TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Großer Beleg

Reinforcement-Learning Windturbine Controller

Nico Westerbeck
(Mat.-Nr.: 3951488)

Betreuer: Dr. Dmitrij Schlesinger

Dresden, 30. November 2019

Aufgabenstellung

0.1 Background:

Here at the HFI Experimental fluid mechanics group, we have developed an open source project called QBlade. QBlade is a simulation tool used for testing wind turbines in the hostile environment that they normally operate. We normally tackle problems of aerodynamic or structural optimization but we have also a research focus on the development of the control systems of the wind turbines. We currently have a research effort looking at developing cluster-based controllers building on the work of Professor Bernd Noack who is a guest professor at our group. In the last year or so (Nair, A. G., Yeh, C.- A., Kaiser, E., Noack, B. R., Brunton, S. L., & Taira, K. (2018). Cluster-based feedback control of turbulent post-stall separated flows. Journal of Physics Fluid Dynamics, (M), 1-32. Retrieved from http://arxiv.org/abs/1809.07220). AI projects such as openAI have enabled he rapid development of neural network in the field of control using reinforcement learning. The goal of this project is to use QBlade as a wind turbine simulator and attempt to control the pitch and rotor speed in a way that doesn't cause the wind turbine to shatter but instead to yield energy, i.e. reward and death condition. This first stage of work should be considered as exploratory but will hopefully open up avenues of controlling active flow control elements such as flaps.

0.2 Tasks

The major tasks of the project are as follows:

- Build up and interface between QBlade and python the model code so that an external code can run as a controller within a QBlade simulation.
- Gain a rough understanding of the mechanics of wind turbines and their controllers.
- Research reinforcement learning methods suitable for use as a windturbine controller and perform a literature review on these approaches.
- Create a reinforcement learning agent which uses the Qblade interface for controllers to control a windturbine.
  - Inputs to the agent could be defined by the standartized controller input format to Nordex turbines, which consists of 39 real-valued sensor-inputs. However, initial tests can be conducted with whichever inputs are easiest to tackle. If required, further hidden state from the simulation can be exported to enrich data quality. If aiming for industrial quality, more inputs

    and also sensor faults could be optionally incorporated.

  - Outputs are in a minimum version pitch angles for the 3 blades and turbine torque. Optionally the agent should be able to control active element such as flaps on the blades.

- Optimize the agent to deliver maximum energy yield.
- Optimize under respect of certain boundary conditions (maximum pitch acceleration, maximum power, maximum blade load, blade touching the tower) and optionally other boundary conditions like long term turbine wear.
- If necessary for the training process, scale the simulation to run at a larger scale.
- Implement and attempt to get the agent to perform something close to sensible control of the wind turbine. Optionally evaluate the results against existing controllers and try to outperform them.
- Optionally, create a conference paper, poster or blog post etc.. on the results.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

Reinforcement-Learning Windturbine Controller

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 30. November 2019

Nico Westerbeck

Inhaltsverzeichnis

0.1 Background: 2
0.2 Tasks 2

1 Abstract **3**

2 Intro **4**

3 Background **5**
3.1 Windturbine control 5
3.2 QBlade 6
3.3 Reinforcement learning 8
3.3.1 Environment assumptions 8
3.3.2 Definitions 9
3.3.3 Q-Learning 11
3.3.4 Policy Gradients 12
3.3.5 Deterministic Policy Gradients 14
3.3.6 DDPG 14
3.4 RL on windturbines 14

4 Experimentation **15**
4.1 Gym Experiments 15
4.2 Starting with qblade 15
4.3 Designing reward functions 16
4.4 Aiding exploration 18
4.4.1 Action Noise 18
4.4.2 Random exploration 18
4.4.3 Parameter noise 19
4.5 Zero Gradients 19
4.5.1 Simplifying the architecture 19

4.5.2 Normalization 19
4.6 High action gradients 20
4.6.1 Gradient actionspace 20
4.6.2 Feeding past timesteps 20
4.6.3 Clipping action gradients 21
4.6.4 Pretraining the policy 21
4.7 Other improvements 21
4.7.1 Prioritized experience replay 21
4.7.2 Data augmentation 22
4.8 Exploding Q-Loss 22
4.8.1 Huber loss 22
4.8.2 Double critic 23
4.8.3 Batch normalization 23
4.9 Automatized hyperparameter search 23

5 Evaluation 24

5.1 First working version 24
5.2 HParam tuning 24
5.3 Comparison to industry controllers 25

6 Future work 26

6.1 Reward functions 26
6.2 Switch reward functions during training 26
6.3 Expert policy training 26
6.4 Expert policy in instable conditions 27
6.5 Active control elements 27

7 Conclusion 28

Literaturverzeichnis 29

1 Abstract

Small

2 Intro

Recent advancements in reinforcment learning have managed to tackle more and more complex problems, like StarCraft or Go. The range of topics, RL is applicable to, increases. However, so far, there hasn't been an effort to control a windturbine with a reinforcement learning algorithm. Current state of the art linear controllers are performing well in maintaining turbine control, and the optimization margin for these is small, as they achieve the theoretical maximum energy output up to rated windspeed and manage to keep the turbine intact above rated windspeeds.

Trying it nontheless can still provide for some foundational knowledge for future developments. Park control becomes a bigger topic in wind-energy, where controlling all windturbines of a park together can increase overall energy yield or protect windturbines further down the windstream from wakes by turbines up front. C

3 Background

This section aims at facilitating the background knowledge necessary for understanding this paper. We fill first introduce windturbines and common terms around that

3.1 Windturbine control

There are two mayor types of wind turbine designs, Horizontal Axis Wind Turbine (HAWT) and Vertical Axis Wind Turbine (VAWT), which differ by their rotation axis. In this work, we will only look at HAWT and all mentions of windturbines mean HAWT type turbines. Such a turbine is made up by 3 big components, a tower on which a nacelle is rested which itself has a rotor in front. The joint between tower and nacelle allows for rotation to turn the rotor into the wind. Except for recent ideas [HLD], the best value for the rotational direction was to always yaw the rotor directly into the wind, and for the sake of simplicity we will omit this control parameter from our simulation and leave it at 0 degrees with wind facing straight onto the rotor. In the nacelle there is, most importantly, an electrical generator, which can be given a specific torque. It will then slow down the rotor with that torque and at the same time generate energy proportional to rotor speed and torque. This torque is one of the two more important control parameters, together with blade pitch. The blades are designed to operate on maximal aerodynamic efficiency when they are pitched at 0 degrees, increasing the pitch angle will (except for possible slight improvements in the first few degrees) reduce aerodynamic efficiency of the rotor.

