Condition monitoring system for wind turbines

– based on deep autoencoders

Master's thesis in Complex Adaptive Systems

Niklas Renström

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden, 2019

Condition monitoring system for wind turbines

– based on deep autoencoders

Niklas Renström

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden, 2019

Condition monitoring system for wind turbines
– based on deep autoencoders
Niklas Renström

Supervisor: Pramod Bangalore, Greenbyte AB
Examiner: Kristian Gustafsson, Department of Physics, Chalmers University of Technology

Department of Physics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Cover image: Wind industry professional on a site with ten wind turbines. Processed photo.

Typeset in LaTeX
Gothenburg, Sweden 2019

Abstract

Wind turbines consist of many mechanical, electrical and hydraulic components. Failures in any of these can lead to high financial loss, both as actual repair costs and from lost production time. Often, failures do not occur instantaneously but rather as a consequence of sequential degrading. Therefore, many failures can be detected in advance using so-called condition monitoring systems. Through their supervisory control and data acquisition system, modern wind turbines store information about their operating state. Among other things, signals such as produced power, wind speed, and various component temperatures are recorded. In this thesis, a condition monitoring system that leverages this data is developed. The system is based on deep autoencoders, a type of neural network that learns to reconstruct its input data. By training an autoencoder on data from a healthy wind turbine it can learn the dependencies between different SCADA signals under normal conditions. If it then gives a poor reconstruction for new data, it is likely that something has changed in the internal dynamics of the wind turbine which could indicate a degraded component.

Previously, many similar systems have been developed. These have shown good results and detected multiple component failures up to months in advance. However, they usually only monitor one component at a time and are therefore not able to provide a complete condition monitoring system. Autoencoders, which do not suffer this problem, have also been investigated but not at a larger scale.

In this thesis, a relatively large, labeled dataset was utilized. With this data, the efficiency of condition monitoring systems based on autoencoders was tested on a variety of real faults. Moreover, the influence of various properties of the autoencoder was investigated. The results of the investigation showed that an autoencoder based condition monitoring system is capable of detecting a variety of failures in wind turbines. Finally, suggestions for future developments are discussed in the thesis.

Keywords: *Wind turbine, condition monitoring system, anomaly detection, preventive maintenance, SCADA, machine learning, unsupervised learning, neural network, autoencoder, hyperparameter selection*

Acknowledgments

I would like to thank the whole of Greenbyte for making this thesis possible. In particular, I would like to thank Pramod for his patience and guidance, Edmund for his never-ending help with all sorts of strange issues and Thomas for his light-heartedness and support.

I would also like to thank my examiner Kristian for all the reviews and text discussions; they were not required of you and I am grateful for your aid.

Not only for this thesis, but for my entire education: Tack Mamma, Pappa, Ola och Anette—utan er hade det inte varit samma sak!

Finally, I would like to thank all my friends—past, present, and future—you continue to teach me new things and make life not boring.

Contents

Glossary

AE	Autoencoders
ANFIS	Adaptive neuro-fuzzy inference systems
ANN	Artificial neural network
DAE	Deep autoencoders
DBSCAN	Density-based spatial clustering of applications with noise
DFN	Deep feedforward neural network
ELM	Extreme learning machine
EWMA	Exponentially weighted moving average
GPR	Gaussian process regression
kNN	k-nearest neighbours
MAPE	Mean absolute percentage error
MD	Mahalanobis distance
NSET	Non-linear state estimation technique
PCA	Principal component analysis
RBM	Restricted Boltzmann machines
ROC	Receiver operating characteristics
RUL	Remaining useful life
SCADA	Supervisory Control And Data Acquisition
SDP	State-dependent parameter model
SGD	Stochastic Gradient Descent
SLFN	Single hidden layer feedforward neural network
SOM	Self-organizing map
SVM	Support-vector machines
WT	Wind turbine
ZCA	Zero-phase component analysis

Chapter 1

Introduction

1.1 Background and problem formulation

Climate change is one of the greatest challenges of our time according to the UN [1]. The Paris Agreement, signed by 195 countries and entered into force on the 4th of November 2016 aims to limit the rise of global temperature to 2 °C over pre-industrial levels [2]. For this to be achieved it is imperative that energy production is moved from non-renewable resources such as oil and coal to renewable resources such as wind and solar. Facilitating this change requires economic incentives for the construction of renewable energy and one of the most straightforward ways of achieving this is by reducing the overall production cost.

Wind power stands for a large part of the world's current renewable electricity with 23.6 % of the total capacity in 2017 and 28% of the newly installed capacity between 2016 and 2017 [3]. Operations and maintenance costs contribute between 11% and 30% to the levelized cost of energy [1] for onshore wind farms and an even greater part for offshore wind farms [4]. A cost reduction in this area will therefore greatly impact the competitiveness of wind power and green energy as a whole.

Wind turbines (WT) are complex machines subjected to highly variable environmental conditions. They can fail in numerous different ways and failures are usually expensive to repair and cause long downtimes. This is especially true for WTs located at inaccessible locations such as offshore. Many failures are not caused by a sudden event but rather stem from continuous wear and tear and the subsequent degrading of components such as gears and bearings. Consequently, many critical failures could be prevented if the worn down component was identified.

Due to the large scale of modern-day wind farms, manual inspection of all turbines is very labor intensive and, instead, a lot of effort has been put

[1] The levelized cost of energy is defined as the price of electricity required for an installation to break even over its lifetime [4].

into developing automated systems that can detect damaged components. Many different techniques can be used to develop such condition monitoring systems (CMS) and among the most common are vibration analysis, oil analysis and strain measurements [5]. However, such systems require the installation of additional measurement devices which typically comes with high installation costs [6]. Furthermore, while vibration analysis-based systems that monitor mechanical parts such as the gearbox typically perform well, other systems that monitor electrical and hydraulic parts are not equally developed and are less capable [6]. Because of these drawbacks, there is a demand for different kinds of CMS's that do not require significant installation costs and perform well for the entire system.

1.2 CMS based on the Supervisory Control And Data Acquisition system

All modern WTs have a supervisory control and data acquisition (SCADA) system installed. This system allows the user to remotely control and monitor the WT. It also records a number of different signals from the WT's operation that can be used for condition monitoring. The signals can be sorted into four different categories: environmental signals, such as wind speed and ambient temperature; monitoring signals, such as gearbox oil temperature and gearbox bearing temperature; electrical signals, such as power output and generator voltages; and control variables, such as pitch angle and yaw angle. The data can be seen as a digital representation of the WT and a lot of work has been put into developing a CMS based on this SCADA data.

Most SCADA-based CMS can be divided into three steps: *preprocessing*, *anomaly detection*, and *postprocessing*. In the preprocessing step the SCADA data is collected and prepared for further analysis. Typically this involves removing timestamps where some signal was missing and scaling all signals to be in the same range.

It is in the anomaly detection step where the actual data inspection happens. Usually, some kind of model is built from healthy data and is then used to score the "normality" of new data. The effectiveness of a given model can make or break the entire CMS and a lot of research has been put into investigating different anomaly detection models and finding the ones best suited for WTs.

The incoming SCADA data can be quite noisy. Wind gusts, turbine startups, and sensor misreadings can all trigger an anomaly detection model without necessarily implying that the WT is malfunctioning. For this reason, it is important to further analyze the outputs of an anomaly detection model in order to distinguish true faults from noise; this is done in the postprocessing step. Usually, postprocessing systems work under the assumption

that, given a true change in the WT dynamics, the anomaly detection model will be triggered more often and indicate larger deviations from normality than what would happen under normal conditions. Typical methods include threshold systems and moving averages, but, since the postprocessing is dependent on the anomaly detection model used, there is no "best" method.

1.2.1 Previous studies

Various researchers have investigated the effectiveness of a SCADA-based condition monitoring system for WTs. Mainly, different anomaly detection models have been compared. An extensive literature review can be found in Appendix A and Table 1.1 presents a summary of the models and methods investigated. It can be observed that the gearbox and generator are the most investigated components. The interest in gearbox and generators is expected as they fail often, detecting failures in them is relatively simpler, and there exist economic incentives to detect failures in these components at an early stage.

Model/Method	**References**	**Case studies presented for**
Self-organizing map	[7]	Gearbox
Nonlinear state estimation technique	[8]	Gearbox
Density-based clustering	[9]	Generator
Polynomial fit	[10], [11], [12]	Generator, Blades, Main bearing
Statistical methods	[11], [13], [14], [15]	Gearbox, Generator, Blades
Gaussian Processes Regression	[16], [17]	Not applicable
Adaptive neuro-fuzzy inference system	[16], [18], [19]	Various components
Single hidden layer feed-forward neural network	[20], [21], [22], [23], [14], [16], [24]	Gearbox, Generator
Deep feed-forward neural network	[12]	Gearbox
Autoencoder	[7], [25], [26]	Gearbox, Generator, Blades

Table 1.1: Summary of the models and methods presented in Appendix A used for detecting different faults.

Methods based on neural networks; single hidden layer feed-forward neu-

ral network (SLFN), deep feedforward neural networks (DFN) and adaptive neuro-fuzzy inference systems (ANFIS) are the most applied and have in comparative tests often outperformed other kinds of anomaly detection models [16], [22], [14]. Typically, these models work by reconstructing one of the available SCADA signals from a subset of the others. Next, the value of the reconstructed signal is compared with the actual signal value and a *reconstruction error* is calculated. This serves as the basis for an anomaly detection model where a higher than usual reconstruction error indicates that something has changed in WT dynamics. These models have shown good results by detecting faults well in advance. Since these models only reconstruct a single signal, however, a fault will only be detected if it affects that signal. Therefore, these models lack an overall system perspective and cannot detect all faults that can occur in a WT. In [18] and [19] Schlechtingen et. al attempted to employ one neural network model for each subsystem of a WT to get more complete coverage and their system was successful at detecting a variety of faults. To select a proper mapping between inputs and outputs, a combination of data reduction techniques and a physical understanding of the system was, however, required. Since different types of turbines often have different sets of signals this could become troublesome if the system was to be employed at scale. Moreover, the system requires a large number of neural networks to be trained which can also cause issues for scalability.

