e-2))


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


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

```
Randomly initialize critic network Q(s,a|θ^Q) with weights θ^Q
 and actor π(s|θ^π) with weights θ^π.
Initialize target network Q' weights θ^Q' ← θ^Q
 and π' with weights θ^π' ← θ^π
Initialize replay buffer R
for episode = 1, M do
 Initialize a random process 𝒩 for action exploration
 Receive initial observation state s_1
 for t = 1, T do
  Select action a_t = π(s_t|θ^π) + 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 = π'(s_{i+1}|θ^π')
  targetCriticQ = Q'(s_{i+1},targetActorAction|θ^Q')
  Set y_i = r_i + γ*targetCriticQ  #only in nonterminal states

  Update critic by minimizing the loss: L = 1/N Σ_i (y_i − Q(s_i, a_i|θ^Q))^2
  Find the sampled policy gradient:
    onlineActorActions = π(s_i|θ^π)
    ∇_{θ^π} J ≈ 1/N Σ_i ∇_a Q(s_i,onlineActorActions|θ^Q) ∇_{θ^π} π(s_i|θ^π)
  Update the actor policy using the sampled policy gradient
  Update the target networks:
    θ^Q' ← τ * θ^Q + (1 − τ)θ^Q'
    θ^π' ← τ * θ^Q + (1 − τ)θ^π'
 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] Richard Bellman. *Dynamic Programming*. Princeton University Press. ISBN: 978-0-691-14668-3. URL: http://press.princeton.edu/titles/9234.html.

[3] 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).

[4] 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).

[5] 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:1509.0 [cs, stat]* (Sept. 2015). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971 (visited on 08/12/2017).

[6] 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).

[7] 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.

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

[9] 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).

[10] 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).

[11] 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.

[12] 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).

[13] 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).

[14] 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

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

signature

city, date