As described in [BJSB, sec 8.3], the normal controller design for windturbines divides the operation of a turbine into two parts, below rated and above rated. Rated is a windspeed, in which the turbine gives maximum energy on highest aerodynamic effiency. At this point, the generator runs on its maximum torque and the blades are pitched to the optimum angle of 0 degrees. At this point, the tip-speed-ratio, which is a good indicator for loads on the blades is at maximum too. Below that windspeed, the rotor generates less rotational torque than the maximum generator torque, thus the generator torque needs to be reduced to not slow the rotor down too much. Slowing the rotor down complicates the energy generation process, and though variable-speed wind turbines allow for some play, all windturbines are limited to a certain operational range. Fixed-speed wind turbines can only run on one single rotational speed and thus need to change the torque more agressively. Above rated windspeed, the blades need to be pitched out to stop the rotor from spinning at faster speeds than what the turbine was designed for. The generator torque can be kept at maximum in this area, but to protect the turbine from damage, aerodynamic efficiency of the rotor is reduced by turning the blades along their long axis and thus moving their angle of attack against the incoming air into a less optimal range.

Additionally to this trivial control, optimizations to reduce vibrations and stress on the structures are implemented. The blades can be pitched individually and quickly enough to allow for different blade pitches during a single rotation of the rotor, which can be used to reduce the turbulence that hits the tower or to account for different windspeeds closer to the ground and further up in the air. Also, both generator torque and blade pitch can be used to counteract natural resonant frequencies of the structure, reducing material stress through extensive swinging of structural parts.

Usually this control is implemented by two PID controllers, which are hand-designed as described in [BJSB, sec 8.4]. Below rated windspeed, the controller for torque is active, above rated the one for pitch. The parameters to those controllers can be calculated according to the laws of control theory, and the resulting controllers work reasonably close to theoretical maximum. Still, in this work, we are trying to replace these hand-designed controllers with a reinforcement-learning algorithm.

3.2 QBlade

Our data source in this work is the open-source simulation tool QBlade [MWP$^+$] developed at TU Berlin. QBlade is an accessible and performant tool with the primary purpose of designing and simulating wind turbines in a graphical user interface. Its simulation results are on par with current state-of-the-art proprietary simulation tools and it yields good computational efficiency. It uses an algorithm which approximates the 3D blade structures with 2D structures and corrects the results through several error terms. Though a full CFD simulation would yield higher accuracy, our computational resources don't allow for that. To be used in reinforcement learning, we designed an interface, over which the QBlade simulation can be embedded into an external environment. Most machine-learning frameworks are written in python, so we decided to compile QBlade into library format and expose the most fundamental functions to allow it to communicate with any programming language that can load libraries. As python has a ctypes interface to load c-code, we could link a python agent to the QBlade C++ environment.

QBlade allows different simulation scenarios, for our testing we decided to only use the NREL 5MW [JBMS] turbine with the default structural model and using all implemented correction mechanisms to achieve the most realistic data possible. As reference data to this turbine is easily available and publicly published by NREL, this allows us for good cross-validation.

The interface is made of 7 functions, which allow for loading a project, resetting the simulation, setting controller inputs and getting environment measurements and advancing the simulation a timestep. The observations returned in getControlVars match those that are visible to standart industry controllers, which in turn are modelled after what can be measured on a real windturbine. Concretely, we measure:

```
* rotational speed [rad/s]
* power [kW]
* wind velocity [m/s]
* yaw angle [deg]
* pitch blade 1 [deg]
* pitch blade 2 [deg]
* pitch blade 3 [deg]
* tower top bending local x [Nm]
* tower top bending local y [Nm]
* tower top bending local z [Nm]
* out-of-plane bending blade 1 [Nm]
* out-of-plane bending blade 2 [Nm]
* out-of-plane bending blade 3 [Nm]
* in-plane bending blade 1 [Nm]
* in-plane bending blade 2 [Nm]
* in-plane bending blade 3 [Nm]
* out-of-plane tip deflection blade 1 [m]
* out-of-plane tip deflection blade 2 [m]
* out-of-plane tip deflection blade 3 [m]
* in-plane tip deflection blade 1 [m]
* in-plane tip deflection blade 2 [m]
* in-plane tip deflection blade 3 [m]
* current time [s]
```

Because it has no semantic meaning, we mask away current time from our reinforcement learning agent. The inputs we can give to the turbine are:

```
* generator torque [Nm]
* yaw angle [deg]
```

```
⋆ pitch blade 1 [deg]
⋆ pitch blade 2 [deg]
⋆ pitch blade 3 [deg]
```

After a while of experimenting with the unrestricted simulation, we added some bounds to our control parameters. In our final version, we had the following restrictions: As mentioned before, we set the yaw angle to always 0 degrees, as in reality controlling the orientation of the nacelle is trivial and in our simulations, wind will always come from the front. The generator torque on our NREL 5MW turbine has a sensible range of 0 to 4×10^6 Nm. Because changing torque from zero to maximum in one timestep is not realistic for a real turbine, we restrict it to move a maximum of 3×10^5 Nm per one timestep of 0.1 seconds. If our agent chooses a value outside of that area, we set it to the closest value inside of that area. Blades can be pitched between 0 and 90 degrees, 0 being not pitched out at all and operating at maximum efficiency and 90 resulting in no aerodynamic torque from the rotor whatsoever. A sensible pitch motor can only turn the blades at a limited speed, we assume a limit of 5 degrees per second and again set controller inputs to a sensible value inside that. The simulation theoretically accepts a full pitch change in one timestep, however as inertia on the blade is so high for such a change, the blades break instantly and the rest of the simulation needs to be reset.

QBlade simulates blade damage, also high rotational speeds cause blades to fall off. QBlade is not validated to realistically simulate these extreme conditions and thus posed a problem to our controllers. In the course of our experiments, we decided to reset the simulation at an rotational speed of 4 rad/s, as our turbine operates at ca 0.8 rad/s. Also, high generator torque inputs cause the simulation to rotate the rotor backwards with a negative energy yield, effectively creating a multimillion-dollar leafblower. As this is neither a realistic scenario, we also reset the simulation at -0.5 rad/s.

