TECHNISCHE UNIVERSITÄT DRESDEN

FACULTY OF COMPUTER SCIENCE
INSTITUTE OF SOFTWARE AND MULTIMEDIA TECHNOLOGY
CHAIR OF COMPUTER GRAPHICS AND VISUALIZATION
PROF. DR. STEFAN GUMHOLD

Großer Beleg

Reinforcement-Learning Windturbine Controller

Nico Westerbeck
(Mat.-No.: 3951488)

Tutor: Dr. Dmitrij Schlesinger

Dresden, December 7, 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 December 7, 2019

Nico Westerbeck

Contents

0.1 Background: . . . 2
0.2 Tasks . . . 2

1 Abstract **3**

2 Background **4**

2.1 Windturbine control . . . 4
2.1.1 Why we do what we do . . . 6
2.2 QBlade . . . 7
2.3 Reinforcement learning . . . 8
2.3.1 Environment assumptions . . . 8
2.3.2 Definitions . . . 9
2.3.3 Q-Learning . . . 11
2.3.4 Policy Gradients . . . 12
2.3.5 Deterministic Policy Gradients . . . 15
2.3.6 DDPG . . . 16
2.4 RL on windturbines . . . 18

3 Experimentation **21**

3.1 Gym Experiments . . . 21
3.2 Starting with qblade . . . 21
3.3 Designing reward functions . . . 22
3.4 Aiding exploration . . . 24
3.4.1 Action Noise . . . 24
3.4.2 Random exploration . . . 25
3.4.3 Parameter noise . . . 26
3.5 Zero Gradients . . . 26
3.5.1 Simplifying the architecture . . . 26
3.5.2 Normalization . . . 26

3.6 High action gradients . . . 27
3.6.1 Gradient actionspace . . . 27
3.6.2 Feeding past timesteps . . . 28
3.6.3 Clipping action gradients . . . 28
3.6.4 Pretraining the policy . . . 28
3.7 Other improvements . . . 28
3.7.1 Prioritized experience replay . . . 29
3.7.2 Data augmentation . . . 31
3.8 Exploding Q-Loss . . . 31
3.8.1 Huber loss . . . 31
3.8.2 Double critics . . . 32
3.8.3 Batch normalization . . . 33
3.8.4 Regarding death conditions . . . 33
3.8.5 Clipping observations . . . 34

4 Algorithm **35**
4.1 QBlade . . . 35
4.2 Core algorithm . . . 36

5 Evaluation **38**
5.1 Hold speed . . . 38
5.2 Hold rated power . . . 40
5.3 Hold power . . . 41
5.4 Pendulum . . . 41
5.5 Discussion . . . 41

6 Future work **43**
6.1 Scale up . . . 43
6.2 Validate Prioritized Experience Replay (PER) . . . 43
6.3 Train PID inputs . . . 43
6.4 Reward functions . . . 44
6.5 Expert policy training . . . 44
6.6 Expert policy in instable conditions . . . 44
6.7 Active control elements . . . 45

Bibliography **46**

1 Abstract

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, these advancements were only partially applied to the area of windturbine pitch and torque control. 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. However, these simple controllers can only deal with a limited amount of data inputs and are preventing turbine damages. We are evaluating the use of an actor-critic reinforcement learning algorithm to solve continous control of a windturbine, hoping to lay a foundational work for later improvements. We find that utilizing a wildly proven continous control algorithm does not solve our windturbine problem. Finally, we demonstrate that this control is possible, however we do not achieve results outperforming industry controllers.

2 Introduction

With the issue of global warming, we are facing maybe the biggest challenge to our generation. Though wheather models have predicted the effect since the 1980s, only in the recent years this topic has been brought into societal focus. CO2 is seen as the main cause for a tendency in the global weather to heat up. Since our industrial age, humankind has been releasing CO2 into the athmosphere in large scales. If however we continue doing so the way we are doing now, we will likely increase global temperature by a degree which will make living on this planet difficult. With the Paris agreement, the United Nations decided to limit global warming to 1.5 degrees, for which massive reductions in CO2 emissions are necessary. As energy production holds one of the biggest parts in the global CO2 emissions, we need to replace CO2 emitting power plants. For this, renewable energies have played a major role, and as a big part of this also windpower.

Winds are a direct result of sunshine. The sun heats different parts of the earth to different temperatures, as the surface of the earth isn't uniform. The air over hotter regions such as close to the equator tends to rise, while on mountains and the poles it tends to fall. Combine this with a vast and complex interplay of humidities, Coriolis effects, mountains, cloud formation and enough effects to fill an entire branch of science and you will get wind. Windturbines operate by drawing energy from this weather phenomenon. Large rotors placed in windy regions which are connected to a generator yield electrical power. As weather is a complex phenomenon, wind tends to come from different directions with different speeds. This makes drawing energy from it more complex, or less efficient. If we were to draw energy from both storms and light breezes with the same turbine, we would not be efficient. Thus, modern windturbines are built to be adjustable. With these adjustments, a turbine can operate sensibly over a broader range of wind conditions. However, somehow, these adjustments need to be automatically adapted to the wind condition. This is the task of the turbine controller. We are attempting to learn such a controller.

Machine learning has developped immensly over the last years. Benefitting from Moore's law, huge amounts of computational power can be used to train more and more complex models. Machine learning is generally split into three big branches. Supervised learning, unsupervised learning and reinforcement learning. In supervised learning, some form of fitting is done on a dataset.

3 Background

This section aims at facilitating the background knowledge necessary for understanding this work. We will first introduce windturbines and common terms around this area of research. Then we will motivate our work precisely. We will give insights on the simulation tool we used and then derive the reinforcement learning algorithm we decided to use. Lastly we will present another paper which tried to combine reinforcement learning and windturbines.

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. To explain this, we added reference plot of the windturbine model which we will use in our further testing, the NREL 5MW reference turbine [JBMS]. Figure 2.1 describes several properties of that turbine in relation to windspeed. Rated windspeed is a windspeed, in which the turbine gives maximum energy on highest aerodynamic effiency. In our example figure, this point is at 11.4m/s inflowing wind. At this point, the generator runs on its maximum torque

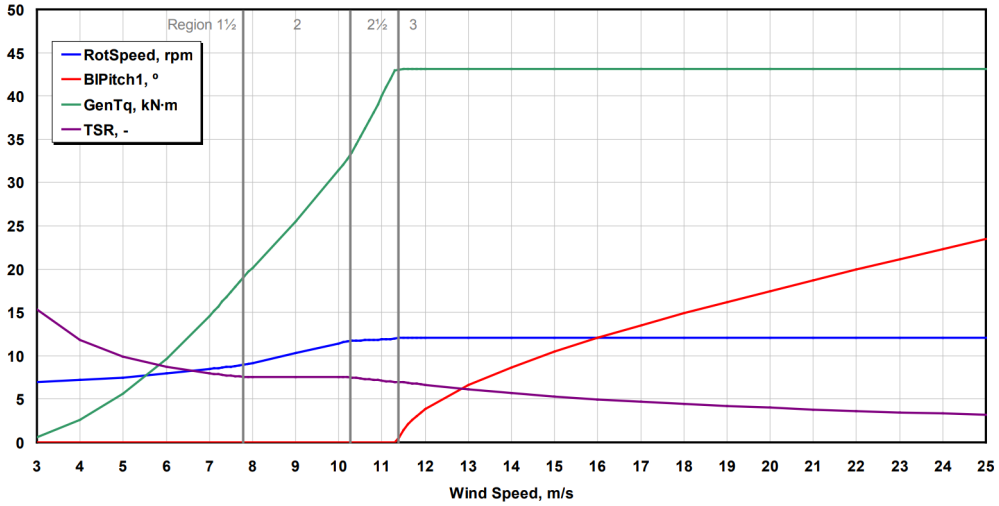


Figure 3.1: Control curve of the NREL 5MW turbine, from [JBMS]

(green graph) and the blades are pitched to the optimum angle of 0 degrees (red graph). At this point, the tip-speed-ratio (purple graph) is at its designed spot. When designing the turbine blades, this tip speed ratio plays an important role. 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 aggressively. In our reference turbine, you can see how the rotor speed is kept above 6.9 rpm, which is the cut-in speed of the NREL 5MW turbine. As the generated power of a windturbine is proportional to its rotor speed times generator torque, we produce less than our rated energy in this area. 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. See the red line in the reference plot - blades are kept at zero below rated and from then on control is done only through blade pitch change. The generator torque can be kept at maximum in this area and such will also the energy production stay at maximum. 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. This is done in a way that the rotational speed of the rotor stays exactly constant.

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. In our 5MW reference turbine, only blade pitch is used to counteract resonances.

