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 13, 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 13, 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

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:

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

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, it can always only be talked about the expected value 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. An episode thus consists for the agent of a sequence of observations, actions and rewards ($S \times A \times \mathbb{R}$) 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

[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 $o(s_e)$

[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.

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 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 from this state on:

$$V^{\pi}(s_t) := \mathbb{E}_S\Big[\sum_{t'=t}^{t_t}(\gamma^{t'-t} * r_{t'}^{\pi})\Big] \tag{2.1}$$

Using the discounted future reward is useful because in an indeterministic environment it gets less likely that the agent actually reaches this state, and to make the agents perform good actions as quickly as possible.

The actual, underlying Value of a state s is defined as the value of the state when using the best possible policy, which corresponds to the maximally archievable reward starting in state s:

$$V^{*}(s_t) := max_{\pi}V^{\pi}(s_t) \tag{2.2}$$

While *passive reinforcement learning* simply tries to learn the Value-function 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 upcoming state: $\pi^* := argmax_{\pi}V^{\pi}(s)\forall s \in S$. Knowing what an optimal policy does, and using 2.1 and 2.2, it is possible to re-write the definition of the value of a state recursively as

$$V^{*}(s_t) = max_{\pi}\mathbb{E}_S\Big[\big(\sum_{t'=t}^{t_t}(\gamma^{t'-t} * r_{t'}^{\pi})\big)\Big] \tag{2.3}$$

$$= max_{\pi}\mathbb{E}_S\big[(r_t^{\pi} + \gamma * V^{\pi}(s_{t+1}^{\pi}))\big] \tag{2.4}$$

This is known as the *Bellman Equation*, which allowed for the birth of dynamic programming. It rewrites the value of the 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.

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 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.

$$Q^{\pi}(s_t, a_t) := \mathbb{E}_S\big[r_t^{a_t} + \gamma * V^{\pi}(s_{t+1}^{a_t})\big] \tag{2.5}$$

With the optimal, maximally archivable action-value Q^* being respectively

$$Q^{*}(s_t, a_t) = \mathbb{E}_S\big[r_t^{a_t} + \gamma * V^{*}(s_{t+1}^{a_t})\big] \tag{2.6}$$

$$= max_{\pi}\mathbb{E}_S\big[(r_t^{a_t} + \gamma * V^{\pi}(s_{t+1}^{a_t}))\big] \tag{2.7}$$

For the Q-value, the Bellman equation holds as well: If the optimal value $Q^*(s_{t+1}, a_{t+1})$ was known for all possible actions, then the optimal action at time t is the one maximizing the sum of immediate reward and corresponding Q-value[3]:

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

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

$$V(s_t) = max_{a_t} Q(s_t, a_t) \tag{2.9}$$

When an agent knows the Q-value for each action of a state, it can easily infer the optimal action in state s_t as $a_t^* := argmax_{a_t}(Q(s_t, a_t))$ and thus the optimal policy π^*, guaranteeing maximum future reward at every state. The goal of a model-free RL agent is thus to get a maximally precise estimate of Q^*, yielding maximal reward for every state. For that, it does not need to explicitly learn the reward- and transition function, but instead can model only the Q-function. Its policy is then to simply always take the action yielding the highest value for every state (a *greedy* policy[4]). 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 calculated using a kind of nonlinear function approximator.

Throughout exploration of the environment, the agent collects more information of it, continually updating its estimate Q^π. 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 $\hat{Q}$ of Q^*. The loss of its current estimate could be seen as the squared difference $(\hat{Q}\text{-}Q^*)^2$, however as the agent has no knowledge of Q^*, 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^*. Using the recursive definition of a Q-value given in the Bellman equation 2.8 allows for a technique called *temporal difference learning*[**sutton1988**]: 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 model, the new estimate will be closer to the actual function Q^* than the original value:

$$\begin{aligned}\hat{Q}^\pi(s_t, a_t) &= r_t + \mathbb{E}_S\left[\gamma * max_{a_{t+1}} \hat{Q}^\pi(s_{t+1}, a_{t+1})\right] &(2.10)\\ &\approx r_t + \gamma * r_{t+1} + \mathbb{E}_S\left[\gamma^2 * max_{a_{t+2}} \hat{Q}^\pi(s_{t+2}, a_{t+2})\right] &(2.11)\end{aligned}$$

Keeping in mind that $\hat{Q}^\pi$ is only an estimator of the Q^*-values of the underyling model, it becomes clear that equation 2.11 is closer to the actual Q^*, as it incorporates more information stemming from the model itself.

[3] 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[**Bellman1957**]"

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

In temporal difference learning, the mean-squared error of the *temporal difference* from the Bellman equation, $r_t + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$, gets minimized via iterative approximation. Because the optimal target-values $Q^*(s_{t+1}, a_{t+1})$ are not available, they are substituted with the *more informed guess*, stemming from observations in the environment. 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 $TD := (r_t + \gamma * \hat{Q}_i^\pi(s_{t+1}, a_{t+1})) - \hat{Q}_i^\pi(s_t, a_t)$. This algorithm of calculating the temporal difference error is known as SARSA, and it is an example of *on-policy* learning. In on-policy learning, the agent uses his own policy in every estimate of the Q-value.