3.3 Reinforcement learning

This section should aim to give a brief overview into Reinforcement Learning assumptions and derive briefly a couple of methods until in the end, we will describe Deep Deterministic Policy Gradients (DD-PG), the method we utilized to control our virtual windturbine.

3.3.1 Environment assumptions

At first, we need to do some assumptions on our environment ε. The concept environment encapsulates the reality or simulation that the reinforcement learning algorithm works in. This could be an actual cyber-physical system or a simulation. The first assumption to this environment is that this system operates over time and we can discretize time into timesteps. Our environment here is the simulated wind turbine, but it could be a computer game [LHP$^+$], a physics simulation[BCP$^+$] or even an actual existing wind turbine [KJT]. In every timestep, the environment supplies us with an observation x and a scalar reward r for taking an action a in that state. In theory, the full trajectory $(x_0, a_0, ..., x_t)$ could be needed to describe the full state of the simulation at timestep t s_t. However, the algorithms we looked at assume that the state progressions can be modelled as a Markov Decision Process (MDP) of observations, as in that what will happen at timestep t+1 is only dependent on the observation at timestep t, not also on for example t-10. In MDP, the state transition probability $p(x_{t+1}|x_t, a_t)$ is fully descriptive, as in it is equal to $p(x_{t+1}|x_0, a_0, ..., x_t, a_t$. Because we assume the observation x to be fully descriptive, we will use the term state as equivalent to observation $s_t = x_t$. In reality, the observed state doesn't have to represent the entirety of the state of the simulation, in fact it usually is only a small subset. For our wind turbine example, the observed state is limited to what can be measured with sensors on a wind turbine, in an arcade game it could be the pixel output or in a physical simulation it could be some key values in the simulation.

The reward r is a scalar value which judges the action taken in the current state of the simulation. A higher reward means the action is better than that of a lower reward. Sometimes, the reward function is obvious from the system, as in an arcade game it would surely be the score or in chess it could be 1 for win, -1 for lose and 0 for not decided yet. In both cases, sometimes a certain timespan passes before a reward for an action kicks in, which is also to be learned. In our paper, we will dedicate a section on how we designed our reward function, whereas in general reinforcement learning theory, it is assumed to exist and be supplied by the environment.

In every timestep, the agent supplies an action a. This action could be button presses in an arcade game or blade pitch in a windturbine simulation. The mission of the agent is to pick actions which maximize cumulative reward. In other words, it should steer the environment to achieve best performance.

Additionally to the state and reward, an environment can optionally send out a done signal d, which indicates a state in which further simulation is not possible and the environment wants to be reset to an initial state. In a chess game, this would be a loss or a win, and with a windturbine that could for example be fatal structural damage.

3.3.2 Definitions

TODO HUGE MESS FOLLOWING Before diving into the algorithm, we would define some parameters. From our environment, we have

- s_t a single state at timestep t
- r_t the reward at that timestep t
- a_t an action taken at timestep t
- ε the unknown transition probability matrix of the environment
- d_t a flag (0 or 1) which indicates a terminal condition in the environment
- $(s_t, a_t, r_t, d_t, s_{t+1})$ a state transition, going from state s_t, taking action a_t, getting a reward for that action r_t and a terminal flag d_t and reaching state s_{t+1}

We furthermore define $\pi_{prob} : S \rightarrow P(A)$ a probablistic policy that returns a probability distribution over actions for given state and $\pi : S \rightarrow A$ a deterministic policy that returns a single action for a given state. We will use deterministic policies in all of the paper except for the Q-Learning explanation, which is easier to grasp with probablistic policies.

$R_\pi(t) = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ A discounted return, summing up all returns from timestep t up to some point in the far future T while applying a discounting term $\gamma \in [0, 1]$. The discounting gives less significance for rewards far in the future than rewards close to now. Also, as we need to assume state/action transitions in the future, the return is dependend upon a policy. If we wanted to compute this return, we would need to calculate the expectation over a construct made of all possible trajectories $(s_0, a_0 \leftarrow \pi_{prob}(s_0), s_1, ...)$, combined with the initial state distribution $p(s_0)$ and the environment Markov Chain $p(s_t|s_{t-1}, a_{t-1}$. As we don't know the environment probability distributions, we need to find another way.

- $(s_0, a_0, s_1, a_1, ..., a_{t-1}, s_t) \rightarrow \tau$ a trajectory of many states and their actions.
- $R(\tau) = \sum_t r_t$ an undiscounted return from a trajectory

$V(s)$ is the value of a state defined as accumulating all rewards from this step infinitely into the future: $V(s) = \sum_t r_t$. There are a couple of problems with this function, at first summing infinitely into the future might yield infinite or otherwise unusable results, which is why we introduce a discounting factor γ in $V(s) = \sum_t r_t * \gamma^t$. And furthermore and more obviously, inputting a state to the function, we can not

HUGE MESS OVER

Some terms which we use

- **Policy** A function that maps from states to actions. There are deterministic policies $\mu : S- > A$ and stochastic policies $\pi : S- > P(A)$.
- **Return** A return with respect of a policy and a starting state s gives the accumulative reward of following that policy from state s until the future. $R_\pi(s_t) = \mathbb{E}_{s_i \sim \varepsilon, a_i \sim \pi, i > t} r(s_i, a_i)$. As we have a stochastic environment and policy, we have to add an expectation term over actions taken

and states resulting from those actions. It is the expected accumulative reward of following that policy. Furthermore a return can be discounted or undiscounted. Discounting means we multiply a discounting factor $\gamma \in [0,1]$ to rewards in the fashion of $r_0\gamma^0 + r_1\gamma^1 + r_2\gamma^2...$, which results in distant future rewards being weighted less than closer ones.

- **Optimal policy** The theoretical construct of a policy that always takes the best action with respect to accumulative reward. Though we rarely know this policy, there must always be at least one policy which gives the highest returns of all possible policies
- **Q-Value** The return of taking action a in state s plus the return following a policy after that $Q_\pi(s,a) = r(s,a) + \mathbb{E}_{s' \sim \varepsilon} R_\pi(s')$
- **Trajectory** A trajectory is a sequence of states and actions $(s_0, a_0, s_1, ..., a_{i-1}, s_i)$

3.3.3 Q-Learning

At first we will present a very basic algorithm for a machine learning agent, which is based on the basic Bellman-Equation [Bel]. It is proven, that

$$Q*(s,a) = \mathbb{E}_{s' \sim \varepsilon | s,a}[r(s,a) + \gamma \max_{a'} Q*(s',a')] \tag{3.1}$$

converges to the optimal solution when using iterative value updates. I will quickly explain this Bellman Equation. This equation uses the notion of a state-value function, also called Q function. In the Bellman Equation, it describes the total accumulated return until infinity taking an action a in state s, receiving reward r, ending up in state s' and then following the optimal policy afterwards. γ acts as a discounting factor just as in our return function. It is even possible to implement an algorithm from this. We could create a lookup table for all s and a, initialize it all zero and whenever observing a state transition, we could update the table at that state and action after the above notation.