Usually this control is implemented by two PID controllers, which are hand-designed as described in [BJSB, sec 8.4]. PID-Controllers in general have one real-valued input over time and deliver an output. This is done by calculating an error term between the input and a desired input. The output is calculated as a mixture of the error itself (P - proportional to the error), integrating the error over the last few timesteps (I - integral of the error) and calculating a derivative to the last timestep (D - derivative). The output is the sum of these 3 terms each factorized with a constant factor. However back to windturbines. Usually, we implement two PID controllers to control a turbine, a torque and a pitch controller. 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. Sometimes, this strong split into two operational ranges is relaxed a bit, as in the 5MW controller, the pitch part counteracts resonances also below rated speed. From this simple concept, the resulting controllers work reasonably close to theoretical maximum already. Our reference turbine has a peak power coefficient of 0.482 - meaning that at optimal wind speed, 48% of the energy of the wind is converted into electricity. In fact, the theoretical maximum is given by the Betz limit of 59.3% and if we account for electrical and frictional losses, the realistic maximum is even lower. So there is not much potential to be alleviated. Also, only a fraction of this potential lies in the responsibility of the control circuit. Blade design has so far been the major area for improvements.

Still, in this work, we are trying to replace these hand-designed controllers with a reinforcement-learning algorithm. We are not expecting to exceed the performance of a industry controller. So why do we even do this?

3.1.1 Motivation

As you might have seen before, all controllers follow a certain style. They take a single input variable, such as rotor speed, and adjust a single output such as torque. This type of controller is called Single Input Single Output (SISO). SISO controllers are easy to design and work reliably. They can also be interconnected, such as with the pitch controller which gets a vibration input and a rotor speed input to combine them to one output. The main challenge with wind turbine control at the moment is however not to optimize for power, but to optimize for life-time [vKPN$^+$, Chapter 4]. The controllers are dealing well with adjusting energy to a good level, but they are missing out on keeping the windturbine intact over a long life time.

Also, the outputs get more complex. Back in the early days of wind turbines, you could control 3

variables - pitch, torque and yaw. Nowadays, we can control each of the blades individually and with a quick enough response time to wiggle the blades around within one single rotor rotation. Research on adding active control elements to the blades such as flaps is happening, which would add many more control parameters. You might have seen flaps on an aircraft wing, extending the length of the wing or adding gaps at takeoff and landing to account for the low wind speeds hitting the aircraft wing in these flight scenarios. This could also benefit wind turbines at low wind speeds, however building a good controller for them is challenging. Tackling all this with SISO is possible, but a lot of manual work.

Next, the inputs also get more complex. It is getting cheaper to install high-precision sensors, and many of them. Wind measurement techniques such as LIDAR based aenometers are able to efficiently and precisely predict incoming wind situations. Load measurements can be done in real-time across a turbine blade and in theory, it would be possible to incorporate these into control systems. However, accomodating all these input parameters in SISO systems is difficult as we would have to hand-model each of the inputs and how exactly it ends up in the output.

We hope to lay a foundational step in solving these issues by using an end-to-end machine learning algorithm which is capable of learning a wind turbine control scenario with many inputs and outputs. Also, we must admit that we are generally interested in playing around with reinforcement learning, it's fucking cool.

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. We have 23 observations and 5 actions.

After a while of experimenting with the unrestricted simulation, we added some bounds to our control parameters and hid some observations and actions. Restrictions to our final version are described in section 4.1.

3.3 Reinforcement learning

So, why are we using reinforcement learning? We have posed the challenge earlier, that we want to learn a control system with a high number of inputs and outputs. Reinforcement learning has proven to be able to tackle high-complexity problems, both in input and output dimension. There have been successes in learning to play atari games based on pixel inputs [MKS$^+$a] and outperform humans, there have been successes in playing games with very long-term strategies such as go [SHM$^+$], and research is continuing to improve. Because of all these successes, we try to automatically learn a windturbine controller.

Unfortunately, reinforcement learning isn't possible without a good amount of maths, so the following sections will be mainly mathematical.

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. We denote the probability of a state transition to a specific s' as $p(s'|s, a; \varepsilon)$ and the distribution over all states s' as $P_{s' \sim \varepsilon|s,a}$ and when s, a is clear, we omit it.

The reward $r(s, a)$ is a scalar value which judges the action a taken in the current state s 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.

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.

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. Let's write down this mission. We want to maximize:

$$J = \mathbb{E}r \tag{3.1}$$

This formula is still a bit incomplete - we didn't yet specify our expectation. And wasn't reward defined on states and actions? But that is because we are still missing some definitions. Luckily, there are some ahead.

3.3.2 Definitions

Additionally to the terms defined by the environment, we introduce some terms which are common to reinforcement learning

- **Policy** A function that maps from states to actions. There are deterministic policies $\mu : S- > A$ and stochastic policies $\pi : S- > P(A)$. If we approximate a policy by a function approximator parametrized with θ, we write $\pi(s;\theta)$ or omit the parameters when clear.
- **Value** A value with respect to a policy and a starting state s gives the accumulative reward of following that policy from state s until infinitely in the future. $V^{\pi}(s_t) = \mathbb{E}_{s_i \sim \varepsilon, a_i \sim \pi, i>t} r(s_i, a_i)$. 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 value 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.
- **Return** If we measured a value function, that would be the return. Obviously we can't measure infinitely, so a return is only defined on a finite trajectory of state-transitions, but otherwise is equal to a value.
- **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 value of all possible policies in a finite MDP
- **Q-Value** The reward of taking action a in state s plus the value following a policy after that $Q^{\pi}(s, a) = r(s, a) + \mathbb{E}_{s' \sim \varepsilon} V^{\pi}(s')$. When following the optimal policy, we simply write $Q^*(s, a)$. Usually, our Q functions are discounted. Note that we can also define Values on Q-Values: $V^{\pi}(s) = \mathbb{E}_{a \sim \pi} Q(s, a)$. If we approximate Q by a function approximator parametrized with θ, we write $Q(s, a; \theta)$ or omit it in case it is clear.
- **Actor-Critic** Actor is just another name for a learnt policy and critic just another name for an estimated Q-Value function. These two frequently appear together, if that happens we are using an actor-critic algorithm.
- **State-density** According to [SMSM], $d^{\pi}(s) = \lim_{t->\infty} p(s_t = s|s_0, \pi)$ describes how likely it is to be in state s when following a policy π forever. If we for example have a policy which plays go as well as the authors of this paper, states with a lot of enemy stones and very few own stones might be very likely, and a state without enemy stones at all would be very unlikely. Also according to [SMSM] this distribution is independent of the starting state s_0 and only dependent

on the policy and of course the environment. In later papers, commonly a discounted version of this is used. $\rho^\pi(s)$ weights future probabilities less and thus can be described as the probability of being in state s *soon* when following policy π. We will omit a precise definition here, you can look it up at [SLH$^+$, Section 2.1].

Now, with all these definitions at hand, we can specify our mission a bit better. We want to maximize

$$J^\pi = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi} r(s,a) = \int_S \rho^\pi(s) \int_A \pi(s) r(s,a) \tag{3.2}$$

In other words, we want a very high probability of being in a state (ρ^π) which might give good rewards, and in that state we want a high probability of taking an action (π) which might give a good reward. In our wind turbine, rewards could for example be the all the costs and incomes summed together, so at every point in time we could count income by energy yield or costs because of maintenance (with our controller more likely costs for rebuilding the entire turbine). This work would be easy if we could directly calculate this equation but clearly, there is a bit of a problem ahead. Firstly, we don't know ρ as we don't exactly know the environment we are modelling. We can observe some rewards, states and actions, but only very few of them. We might then see that using 0.3MN of torque at rotational speed 11rpm yields good energy yield, but how about rotational speed 11.1rpm? We might have never seen that. Also, rotating the turbine at 15rpm might be a good strategy to immediately achieve high power outputs, but we might need to fix our turbine after running this policy for just a few days and incur a heavy penalty. So we don't know how rewards will spread out in the future. Luckily we have defined Q-Values above. We could use them somehow, so let's discuss how to learn them.

3.3.3 Q-Learning

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

$$Q^*(s,a) = \mathbb{E}_{s'\sim\varepsilon|s,a}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{3.3}$$

converges to the optimal solution when using iterative value updates. We 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

$$y_i = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q^*(s', a'; \theta_i)] \tag{3.4}$$

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

$$L(\theta_i) = \mathbb{E}_{s \sim \varepsilon}[(y_i - Q^*(s, a; \theta_i))^2] \tag{3.5}$$

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

$$\Delta_{\theta_i} L(\theta_i) = \mathbb{E}[(y_i - Q^*(s, a; \theta_i)) \Delta_{\theta_i} Q^*(s, a; \theta_i)] \tag{3.6}$$

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 an optimal policy. In [MKS+a], 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. 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 REward Incre-

ment equals Nonnegative Factor x Offset Reinforcement x Characteristic Eligibility (REINFORCE) (no joke) and was originally presented by [Wil]. Then we will add a Q term to it according to [SMSM] and call it Policy Gradients (PG). [DWS] generalized that term and made an off-policy version of policy gradients, which we will not discuss in detail. 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 Deep Deterministic Policy Gradients (DDPG) by [LHP$^+$], which is what we used to start our experimentation.