Lately, anomaly detection models have been constructed using a different kind of neural network, the autoencoder (AE). These networks have proven successful at detecting both blade issues [25] and sensor failures [26]. Since autoencoders reconstruct all of their input signals they are capable of detecting faults in many different components of a WT. Furthermore, since only one network needs to be employed per WT, some of the scalability issues previously mentioned are mitigated.

1.3 Aim of the thesis

In this thesis, a condition monitoring system for wind turbines which utilizes SCADA data will be developed. Drawing upon the valuable knowledge generated by previous academic research, this system will utilize deep autoencoders for anomaly detection. Autoencoders have the theoretical capability to provide a complete condition monitoring of a WT and they should be possible to employ at a large scale. In previous studies [25], [26], autoencoders have proven capable of detecting a variety of faults, but they have still not been thoroughly investigated at a large scale with realistic, 10 minute sampled SCADA-data.

One of the main contributions of this thesis is to study the effectiveness of autoencoders as anomaly detection models for wind turbines. A relatively

large dataset is available where six different types of anomalies could be identified using service logs. This enables investigating different properties of the autoencoder in terms of their actual influence on anomaly detection performance.

1.3.1 Scope of the thesis

This thesis focuses on investigating anomaly detection models for WTs based on deep autoencoders. To test these, a full CMS with additional preprocessing and postprocessing steps is employed. These, however, are of a secondary nature and no thorough comparison between different possibilities is performed.

Different autoencoders are the only anomaly detection models tested. The purpose of this work is not to do an exhaustive comparison of all possible models but rather to find one or a few that works well from a system-wide perspective. Further, since many anomaly detection models only monitor at a component level, a fair comparison between these and autoencoders would require the development of one such model per component which is a major work beyond the scope of this thesis.

1.4 Thesis structure

The rest of this thesis is structured as follows:

- Chapter 2 provides the theoretical background to the methods and models employed such as artificial neural networks and exponentially weighted moving averages. In particular, it provides detailed information on the training and application of artificial neural networks as this is a major part of the thesis.
- Chapter 3 describes the process of designing autoencoder models for anomaly detection and gives information on how and why certain design decisions were made. It further discusses the preprocessing of SCADA data, practical implementation details and presents a study of different hyperparameters' effect on training and validation errors.
- Chapter 4 presents a large validation study which, using a labeled dataset, compares different autoencoder models and evaluates their effectiveness as anomaly detection models. Further, the postprocessing of reconstruction errors is discussed.
- Chapter 5 discusses the findings in the thesis and proposes future work. Finally, the thesis is concluded.

Chapter 2

Theoretical background

This chapter presents a theoretical background to the tools and concepts used in this thesis.

2.1 Artificial Neural Networks

Modern day computers are capable of performing trillions of computational operations every second [27] and have revolutionized the way our society works. Still, some tasks that the human brain perform on a daily basis such as holding a conversation or understanding pictures are very difficult for a computer. The field of Artificial Intelligence is devoted to the simulation of such *intelligent behaviour* in machines and one of the most successful models is the Artificial Neural Network (ANN). ANNs are loosely based on the structure of the human brain and can perform complex information processing. Typically, they can be described as a transfer function from some input $\boldsymbol{x}$ to some output $\boldsymbol{y}$. The inputs can be anything from instrument readings to light-intensity of pixels but they are almost always encoded as a single or a vector of real numbers, $\boldsymbol{x} = (x_1, x_2, \ldots, x_m), \quad x_i \in \mathbb{R}$. The outputs are given in a similar form, $\boldsymbol{y} = (y_1, y_2, \ldots, y_l), \quad y_i \in \mathbb{R}$, but can be interpreted as the value of a different signal, as the probability of an image belonging to a certain class or as anything else that can be represented by real numbers.

The following Sections goes more in-depth of the various parts of an ANN and how it can be trained and "learn" from input data.

2.1.1 Model of a neuron

The human brain is a network of neurons (or nerve cells) that receive, process and transmit electrical or chemical information. Similarly, an ANN is a network of artificial neurons. Each artificial neuron receives a set of input variables and produces an output. First, all of the inputs are weighted and

then they are combined with a bias in a sum. Next, this weighted sum is passed to an activation function $\varphi(\cdot)$ which introduces a nonlinearity to the transfer function of the neuron. A diagram of the procedure is shown in Figure 2.1.

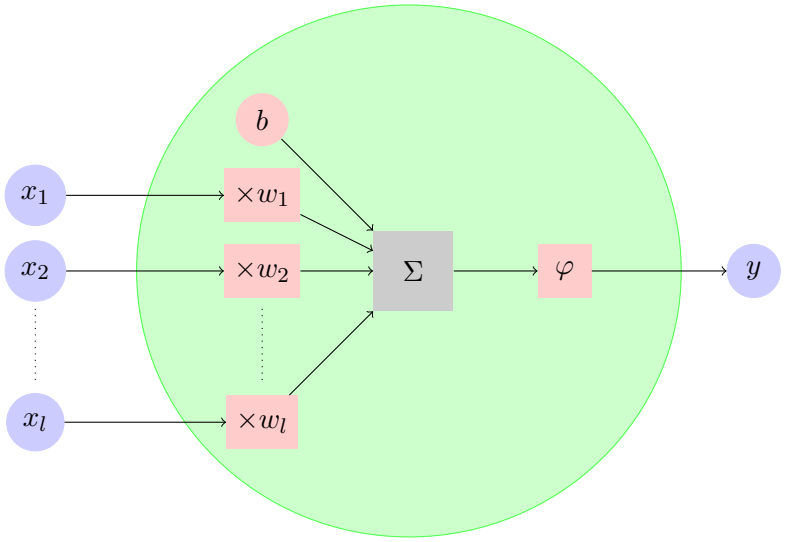

Figure 2.1: An artificial neuron that transforms the input $\boldsymbol{x}$ as

$$\boldsymbol{x} \to y = \varphi(v), \quad v = \sum_i w_i x_i + b$$

2.1.2 Activation functions

The activation function $\varphi(\cdot)$ in Figure 2.1 is important since it introduces a nonlinearity to the transfer function of a neuron. Without this, it doesn't matter how we combine different neurons: if all we do is pass linearly transformed data around all we're going to get in the end is linearly transformed data.

There are many different activation functions that one can think about and below some are described, starting from the simplest ones to the ones currently in use.

Step function

The real neurons in the human brain fire when the input signal is strong enough. Inspired by this, one of the first artificial neurons, the *Perceptron*, used a simple step function to make a binary classification of points [28].

A step function is defined as

$$\varphi(v) = \begin{cases} 1 & \text{if} \quad v > 0, \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

and can thus have an output of either 0 or 1 depending on if the weighted sum is negative or positive.

Logistic function

Since most ways of training a network (see Section 2.1.6) relies on derivatives it is beneficial if the activation function of a neuron is differentiable everywhere. The logistic function,

$$\varphi(v) = \frac{1}{1 + \exp(-v)}, \tag{2.2}$$

has this property and can be seen as a "smooth step function". This and similarly shaped (together called sigmoid) activation functions have been widely used in ANNs, especially for classification.

Rectified linear unit

A problem with both the step and the logistic function is that of vanishing gradients. If the input v is too high or too low the gradient of the activation function will be zero or close to zero and the algorithms used for training (Section 2.1.6) cannot sense in which direction they should shift the weights to get a better output. To counteract this problem the activation function called Rectified Linear Unit (ReLU) is commonly used. ReLU maps negative inputs to zero and positive inputs to themselves:

$$\varphi(v) = \begin{cases} v & \text{if} \quad v > 0, \\ 0 & \text{otherwise} \quad . \end{cases} \tag{2.3}$$

In comparison with the logistic function, the derivative is quick to calculate and, for positive inputs, the problem with vanishing gradients is removed. For negative inputs, the problem still exists and is referred to as "dying ReLU". In some cases, however, this can be a strength instead of a weakness since it causes a regularization (Section 2.1.7) of the network. During training, some nodes will become deactivated for some of the inputs and the network will thus get a lower and lower complexity as the training proceeds.

Because of these benefits, ReLUs have become the most commonly used activation function for deep neural networks in the most recent years [29].

2.1.3 Network structure

A single artificial neuron is not capable of modeling very complex phenomena. In fact, it was shown that the Perceptron, one of the first artificial neurons which learnt to classify points into one of two classes, only had the capability to solve linearly separable problems [30]. To overcome this problem researchers started investigating connecting neurons to each other in a network, just as in the human brain. Several network structures have been proposed and different structures have proven successful at solving different kinds of problems. Below, three different architectures, most closely related to the work in this thesis are discussed. They are all subsets of the class *feedforward* neural networks, in which neurons are organized in consequent layers and information flows forward from one layer to the next.

2.1.4 Single hidden layer feedforward neural networks

The simplest kind of feedforward neural network is the single hidden layer feedforward neural network (SLFN). The network consists of the input nodes, a single layer of artificial neurons (referred to as the hidden layer), and the output node(s). Each neuron in the hidden layer receives a weighted sum of the inputs and this value is passed through its activation function which produces an output, just as in Figure 2.1. After the hidden layer, all outputs are once again combined in a weighted sum to produce the final output of the network. The whole architecture is shown in Figure 2.2.

Training the network refers to adjusting all its weights so that it produces the desired output. In [31], Hornik et. al showed that, given a large family of activation functions, such a network is a *universal approximator*, i.e. it is capable of approximating a large set[1] of transfer functions with an arbitrary precision given a sufficient number of artificial neurons in the hidden layer.

2.1.5 Deep feedforward neural networks

According to the universal approximation theorem [31] a SLFN is capable of approximating almost any transfer function. However, the number of hidden units required is not fixed and, in the worst case scenario, it can grow exponentially with the number of input configurations that need to be distinguished [32]. To overcome this problem one can add *depth* to the network. Depth means that instead of one large hidden layer the network has several smaller hidden layers put in sequence, see Figure 2.3.