With the help of this function, we can compare different actions in our current state. However, this table is only computationally feasible when we have small finite state and action spaces. If any of the two spaces is big or even continuous, we have to replace the table by an estimator for Q.

Even with an estimated Q, though no longer guaranteed, we can still converge towards the optimal solution. For training an estimator, we need a loss which we can derive. For this, we define our learning target for learning step i as

$$y_i = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_i)] \tag{3.2}$$

For the moment, let's assume we can still calculate $\max_{a'}$. In fact, in a small finite action space we could just try all possible values of a. Our y is then the current reward added to the highest possible rewards in the future, with other words exactly what we want to have as Q. We use the parameters θ_i in the target, so in theory, when deriving a later loss, we would also have to derive over our targets. However, in Q learning, this is commonly ignored.

Now, on our learning target y, we can define a loss as

$$L(\theta_i) = \mathbb{E}_{s \sim \varepsilon}[(y_i - Q(s, a; \theta_i))^2] \tag{3.3}$$

This already looks close to some squared error term. To calculate the expectation value over the environment ε, we need to do some form of Monte-Carlo experiment, in which we approximate that expectation value by repeatedly drawing samples from the underlying distribution. Luckily, we can easily draw samples from our environment, and we can even reuse old samples. Thus we store all samples in a replay buffer and uniformly draw from it to resemble a Monte-Carlo experiment.

Deriving for θ_i and applying the chain rule, we get the loss derivative

$$\Delta_{\theta_i} L(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i))\Delta_{\theta_i} Q(s, a; \theta_i)] \tag{3.4}$$

Using our Monte-Carlo like batch from the replay buffer, we end up with stochastic gradient descent.

Thus we just derived a method to train a Q function under the assumption of a policy. In [MKS$^+$], the authors assumed a finite state space and could use the greedy policy $argmax_a Q(s', a)$ by searching the entire action space. We are presented with an infinite action space though, which is why we can't implement a greedy policy without high computational expense. So, for our work, we have to also learn a policy and not just a Q estimator. Luckily, our Q learning rule also works for other policies than the optimal policy, so let's try to derive a way to learn a policy.

3.3.4 Policy Gradients

We will at first give an intuitive explanation of a basic algorithm which forms the basis for many modern reinforcement learning algorithms, including the one we chose. The algorithm is called REINFORCE and was originally presented by [Wil]. Then we will add a Q term to it according to [SMSM] and is called Policy Gradients (PG). [**?**] generalized that term and made an off-policy version of policy gradients. That version was then used by [SLH$^+$] and the stochastic policy was replaced by a deterministic one, the algorithm was called Deterministic Policy Gradients (DPG). This algorithm finally lays the basis for DDPG by [LHP$^+$], which is what we used until finally optimizing it towards Twin Delayed Deep Deterministic Policy Gradients (TD3) [FHM].

Let's assume we have a probablistic policy $\pi : S \rightarrow P(A)$ which is parametrized by θ. An intuitive way of improving this policy could be to increase the gradient of the best possible action a^* for a state s:

$$\theta_{i+1} = \theta_i + \alpha \Delta \pi_{\theta_i}(a^*|s) \tag{3.5}$$

We could do this for all states we see and would end up with an iterative way of improving our policy. However, we might not know the best action for any state. Luckily, we derived Q-Learning in the section before and thus have an estimation which action will give which return in this step. So, with our newly gained Q, for all actions and states, do

$$\theta_{i+1} = \theta_i + \alpha Q(s,a) \Delta \pi_{\theta_i}(a|s) \tag{3.6}$$

This way, actions with higher Q values will receive a higher gradient step, and as Q is constant with respect to θ we don't have to derive along Q here. Problematic is though, that the Q term we defined before, always took the best possible actions, which could be vastly different from the actions under our policy. Thus, we need to reformulate Q to account for any policy and not just the one which always takes the best action.

$$Q(s,a) = \mathbb{E}_{s' \sim \varepsilon, a' \sim \pi | s,a}[r + \gamma Q(s', a')] \tag{3.7}$$

However there is still a problem if we integrate this new Q into our policy updates. We assume that we use the same π for exploration while doing training, so this policy is responsible for taking actions in the environment. So let's assume the policy is poorly initialized at the beginning of our training and gives action 1 a probability 4 times as high as action 2, though taking action 2 would yield a 10% better return in our Q function. Now, as we are taking action 1 4 times as often as action 2, we will also perform gradient updates on action 1 4 times as often in our Monte-Carlo expectation draw. Though each of the gradient steps is smaller due to the Q factor, the higher frequency of updating action 1 will lead to action 1 getting an advantage over action 2 in this update strategy. To correct for this oversampling bias, we could divide the probability of each of the actions from the update term:

$$\theta_{i+1} = \theta_i + \alpha Q(s,a) \frac{\Delta \pi_{\theta_i}(a|s)}{\pi_{\theta_i}(a|s)} \tag{3.8}$$

We might remember from calculus that $\frac{\Delta x}{x} = \Delta \log x$ so we simplify to

$$\theta_{i+1} = \theta_i + \alpha Q(s,a) \Delta \log \pi_{\theta_i}(a|s) \tag{3.9}$$

This update equation is our basic policy gradient equation. In reinforcement-learning, it has proven helpful to do off-policy learning, in which another policy than the one being trained can generate experiences. Thus, we can store past experiences in a replay buffer. And in fact, we were already able to use a replay buffer in the Q learning part. However we lost that again when we introduced our policy to Q learning. If we remember our expectation term, in which actions were sampled from the policy $\mathbb{E}_{a \sim \pi}$. We needed this expectation term, because we were dealing with stochastic policies. If instead learning deterministic policies, we can exclude the term from the equation.

$$Q^{\mu}(s,a) = \mathbb{E}_{s' \sim \varepsilon | s,a}[r + \gamma Q^{\mu}(s', \mu(s'))] \tag{3.10}$$

Now, our Q is only dependent on the environment distribution, so when we do our Monte-Carlo experiments to update it, we can take our generated samples from the environment and assume they are modelling the underlying environment. However, by introducing our deterministic policy, we ruined our trick of dividing the probability value of an action from its gradient.