Q-learning

In contrast to SARSA stands the *off-policy* algorithm *Q-learning*. This algorithm 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}.[5] As the previous definition of Q-values was only correct in non-terminal states, a case differentiation must be introduced for terminals 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 [4]:

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

The temporal difference error for time t is accordingly defined as

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

A Q-learning agent must thus for his learning step 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). Q-learning is considered an off-policy algorithm, because it learns about the greedy policy $a = argmax_{a'} Q(s, a')$, while not necessarily following it.[6]

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 [9] that for finite-state Markovian problems with nonnegative rewards the update-rule for the Q-table $\hat{Q}(s_t, a_t) \leftarrow r_t^{a_t} + \gamma * \hat{Q}^\pi(s_{t+1}^{a_t}, a_{t+1})$ converges to the optimal Q^*-function, making the greedy policy π^* optimal[7].

As however in practice the problems using a table as the Q-functions are only very limited scenarios, an update rule like this is irrelevant. Instead, a better idea is

[5] A slight deviation from this *double-Q-learning*, an algorithm I will go into detail about lateron.

[6] This is because in its actual policy, the agent includes some way to explore the environment, more on that later.

[7] Of course the agent will need some kind of exploration technique first, more on that later

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 gradient descent, updating the parameters of the Q-function into the direction of the newly acquired knowledge. The L2-Loss at iteration i with model-parameters θ_i is thus defined as the following:

$$L_i(\theta_i) := \mathbb{E}_{s,a,r}\left[\left(y_i(\theta_i) - \hat{Q}_i^{\pi}(s_t, a_t; \theta_i)\right)^2\right] \tag{2.14}$$

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. A special focus must also lie on *Convolutional Neural Networks* (**CNN**s) [10], mainly used in image processing. As mentioned before, it is (in theory) not only possible to use a Q-table to estimate the Q^*-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 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. The update rule in Deep Networks depends on the gradient with respect to its, $\nabla_{\theta_i} L(\theta_i)$. These weights correspond to the parameters of the $\hat{Q}$-function of the agent. While there are attempts to use Artificial Neural Networks for Q-learning as early as 1993[SOURCE], some key components of modern Deep-Q-Networks (**DQN**s) 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 right after the observation occured to the agent. As consecutive steps of MDPs tend to be correlated, the update using gradient descent is prone to oscillation in its result, thus never converging to an optimal Q^*-function. It was not until *Deepmind*'s famous papers in 2013[3] and 2015[4], 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

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

Network, neither only the last temporal difference error TD_t is considered, nor the entire sequence $TD_0, .., TD_{t_t}$. 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 Deepminds 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 function, even though still no formal mathematical proof of that 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[5], 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 (2.14), 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 as used in [4]:

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

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 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 [2]. 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

Dueling Q-Learning

Using Recurrent Networks

2.4 Policy Gradient Techniques

2.4.1 Actor-Critic architectures

All previously introduced techniques are adaptations of the Q-learning algorithm [7],[8]. In Q-learning, the algorithms learns via temporal differences the state-action value Q for the greedy policy $\pi = argmax_a Q(s, a) \forall s \in S$. As purely using this policy hinders the agent from *exploration*, Q-learning algorithms must always be off-policy, and actually follow a slightly different policy than π. 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.

However in a lot of scenarios, the action space is not discrete, but continuous: $A \subseteq \mathbb{R}^n$. In such situations, the approach is to model the policy explicitly, with another function approximator. This gives rise to *actor-critic* architectures. In an actor-critic approach, there are two function approximators: The *critic* uses temporal differences to estimate the Q-value of states $s \in S$ and actions $a \in A$. 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 $a = \pi_\theta(s)$. Note however, that this will (again) lead to the necessesity of off-policy algorithms, as a purely deterministic policy does not allow for adequate exploration of state-space S or action-space A.

Standard actor-critic algorithms necessarily rely on the *policy gradient theorem*, which states a relation between the gradient of the policy and the gradient of the performance function:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi}[\nabla_\theta log \pi_\theta(a|s) \hat{Q}^\pi(s, a)] \tag{2.16}$$

Deterministic Policy Gradient

The idea in the Deterministic Policy Gradient 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 Q-function using a differentiable function approximator, and then updates the *policy* parameters (the ones of the actor) in the dirction of the approximate action-value gradient.[9]

As mentioned at the beginning, the state transition of MDPs is in general indeterministic. Because of that, one can only talk about state distributions, depending on the policy of the agent: $\rho^\pi(s)$. An agent's goal is to gain a policy that can follow the trajectory of that state distribution with the highest *expected* reward. Thus, we can only write the performance ojective J of a policy π as expectation, which is technically the integral over the state- as well as the action-distribution, each depending on the policy.

$$J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi}[R(s,a)]. \tag{2.17}$$

The idea behind policy gradient algorithms is accordingly to adjust the parameters θ of the policy in the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$.

Deep DDP

The *Deep DPG Algorithm* is an off-policy actor-critic, online, active, model-free, deterministic policy gradient algorithm for continous action-spaces.

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"

[9] *"The critic estimates the action-value function while the actor ascends the gradient of the action-value-function"* (cf. [6])

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

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

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

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] 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).

[3] 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).

[4] 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.

[5] 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).

[6] 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).

[7] 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.

[8] 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).

[9] 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).

[10] 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