Let's assume we have a probablistic policy $\pi : S \to P(A)$ which is parametrized by θ. An intuitive way of improving this policy could be to increase the gradient proportional to the reward an action yielded:

$$\theta_{i+1} = \theta_i + \alpha r(s,a) \Delta \pi_{\theta_i}(a|s) \tag{3.7}$$

We could do this for all states we see and would end up with an iterative way of improving our policy. In fact, this is similar to the REINFORCE algorithm (if we add our importance sampling trick from below). However, we might not know rewards for any action, state or have noisy rewards. 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

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

This way, actions with higher Q values will receive a higher gradient step. 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 used, 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 our policy instead of the optimal policy

$$Q^\pi(s,a) = \mathbb{E}_{s' \sim \varepsilon, a' \sim \pi | s,a}[r + \gamma Q^\pi(s', a')] \tag{3.9}$$

Now, in theory, we can not regard Q as constant with respect to θ anymore, because it is dependent on the policy. [SMSM] claim though that it can be ommitted and still regarded as constant. 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 and also responsible for which experiences we see during our training. 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 update steps is smaller due to the lower 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 employ a trick which is called importance sampling. Because we know the bias from the probability distribution of our policy, we divide that probability from the update term:

$$\theta_{i+1} = \theta_i + \alpha Q^\pi(s,a)\frac{\Delta \pi_{\theta_i}(a|s)}{\pi_{\theta_i}(a|s)} \tag{3.10}$$

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

$$\theta_{i+1} = \theta_i + \alpha Q^\pi(s,a)\Delta \log \pi_{\theta_i}(a|s) \tag{3.11}$$

So, let's sum up what we have done before and formulate our goal again. Before, we ignored for brevity that we are dealing with expectation calculations and that we are doing these updates under a stochastic environment, so to write it down formally correct, we need to remember the density function of a state under a policy from section 2.3.2. Using the discounted version approximates the Monte-Carlo experiments we are doing better than the undiscounted version, as effectively we do not train forever on one constant policy, but change it frequently, so we will use ρ^π to approximate how likely we are in a state s when using policy π.

Having this ρ, let's write our objective function J which we want to maximize.

$$J(\pi) = \int_S \rho^\pi(s) \int_A \pi(s,a) Q^\pi(s,a) da ds = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi}[Q^\pi(s,a)] \tag{3.12}$$

This looks similar to equation 2.2, just that we use Q now.

[SMSM] proves that deriving this yields

$$\Delta J(\pi) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi}[\Delta \log \pi(a|s) Q^\pi(s,a)] \tag{3.13}$$

This equation is very famous in reinforcement learning and is called policy gradients. And by coincidence, it is equivalent to what we intuitively derived in equation 2.11. We decided to skip the actual proof and use this intuitive explanation instead, and we used the intuitive explanation in [SB, Chapter 13, Section 3, after Equation 13.8] to explain the nature of this policy gradient equation.

In reinforcement-learning, it has proven helpful to do off-policy learning, in which another policy β than the one being trained can generate experiences. We would then reformulate our expectation to $\mathbb{E}_{s\sim\rho^\beta, a\sim\beta}$. Thus, we could store past experiences in a replay buffer and learn from that. In fact, we were already able to use a replay buffer in the Q learning part. If we wanted to do that on our policy however, we would need to correct for the bias of the other policy aswell with importance sampling. However we will not derive stochastic off-policy updates, because using the DPG algorithm, this problem is solved.

If you want to learn about Off-Policy Actor Critic (Off-PAC) with stochastic policies, we recommend to read [DWS]

Additionally to this problem, another problem will be solved by DPG as well: If we remember our Q 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 policy expectation from the equation. But before going through all the advantages of deterministic policies, let's first explain it.

3.3.5 Deterministic Policy Gradients

Before [SLH$^+$], stochastic policies were preferred because the stochastic nature aided exploration and because there was simply no theory on how to derive deterministic policy gradients. The idea behind deterministic policy gradients is to replace the stochastic policy $\pi : S- > P(A)$ with a deterministic one $\mu : S- > A$. Also, we are switching from small finite action spaces to continous action spaces. Now, updating this policy according to the equations above does not work anymore due to two problems. At first and most significantly, above we needed all a or at least some form of maximization to calculate the full gradient of $\pi(a|s)$. This is difficult with a continous action space now. Furthermore our importance sampling trick, our correction for overestimation bias based on the bias of the policy, does not work anymore. We don't now the bias of that policy, because it is deterministic and will yield only a single action for any state.

So, following the policy gradient has become difficult. However, let's have a look at our Q function. We can replace our stochastic policy with a deterministic one and get

$$Q^{\mu}(s,a) = \mathbb{E}_{s'\sim\varepsilon|s,a}[r + \gamma Q^{\mu}(s', \mu(s'))] \tag{3.14}$$

Now, our Q is only calculating an expectation on the environment distribution, not anymore both on $s' \sim \varepsilon, a' \sim \pi$. This is an advantage, because computing only one expectation requires less samples and we do not have to worry for any bias from the policy in Q-updates. 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.

If we are visualizing our Q as a function of state and action, we get a fully differentiable surface. If we derive this surface along the action dimension, we get a gradient which points in the direction of a better action for this state. We could direcly follow this gradient and calculate our policy update from it:

$$\theta_{i+1} = \theta_i + \alpha \mathbb{E}_{s\sim\rho^{\mu_{\theta_i}}}[\Delta Q^{\mu_{\theta_i}}(s, \mu_{\theta_i}(s))] \tag{3.15}$$

[SLH+, Theorem 1] proves that this gradient always exists. We now managed to remove any expectation that is related to our action space from our gradient updates. This is beautiful because at first, we do not have to sample over many actions to learn this expectation. But more importantly, this term also works off policy, and without importance sampling. We can use a different exploration policy β and replace our expectation term $\mathbb{E}_{s\sim\rho^{\mu}}$ with $\mathbb{E}_{\rho^{\beta}}$ here completely unpenalized. Why is this so? [DWS] proved off-policy gradients possible, but had to use importance sampling to correct for the bias in β. In fact, generally, it is not possible to do this replacement. However, [SLH+, Section 4.3] formulated a compatibility theorem. They proved, that a linear function approximator Q^{ω} which minimizes the MSE between Q^{ω} and Q^{μ} will retain the gradient, even when trained on samples generated by β, and called this Q *compatible*. However, in practice we are not minimizing that loss and neither are we using linear function approximators. [BPS+] proved that using a non-linear function approximator to predict the value function $V(s)$ will converge towards a local optimum. As we can formulate our Q by using V, we transfer this to our scenario. However, DPG is not definitely proven to converge for a non-linear function approximator, as this transfer was, to our knowledge, never fully formulated. However, in practice, it is still trained off-policy. In our own experiments, we found that using a wildly different policy for exploration than for training did in fact result in worse behavior, and [FMP] observed the same behavior. Although there is some evidence against this being possible, we will from now on just assume that DDPG is trainable off-policy because it delivered good results in the past and because the paper stating it is possible has been cited more than 100 times more often than the one saying it's not. Science!

So let's restate our objective function for the actor.

$$J(\mu_\theta) = \int_S \rho^{\beta}(s) Q^{\omega}(s, \mu_\theta(s)) ds = \mathbb{E}_{s\sim\rho^{\beta}}[Q^{\omega}(s, \mu_\theta(s))] \tag{3.16}$$

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, traditional Q learning updates the Q network parameters based on a target y calculated on the Q network parameters themselves. This introduces instability and to tackle this, [MKS+a] introduced a second Q network Q' which they call target network, for which the parameters are held constant over a period of updates. They calculate the y based on the second network Q'. After a number of steps, the parameters from the actual Q network are then copied over to the target.

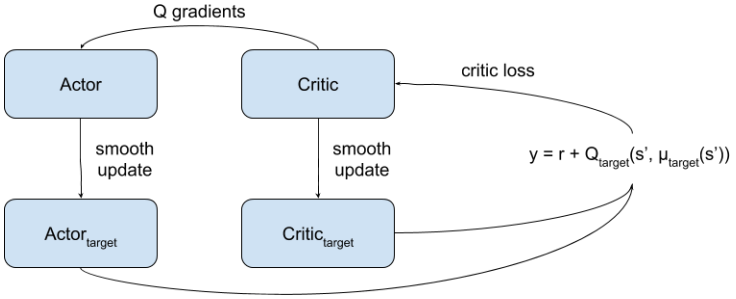


Figure 3.2: DDPG Network

The stability gain in their paper depends greatly on the number of steps after which to perform an update. In DDPG, they instead use smooth 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 0.001. Additionally to the target Q-network, they also found that a target policy μ' improves stability, and they update it the same way as the main policy. But this is a lot at once. Let's have a look at the Q update functions

$$y = r(s,a) + \gamma Q'(s', \mu'(s'; \theta^{\mu'}); \theta^{Q'}) \tag{3.17}$$

$$L(\theta^Q) = \mathbb{E}_{s\sim\rho^\beta, a\sim\beta, s'\sim\varepsilon}[(Q(s,a;\theta^Q) - y)^2] \tag{3.18}$$

We sketched an overview in figure 2.2. Remember that actor was another name for policy and critic another name for Q estimator. Notice how for calculating the Q targets, they used both the target actor μ' for generating actions which are then evaluated by the target critic Q'. This is their version of the improvement over DQN for their instable Q function, as they also observed instabilities. Note how the Q function here does not really predict the value of following a specific policy anymore, as before we could always denote which policy Q was trained on. Also, this Q is approximated by a non-linear function approximator instead of the linear one from compatible DPG. [LHP+] do not provide a proof of convergence for this new, mixed Q update. Still, we have an off-policy way of improving the critic now. The actor is directly copied from DPG even though it does not fulfil the *compatible* property.