The sequence of layers allows the network to decompose the transfer function: to identify a cat in an image it can first identify that there is an animal and in the next step that the animal is a cat. The following example illustrates why depth has the potential to greatly reduce the total number

[1] Borel measurable functions

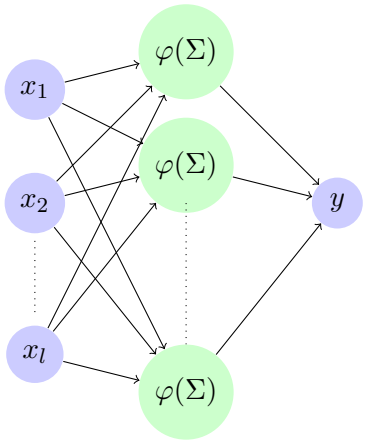

Figure 2.2: Architecture of a single hidden layer feedforward neural network.

of neurons. Suppose we want to classify the type and colour of a vehicle in an image. We may have images with bikes, motorcycles, and cars and each one of these might have the colour red, green or yellow. If we want to perform the classification in one step we would have to separate $3 \times 3 = 9$ different combinations and need equally many decision units. However, we can split the decision making into two steps and first determine the type and then the colour. This way we disentangle the decisions and since they do not depend on each other we would only need $3 + 3 = 6$ decision units.

There is no way of proving exactly how beneficial depth is and when it is needed since the transfer functions we wish to model vary a lot and are inherently unknown. There is, however, empirical evidence showing that greater depth seems to give better generalization results for a wide variety of tasks [32].

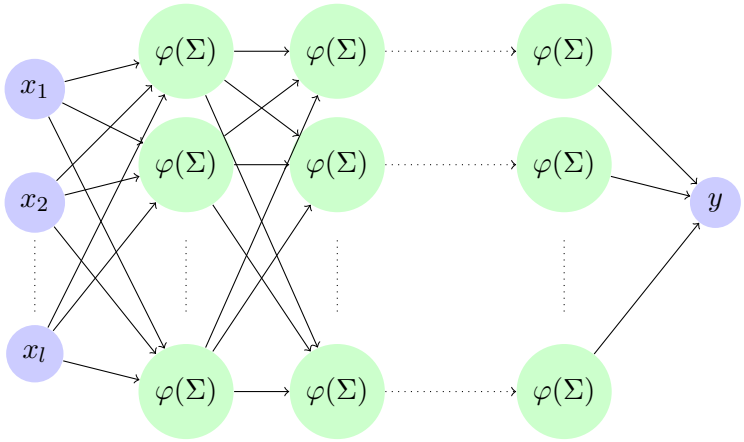

Figure 2.3: Architecture of a single hidden layer feedforward neural network.

2.1.6 Training of a network

Since the transfer functions we wish to model with ANNs are inherently unknown, the weights of the networks need to be *learnt* from data. This procedure is referred to as *training* of a network. A training dataset, containing a set of inputs $\{\boldsymbol{x}_i\}_{i=1}^N$ and their respective targets $\{\boldsymbol{y}_i\}_{i=1}^N$, is provided. The inputs are fed through the network and the outputs $\{\hat{\boldsymbol{y}}_i(\boldsymbol{x}_i)\}_{i=1}^N$ are compared to the desired output $\{\boldsymbol{y}_i\}_{i=1}^N$ using a *cost function* $C(\{\hat{\boldsymbol{y}}_i\}_{i=1}^N, \{\boldsymbol{y}_i\}_{i=1}^N)$. Typical cost functions include mean squared error:

$$C(\{\hat{\boldsymbol{y}}_i\}_{i=1}^N, \{\boldsymbol{y}_i\}_{i=1}^N) = \frac{1}{N}\sum_{i=1}^{N}(\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2, \tag{2.4}$$

and mean absolute error:

$$C(\{\hat{\boldsymbol{y}}_i\}_{i=1}^N, \{\boldsymbol{y}_i\}_{i=1}^N) = \frac{1}{N}\sum_{i=1}^{N}|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i|. \tag{2.5}$$

The problem of setting the weight parameters now becomes equivalent to minimizing the value of the cost function and various sorts of optimization algorithms can be used. Today, the most popular algorithms are based on using information from the derivatives of the cost function with respect to the weights. Methods that only use the first derivative are referred to as *gradient descent*-based methods and are described below. There has been some work using methods which include second derivatives but these have not proven as useful for deep networks [32] and will therefore not be presented here.

Gradient descent

Gradient descent can be described as finding the lowest point on a surface by first taking a step in the direction with the steepest slope downward, then reevaluating where the slope is now the steepest, taking another step in this direction and so on. In mathematical terms, the surface corresponds to the value of the cost function and the steps corresponds to small changes in the dependent variables, i.e. weights for a neural network. The slope is determined by the partial derivatives of the cost function and it can be proven that the steepest slope corresponds to the gradient vector of the cost function. Therefore, gradient descent calls for taking successive steps in the direction of the negative gradient of the cost function. Gradient descent for a neural network is explicitly stated in Algorithm 1 where the vector $\boldsymbol{w}$ refers to *all* the weights of the neural network. The speed with which the network learns is dependent on the variable ε which is the length of a step and referred to as the *learning rate*. The value of ε is often important for the convergence of gradient descent. A low learning rate will lead to a

while *Convergence criteria not met* **do**
  Calculate $\boldsymbol{\nabla}_{\boldsymbol{w}} C(\boldsymbol{w})$;
  New weights $\boldsymbol{w} = \boldsymbol{w} - \varepsilon \boldsymbol{\nabla}_{\boldsymbol{w}} C(\boldsymbol{w})$;
end

Algorithm 1: Gradient descent for a neural network.

slow convergence while a high one might cause the model to oscillate and never converge. In practice, one often tests several different rates to find the one leading to the best results. Such optimization of the learning algorithm itself is referred to as *hyperparameter optimization* where the learning rate is one of many hyperparameters [32].

The gradient descent algorithm works well for functions that only have one global minimum. If, however, a function has many local minima or other flat areas such as saddle points, the algorithm can get stuck in one of these areas. Some methods that help the algorithm escape these regions are presented below. Additionally, these methods can significantly decrease the time required to train the network.

Stochastic algorithms

Computing the gradient using the entire dataset can be computationally heavy when the number of samples becomes large. The gradient on the training dataset is really only an estimation of the gradient for unseen examples; another estimate of the gradient can be calculated from a subset of the training dataset. For a subset with n samples the estimate's standard error goes as $\sigma/\sqrt{n}$ where σ is the standard deviation of the distribution of data samples [33]. The improvements in accuracy are thus less than linear in n and, since the computation time *is* linear, we will get diminishing returns for using more samples to estimate the gradient.

It has been shown empirically that most optimization algorithms converge much faster if they are allowed to calculate approximate gradients on subsets of the training dataset [32]. Such algorithms are called *minibatch* or *stochastic* algorithms and one of the simplest and most commonly used is *stochastic gradient descent* (SGD) which works just as normal gradient descent except that at each step a sample of size n is drawn from the training dataset to provide an estimate of the gradient. This can also alleviate the problem of converging to a local minimum since the gradients calculated on different minibatches can point away from the minimum even if the gradient on the entire dataset would be zero.

The choice of the minibatch size n is not straightforward since a smaller n introduces more noise to the calculated gradient which can cause the error of the model to fluctuate heavily while a larger n will increase stability but also significantly slow down the computation time for each estimate.

Therefore, n is, similarly to the learning rate ε, a hyperparameter to be selected. In practice, one often tests several different values in order to find the right compromise between stability and computational effort. The minibatch estimation introduces a level of noise to the gradient. Hence, the choice of learning rate ε is even more important than for the deterministic gradient descent. In fact, for the algorithm to converge at all one must successively decrease the learning rate [32]. Otherwise, the minibatch noise would forever cause the model to oscillate around a minimum.

Momentum

To speed up learning and also provide a tool for escaping flat regions some algorithms incorporate *momentum*. Momentum algorithms accumulate an exponentially weighted moving average (see Section 2.3) of the previous gradients and continue to move in their direction. More concretely, momentum introduces a velocity $\boldsymbol{v}$ and updates this as an exponentially weighted moving average of the negative gradient. The weights are then updated by adding $\boldsymbol{v}$. The update scheme is shown in equation (2.6) where a hyperparameter $\lambda \in [0, 1)$, determines how quickly the contributions of earlier gradients decay.

$$\boldsymbol{v}_{k+1} = \lambda \boldsymbol{v}_k - \varepsilon \boldsymbol{\nabla}_{\boldsymbol{w}} C(\boldsymbol{w}_{k+1}) \tag{2.6}$$

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \boldsymbol{v}_k \tag{2.7}$$

Momentum derives its name from a physical analogy in which the weight vector $\boldsymbol{w}$ is a particle moving through space with velocity $\boldsymbol{v}$. It experiences a driving force from the negative gradient and a dampening force akin to viscous drag which causes the exponential decay.

Algorithms that incorporate momentum have empirically shown quicker and better convergence than those without [32]. The decay parameter λ is, however, a new hyperparameter that needs to be considered. Hence, the finetuning of algorithms with momentum is more complex than that of those without.

Adaptive algorithms

We have previously discussed how the learning rate ε needs to be fine-tuned for gradient descent algorithms to be effective. Not only does one need to correctly set the initial value but, in the case of SGD, one also has to choose a proper scheme for decreasing it. To overcome these problems many researchers have looked into schemes to automatically adjust this parameter. Typically the learning rate is decreased proportionally to the squared values of the previous gradients ensuring a large learning rate for flat areas and a

smaller one for steep ones. Empirically, algorithms that include some adaptive learning scheme have been shown quite robust to the hyperparameter selection and some of the most commonly used ones are AdaDelta, RMSProp and ADAM. That said, plain SGD and SGD with momentum are also commonly used and produce similar results once correctly tuned [32].

2.1.7 Regularization for neural networks

For a neural network, good performance on seen data is not the ultimate goal. Rather, we want it to perform well on new, previously unseen inputs. This is called the network's *generalization capability* and in practice, it can only be optimized indirectly with the training data. In a perfect world, this would not matter since any information that could be extracted from seen data would also be featured in new data and therefore useful to learn. In reality, however, all data is contaminated in some way by measurement noise or missing values and if the network is too capable it will also fit on this incorrect data. This is referred to as *overfitting* and must be limited when designing a good machine learning algorithm.