TODO: deterministic policy gradients

3.3.5 Deterministic Policy Gradients

3.3.6 DDPG

DDPG is a combination of DPG and Deep Q-Networks (DQN). [LHP$^+$] tackle several problems in the two algorithms to combine them. At first, DQN updates the Q network parameters based on a target calculated on the Q network parameters. To tackle this, [MKS$^+$] introduced a second Q network which they call target network for which the parameters are held constant over a period of updates. After a number of steps, the parameters from the actual Q network are then copied over to the target. In DDPG, they instead use soft target updates, where every step, the parameters of the actual network fade over to the target: $\theta_{target} = \tau\theta + (1 - \tau)\theta_{target}$. The fade-over parameter τ was recommended to be set to a small number like 0.01. Additionally to the target Q-network, they also found that a target policy

3.4 RL on windturbines

We are aware of one paper [KJT] which tried RL on windturbines already, we want to dedicate a section to it. The team behind that paper built a miniature model of a windturbine and used a variation of the policy gradient algorithm to control it.

4 Experimentation

Unfortunately, we are not able to provide a scientific comparison between different methods, as our learning proved too instable to derive sane metrics from it. Thus, this section will be structured as an assortment of problems and their solutions

4.1 Gym Experiments

At the start of our experimentation, we took off with a well-known problem and a proven implementation. We used the OpenAI-Pendulum task, in which a pendulum in gravity needs to be held upright by applying a force to it. OpenAI Gyms generally serve as benchmarks for reinforcement learning algorithms and most foundational papers and articles evaluate their solutions based on how well they perform in OpenAI Gyms. Starting with a reference-implementation in pytorch by [Dee], we experimented with the pendulum environment. We observed that the algorithm got stuck in a local optimum, where it rotates the pendulum instead of holding it upright, for long periods of the training process. Only with high noise, we could get it to hold a pendulum upright. We also tried to run the HalfCheetah task, where a two-dimensional dog-like being should be brought to running and moving its joints, but the results were more a crawling and shaking towards the goal. However, we hoped that the implementation might yield results on our windturbine environment already and exchanged the OpenAI gym for qblade.

The first runs looked marvellous, perfectly holding rated speed, until we found out that those perfect values were prescribed by a setting in the qblade project file, and disabling this setting marked the beginning of our actual experimentation phase.

4.2 Starting with qblade

On our first experimentation steps, we did not yet have experience with how the qblade simulation behaves with a random or not well designed controller. Our observation was, that the simulation was resting between two different states, which we want to call Forward Maniac Mode (FMM) and Backward Maniac Mode (BMM).

FMM is characterized by a rotational speed of ca 5 times rated speed, at 4.3 rad/s. At this speed, for some reason, the rotor will not speed up anymore. However, severe vibrations and sometimes the case of a disjointed blade happen. Especially in the case of a blade flying off, deflection values skyrocket up to several orders of magnitude higher than values seen in normal operation, and with one or more missing blades, the rotor will quickly stop rotating. This state is usually reached by a too low generator torque while not pitching out blades and sometimes stays stable for several thousand steps until a blade falls off.

BMM is characterized by a negative rotational speed, up to -3.8 rad/s. This can be reached by setting a higher generator torque than the rotor torque which is produced by aerodynamic lift on the blades. Also in this state, extreme vibrations and sometimes blades flying off can be observed, and like FMM this state is stable until a blade falls off.

In neither of these modes, the simulation is within realistic bounds and the observations gotten from the simulation at this part are not credible. After we observed that the controller itself wasn't able to avoid these states on its own, we inhibited reaching these states by outputting a death condition when being in negative rotational areas or high positive. After a death, the simulation gets reset to initial state and the controller has more chances to learn.

4.3 Designing reward functions

As qblade doesn't provide a built-in reward function, we needed to craft our own reward function based on the observed state. We considered several possible variants. We will first introduce all of them and give a more detailed explanation later

- **Rated speed** Hold a rated speed. Observe current rotational speed s_{rot} and with given rated speed s^*_{rot} calculate reward $r(s) = -\left|\frac{s_{rot}-s^*_{rot}}{s^*_{rot}}\right|$
- **Rated power** Hold a rated power. Observe current power s_{pow} and with given rated power s^*_{pow} calculate reward $r(s) = -\left|\frac{s_{pow}-s^*_{pow}}{s^*_{pow}}\right|$
- **Maximal power** Generate the maximum power possible. Observe current power s_{pow} and with given normalization constant c calculate $r(s) = s_{pow} * c$
- **Penalize current stress** Penalize high blade bending. Observe current bending s_{bend} and with given normalization constant c and another reward function $r(s)$ calculate new reward $r'(s) = r(s) - c * s_{bend}$
- **Penalize accumulated stress** Penalize accumulated structural stress. With $x = rainflow(s_0...s_t$ being the rainflow table and $p(x)$ being a function that maps from rainflow outputs to a scalar penalty, calculate $r'(s_0...s_t) = r(s_t) - p(rainflow(s_0...s_t))$

- **Penalize action gradients** Penalize high action gradients. With a_t being the current action, a_{t-1} the last action, c a penalty constant and $r(s)$ another reward function, calculate: $r'(s, a) = r(s) - c * (a_t - a_{t-1})$