$$J(\theta^\mu) = \mathbb{E}_{s\sim\rho^\beta}[Q(s, \mu(s;\theta^\mu); \theta^Q)] \tag{3.19}$$

Let's derive these two equations. First the policy gradient:

$$\Delta_{\theta^\mu} J(\theta^\mu) = \mathbb{E}_{s\sim\rho^\beta}[\Delta_{\theta^\mu} Q(s, \mu(s;\theta^\mu); \theta^Q)] = \mathbb{E}_{s\sim\rho^\beta}[\Delta_a Q(s,a;\theta^Q|a=\mu(s))\Delta_{\theta^\mu}\mu(s;\theta^\mu)] \tag{3.20}$$

And the Q update:

$$\Delta_{\theta^Q} L(\theta^Q) = \mathbb{E}_{s\sim\rho^\beta, a\sim\beta, s'\sim\varepsilon}[(Q(s,a;\theta^Q) - y)\Delta_{\theta^Q} Q(s,a;\theta^Q)] \tag{3.21}$$

Similar to other reinforcement learning algorithms, we will again not compute the full gradient but use stochastic gradient ascend with batch learning. To rephrase gradient ascend to gradient descend, they multiply the policy loss with -1. In fact, in the original DDPG algorithm, they used Adam [KB] optimization instead of stochastic gradient descend. As activations, they used relu [Aga]. Their network architecture for the actor consisted of two fully connected layers with 400 and 300 units and instead of relu they used tanh activations in the last layer to bound actions. For the critic, they had a version which learnt upon pixel outputs and used convolutional layers, while we only look at the low-dimensional version. For this, they also used a two layer variant with 400 and 300 units. State inputs were fed through the whole network, while actions were concatenated in onto the second layer, skipping the first.

Furthermore, to aid exploration, they added noise in form of a Ornstein-Uhlenbeck process [UO].

For readability, we added a copy of their algorithm in Algorithm 1, with minor adjustments to fit our notation.

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 REINFORCE algorithm to control it.

Their miniature model was built from cheap and low-scale parts, they simulated a wind-tunnel by attaching several fans in front of a wooden tube and at the end placed their turbine. Figure 2.3 illustrates this. Each of the blades of that miniature turbine is individually controllable and through a correction term, they are able to achieve independent pitch control which is able to pitch the motors to a value based on rotor position. Though this turbine rotates many times faster than a normal, full scale wind turbine, the underlying principles are the same (500rpm vs 11rpm). For us most importantly, they used reinforcement learning to control this turbine.

As training input, they used a single observation in contrast to popular reinforcment learning which operates on high-dimensional observation space. This observation was power output of the turbine, and they treated it as reward to the system, not having any state observations. Their action space consisted of electrical load on the generator and blade pitch, which was uniformly applied to all 3 blades. In fact, they designed their policy as to return a gaussian distribution over these two parameters directly without

Algorithm 1: Vanilla DDPG algorithm

Randomly initialize critic network $Q(s, a; \theta^Q)$ and actor $\mu(s; \theta^\mu)$ with weights θ^Q and θ^μ

Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer $\mathcal{R}$

for *episode = 1, M* **do**

Initialize a random process $\mathcal{N}$ for action exploration

Receive initial observation state s_1

for *t = 1, T* **do**

Select action $a_t = \mu(s_t; \theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

Execute action a_t and observe reward r_t and observe new state s_{t+1}

Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$

Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{R}$

Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}; \theta^{\mu'}); \theta^{Q'})$

Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i; \theta^Q))^2$

Update the actor policy using the sampled policy gradient:

$\Delta_{\theta^\mu} J \approx \frac{1}{N} \sum_i \Delta_a Q(s_i, a; \theta^Q | a = \mu(s_i)) \Delta_{\theta^\mu} \mu(s_i; \theta^\mu)$

Update the target networks:

$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$

$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$

end

end

Figure 3.3: The mini wind-turbine from [KJT]

any input: $\pi : \varnothing \to P(\mathbb{R}^2)$. To run the training, they kept the wind speed constant, let the turbine spool up to speed and then let it run with a set of parameters returned by the policy. They ran these parameters for 2 seconds on the turbine, averaged power output over that time into a reward and then ran an iteration of their algorithm to generate new parameters, starting with random parameters. They first tested the REINFORCE algorithm, with operates without a Q network and trains a policy directly on rewards. This algorithm was discussed above and utilizes the following gradient equation:

$$\Delta_{\pi^\theta} J = \mathbb{E}_{a\sim\pi, r\sim\varepsilon|a} r \Delta \log \pi(a;\theta) \tag{3.22}$$

Note, how they denoted J in their paper as rewards, while we are using r for rewards and J for objective functions. Also, they used ω as a symbol for the action taken, while we use a. Also note how their policy is not dependent on a state but directly predicts actions. They did not explicitely note what they formulated their expectation term on, but we added it here for clarity. Note the similarity to what we discussed in equation 2.7 and 2.13. In addition to this, they came up with an algorithm which they call Trust Region Policy Search, and which is astonishingly different from Trust Region Policy Optimization. With a few additions, they incorporated off-policy learning and second order error approximation into their algorithm, and showed later that it performs nicely. They were able to find optimal pitch and load settings as soon as 15 timesteps after the start of the simulation.

Having described this paper, we want to also summarize what we do differently and why. First, they had a simpler environment. While a full scale windturbine has a strong tendency to destroy itself when outside of their operational margin, their turbine did not fly apart when not applying braking. Also, they were setting electrical resistance, not rotor torque. This system applies a braking force constant to rotational speed times resistance parameter, and thus already acts as a P controller. We were setting torque, so we had to adjust it regularly to rotor speed. In general, their turbine behaved more concavely opposed to our turbine, which with a constant parameter set, would settle to either rotating quickly backwards, forwards or not at all. Because of this, they were able to give the turbine some time at the beginning to stabilize before applying reinforcement learning while we needed to start with the standing turbine. Lastly and most importantly, they aimed to find optimal control settings for a constant wind speed. To create a full controller setup from this, they would have to repeat this process for every windspeed and use the resulting look-up table to derive a PID-controller. We however want to create an end-to-end system and want to experiment with the possibly higher expressive nature of predicting actions based on a full state input. We want to create a version as general as possible. Also, as DDPG is a direct successor on REINFORCE and has proven to outperform it in many areas, we do not reimplement their paper.

Further works on reinforcement learning on windturbine are also [SAZFG$^+$]. This work uses traditional

supervised learning to fit a neural network of 10 neurons to the straight line that describes the relation between yaw angle and power output and because what they got looks similar to a Q function they call it reinforcement learning. [FGFGG] published in a proprietary journal.

4 Experimentation

For everyone who is interested in a recap of our design process, in this section we will provide such. We will again present our algorithm in a later section. To understand our design decisions, it might be interesting to read this passage, but you can also skip this passage and just read the rest.

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. We managed to find a solution with our DDPG implementation. We hoped that the implementation might yield results on our windturbine environment as well 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: $r'(s,a) = r(s) - c * (a_t - a_{t-1})$

- **Penalize death conditions** Penalize when a death condition is reached. With $r(s)$ being another reward function, $d(s) \in 0, 1$ being our indicator for deaths and c a constant penalty, calculate $r'(s) = r(s) - c * d(s)$

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.

Penalize deaths is simple to implement but yields a not differentiable loss function at deaths. Especially because we are learning a continuous value estimation, it will most likely smooth out along a death and likely underestimate the death penalty and underestimate values close to the death. Alternatively we could imagine penalizing getting close to a death condition continously.

Most of our experimentation we did with rewarding rated speed and penalize deaths, as we expected this reward function to be the easiest to learn. In a real application, crafting a good reward function will be a bigger challenge, but this is out of the scope of this work.

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

We first tried a gaussian noise instead of the recommended Ornstein-Uhlenbeck process (OU-Noise) [UO] from DDPG: $n \leftarrow \mathcal{N}(\mu, \sigma^2)$. We didn't see any improvements over our old behavior, so we switched to OU-Noise. Additionally to the DDPG paper, we let the sigma parameter decay over time, so that the noise gets less when training has proceeded. We hope to only need this noise in the first stage of the training and then, later, see better results. Our decay is parameterized by a start sigma, end sigma and end step and then held constant at the end sigma.