To know whether a network is overfitted or not it is important to be able to test it on unseen data. Therefore, a given dataset is usually split into two parts: the *training* dataset and the *testing* dataset where the former is used to train the network and the latter to infer how well it performs on unseen data.

Apart from optimizing the weights of the network, the training phase is also used to determine parameters of the model or optimization algorithm itself, so-called *hyperparameters*. These include the learning rate and momentum parameters discussed earlier but also parameters such as the overall structure of the network or the choice of optimization algorithm. If these were learnt on the training dataset they would always be set so as to give the network as high capacity as possible and thus making it keen to overfit. If they were induced from the testing dataset, however, the testing error would no longer be an unbiased estimate of the generalization error of the network since the model had been selected to favour this data. Therefore, it is common practice to split the training dataset into two parts, one for optimizing the weights and one for determining all the various hyperparameters. Usually, the former part is larger and still referred to as the *training* dataset while the latter is smaller and referred to as the *validation* dataset.

Since overfitting occurs when the model is too capable one of the simplest ways to avoid it is by limiting the network in some way. If, however, a too simple model is used, the target function would be missed altogether. Fitting a sinus wave with e.g. a linear model would make both the generalization and training errors huge. Consequently, instead of simply restricting the number of nodes in a network it is often better [32] to *regularize* the training process by inducing some prior knowledge on how the network should

behave. Regularization methods include restricting the size of the weights and adding additional noise to the data. Some of these are discussed below.

Early stopping

One of the simplest ways to regularize a network is to keep track of the error on the validation set and stop training the network once the error seizes to decrease. In practice, one stores the configuration with the lowest validation error and if no improvements are made for a fixed number of iterations p (called *patience*) training halts and the stored configuration is used.

Due to its effectiveness and simplicity, early stopping is one of the most commonly used regularization techniques for deep as well as shallow neural networks.

Penalizing weights

Another regularization technique is to force weights toward a fixed value $\boldsymbol{w}_0$, chosen to be a value which we expect the weights to take. Typically we have no prior information on what the weights should be and we simply set $\boldsymbol{w}_0 = 0$ and the method penalizes big weights. Two common ways of penalizing the weights are L^2 or ridge regression and L^1 or lasso regression. Both methods add an extra term to the cost function:

$$\tilde{C}(\boldsymbol{w}) = C(\boldsymbol{w}) + \alpha\Omega(\boldsymbol{w}).$$

For ridge regression, the extra term equals half the L^2-norm of the weights:

$$\Omega_{\text{Ridge}}(\boldsymbol{w}) = \frac{1}{2}||\boldsymbol{w}||_2^2.$$

It can be shown that, from a probabilistic point of view, optimizing with ridge regression is the same as having a normal distribution as a prior distribution on the weights and then solving in a maximum likelihood fashion [32].

There is a close connection between ridge regression and early stopping. Ridge regression tries to limit the L^2-norm of the weights which can be interpreted as limiting their distance from the origin. With early stopping one essentially limits the number of steps the weights are allowed to travel from an initial value, usually the origin. Thus there is a maximum distance that the weights are allowed to travel. For a simple kind of loss function, a quadratic function, the methods have in fact been proven perfectly equivalent [32], and, in practice, one method can often be used as a substitute for the other.

Similarly to Ridge regression, Lasso regression also penalizes large weights but this time it's the L^1-norm that's being penalized:

$$\Omega_{\text{Lasso}}(\boldsymbol{w}) = |\boldsymbol{w}|.$$

Instead of forcing small weights overall, this causes some weights to be small and others to vanish completely [32]. Thus the network becomes sparse with active connections between only some of the nodes.

Dropout

Dropout refers to randomly dropping connections between network nodes during training. At each iteration, a random number of weights are set to zero and different features are thus hidden from the network. From a different perspective, at each step, a different network is used for inference and the weights are adjusted accordingly. For this reason, dropout can be seen as a cheap way of implementing *bagging*, i.e. training several different models and combining the results. Especially for deep networks, dropout has been successful, often outperforming other regularization techniques such as penalizing weights [32].

2.1.8 Enabling training of deep neural networks

In the early '00s, DNNs were considered too difficult to optimize and most machine learning research focused on other methods such as support vector machines and random forests. The main issue is that when backpropagation is applied to DNNs the resulting gradients can vary greatly in size between different layers. This causes some layers to learn faster than others and the network can get "stuck" in a region where its activation functions are saturated but its performance is suboptimal. Most of the time, the gradients decrease from one layer to the next as a consequence of successive multiplications of matrices with maximum eigenvalues smaller than 1. The problem is known as the "vanishing gradient problem" and it was not until 2006 when Hinton et. al [34], [35] applied so-called "layer-wise pretraining" to overcome it that DNNs reached state-of-the-art performance. Since then, many methods and tools have been developed to achieve efficient training of DNNs and some of these are described below.

Pretraining, weight initialization and activation functions

In [34] Hinton et. al applied Restricted Boltzmann machines (RBM) to "pretrain" the weights of a DNN model and subsequently achieve state-of-the-art classification errors on the MNIST dataset and reignited the research into neural networks. The theory behind RBMs is beyond the scope of this thesis but, in short, they can be described as a two-layer model that encodes a representation of its input variables (layer one) in a set of hidden variables (layer two) [32]. RBMs are equivalent to a kind of ANN with sigmoid activation functions and the idea of pretraining is to stack layers of RBMs. The first layer is trained to encode the inputs from the dataset to the second layer and the outputs of this are then used to encode another

representation in the third layer and so on. Finally, the last layer of the ANN is tuned in a traditional way to solve the task at hand. This solves the problem of saturated activation functions by moving the network parameters to a "good" initial region and then learning from there.

Following the success of pretraining, Glorot et. al [36] investigated why traditional training had failed by looking at the activation of hidden units as well as the sizes of backpropagated gradients for different layers. They found that, at the beginning of training the gradients quickly decayed from one layer to the next, and, as a likely consequence of this, the layers stopped adapting one after the other. Using a linear toy model they proposed a new initialization strategy to allow gradients to persist in size as they propagated through the network. According to this *Glorot initialization* the weights for each layer should be initialized from the distribution

$$W_{ij} \sim U(-\sqrt{\frac{6}{\#\text{ inputs} + \#\text{ outputs}}}, \sqrt{\frac{6}{\#\text{ inputs} + \#\text{ outputs}}}). \tag{2.8}$$

With this initialization, the authors obtained equally sized activation values and gradients across all layers and were able to successfully train DNNs with up to five hidden layers. Moreover, they found that the choice of activation function made a big difference with the traditionally popular sigmoid function being suboptimal. Subsequent works [37], [38], [39] showed good results for the ReLU activation function which has the benefit of not saturating in the same sense as sigmoidal activation functions.

In [40] Sussilo showed that, since the matrices in a feedforward NN are all different, the propagation of gradients from one layer to the next can be interpreted as a random walk. If the activations were adjusted with a *gain factor* g, which depends on the network architecture, the vanishing gradient problem could mostly be avoided. Using this *Random walk initialization* Sussilo was able to effectively train networks as deep as a thousand layers.

After this, many different initialization schemes have proven successful in training DNNs, especially in combination with ReLU activation functions. However, none of the schemes consistently leads to the best results which, according to [32], can be for three reasons:

- It may not be optimal to preserve the norm of a signal throughout the network.
- The properties imposed by initialization may not persist after learning has begun to proceed.
- The strategy might succeed at improving convergence speed but will also increase the generalization error.

For this reason, the authors recommend treating the initialization as a hyperparameter to be searched for.

Batch Normalization

The main issue with depth is that it introduces highly non-linear dependencies on the parameters: an ever so slight change of the first layer's output will cascade through the network and either grow or decay exponentially depending on the weights in the following layers. A gradient-based learning algorithm either has to use a very low learning rate or utilize higher order information to address this issue. There are some learning algorithms that use second-order derivatives but these are computationally expensive and in deep networks even third or fourth order interactions can be important.

Batch normalization is an elegant solution to this problem. In essence, it re-parametrizes the outputs of any layer making it less sensitive to changes in the previous layers. With $\boldsymbol{H}$ being a minibatch of activations with rows as samples and columns as features, the first step is to transform its elements H_{ij} to have zero mean and unit standard deviation:

$$H'_{ij} = \frac{H_{ij} - \mu_j}{\sigma_j},$$

where μ_j and σ_j are the mean and standard deviation of the j'th feature calculated on the minibatch. This step ensures that any gradient-based optimization algorithm will not propose a change that only alters the mean or standard deviation of a layer's activations since any such changes would be canceled out by the normalization.

During the training, running averages of the means and standard deviations are calculated and these are then used for inference at test time. This way, all test examples are normalized the same way and even a single example can be evaluated.

The normalization can limit the capacity of the network and to counteract this it is common that each normalized hidden activation H'_{ij} is replaced with the affine transformation $\gamma_j H'_{ij} + \beta_j$. γ_j and β_j are parameters trained by the network that once again allow the activations to have any mean or standard deviation. All in all, batch normalization allows the new parametrization to represent the same family of functions as the old ones but changes the learning dynamics. While the mean and standard deviation of H_{ij} can depend on complicated interactions of the previous layers, the mean and standard deviation of $\gamma_j H'_{ij} + \beta_j$ will only depend on β_j and γ_j respectively. This significantly decreases the problems with deep learning and gradient descent.

In addition to simplifying the training, batch normalization has a regularizing effect that helps with generalization. Each time the network sees an example x_i it is normalized differently depending on what other examples are in the same batch. This has the effect of adding random noise to the samples after each layer and forces the network to be more robust to inherent noise in the training dataset.

2.1.9 Autoencoders

An autoencoder is a neural network which tries to transfer the input $\boldsymbol{x}$ onto itself. To force autoencoders to not only copy the input perfectly they are restricted in either their architecture or in their training procedure. These restrictions force the autoencoder to learn only the most important features of the training data since it cannot model everything correctly. The learnt features often provide a better representation of the data and autoencoders have, among other things, been used for anomaly detection, dimensionality reduction, and feature extraction.