Rated speed is the easiest of the rewards, as holding a rated speed can be achieved by either pitch or torque or a combination of the two. So both a high pitch and a high torque will stop it from running faster than it should. We used this reward for our first working version. **Rated power** is a bit more difficult. For reaching rated power, a certain aerodynamic torque is required, so pitching the blades out completely won't deliver rated power. However especially if setting an artificial rated power lower than what the turbine was built for, there is a certain play in how strongly to use blade pitch and how much needs to be done through torque. **Maximal power** is an intuitive reward, but might yield unrealistic results. This rewards high power regardless of the stress that is induced to the turbine, so a good policy might spin the turbine at high speeds beyond any turbine specification and then apply maximum torque. So this reward makes more sense when combined with a stress penalty. **Penalize current stress** is the easiest stress penalty, where a penalty proportional to the current bending of the structure can be used. As there are several bending modes on most of the components of the windturbine, a combination or selection of modes and components should be taken before. We decided to limit us to out-of-plane bending of the blade tips. This is a simple variant which leaves out bending in the middle of the blades, bending in the rotational plane, vibration or bending of the tower and torsional stress on the shaft. **Penalize accumulated stress** is a more complete, but time dependent variant. Rainflow Counting (RFC) is a commonly used method for calculating structural fatigue in windturbines [BW]. RFC takes in a bending signal over time and returns a table with amplitudes and their frequency, as in how often the structure swung how far. It filters out some higher-frequency swinging in between two lower frequency swings and thus also works for structures that swing in more than one frequency. To use it as a penalty, we needed to create a function that maps from that table to a scalar penalty. This penalty accumulates past stress, so it increases over time. We suspect this penalty to be hard to learn, as it reacts slowly to changes and a once given penalty will never be lifted again. **Penalize action gradients** is an option which is useful in contexts where high action gradients are impossible on the real system. Instead of clipping them, penalizing them could include action gradients into the learning process. We only expect sensible results with a stateful agent which knows at least the last action it took or one which has recurrent connections, as otherwise the last action taken is unknown to the agent.

Most of our experimentation we did with rewarding rated speed, as we expected this reward function to be the easiest to learn.

4.4 Aiding exploration

As in the beginning we observed our algorithm to get stuck in either FMM or BMM, we wanted to improve exploration. The term exploration in reinforcement learning means how well the agent explores the state space of the environment. Exploration is a common problem in reinforcement learning, as algorithms tend to stay in local maxima. Intuitively, we are optimizing a policy to move towards an optimal state, while computing what is the optimal state based on what we observed. Thus, if a certain policy already gets locally good values, it will move towards that locally optimal state and never explore out of it, thus the Q network will not know about the good values beyond that.

Additionally, as we discount future rewards, if the slope between the local optimum and the global optimum is too high and wide, the higher return of the global optimum will not propagate until the local optimum because of the discounting done close to it, and though the Q-network knows the global optimum, the gradients along the Q-function still lead into the local optimum.

We suspected the first problem to happen, as neither of the two modes had good rewards, and going out of these modes continously improves rewards. To feed a network with new values closer to an optimum, a common practice in reinforcement learning is to add noise.

4.4.1 Action Noise

In [MKS$^+$] they use an epsilon-greedy strategy, where with a certain small probability epsilon, a random action is taken. As we have a continous action space, we can add a little bit of noise towards every action: $a' = a + n$. We first tried a gaussian noise: $n \leftarrow \mathcal{N}(\mu, \sigma^2)$. We didn't see any improvements over our old behavior, so we switched to an Ornstein-Uhlenbeck process [UO] as recommended in the DDPG paper [LHP$^+$, formula (7)]. Additionally to the DDPG paper, we let the sigma parameter decay over time, so that the noise gets less when training has proceeded.

4.4.2 Random exploration

The TD3-paper [FHM] utilizes a random exploration phase, in which a completely random policy [MB] initially explores the environment, before the actual agent switches in. We implemented the random walk at first through gaussian noise and then through an Ornstein-Uhlenbeck process. The later version creates less vibration through less excessive changes in control parameters. As we found that both random policies rarely explore a sensible operational range, we later added an expert PID controller, which implements torque control, and combined that with our Ornstein-Uhlenbeck process. We set the parameters to that controller by experimentation as we did not need the actual control parameters of the controller. However it could be imagined to use an established controller here. The PID controller was easily able to keep the windturbine in a sensible operation range, and without noise it resulted in a smooth and constant operation. Though our PID controller is certainly inferior to industry-grade controllers, we will view it as an expert controller which is already achieving a certain degree of quality.

4.4.3 Parameter noise

As we still didn't see good results, we tried adding parameter-space noise [PHD$^+$] to the actor function, as the paper promised better exploration. We could not observe any improvements and thus deactivated parameter space noise for the rest of the experimentation.

4.5 Zero Gradients

All our efforts to aid exploration by adding noise did not yield better results, so we had a look at our gradients and found that all our actor gradients are zero. The critic had gradients, but only in the later levels of the net. The problem of vanishing or exploding gradients is typical for deep learning, but not so common in flatter architectures.

4.5.1 Simplifying the architecture

We tackled this problem by at first reducing the number and size of layers from the example implementation to one fully connected layer of 64 neurons in the critic and one fully connected layer of 32 neurons in the actor. The reference-implementation of [Dee] utilized 3 levels in the critic with 1024, 512 and 300 fully connected neurons and in the actor 2 levels of 512 and 128 fully connected neurons. Reducing complexity partially tackled the problem of zero gradients, we could then observe small gradients which however vanished over time.

4.5.2 Normalization

More effectively, we added data normalization. We normalized the data so states, actions and reward always stayed between 1 and -1. This finally fixed our problem of zero gradients and we could observe normal learning. We recommend this technique as a default technique for every reinforcement learning algorithm. As in our simulation, where our state reflects measurements from very different parts of the simulation, some values in the state-array were 8 orders of magnitude different from others. As most of the high-magnitude values are vibrations, the net could not create a link between these and the state-action, while low-magnitude but insightful observations like rotational speed were not considered due to their small absolute values. At first, we used observed states and rewards from the random exploration phase to calculate maximum/minimum values. As sometimes after random exploration, extreme values were observed, we started normalising on the replay data of a previous run. We could also imagine manually setting normalization bounds on low-dimensional problems, as sometimes human knowledge about the possible state space is present.

4.6 High action gradients

We observed that our controller delivers high action gradients and reacts strongly between timesteps, sometimes jumping from no pitch/torque all the way to the maximum. This creates vibrations in the structure and especially high blade pitch changes lead to blades breaking or falling off. We tried different methods to circumvent this.

4.6.1 Gradient actionspace

At first, we implemented gradient actionspaces, where the controller output is not an absolute pitch or torque value but a difference to the last output, starting with everything set to 0. For aiding training, we added the actual actions taken to the observation space of the next step. This way, we could clip the action gradients by reducing the gradient action space, so that in each timestep, only a fraction of the actual action space could be traversed. If the controller would output a positive change at maximum pitch/torque or a negative change at minimum pitch, this would be ignored. This did effectively limit action gradients, but also did not lead to any sensible results, so we deactivated this again.

4.6.2 Feeding past timesteps

Another idea was that we feed the last n timesteps concatenated to the current observations. We hoped that this way, the agent could derive its own gradients internally. The result of this were high and very noisy q-losses and no sensible results still. Though gradients looked sensible, we suspect that the high Q-losses resulted in too high steps along the gradient and thus diverging behavior.