4.4.2 Random exploration

The TD3-paper [FvHM] 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 and came up with the following equation, only dependent on rotational speed rot: $\mathcal{C} : (\mathrm{rot}) \rightarrow (\mathrm{torque}, \mathrm{pitch}), \mathcal{C}(\mathrm{rot}) = (2 \times 10^7 * \mathrm{rot} - 12 \times 10^6, 128 * \mathrm{rot} - 103)$. These actions were clipped as in section 3.6.3. So effectively, we designed a P controller, and only using this, we could already achieve relatively stable control. As further work, it could be imagined to use an established controller here, but we did not need this for our purposes. The PID controller was easily able to keep the windturbine in a sensible operation range, and without adding noise it resulted in a smooth and constant operation. We summarize this in algorithm 2

Algorithm 2: Expert exploration

Use replay buffer $\mathcal{R}$ from normal training

for *episode = 1, E* **do**

  Initialize expert controller $\mathcal{C}$ and random process $\mathcal{N}$ for expert exploration

  Receive initial observation state s_1

  for *t = 1, T* **do**

    Select action $a_t = \mathrm{clip}(\mathcal{C}(s_t) + \mathcal{N}_t)$ Execute action a_t and observe reward r_t, death condition d_t and new state s_{t+1}

    Store transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{R}$

    On death condition, reset the environment.

  end

end

We did however not really observe better results with expert exploration enabled. We tried combining it with the algorithm from section 3.6.4, because we were suspecting that the large difference between the actor distribution and the expert distribution lead to problems in learning the respective Q distribution. We argued before that we suspect DDPG not to be able to learn completely off-policy but just a bit off-policy. Together with the pretraining algorithm, we had some slight improvements, but these didn't justify the extra computation time needed for this. Also, though an expert controller which yields somewhat sensible result is easy to build for each turbine, this is a step of human interaction and expert

knowledge, which machine learning generally tries to minimize in algorithm design.

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. Reducing complexity partially tackled the problem of zero gradients, we could then observe small gradients which however vanished over time. Later, we found that two layers of 64 and 32 in the critic and 32 and 16 neurons in the actor also still yielded good gradients with higher generalization potential.

4.5.2 Normalization

More effectively, we added data normalization. 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 they were measured in very different units. 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.

We normalized the data so states, actions and reward usually stayed between 1 and -1. This is a normal technique in machine learning, and we are irritated by the fact that the reinforcement learning community largely made no mention of normalization. At times, rewards were clipped [MKS+a] and observations

were sometimes, especially when working on pixel inputs, cropped to be of square dimensions. Only in 2016, [vHGH$^+$] brought up the topic. They propose an adaptive normalization mechanism which integrates into a generic Reinforcement Learning (RL) algorithm and which can normalize all inputs adaptively. We decided to implement a more straight-forward variant. At first, we used observed states and rewards from the random exploration phase to calculate the 5%, 95% quartile values. We calculated normalization constants from it to linearly normalize those quartiles to $[-1, 1]$. Plain min/max normalization worked less well. As sometimes during random exploration, no extreme values were observed and later extreme values lay far outside of $[-1, 1]$, we instead normalized on replay data of a previous run. We could also imagine manually setting normalization bounds on low-dimensional problems, as human knowledge about the possible state space is usually present. In case when human knowledge about input magnitudes is not present, we recommend the mechanism from [vHGH$^+$], or if you are willing to accept the extra computational effort of running once without normalization, our version.

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.

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. Also, a PID controller has a view of the past, with this extra information, the countroller would be able to generalize further. The result of this however were high and very noisy q-losses and q loss explosions, so we disabled this again.

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. We defined our sensible gradient vector $g = (3 \times 10^4, 0.1)$ for pitch and torque. Clipping action a and previos action a' can be defined as $\text{clip}(a_i, a'_i) = \min(\max(a_i, a'_i - g_i), a'_i + g_i)$. The results of this are again clipped to be in the action space. 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. We summarize this in algorithm 3, parametrized with training time $T = 10000$ and batch size $N = 32$

The training is relatively straightforward, however because we decided to abolish expert exploration, we also abolished this.

4.7 Other improvements

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

Algorithm 3: Policy pretraining

Use prefilled replay buffer $\mathcal{R}$ from expert exploration

Use expert policy $\mathcal{C}$ from expert exploration

for *t = 1, T* **do**

Sample a random minibatch of N states (s_i)

Calculate noise-free expert actions $a_i = \mathcal{C}(s_i)$

Update the actor by minimizing the loss: $L = \frac{1}{N}\sum_i(\mu(s_i) - a_i)^2$

end

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. This method is called Prioritized Experience Replay (PER). According to how much they benefit training means that we add a priority to the experiences, which we somehow derive from the performance of the algorithm on that samples. In their paper, they showed how especially cliffwalk problems, but effectively all problems, benefit from using prioritized experience replay. We argue that our windturbine is similar to such a cliffwalk, as random exploration usually only shortly passes sensible operation range and then quickly destroys the turbine. Seeing this short high reward might not be enough for the controller to learn that being there is good. Furthermore, the technique is generally promising as in Rainbow-DQN [HMvH$^+$], PER made up for most of the gain on the algorithm. In the original case, they use it on DQN, but we will apply it to DDPG. There hasn't been much work on how to use PER in conjunction with DDPG, in fact we only found two papers: [HZ] and [ZLZH]. We will at first present the general prioritized experience replay algorithm, then we will discuss the papers and then present how we incorporated PER.

As knowing how much a sample benefits training is difficult, [SQAS] propose to approximate this importance by the Temporal Difference Error (TD-Error) $\delta = y - Q(s, a)$. 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 priority for each item p_i is stored in the replay buffer. In theory, we would have to calculate this TD-Error every time we update our critic. This would mean to feed our entire replay buffer through the critic on every update step. Clearly, this is a huge performance hit, which is why in the algorithm, they approximated by only updating the error on samples that are anyways being used for training. We calculate the TD-Error anyways for the critic update, so we can store this without performance penalty. New samples are always stored with maximum priority to make sure they are sampled at least once.

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 pure priority based sampling ($\alpha = 1$). Because we introduced oversampling bias by sampling some experiences more often than others, a correction term is applied to the TD-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. This importance sampling is a common technique in reinforcement learning and we have addressed it before in our background part.

In addition to sampling directly based on $|\delta|$, [SQAS] propose a variant where they sample according to the rank of an experience: $p_i = \frac{1}{\text{rank}(i)}$ where rank is defined as the position in the replay memory when sorted according to $|\delta|$. This method is more robust to outliers and slightly increases performance, but because of the extra computational effort of having to calculate the rank every time, we decided to go for the first method. Also, [ZLZH] measured worse performance of the rank based version in contrast to even plain DDPG.

The importance sampling is also the reason why we deem it not so trivial to introduce PER to DDPG, as there is no specified way how to integrate this into policy updates. But first we will describe two other efforts. [HZ] implemented it straightforward on the Q-Learning part of DDPG and ignored any effects on policy updates. They measured improvements tackling the OpenAI Pendulum task in contrast to vanilla DDPG. However, they only evaluated this pendulum task, which we already suspect not to be representative of our windturbine problem. We prefer to rely on the work of [ZLZH] - the team has formulated experience replay as a learning problem and they trained another prediction network on which replay to choose. They evaluated their learnt experience replay against sampling based on rank or priority and found their algorithm to outperform both, while rank-based sampling performed far worse than even vanilla DDPG and priority based sampling slightly better than vanilla DDPG. However, the team completely disregarded importance sampling in their work. Also, we deem it as unnecessary to introduce yet another learning task, especially because we are calculating on weak hardware.

Because neither of the works seemed solid enough for us to implement it, but because the learning improvements in Q-learning are very promising, we decided to create our own mixture of PER and DDPG. We decided to implement the Q-Learning part straightforward as in [SQAS] and including importance sampling. We used the reformulated TD-Error of DDPG with the target actor and critic. To avoid oversampling bias in the policy, we only applied PER-sampling to the training part of the critic and trained the actor on uniformly sampled data. It can be argued, that as the policy μ is anyways being trained on the state density function of another policy ρ^β, it can deal with that oversampling bias. However we doubt adding more bias is beneficial to the training, so we decided to sample uniformly for the policy.

Because we saw performance problems storing the priorities alongside the experiences and directly sampling from a 100000-item weighted array, we followed the advice of PER to use a sumtree datastructure, a binary weighted tree in which each node holds the sum of weights of the children. We implemented sampling as a tree walk starting at the root node. At every node, we normalized the priorities of the two child nodes to probabilities, did a random draw and recursively repeated until we reached a leaf. This recursive sampling technique retains the sample probabilities of the leaf nodes but is cheaper in complexity than to normalize the whole array each timestep and do a random draw over the whole probability distribution. All our operations on the replay buffer are in $O(log(n))$ complexity now.

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.

[FvHM] addressed something similar on actions generated by the target policy, see section 3.8.2 for this.

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 [FvHM] 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. In fact, the authors of DQN have caused some confusion on this in the RL-community. In their journal paper [MKS$^+$b], they write ẅe also found it helpful to clip the error term from the update [...] to be between -1 and 1", which could be interpreted as loss clipping.