Internally, an autoencoder can be viewed as consisting of two parts, an encoder function that produces a *code* $\boldsymbol{c}$ from the input, $\boldsymbol{c} = e(\boldsymbol{x})$, and a decoder function that tries to reconstruct the input from this code, $\boldsymbol{r} = d(\boldsymbol{c})$. Typically, the network is built in a feed-forward fashion and the code $\boldsymbol{c}$ simply corresponds to a hidden layer. The architecture of such a network is shown in Figure 2.4. This network only has one hidden layer, the code layer $\boldsymbol{c}$. However, just as depth can be beneficial for traditional feedforward networks it can be successfully used for autoencoders as well. Such an autoencoder is shown in Figure 2.5.

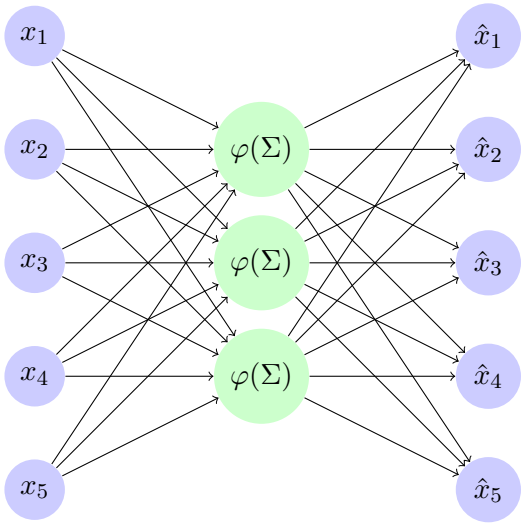

Figure 2.4: Architecture of a shallow autoencoder.

In this thesis, autoencoders will be used to model the distribution of wind turbine data and to determine whether new data points come from this distribution or not. The idea is simple: during training, the autoencoder learns to reproduce data from a distribution similar to the training distribution. If new data is reproduced well it is likely that it's coming from the same distribution as the training data. If, on the other hand, it is not reproduced well and the reconstruction error is large then it is likely that

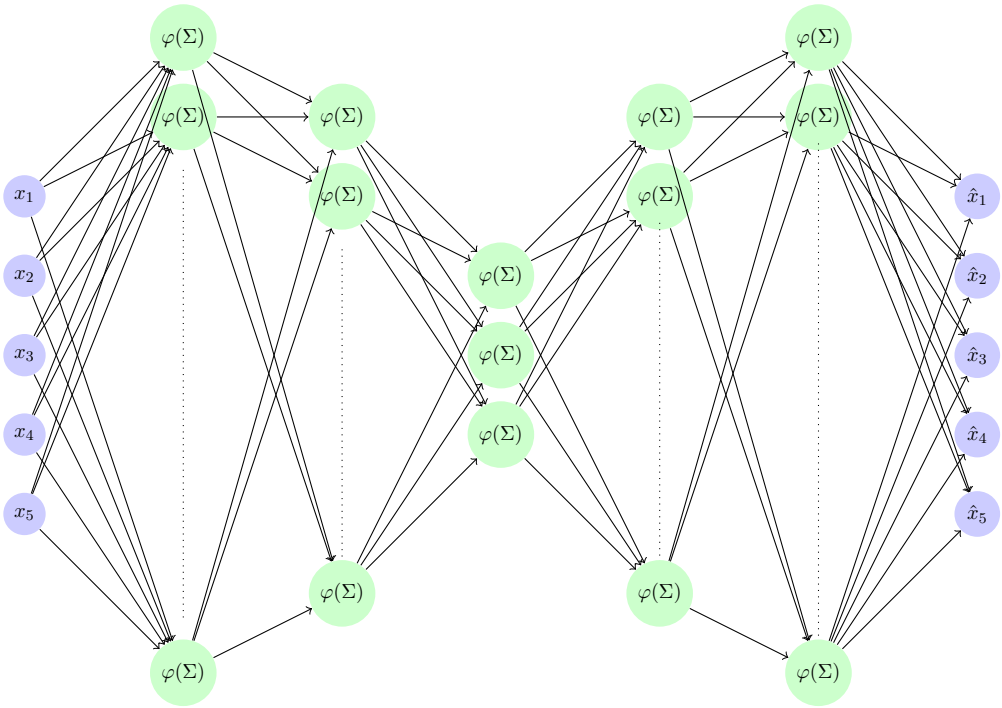

Figure 2.5: Architecture of a deep autoencoder.

this data comes from a different distribution than the training data.

Undercomplete autoencoders

One way of restricting the network is to set the code to have fewer neurons than there are inputs. The network is then forced to represent the data in a lower dimension and will only learn its most prominent features. These autoencoders are referred to as *undercomplete* autoencoders and the networks in Figures 2.4 and 2.5 are of this type.

Denoising Autoencoders

A different way of ensuring that an autoencoder learns useful features of the data distribution is to add noise to the input $\boldsymbol{x}$ during training. The autoencoder then learns to denoise the input which forces it to learn the underlying structure of the data distribution. If noise with an expected value of zero is applied, the autoencoder learns to contract a point $\tilde{\boldsymbol{x}}$ to the nearest point in the underlying distribution. This is illustrated in Figure 2.6

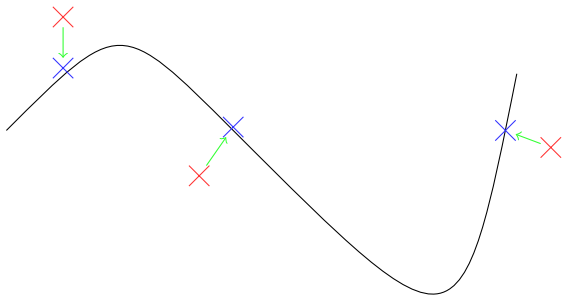

Figure 2.6: A picture illustrating the mapping learnt by a denoising autoencoder. The solid line represents the underlying distribution of the data and the blue x's correspond to training data drawn from this distribution. During training of the autoencoder these are corrupted to the red x's and the autoencoder learns to contract everything toward the solid line as is shown by the green arrows.

2.2 Mahalanobis distance and whitening

The Mahalanobis distance (MD) is a unitless distance between a point $\boldsymbol{x}$ and a distribution D or alternatively between two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ from the same distribution D. It can be seen as a multi-dimensional generalization of the idea to measure how many standard deviations away from the mean of a distribution a point is. First introduced by P.C. Mahalanobis in 1936 [41] it is defined for a point $\boldsymbol{x}$ as

$$\mathrm{MD}(\boldsymbol{x}) = \sqrt{(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})}, \tag{2.9}$$

where $\boldsymbol{\mu}$ is the mean vector of the distribution and $\boldsymbol{\Sigma}$ its covariance matrix. Similarly, it can be defined as a distance between two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$:

$$\mathrm{MD}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \sqrt{(\boldsymbol{x}_1-\boldsymbol{x}_2)^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_1-\boldsymbol{x}_2)}. \tag{2.10}$$

Since it is calculated through the covariance matrix, the Mahalanobis distance is invariant under a scaling or any other form of linear transformation. It has been used for anomaly detection in many different fields and Bangalore et. al applied it successfully in the context of detecting gearbox faults in a WT [23].

Closely related to the MD is the concept of whitening. While the MD uses the covariance matrix to calculate a unitless distance between points in a distribution, whitening refers to transforming all points so that they have the identity matrix as covariance matrix. The concept is best illustrated graphically and in Figure 2.7 three stages of a transformation are shown. In

the first plot samples from a multivariate Gaussian distribution with covariance matrix $\mathbf{\Sigma} = \begin{bmatrix} 1 & 1/2 \\ 1/2 & 2 \end{bmatrix}$ and zero means. Because of the codependency between the features, the distribution takes an elliptic form. In the second plot, traditional normalization by dividing each feature by their respective standard deviation (1 and 2 in this case) is applied. One can see that this brings the x and y-axes to the same scale but does nothing to remove their codependency. In the final picture, ZCA-whitening is applied on the original data. By removing the codependencies the data becomes symmetrically distributed with each feature being more representative of underlying the features.

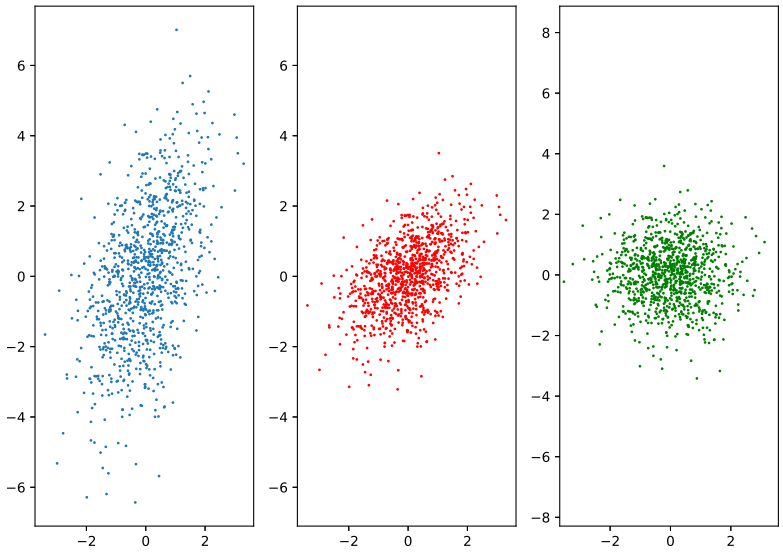

Figure 2.7: From left to right: (Blue) Samples from a multivariate Gaussian distribution. (Red) The same samples scaled by their standard deviation. (Green) The samples after a ZCA-whitening transform.