4.6.3 Clipping action gradients

We eventually solved the problem with the high action gradients by still letting the controller output absolute actions across the entire action spaces, but if gradients exceed a certain threshold, the actual action taken is set to the nearest value with sensible gradients. Though the actions returned from the policy are still wildly unstable, they at least didn't result in death of the turbine anymore and still we didn't hinder convergence through our gradient actionspace. This resulted in our first ever controller keeping the turbine within sensible operational range for at least a part of the training, though diverging quickly after.

We at first hid this clipping from the replay buffer, but then decided to store the actual values taken, as the models then don't have to also learn the clipping effect. We observed better learning when not hiding the clipping from the replay buffer.

4.6.4 Pretraining the policy

We added a small period of direct pretraining of the policy on the actions taken by the expert policy during random exploration. We hoped for a more sensible default policy at the beginning and indeed observed reaching sensible behavior a little earlier.

4.7 Other improvements

Because still generally not performing well, we added some techniques which we thought would aid general performance.

4.7.1 Prioritized experience replay

[SQAS] proposed a method to sample experiences from the replay buffer according to how much they benefit training, and not just uniformly. In their paper, they showed how especially cliffwalk problems, but effectively all problems, benefit from using prioritized experience replay. In their case, they use it for DQN, but we will apply it to DDPG, as our windturbine problem is also such a problem in which rewards are sparse.

As knowing how much a sample benefits training is difficult, they propose to approximate this importance by the temporal difference error $\tau = y - Q(s, a)$. This difference, which in Q learning is comparable to the loss of the approximator, tells us how far off the prediction was in this step. So, alongside with the experiences, a sampling probability for each item p_i is stored in the replay buffer. To be able to efficiently sample from it, we implemented it as proposed in their paper in the form of a sum-tree.

When replaying, the sampling probability of a sample is set to $P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$, α being a hyperparameter allowing to fade between uniform sampling ($\alpha = 0$) and only priority sampling ($\alpha = 1$). New samples are always stored with maximum priority, and to correct for oversampling bias, a correction term is applied to the temporal difference error: $w_j = \frac{(N * P(j))^{-\beta}}{\max_i w_i}$. It is simply multiplied to the real TD-error. β is a parameter describing how strongly to correct oversampling, where $\beta = 1$ is full correction and $\beta = 0$ no correction at all.

We decided to split policy and critic updates, as the oversampling effects could also happen to policy updates, and still sample uniformly for policy updates.

4.7.2 Data augmentation

We still had some parts of the code left where there wasn't any noise involved, so we decided to add some to the training process. Data-augmentation with noise is a common technique when using neural networks [Bis, p.347] to reduce overfitting, so we expected it to also make our model generalize better. There have been some experiments with data augmentation in reinforcement learning [CKH$^+$], but it is not often mentioned on tutorials or blog posts. We implemented by adding a small noise term to the sampled states, actions and next states (s, a, s') when performing experience replay. We didn't really see any indication of bad generalization, but as implementing it wasn't a big issue and normally doesn't bring any disadvantages, we still added this feature and set the noise level to a very small value.

4.8 Exploding Q-Loss

We observed that our Q-Loss, especially after a death, explodes into e15 magnitude values, while most of the times it stayed around e2 magnitude. Instable Q-Losses are a general problem with DQN and DDPG, and some literature [FHM] proposes to use two Q networks, of which only the minimum is used for training.

4.8.1 Huber loss

We could already greatly improve our results by using huber-loss [Hub] for the critic instead of mean squared error loss as described in DDPG paper. However we still saw some loss explosions.

4.8.2 Double critic

Adding twin critic networks such as in [FHM] eventually solved our problem with loss explosions completely, however the thing didn't reach sensible states anymore. To use it in conjunction with Prioritized Experience Replay (PER), we needed to decide for a priority formula and decided to take the minimum of the two q values, after the maximum showed biasing towards one of the networks.

4.8.3 Batch normalization

As in deep learning, batch normalization is helpful for increasing lr and thus to skip over local maxima, we tried to add batch normalization between the layers. The authors suspected problems with the noise in RL, however it looked good.

4.9 Automatized hyperparameter search

To test all the above things, we tried simulations with different hyperparameters.

5 Evaluation

It must be stated that we did not achieve a single run which stayed within sensible operational range for longer than 30K steps. Even our handcrafted PID controller for which we took the first set of parameters that somehow works widely outperforms our learnt policies, an industry controller is even better. This result in itself is notable, because it needed a whole set of additions to plain DDPG to get first sensible results on our windturbine environment. So apparently, our windturbine simulation is more complex to learn than the OpenAI gym examples, which are usually used as benchmarks for Reinforcement Learning (RL) performance, or we are having errors in our programme. As the final product we developped is close to [FHM] and has improvements from a wide variety of papers, we conclude that tackling wind turbine control with RL will need more research. Though we were not able to outperform industry controllers, the speed in which the area of RL is advancing makes us hopeful to achieve good results in the near future. With our work on integrating qblade, we provided an open-source, accessible framework to develop future reinforcement learning controllers for wind turbines.

5.1 First working version

Our model consists of the DDPG algorithm with following additions: We integrated prioritized experience replay to

How we achieved first successes, hparams, etc

5.2 HParam tuning

We tried different hyper parameters. To evaluate our controller, we measure the time the simulation stayed within a sensible range. We define sensible as rotational speeds between 0 and 1.2 rad/s. We let each simulation run 1 million steps. As during our random exploration, the simulation was always in sensible range, we remove the random exploration phase from our evaluation.

We tried

- increasing/decreasing both learning rates

- increasing/decreasing tau and gamma
- increasing/decreasing batch size
- increasing/decreasing replay noise

5.3 Comparison to industry controllers

6 Future work

6.1 Reward functions

We did all our training under the simple hold-speed reward function. A realistic reward function however would incorporate power generated and damage incurred. As soon as a stable agent under the hold speed reward function was created, using different rewards could bring the performance of the controller closer to that of industry controllers. A complex reward function like what we proposed in section 4.3 however also makes learning more difficult, and thus would need more work in refining the algorithm.

6.2 Switch reward functions during training

To combine the learning ease of a simple reward function with the performance of a complex reward, we could imagine fading over between a simple and complex reward function. In early stages of training, the network is conditioned to only output anything sane, while in later training it could optimize power or damage prevention more.