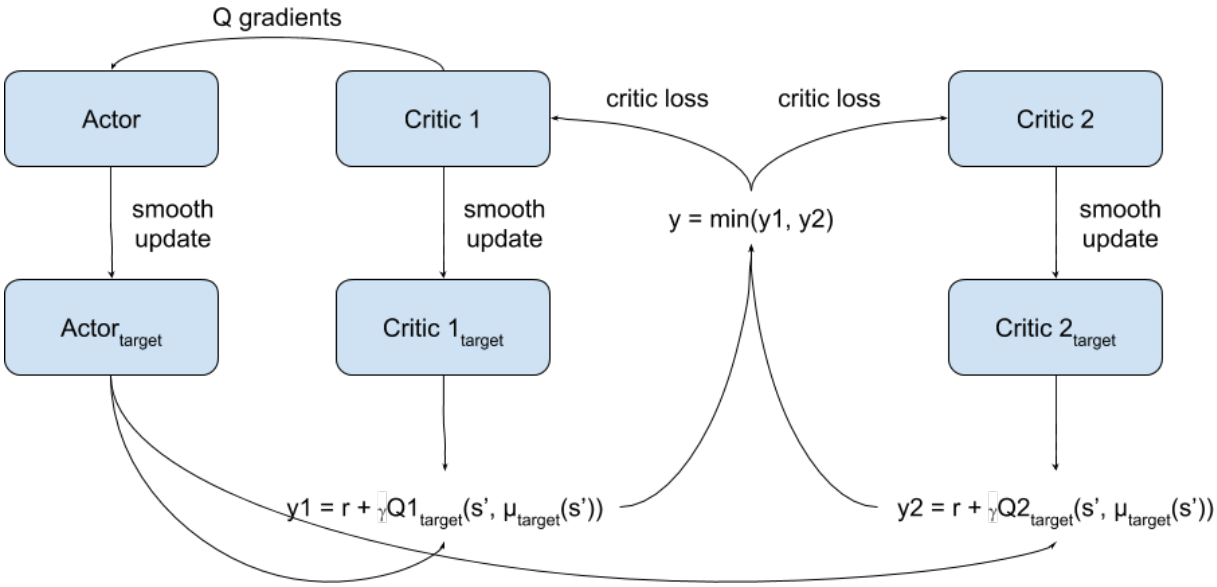


Figure 4.1: Clipped double Q learning

However they actually switched to absolute error instead of square error outside those boundaries, which is the definition for huber loss.

Although loss explosions gotten less frequent and smaller in magnitude with this update, we still saw some. In general we could observe an improvement in training, so we kept this active.

4.8.2 Double critics

We also had a look at [FvHM]. They propose a set of three mechanisms that address function approximation error in DDPG, and they call it Twin Delayed Deep Deterministic Policy Gradients (TD3). At first, they show in section 4.1 that Q learning with a deterministic policy tends to overestimate true Q values. This has been observed in normal Q learning as well, but hasn't been generalized to DDPG before. They show, that the interaction of the policy and the Q network leads to a residual error accumulating over the recursive nature of Q updates. To mitigate this, they at first proposed to use two Q networks and choose the minimum of the two to compute the target y. They combine this with the target network idea, so effectively we end up with 6 networks now. They called this technique Clipped Double Q learning. We illustrate this in figure 3.1. The Q target is computed as

$$y = r(s, a) + \gamma \min_{i \in 1,2} Q'_i(s', \mu'(s'; \theta^{\mu'}); \theta^{Q'_i}) \tag{4.1}$$

As they observed that a quickly changing policy introduces instability into the Q learning process, they proposed to update the policy less often than the actor. They do this simply by updating the policy only

when $t \mod d = 0$ where t is the timestep and d describes how many timesteps to wait before an actor update. They call this *delaying* policy updates. Lastly they proposed adding noise in the target policy calculations to achieve smoothing around policy results. So they changed the target updates again to:

$$\tilde{a} = \mu'(s'; \theta^{\mu'}) + \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -\tilde{c}, \tilde{c}) \tag{4.2}$$

$$y = r(s, a) + \gamma \min_{i \in 1,2} Q'_i(s', \tilde{a}); \theta^{Q'_i}) \tag{4.3}$$

Where $\tilde{a}$ is the target action with added noise on it. In their appendix they describe that they furthermore clip this target action to sensible action range.

When implementing this, we found that we now have two TD-Errors which left us with a choice how to implement PER in conjunction with this. We could either use one of the two errors. When implementing this, we found that the critic whose error we chose for PER performed way better than the other, effectively rendering it useless to have two critics. We also tried to use the maximum and minimum of the two, which yielded more stable results, where minimum seemed better than maximum. Finally, we ended up with adding the two errors for prioritisation.

However, we could not get TD3 to work properly, with or without PER, and didn't see any convergence.

4.8.3 Batch normalization

As in deep learning, batch normalization is found helpful for increasing lr and thus to skip over local maxima [BGSW]. It works by normalizing the outputs between the layers to be of zero mean and unit standart deviation, and for not losing generalization adding an extra learnt parameter per output dimension. We did get a performance boost when using this, however we ended up with NaN values after a while of running it, especially when experiencing q loss explosions. We can explain this because our prioritized experience replay most likely sampled only one item at a timee. To circumvent this, we clipped individual priorities. As these unexpected nan-values made training more complicated, we decided to not use batch normalization.

4.8.4 Regarding death conditions

In DQN, death conditions are accounted for by changing the Q target in the terminal step to be $y = r$ instead of $y = r + \gamma \max'_a Q(s', a')$. This way, behavior after a death will not be accounted for by the Q function. As we are seeing many death conditions, we thought it could be useful to include this into our DDPG algorithm. So now, instead of $y = r + \gamma Q'(s', \mu'(s'))$ we change it to be $y = r + \gamma Q'(s', \mu'(s')) * (1 - d)$ where $d \in 0, 1$ symbolize the death conditions.

4.8.5 Clipping observations

Finally, we found that the solution to our problem lay in qblade. In certain conditions, qblade returned observations 20 orders of magnitude away from normal operation. As we based our reward calculation on the observations, the reward was also far off from normal reward. This condition happened only every 200k steps or so, and as we did not log every timestep, these spikes didn't end up in our logs. We decided to filter out these spikes by clipping rewards and observations after normalization to $[-3, 3]$.

5 Algorithm

After our experimentation phase, we will present our algorithm, on which we perform our evaluation. At first, we describe how we encapsulated the qblade environment. At second, we will present our replay buffer, our network structure, our exploration strategy and our update algorithm.

5.1 QBlade

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 ca 4×10^6 Nm, we used a range of 0 to 8×10^6 as action space, as higher torque can be simulated by qblade and this way made it easier for the controller, as it can apply high torques to slow down quickly rotating blades. 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. We however observed problems with our controller when reaching these extreme limits of the simulation In the course of our experiments, we decided to reset the simulation at an rotational speed of 3 rad/s, as our turbine normally 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. We tried to clip at 0, but this resulted in so frequent deaths that learning actions at this border was not possible.

When resetting the simulation, we let it run for 100 timesteps with zero action and state, as the elastic

structure needs a bit of time to move from simulation default to elastic equilibrium. In this time, simulation results do not make sense yet. Also, every 100000 timesteps or after seeing NaN values we reload the project file to maximize simulation stability.

We hid all observations except for power and rotational speed from the algorithm and also merged all pitch controls into a single variable.

5.2 Core algorithm

As DDPG is capable of learning from a replay of experiences, we implemented a replay buffer with capacity 1e6. To enable PER, we needed to sample according to priorities proportional to a metric, in our case absolute TD-Error. We implemented the replay buffer as a sum tree datastructure, a binary weighted tree in which each node holds the sum of the weights of the nodes below. Also, we did not store probabilities right away but kept the unnormalized experiences $p_i = (|\sigma| + \epsilon)^\alpha$ with an epsilon of 1×10^{-6} and $\alpha = 0.5$.

We implemented our actor model according to the DDPG paper, but with less neurons per layer. Instead of 400 and 300 neurons, we use 32 and 16 neurons. Our critic is also according to the DDPG paper but with less neurons. Instead of 400 and 300 we use 64 and 32 neurons. We use batch normalization between the layers. All states and rewards seen by the network are normalized to be between $[-1, 1]$ 90% of the time and are clipped to $[-3, 3]$. Actions are normalized to $[-1, 1]$. Learning rate for both actor and critic are 1e-4 and we use adam optimization. We train with batch size 128, discounting factor γ = 0.01 and target update rate τ = 0.05. Weights are initialized by uniform random in $[-0.5, 0.5]$.