Any procedure that linearly transforms a distribution to have the identity matrix as a covariance matrix can be called a whitening transform. In fact, since the covariance matrix does not change under rotation of the data, there exists an infinite number of different whitening procedures. The first and perhaps most common whitening is called principal component analysis-whitening or PCA-whitening[2] and is done by projecting the sampled data points onto the eigenvectors of the covariance matrix and then normalizing them by dividing with the square root of the eigenvalues. Formally, denoting

by $\boldsymbol{U}$ the matrix of eigenvectors:

$$\boldsymbol{U} = \begin{bmatrix} | & | & | \\ \boldsymbol{u}^{(1)} & \cdots & \boldsymbol{u}^{(M)} \\ | & | & | \end{bmatrix},$$

and by $\boldsymbol{D}^{-1/2}$ the diagonal matrix containing the square root of the eigenvalues λ_i:

$$\boldsymbol{D}^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 & \cdots & 0 \\ 0 & \ddots & \cdots & 0 \\ \vdots & \cdots & \ddots & 0 \\ \vdots & \cdots & \cdots & \frac{1}{\sqrt{\lambda_M}} \end{bmatrix},$$

the PCA-whitening is defined by:

$$\boldsymbol{x}_{\text{PCA}} = \boldsymbol{D}^{-1/2}\boldsymbol{U}^T\boldsymbol{x} = \begin{bmatrix} \boldsymbol{u}^{(1)T}\boldsymbol{x}/\sqrt{\lambda_1} \\ \vdots \\ \boldsymbol{u}^{(L)T}\boldsymbol{x}/\sqrt{\lambda_L} \end{bmatrix}, \tag{2.11}$$

where one in the final expression can see how it is the projection onto the space spanned by the eigenvectors $\boldsymbol{u}_i$ of $\boldsymbol{U}$.

A different whitening is referred to as zero-phase component analysis-whitening or ZCA-whitening uses $\boldsymbol{U}$ to rotate the PCA-whitened variables back to their original coordinate system:

$$\boldsymbol{x}_{\text{ZCA}} = \boldsymbol{U}\boldsymbol{x}_{\text{PCA}} = \boldsymbol{U}\boldsymbol{D}^{-1/2}\boldsymbol{U}^T\boldsymbol{x}. \tag{2.12}$$

ZCA-whitening has the useful property that its whitening matrix is equal to the square root of the inverse covariance matrix $\boldsymbol{\Sigma}$:

$$\boldsymbol{W}_{\text{ZCA}} = \boldsymbol{U}\boldsymbol{D}^{-1/2}\boldsymbol{U}^T = \boldsymbol{\Sigma}^{-1/2}.$$

This can be proven by using the fact that the covariance matrix, being symmetric and positive semi-definite, can be diagonalized using its eigenvectors and eigenvalues:

$$\boldsymbol{\Sigma} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^T.$$

From its definition, it follows that the squared distance between two

[2]The transformation is as its name implies, closely related to the unsupervised learning algorithm PCA reduction. The difference is that in PCA reduction the data is projected onto only a few of the eigenvectors to produce a more compressed representation. Moreover, since the goals are different, scaling is not performed.

ZCA-whitened points is equal to the squared Mahalanobis distance:

$$\begin{aligned}
&(\boldsymbol{x}_{\text{ZCA}} - \hat{\boldsymbol{x}}_{\text{ZCA}})^2 \\
=&(\boldsymbol{\Sigma}^{-1/2}(\boldsymbol{x} - \hat{\boldsymbol{x}}))^T(\boldsymbol{\Sigma}^{-1/2}(\boldsymbol{x} - \hat{\boldsymbol{x}})) \\
=&(\boldsymbol{x} - \hat{\boldsymbol{x}})^T(\boldsymbol{\Sigma}^{-1/2})^T\boldsymbol{\Sigma}^{-1/2}(\boldsymbol{x} - \hat{\boldsymbol{x}}) \\
=&(\boldsymbol{x} - \hat{\boldsymbol{x}})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \hat{\boldsymbol{x}}) \\
=&(\text{MD}(\boldsymbol{x}, \hat{\boldsymbol{x}}))^2. \qquad (2.13)
\end{aligned}$$

Furthermore, it can be proven that ZCA-whitening is the whitening that minimizes the total squared distance between the original and whitened variables [42]. For these reasons, ZCA-whitening was the whitening of choice in this work.

2.3 Exponentially weighted moving average

Time series data, i.e. data sampled and indexed by time can be used to monitor processes. If the value goes above or below a preset threshold an operator is alerted and the process investigated. Real life data is, however, often noisy and if this noise is large enough it can make the value cross the thresholds even if the underlying process is fine. To counteract this issue, one often wishes to smooth the signal so that noise is averaged out and underlying trends are more easily detected. For this, a commonly used method is the *exponentially weighted moving average* (EWMA). For a signal x_t the EWMA at timestamp t, denoted z_t, is calculated as:

$$z_t = \lambda x_t + (1 - \lambda)z_{t-1}, \qquad (2.14)$$

where λ is a constant such that $0 \leq \lambda < 1$. The initial value, z_0, is usually set to the expected value of x_t, $z_0 = \mathbb{E}[x_t]$ [43]. The EWMA can be viewed as a weighted average of all past and present observations where the constant λ determines their relative importance. The term "exponentially weighted" comes from the fact that the weights decay exponentially when looking backwards in time from the present timestamp. Using equation (2.14) recursively we find

$$z_t = \lambda \sum_{i=0}^{t-1}(1 - \lambda)^i x_{t-i} + (1 - \lambda)^t z_0. \qquad (2.15)$$

Thus the sequence of weights, $(w_0, w_1, w_2, \dots) = (1, \frac{1}{1-\lambda}, \frac{1}{1-\lambda^2}, \dots)$ form a decreasing geometric sequence which is the discrete version of an exponential function.

To get a feeling for how quickly the EWMA picks up change, we define a response time τ as the time it takes for a constant function to reach $1-1/e \approx$

63% of its value given an initial EWMA-value of $z_0 = 0$. Equivalently, this is the time it takes for the contribution from previous timestamps, z_{t-1} to drop to a level less than $1/e \approx 37\%$. Given equation (2.15), with $x_t = 1 \forall t$, $z_0 = 0$, we have

$$\begin{aligned} 1 - \frac{1}{e} = z_\tau &= \lambda \sum_{i=0}^{\tau-1} (1-\lambda)^i = (\text{Geometric sum}) \\ &= \lambda \frac{1-(1-\lambda)^\tau}{1-(1-\lambda)} = 1 - (1-\lambda)^\tau \\ \implies \tau &= \frac{1}{\log \frac{1}{1-\lambda}}, \end{aligned}$$

which for $\lambda \ll 1$ implies $\tau \approx \frac{1}{\lambda}$.

If the observations x_t are independent random variables with variance σ^2 the variance of z_t is [43]

$$\sigma^2 \Big(\frac{\lambda}{2-\lambda} \Big) \Big(1 - (1-\lambda)^{2t} \Big). \tag{2.16}$$

Based on these *upper* and *lower control limits*, thresholds that the EWMA should not cross are usually defined as

$$\text{UCL} = \mu + L\sigma \sqrt{\frac{\lambda}{2-\lambda}(1-(1-\lambda)^{2t})} \tag{2.17}$$

$$\text{LCL} = \mu - L\sigma \sqrt{\frac{\lambda}{2-\lambda}(1-(1-\lambda)^{2t})}, \tag{2.18}$$

where μ is the mean of the observations, $\mu = \mathbb{E}(x_t)$ and L determines the width of the control limits. Typically L is set to around 3 [43]. The term $1-(1-\lambda)^{2t}$ approaches unity as t grows, and, after the EWMA control chart has been running for several periods, the control limits approach the following steady-state values:

$$\text{UCL}_{\text{SS}} = \mu + L\sigma \sqrt{\frac{\lambda}{2-\lambda}} \tag{2.19}$$

$$\text{LCL}_{\text{SS}} = \mu - L\sigma \sqrt{\frac{\lambda}{2-\lambda}}. \tag{2.20}$$

If the process is not expected to be off-target early on, these fixed thresholds can be used instead.

2.4 Precision and Recall

The performance of an anomaly detection model can be determined by its *confusion matrix*. The confusion matrix is a table which splits the output of

the model into columns according to its actual condition and into rows according to its predicted condition. The structure is shown in Table 2.1 where positive/negative refers to if a sample is an anomaly or not. One can see how it provides a summary of the *false positives* (falsely predicted anomalies), *false negatives* (missed anomalies), *true positives* (detected anomalies) and *true negatives* (correctly detected normal behaviour).

	Actual positive	**Actual negative**
Predicted positive	True Positive (TP)	False Positive (FP)
Predicted negative	False Negative (FN)	True Negative (TN)

Table 2.1: A confusion matrix: a helpful tool for determining the performance of an anomaly detection model.

The data in the confusion matrix can be used in many different ways to provide different performance measures of the model. Two commonly used measures are *recall* and *precision*. Recall, also called *sensitivity* or *true positive rate*, measures how many of the actual anomalies that the model is able to identify. It answers the question "How good is the model at finding anomalies?" and is defined as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{2.21}$$

Precision on the other hand, answers the question "Given a predicted anomaliy, what is the probability that it is an actual anomaly?", or, in other words, "How much can we trust an alert"? It is sometimes referred to as *positive predictive value* and is defined by

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{2.22}$$

For a system that uses a discriminative threshold to make its decision, the specific value of the threshold is very important. If the threshold is set low the system will classify most instances as anomalies. Almost all actual anomalies will be detected and the recall will be high. However, many of the normal cases will also be labeled as anomalies which lowers the precision. With a high threshold, the opposite results are obtained. Consequently, the optimal threshold value depends on what is considered the most important of precision and recall.

The tradeoff between precision and recall can be captured by a so-called precision-recall curve which is a plot of precision (y-axis) versus recall (x-axis) for varying threshold values.

Chapter 3

Designing an autoencoder model

This chapter describes the process of designing an autoencoder and selecting relevant hyperparameters. Furthermore, both the preprocessing of SCADA data and a scheme to train multiple networks in parallel are presented.

3.1 Preprocessing of SCADA data

The SCADA data was extracted from a database containing data for a large number of WTs from many different manufacturers. All the data was recorded at a 10-minute interval, but, for some signals, there were missing recordings for some timestamps. Different turbine types had different sets of signals available and to simplify the comparison between different models a single, 2 MW type turbine was used.

The signals were classified according to "type", i.e. what kind of signal it was. This "type" could be e.g. Power or Temperature but also labels such as Curtailment or Reactive power, that is, signals corresponding to operations or the energy market. To construct a good CMS the signals were first filtered on this "type" and only those that were relevant to the WT dynamics were selected. For the selected turbine there were 33 relevant signals available and these are presented in table 3.1.