6.3 Expert policy training

We implemented expert policy pretraining. Alternatively, we could imagine leaving the expert policy training active the whole time, not just once after random exploration, and to reduce the learning rate of the expert policy training gradually. This way, in initial training, predicting the expert policies actions would have a higher impact than what the Q-function would suggest, whereas in later training with the smaller learning rate, deviations from the expert policy would be punished less and the policy is allowed to deviate further from the expert policy. This would involve two backward passes per training iteration and would require prediction targets from the expert policy on new seen observations

6.4 Expert policy in instable conditions

When the turbine is reaching a dangerous state (high/low rotational speed, high vibrations), we could fall back to our expert controller to save the situation. This would result in a safe controller, that upon insane policies would fall back to sane behavior. It would degrade exploration beyond what we doom safe, so a controller with this fallback method would never explore a death condition. This would not necessarily hinder learning, as a complete death condition might not be necessary to explore to see the decreasing loss gradient leading to it. However, seeing a death condition could result in more extreme Q-predictions and thus more incentive to stay clear of a death.

6.5 Active control elements

There is ongoing research into adding active control elements to windturbine blades such as flaps. A difficulty incurred in that research is the design of a controller, as the high-dimensional action space makes it difficult for a human to engineer a controller. Using our automatically learning agent could enable a first working version to that research, providing a first direction or even lead to performance increases over not using active control elements in a turbine.

7 Conclusion

Literaturverzeichnis

[BCP$^+$] BROCKMAN, Greg ; CHEUNG, Vicki ; PETTERSSON, Ludwig ; SCHNEIDER, Jonas ; SCHULMAN, John ; TANG, Jie ; ZAREMBA, Wojciech: OpenAI Gym.

[Bel] BELLMAN, Richard: The Theory of Dynamic Programming. 60, Nr. 6, S. 503–516. – ISSN 0002–9904

[Bis] BISHOP, Christopher M.: *Neural Networks for Pattern Recognition*. Clarendon Press ; Oxford University Press. – ISBN 978–0–19–853849–3 978–0–19–853864–6

[BJSB] BURTON, Tony ; JENKINS, Nick ; SHARPE, David ; BOSSANYI, Ervin: *Wind Energy Handbook: Burton/Wind Energy Handbook*. John Wiley & Sons, Ltd. – ISBN 978–1–119–99271–4 978–0–470–69975–1

[BW] BERGLIND, J. J. B. ; WISNIEWSKI, Rafael: Fatigue Estimation Methods Comparison for Wind Turbine Control.

[CKH$^+$] COBBE, Karl ; KLIMOV, Oleg ; HESSE, Chris ; KIM, Taehoon ; SCHULMAN, John: Quantifying Generalization in Reinforcement Learning.

[Dee] *Deep Deterministic Policy Gradients Explained*

[FHM] FUJIMOTO, Scott ; VAN HOOF, Herke ; MEGER, David: Addressing Function Approximation Error in Actor-Critic Methods.

[HLD] HOWLAND, Michael F. ; LELE, Sanjiva K. ; DABIRI, John O.: Wind Farm Power Optimization through Wake Steering. 116, Nr. 29, S. 14495–14500. – ISSN 0027–8424, 1091–6490

[Hub] HUBER, Peter J.: Robust Estimation of a Location Parameter. 35, Nr. 1, S. 73–101

[JBMS] JONKMAN, J. ; BUTTERFIELD, S. ; MUSIAL, W. ; SCOTT, G. *Definition of a 5-MW Reference Wind Turbine for Offshore System Development*

[KJT] KOLTER, J. Z. ; JACKOWSKI, Z. ; TEDRAKE, R.: Design, Analysis, and Learning Control of a Fully Actuated Micro Wind Turbine, IEEE. – ISBN 978–1–4577–1096–4 978–1–4577–1095–7 978–1–4577–1094–0 978–1–4673–2102–0, S. 2256–2263

[LHP+] LILLICRAP, Timothy P. ; HUNT, Jonathan J. ; PRITZEL, Alexander ; HEESS, Nicolas ; EREZ, Tom ; TASSA, Yuval ; SILVER, David ; WIERSTRA, Daan: CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING.

[MB] MOZER, Michael C. ; BACHRACH, Jonathan: Discovering the Structure of a Reactive Environment by Exploration. 2, Nr. 4, S. 447–457. – ISSN 0899–7667, 1530–888X

[MKS+] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; GRAVES, Alex ; ANTONOGLOU, Ioannis ; WIERSTRA, Daan ; RIEDMILLER, Martin: Playing Atari with Deep Reinforcement Learning.

[MWP+] MARTEN, D ; WENDLER, J ; PECHLIVANOGLOU, G ; NAYERI, C N. ; PASCHEREIT, C O.: QBLADE: AN OPEN SOURCE TOOL FOR DESIGN AND SIMULATION OF HORIZONTAL AND VERTICAL AXIS WIND TURBINES. 3, Nr. 3, S. 6

[PHD+] PLAPPERT, Matthias ; HOUTHOOFT, Rein ; DHARIWAL, Prafulla ; SIDOR, Szymon ; CHEN, Richard Y. ; CHEN, Xi ; ASFOUR, Tamim ; ABBEEL, Pieter ; ANDRYCHOWICZ, Marcin: Parameter Space Noise for Exploration.

[SLH+] SILVER, David ; LEVER, Guy ; HEESS, Nicolas ; DEGRIS, Thomas ; WIERSTRA, Daan ; RIEDMILLER, Martin: Deterministic Policy Gradient Algorithms. , S. 9

[SMSM] SUTTON, Richard S. ; MCALLESTER, David A. ; SINGH, Satinder P. ; MANSOUR, Yishay: Policy Gradient Methods for Reinforcement Learning with Function Approximation. , S. 7

[SQAS] SCHAUL, Tom ; QUAN, John ; ANTONOGLOU, Ioannis ; SILVER, David: Prioritized Experience Replay.

[UO] UHLENBECK, G. E. ; ORNSTEIN, L. S.: On the Theory of the Brownian Motion. 36, Nr. 5, S. 823–841. – ISSN 0031–899X

[Wil] WILLIAMS, Ronald J.: Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. , S. 28

Danksagung

Die Danksagung...

Erklärungen zum Urheberrecht

Hier soll jeder Autor die von ihm eingeholten Zustimmungen der Copyright-Besitzer angeben bzw. die in Web Press Rooms angegebenen generellen Konditionen seiner Text- und Bildübernahmen zitieren.