Critic updates are done similar to DQN updates with PER. We use importance sampling to correct for PER overestimation bias in the critic and redraw samples uniformly for policy training. We regard death conditions in the Q update as in DQN. Also, we use huber loss instead of MSE. All in all, algorithm 4 summarizes all improvements:

Algorithm 4: Our DDPG algorithm

Randomly initialize critic network $Q(s, a; \theta^Q)$ and actor $\mu(s; \theta^\mu)$ with weights θ^Q and θ^μ

Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer $\mathcal{R}$

Calculate normalization factors based on data from a previous run

for *episode = 1, M* **do**

Initialize a random process $\mathcal{N}$ for action exploration

Receive initial observation state s_1

for *t = 1, T* **do**

Select action $a_t = \text{clip}(\mu(s_t; \theta^\mu) + \mathcal{N}_t)$ according to the current policy and exploration noise

Execute action a_t in ε and observe r_t, d_t and s_{t+1} and normalize these

Store transition with highest priority $(s_t, a_t, r_t, s_{t+1}, d_t, \max_i(p_i))$ in $\mathcal{R}$

Calculate sampling probabilities $P(i) = \frac{p_i}{\sum_j p_j}$

Sample a random minibatch of N transitions $(s_i, a_i, r_i, s_{i+1}, d_i)$ from $\mathcal{R}$ according to P

Calculate importance sampling weights $w_i = (N \times P(i))^{-\beta}$ and normalize $w_i = \frac{w_i}{\max_j w_j}$

Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}; \theta^{\mu'}); \theta^{Q'})(1 - d_i)$

Calculate TD-Error $\delta_i = y_i - Q(s_i, a_i; \theta^Q)$

Store TD-Errors in the replay buffer: $p_i = (|\delta_i| + \epsilon)^\alpha$

Apply importance sampling $\delta_i = \delta_i w_i$

Update critic by minimizing the huber loss: $L = \frac{1}{N} \sum_i \begin{cases} (\delta_i)^2 & \text{if} \delta < 1 \\ |\delta_i| & \text{else} \end{cases}$

Sample a random minibatch of N states (s_i) from $\mathcal{R}$ uniformly

Update the actor policy using the sampled policy gradient:

$\Delta_{\theta^\mu} J \approx \frac{1}{N} \sum_i \Delta_a Q(s_i, a; \theta^Q | a = \mu(s_i)) \Delta_{\theta^\mu} \mu(s_i; \theta^\mu)$

Update the target networks:

$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$

$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$

On death condition, reset the environment

end

end

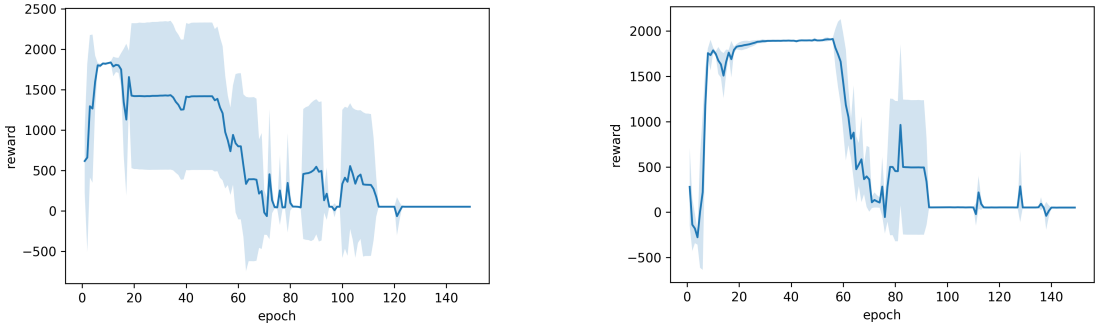
6 Evaluation

For our evaluation, we used a machine with an Intel® i7 4-core processor, 16GB RAM and a Nvidia 1080 GTX with 8GB as accelerator. On this system, a fully realistic run with 300k steps would take roughly 200 hours, of which almost everything would be used by qblade. Partly this is because we didn't manage to enable hardware acceleration because of compilation problems, but computing a fluid simulation is always time-consuming. We could bring the computation time down to 12 hours per run by deactivating wake calculations and using a different structural computation method in qblade. Wake calculations are responsible for calculating the effects of the rotor on the surrounding air. Without this effect, the rotor can draw infinite energy from the air as it can generate lift without slowing down the air. Effectively, this increased instability of the system, as now the rotor does not get slowed down at high rotational speeds, but otherwise we would not have been able to evaluate our work.

We ran every experiment 4 times except stated otherwise. We visualize our measures as confidence plots, for which we calculated the mean and a 90% confidence interval over the results. The mean is displayed as a solid line, while the confidence is plotted as a colored background.

6.1 Hold speed

We start off with the simplest task. The task to solve was to hold a rotational speed of 0.8 rad/s over the time of 2000 steps, starting with a simulation that has artificially been stabilized at the desired rotational speed with 0 degrees pitch and 0 generator torque. Wind-speed is kept constant at 11m/s. The agent gets a reward of 1 per step if it exactly matches rotational speed, deviation from it is linearly punished, 0 rad/s would yield a reward of 0. Thus, the theoretically possible maximum reward would be 2000, holding the turbine at perfect speed for the entire length of the simulation. However, as we start with 0 pitch and torque, action gradient limitations will naturally incur a small penalty at the beginning as these values need to be adjusted to sensible values and the resulting rotational speed deviation needs to be recovered. A perfect policy would thus reach a score of a litte bit below 2000. This task is simple because it can be achieved either through pitching out, increasing generator torque or a combination of both. This task is not realistic, as it can be solved without generating energy whatsoever.



(a) critic lr=1e-3

(b) critic lr=1e-4

Figure 6.1: Learning rate comparison for the critic network

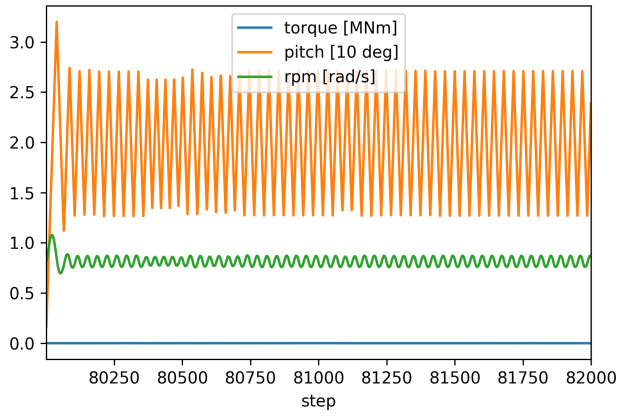


Figure 6.2: Pitch, torque and rotational speed from a sample epoch

We look at two different learning rates for the critic network, 1e-3 (a) and 1e-4 (b). The latter is what was recommended in the DDPG paper.

Each of the controllers reaches close to theoretical optimum and is able to hold it for a number of epochs. With learning rate 1e-3, we reach it somewhere around episode 10, with learning rate 1e-4 a bit later around epoch 25. The time we are holding it before divergence also raises with learning rate. The most aggressive learning rate reached 1828.8 average epoch reward between epoch 7 and 11 and a single maximum of 1917.5 points. On the lower learning rate, it held an average of 1896 points from episode 30 to 56 and reached a maximum of 1968.4 points. The maximum we could reach with our hand-designed PID controller was 1981 points, where the 19 points were lost through the swing at the beginning and then rotational speed was held perfectly.

Interesting to see is the strategy which the agent found. As described before, both increasing pitch and increasing torque reduces rotor speed, so through both it would be possible to control the rotational speed. However, all of our runs decided to implement pitch-only control and kept torque constant at zero, as possible to see in figure 5.2. A possible explanation for this decision could be that with high pitches, lower forces act on the blades, thus rotational speed is easier to control. However, this insight is

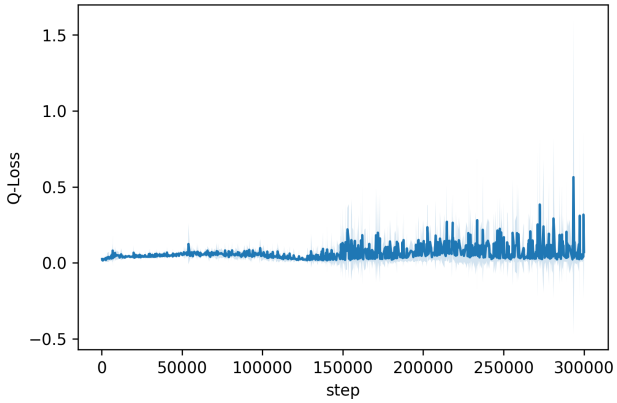


Figure 6.3: Critic q loss during divergence

above our human knowledge of windturbines. Furthermore should be noted that rotational speed is still varying between 0.75 and 0.85 and the pitch is ranging from 15 to 25 degrees (note the scaling of the pitch and torque). In a realistic scenario, oscillations of this magnitude would likely destroy the turbine very soon, as pitching the blades around this quickly creates strong material stress.

With the aggressive learning rate, we can clearly see how quickly the algorithm is diverging. After a short period of convergence around epoch 10, the high learning rate steps the critic out of the minimum and causes instabilities. With even higher learning rates, we observed not reaching convergence at all. Interestingly, we can not see such divergence in the loss. In normal machine learning, a too high learning rate would cause a phenomenon of jumping across a minimum, taking gradient steps greater than the width of the local minimum. This would reduce in an increase in loss. Here, however, the q loss stays small on such divergence, as plotted in figure 5.3 for the diverging run of critic lr=1e-3. This is because we are generating our prediction targets based on delayed versions of the same net, as we do not know a ground truth. As this delayed target net will diverge with the original net if divergence happens slow enough, we do not see it.