Once the signals were selected they were loaded into Python for further processing with the libraries Numpy and Pandas. Here the data was represented in a matrix format were rows corresponded to different timestamps and columns to different signals. First, all rows containing missing values were dropped. Since there was relatively few missing values, all signals from a given timestamp (i.e. the whole row) were dropped even if only one signal was missing. When a turbine is stopped, i.e. it is producing zero power, many of the SCADA signals are still recorded. The behaviour and co-dependencies of these signals are, however, a lot different compared

to when the turbine is running. It might be difficult for an autoencoder to learn the dynamics in both regimes. Moreover, a fault will most likely be apparent during running conditions. Therefore, learning the stand-still operations was considered unnecessary. Consequently, all rows with zero or negative power [1] were dropped. Furthermore, the first three timestamps after a stop were dropped to avoid capturing transient dynamics during the initial startup of the WT.

It is desirable to have all the signals with mean close to zero and a variance close to one for application with standard neural network regularization and training tools. In this thesis, this and an additional step of decoupling signals from each other was performed through ZCA-whitening which is described further in section 2.2. The transformed data matrix had uncorrelated columns and the idea is to help the neural network separate different signals from each other by transforming highly correlated inputs to a set of less redundant features. The whitening required the means and covariance matrix of the data distribution and these were estimated from the training data and then stored so that the data could be reconstructed. In addition to providing the network with uncorrelated data, the whitening also facilitated an easier training with the Mahalanobis distance as an objective function. This is described further in the next section.

[1] Sometimes WTs are supplied energy from the grid to keep running even when there is no wind. This is done to avoid unnecessary wear due to many starts and stops. In these cases, the power produced is recorded as negative.

Signal name	**Unit**
Power	kW
Wind speed	m/s
Wind direction	°
Nacelle Position	°
Generator bearing front temperature	°C
Generator bearing rear temperature	°C
Generator phase 1 temperature	°C
Generator phase 2 temperature	°C
Generator phase 3 temperature	°C
Generator slip ring temperature	°C
Hydraulic oil temperature	°C
Gear oil temperature	°C
Gear bearing temperature	°C
Nacelle temperature	°C
Ambient temperature	°C
Grid inverter temperature L1	°C
Top controller temperature	°C
Hub controller temperature	°C
Controller VCP temperature	°C
Spinner temperature	°C
Rotor inverter temperature L1	°C
Rotor inverter temperature L2	°C
Rotor inverter temperature L3	°C
Grid busbar temperature	°C
Voltage L1	V
Voltage L2	V
Voltage L3	V
Current L1	A
Current L2	A
Current L3	A
Generator RPM	RPM
Rotor speed	RPM
Blade angle (pitch position)	°

Table 3.1: Relevant SCADA-signals for the type of wind turbine used in this thesis.

3.2 Anomaly detection using autoencoders

In this thesis, autoencoders were used as anomaly detection models. In short, an autoencoder takes a number of signals as input, encodes these to a condensed representation in a code layer and then decodes this representation back to reconstructed signals. Restrictions are put on the code layer so that the model cannot simply learn an identity mapping. For more details, see Section 2.1.9.

The choice of autoencoders was inspired by the works in [25] and [26] and further motivated by the fact that autoencoders more directly model the entire WT and not only a subsystem as is the case for a single-output neural network.

To build and train the networks the Python library PyTorch was used. Among other things, this library supports automatic differentiation, GPU-powered tensor calculations and many other useful tools.

The output of an autoencoder model is a large number of reconstructed signals. To simplify the interpretation it is desirable to summarize all of these residuals to a single value. The naive approach is to simply take the mean or root mean square (RMS) of all the signals. However, this has the downside that if the signals are highly correlated they will contribute disproportionally. To illustrate this imagine that we construct a very simple autoencoder modelling three signals: Power, gearbox oil temperature and gearbox bearing temperature. If the turbine experiences a gearbox fault it is likely that both the oil and bearing temperatures are higher than usual and the autoencoder will, if properly trained, reconstruct them as lower than they actually are. Both of these residuals will contribute to the mean/RMS which will increase. If, on the other hand, the WT experiences a different fault, say a fracture in the blades, the power might go down. The autoencoder will then model the power as larger than it actually is and this residual will contribute to the mean/RMS. However, only one of the three signals will have a significant reconstruction error and the mean/RMS will only be increased by half of which it would have been in the previous case. This model is thus biased toward detecting gearbox faults and will be less sensitive to other faults. To overcome this problem, the Mahalanobis distance (MD) was used in the present work. The Mahalanobis distance is a unitless measure which can be seen as a multi-dimensional generalization of measuring how many standard deviations apart two points of the same distribution are. It is described more in detail in Section 2.2. Normally, the Mahalanobis distance is calculated as a quadratic form with the inverse covariance matrix according to (2.10). Because of the ZCA-whitening performed in the preprocessing step, however, the mean squared Mahalanobis distance is equivalent to the mean squared error and this was indeed used for the models constructed. This simplified both training and inference since existing library functions could be used.

One network was trained for each turbine and, initially, one year of SCADA data was used. The time period was chosen so that all seasons were represented. This ensured that many different environmental conditions were present without setting too high requirements on the availability of good SCADA data. Initially, 20% of the timestamps from this year were extracted randomly to provide a validation set while the other 80% were used for training. In the later postprocessing steps, however, it was found that while the network produced low errors for both the training and validation sets it produced higher errors on new data, even if the new data came from healthy conditions. This might have occurred because the data points within the same year are more similar than data points from different years. To get a more representative validation set, the network was instead trained on all the data from one year and then the following six months were used for validation. This provided training datasets of roughly 30 000 timestamps and validation sets of 15 000 timestamps.

3.3 Parallel training with stacked networks

In this thesis, to make a thorough hyperparameter search many networks had to be trained. Training networks on GPUs has become popular in the last few years as they excel at performing many calculations in parallel which is exactly what is required for neural network training. However, when a GPU was tested for training networks in this thesis, it was quickly discovered that it performed equally or worse than a standard laptop CPU. Most likely, the missing speedups were due to the relatively low number of parameters of the networks. The employed autoencoders had no more than 200 000 parameters and were trained on roughly 30 000 data points. In comparison, the record-breaking AlexNet that won the ImageNet LSVRC-2012 competition had 60 million parameters and was trained on 1.2 million images [44].

After measuring the time of different parts in the training it was found that, when training on the GPU, the main bottleneck was not the actual calculations but rather data transfer and other overhead processes. To overcome these problems and fully take advantage of GPU computing, algorithms which enabled training of multiple networks simultaneously on the same GPU were developed. This worked by stacking all internal parameters on top of each other, adding an additional dimension to all tensors: 2-D weight matrices became 3-D tensors, 1-D bias vectors became 2-D matrices and so on. By doing this, existing PyTorch library functions could be used which handled all the GPU-specific instructions.

With the architecture in place a lot of the hyperparameter search could be parallelized and training times were reduced from around 3.7 s per model and epoch to around 0.08 s per model and epoch, that is, GPU training was

approximately 46 times faster per model.

3.4 Hyperparameter selection for the autoencoder models

To be effective as an anomaly detection model the autoencoder does not only need to reconstruct unseen data from a healthy WT well; it also needs to produce a large error for data from a malfunctioning WT. Typically, hyperparameter selection is based on the reconstruction error on a validation dataset. For an autoencoder, this is, however, only fair for parameters that do not directly affect its capacity, i.e. its ability to reconstruct complex datasets. To understand why, imagine a very simple "autoencoder" without any hidden layers that only maps the incoming signals onto themselves. This model would have a perfect, zero reconstruction error not only on the training dataset but also on any dataset used for validation. It would, however, continue to reconstruct perfectly even for data from a malfunctioning WT. Therefore, it would be useless for anomaly detection.

Some hyperparameters only have a minor effect on the autoencoder's capacity; instead, they affect how well a given model fits a certain type of data. Therefore, they can well be determined in a traditional fashion from the validation error. These parameters include the activation function, how weights are initialized, the optimization algorithm and its specific parameters, the minibatch size, and, to some extent, the regularization used. Together they are referred to as *training hyperparameters* in this thesis and the particular choices are discussed in Section 3.4.1.

The autoencoder's architecture, i.e. the number of layers and the number of nodes in each layer, does, however, strongly affect the autoencoder's capacity and can therefore not be selected by just minimizing the validation error. For the selection of these in this thesis, a more qualitative approach was taken which is described further in Section 3.4.2. The proposed method was then verified in a large validation study.

3.4.1 Selection of training hyperparameters

By using all reasonable signals it is very likely that some are not very related to each other. We can make the network use this information by imposing that it is sparse, i.e. that only some neurons are active at a time. To do this ReLUs was used as activation functions. ReLU returns zero for negative inputs and this can force different signals to take different paths through the network. A more in-depth description of ReLU can be found in section 2.1.2.

As an optimization algorithm "ADAM" was chosen because of its theoretical benefits and empirically good results, see Section 2.1.6 for more

details.

After this, additional hyperparameters were selected in stages. First of all, the parameters relevant to the training, i.e. weight initialization, learning rate, and minibatch size were selected in order to minimize the error on a validation set. It was found that, among the common weight initializations, the particular choice did not matter much. In this thesis, the weights were normally distributed with $\mu = 0$ and $\sigma = 1/\sqrt{\#\text{ inputs}}$. The biases were all filled with positive values equal to σ so that all ReLU units were initially active.

For the learning rate, it was found that 0.01 caused the training error to fluctuate heavily through the training which prevented a good fit. However, both 0.001 and 0.0001 converged well, achieving similar low validation errors which shows that 0.01 was probably a too large learning rate. Since convergence was quicker 0.001 and it indeed is the standard for ADAM, it was chosen.

As for the minibatch size, a bigger batch ensures less variance in the gradient which can facilitate a quicker training. However, if it becomes too large with respect to the size of the training set there is a risk of overfitting since the gradients quickly become biased, see section 2.1.6 for further explanation. Indeed it was found that when the minibatch size was increased over 4000 samples the fit on the validation set worsened. To be on the safe side a minibatch size of 256 was chosen.

It is possible that models trained with the same parameters behave differently because of the randomness in weight initialization and stochastic optimization algorithms. During the early experimentation, there was a big variation in the performance of different models. Even with the same hyperparameters, some models showed lower training and validation errors than the others. It is likely that the divergence occurred early on in the training with many of the ReLU activation functions "dying" to select optimal routes for the initial data. The network was then unable to fit data received later in the training as a ReLU cannot come back to being active once dead.