6.2 Hold rated power

A more difficult task to solve, but also a more realistic one it to hold a rated power. Analog to our hold speed task, the agent gets a simulation which was artificially stabilized at 0.8rpm with pitch and torque 0, and should generate exactly 5MW of power during an epoch of 2000 steps. This power is the reference output of the NREL 5MW turbine, and in our wind scenario it should theoretically be able to reach this power. Wind speed was kept constant at 11m/s. Again, the agent gets a reward of 1 for hitting 5MW perfectly, and 0 for having no power at all, so a perfect policy would get something a little below 2000. Unfortunately, we did not get any policy to achieve something close to optimal, the best score we

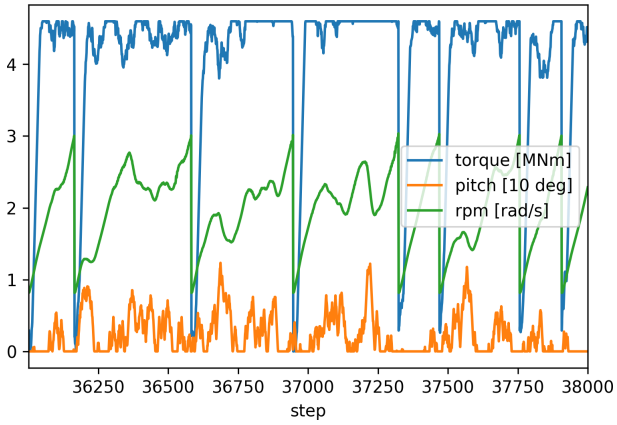


Figure 6.4: An epoch from the hold power task

achieved was 1500 points with an average of 5 kills per episode. Unfortunately, this result wasn't very stable, so we can not plot an average over many runs. An example for a relatively successful epoch is figure 5.4. Here, the controller maxed out torque and tried to hold the turbine stable with pitch. Power output for these insane regions was up to 9MW, which would have instantly roasted any electronics in a real wind turbine. Also, blades would have taken serious damage on these rotational speeds.

At the end, most controllers converged to leaving pitch at zero and left the turbine to die regularily with maximum generator torque.

6.3 Hold power

The next task

6.4 Pendulum

To exclude the possibility that our implementation is buggy, we decided to run our DDPG algorithm on the Pendulum task. With our hyperparameters, we could not observe convergence and only switching back to the algorithm of DDPG, the agent managed to balance out the pendulum after 11 epochs. Especially the network size hyperparameter was important for this.

6.5 Discussion

Commonly, algorithms are evaluated over many different problems. The OpenAI Gym suite is the common suite for evaluating reinforcement learning algorithms, and typical problems in this are balance

problems (all cartpole* and pendulum* tasks) and actuation tasks (gripper*, reacher*, cheetah*). Balance tasks could be argued to resemble our windturbine in a way, as we want to balance an output aswell. However a windturbine is vastly more complex than a pendulum. It exhibits resonant frequencies in addition to the optimization problem which the controller needs to keep under control. Our experiments showed that it didn't. We were able to solve easy tasks, in which resonances of high magnitude still lead to acceptable results, however as soon as it got more difficult, high swinging lead to our control task not being accomplished. Also, noisy sensor measurements were a more difficult task to learn. We could not implement the same size of network and thus not achieve the same learning potential as the authors in DDPG.

7 Future work

Just because we didn't manage doesn't means it's impossible. We could imagine that with some additional work, learning stable policies would be possible. We have some ideas how we could achieve this result.

7.1 Scale up

It was limiting to perform all our experiment on the tiny machine. Limiting factor in this is currently the simulation, even with the lowered accuracy, which currently uses 70% of the computation time, while we only use 30% to train our networks. As currently, linux hardware acceleration is broken in qblade, a nice idea could be to fix this and to run future experiments on a cluster. This would also allow efficient hyperparameter sweeps, as we saw that hyperparameter choice greatly affects the outcome of the simulation.

This would also allow to evaluate different wind scenarios and simulations with enabled wakes.

7.2 Validate PER

We did not validate our combination of PER and DDPG. For our task, it gave better results, but it would be interesting to know whether our algorithm might also yield better results on general reinforcement learning tasks. This might also yield some insights into what makes our case special in comparison to general reinforcement learning.

7.3 Train PID inputs

As PID controllers have proven well in controlling a turbine and neural nets directly performed poorly, we could imagine to predict PID controller parameters and error targets instead of directly building the controller. Alternatively, feeding the algorithm with derivative and integrative inputs similar to what we described in section 3.6.2 could enable the policy to build more resonant-robust results.

7.4 Reward functions

We did all our training under the simple hold-speed or hold-power reward function. A realistic reward function however would incorporate power generated and long-term damages occured. Especially the last part is tricky, as it would somehow have to be included into the scalar reward function. An interesting approach could be to try to train under multi-dimensional reward functions. To our knowledge, there haven't been efforts to accomodate this into Q-learning. Also, we could imagine interpolating between different 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 complex goals such as damage prevention more. This could be achieved by increasing the magnitude of the reward function further down in training, like after focussing on the big chunky goals zoom in on more fine-grained problems.

7.5 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

7.6 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 to what we deem 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.

7.7 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.

Bibliography

[Aga] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU).

[BCP+] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym.

[Bel] Richard Bellman. The theory of dynamic programming. 60(6):503–516.

[BGSW] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding Batch Normalization. page 12.

[Bis] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press ; Oxford University Press.

[BJSB] Tony Burton, Nick Jenkins, David Sharpe, and Ervin Bossanyi. *Wind Energy Handbook: Burton/Wind Energy Handbook*. John Wiley & Sons, Ltd.

[BPS+] Shalabh Bhatnagar, Doina Precup, David Silver, Richard S Sutton, Hamid R Maei, and Csaba SzepesvÃ¡ri. Convergent Temporal-Difference Learning with Arbitrary Smooth Function Approximation. page 9.

[BW] J. J. Barradas Berglind and Rafael Wisniewski. Fatigue Estimation Methods Comparison for Wind Turbine Control.

[CKH+] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying Generalization in Reinforcement Learning.

[DWS] Thomas Degris, Martha White, and Richard S. Sutton. Off-Policy Actor-Critic.

[FGFGG] Borja Fernandez-Gauna, Unai Fernandez-Gamiz, and Manuel Grasa. Variable speed wind turbine controller adaptation by reinforcement learning. 24(1):27–39.

[FMP] Scott Fujimoto, David Meger, and Doina Precup. Off-Policy Deep Reinforcement Learning without Exploration.

[FvHM] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods.

[HLD] Michael F. Howland, Sanjiva K. Lele, and John O. Dabiri. Wind farm power optimization through wake steering. 116(29):14495–14500.

[HMvH+] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning.

[Hub] Peter J. Huber. Robust estimation of a location parameter. 35(1):73–101.

[HZ] Yuenan Hou and Yi Zhang. Improving DDPG via Prioritized Experience Replay. page 10.

[JBMS] J. Jonkman, S. Butterfield, W. Musial, and G. Scott. Definition of a 5-MW Reference Wind Turbine for Offshore System Development.

[KB] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization.

[KJT] J. Z. Kolter, Z. Jackowski, and R. Tedrake. Design, analysis, and learning control of a fully actuated micro wind turbine. pages 2256–2263. IEEE.

[LHP+] 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.

[MB] Michael C. Mozer and Jonathan Bachrach. Discovering the Structure of a Reactive Environment by Exploration. 2(4):447–457.

[MKS+a] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning.

[MKS+b] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. 518(7540):529–533.

[MWP+] D Marten, J Wendler, G Pechlivanoglou, C N Nayeri, and C O Paschereit. QBLADE: AN OPEN SOURCE TOOL FOR DESIGN AND SIMULATION OF HORIZONTAL AND VERTICAL AXIS WIND TURBINES. 3(3):6.

[PHD+] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter Space Noise for Exploration.

[SAZFG+] Aitor Saenz-Aguirre, Ekaitz Zulueta, Unai Fernandez-Gamiz, Javier Lozano, and Jose

Lopez-Guede. Artificial Neural Network Based Reinforcement Learning for Wind Turbine Yaw Control. 12(3):436.

[SB] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, second edition edition.

[SHM+] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. 529(7587):484–489.

[SLH+] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. page 9.

[SMSM] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. page 7.

[SQAS] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay.

[UO] G. E. Uhlenbeck and L. S. Ornstein. On the Theory of the Brownian Motion. 36(5):823–841.

[vHGH+] Hado van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude.

[vKPN+] G. A. M. van Kuik, J. Peinke, R. Nijssen, D. Lekou, J. Mann, J. N. SA¸rensen, C. Ferreira, J. W. van Wingerden, D. Schlipf, P. Gebraad, H. Polinder, A. Abrahamsen, G. J. W. van Bussel, J. D. SA¸rensen, P. Tavner, C. L. Bottasso, M. Muskulus, D. Matha, H. J. Lindeboom, S. Degraer, O. Kramer, S. Lehnhoff, M. Sonnenschein, P. E. SA¸rensen, R. W. Konneke, P. E. Morthorst, and K. Skytte. Long-term research challenges in wind energy“ a research agenda by the European Academy of Wind Energy. 1(1):1–39.

[Wil] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. page 28.

[ZLZH] Daochen Zha, Kwei-Herng Lai, Kaixiong Zhou, and Xia Hu. Experience Replay Optimization. page 7.

Acknowledgments

At first thanks to my advisors Mattew Lennie and Dmitrij Shlezinger for their support. Also, I guess, thanks David Silver - 7 of my most important papers were from you. Thanks to my laptop for almost dying, but then surviving. And big thanks to the mighty god of windturbines for letting me crush several thousand windturbines for science.