Switching from ReLU to its close relatives leaky ReLU or ELU decreased the variance between the models but also increased the average error slightly. The same effect could be achieved with plain ReLU by applying batch normalization between the layers. Batch normalization re-parametrizes the neural network to facilitate an easier training with gradient descent, see section 2.1.8. Because of its sound theoretical background and proven track record of training deep neural networks [32], the combination of batch normalization and ReLU was used.

Next, additional regularization techniques were investigated. Dropout, which has proven successful for many tasks [32], was tried first. Surprisingly it significantly increased the models' reconstruction error by a factor of ten or more. Under closer investigation, it was found that almost all ReLUs were killed during the first iterations and that the network after this produced

the same output for all inputs. This is likely another manifestation of the problems of training deep neural networks described in section 2.1.8 but, due to time constraints, the issue had to be left unresolved and, instead, another regularization method, input noise, was used.

Adding noise to the inputs of an autoencoder is another way of ensuring that the autoencoder does not simply learn an identity mapping. These autoencoders are called *denoising* autoencoders and are described further in section 2.1.9. The details of the selection of input noise are left for the next section as it likely affects the model's capacity.

3.4.2 Selection of model architecture

In this thesis, the autoencoders were shaped like an "X". The first layers, referred to as the *encoder*, started wide but then got consequently narrower until reaching the *code* layer in the middle. This layer was the narrowest and serves as a bottleneck for the information flow through the network. After the code layer, the layers got wider and wider, mirroring the encoder. This part is referred to as the *decoder*. The architecture is similar to that in Figure 2.5 but instead of two layers for the encoder/decoder, three were used. The idea behind the X shape is to allow the network to learn features at higher and higher abstraction levels. In reality, the number of nodes in each layer's is a hyperparameter to be tuned. However, in this work, the X shape was fixed with a constant ratio of 1.5 between consecutive layers in order to limit the search space for hyperparameters. Furthermore, the decoder was a perfect mirror of the encoder with the same number of nodes in layers one and seven, two and six and three and five. The number of layers in the encoder/decoder was fixed to three and no further investigation of the effect of depth was undertaken.

After limiting the search space, the remaining hyperparameters were the size (number of nodes) of the first encoding layer, the size of the code layer and the level of the input noise. If either the code or encoder/decoder size is increased the model's capacity is also increased. The interplay between these parameters is, however, not clear: lowering the code size while increasing the encoder size could either increase or decrease the capacity depending on what amount they were changed with.

The effect of noise is not certain, but, since it acts to distort the input data, it is likely that raising the level of noise has a similar effect as lowering the model capacity. For models with high capacity, input noise might be necessary to make them learn anything but the identity mapping while it for a low capacity model might be redundant.

There is no known method to optimize these parameters using only the training and validation datasets. In the following sections, however, a method is described, that, by using so-called "encoder curves" and "noise curves" intends to accomplish this. The method is highly speculative and

therefore a larger validation study is presented in Chapter 4 where the method's results are compared with actual performance of the models.

Encoder curves

"Encoder curves" refer to curves formed by the reconstruction error on the training and validation sets when the size of the encoder was varied but the code size and noise level were fixed. In Figure 3.1 three such graphs are shown for code size of 20 but for different levels of input noise. Four turbines from the same wind farm were used and all showed similar results. To average out randomness in the training ten networks were trained for each configuration and turbine.

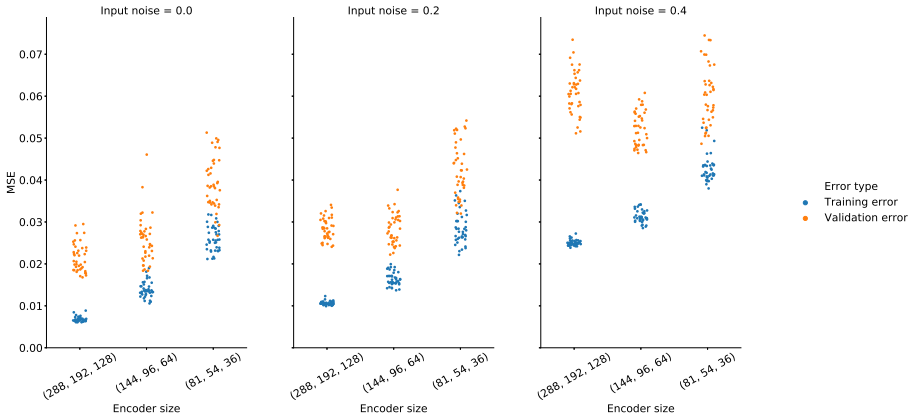

Figure 3.1: So called "encoder curves": reconstruction error on the training and validation sets for different sizes of the encoder/decoder.

"Encoder size" refers to the number of nodes in each layer of the encoder/decoder so that, with code size 20, a value of (144, 96, 64) implies a network with the shape (144, 96, 64, 20, 64, 96, 144). All curves show that the reconstruction error on the training set, denoted "training error", gets lower with higher encoder complexity. For the reconstruction error on the validation set, however, this is only true up to a point: after (144, 96, 64) it remains the same or worsens. This indicates that up to this point the model improves its representation of the true data distribution, but, beyond, it instead starts to fit the noise in the training distribution. The point seems to shift to a lower complexity as the noise increases. It appears that given a fixed code size and noise level there is only a part of the true data distribution that the autoencoder can model. Viewed in this light, it is reasonable to select the encoder size at the threshold as the model then captures as much as possible of the true data distribution but nothing more.

Noise curves

To select an appropriate code size and noise level so-called "noise curves" were used. Similar to encoder curves, a noise curve shows the training and validation errors but now as a function of the input noise with the code and encoder/decoder size being held fixed. Figure 3.2 show five such graphs, all with encoder size (144, 96, 63) but with different code sizes.

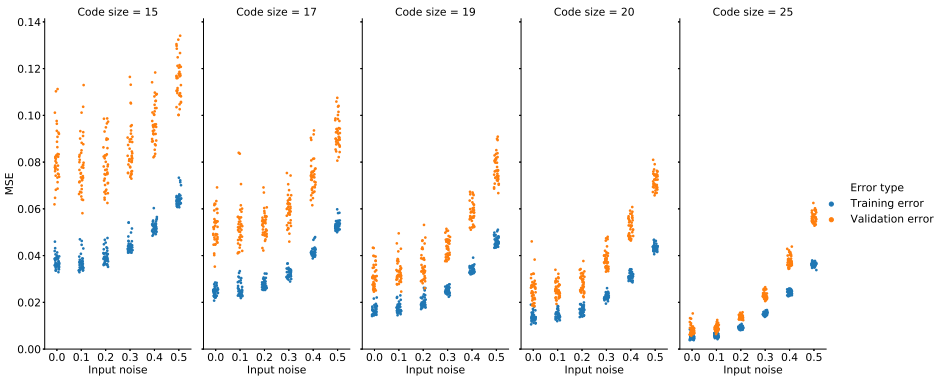

Figure 3.2: So called "Noise curves": reconstruction error on the training and validation sets for different levels of input noise.

At first glance, all curves look the same: for low noise the errors are roughly constant but after a certain threshold both the validation error and the training error increase rapidly. This threshold seems to decrease as the code size increases: for code size 15 the errors start to rise after input noise 0.2 but for code size 25 it happens already after input noise 0.1.

On closer inspection, one can also see that, after a certain point, the validation and training errors start to diverge. This is more clearly seen in Figure 3.3 where the actual differences between the curves are plotted. The threshold seems to decrease as the code size increases: while the gap remains constant until after input noise 0.4 for code size 15 it starts to grow already after input noise 0.1 for code size 25. Note that for the lower code size the two thresholds differ but for the higher ones they seem to converge. It is difficult to pinpoint exactly where the two thresholds are for different code sizes, but, by graphical inspection, an attempt was made. The results are summarized in Table 3.2 where the column "Both" corresponds to the first threshold and the column "Difference" corresponds to the second. As indicated by the table, the two points collapse between code sizes 17 and 19 and afterwards decrease together.

Exactly why this happens is not fully understood. An educated guess,

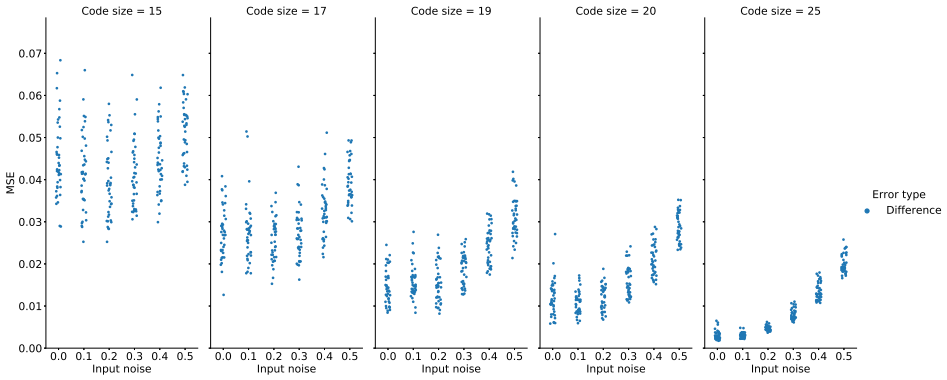

Figure 3.3: The difference between the training and validation errors shown in Figure 3.2.

however, is that the input noise hides some of the features of the training data. If a model has a too low capacity, learning would not be affected at all. For such a model, it is only possible to learn a subset of the features in the data distribution; if some features are masked by input noise, the model will just learn a different subset. It is not until enough features are hidden and the model is forced to learn a smaller subset that performance gets worse. This would explain the first threshold and why models with a lower code size experience it later.

After the second threshold, the validation and training errors start to diverge. Thus, the model's ability to generalize gets worse. This could be explained if some of the features hidden are those that make models able to extrapolate to new, unseen data. Still, why the first and second threshold differ for low-capacity models but not for high-capacity models is unclear. However, since we are looking for an optimal code size in this area and there is a distinct behaviour of the two thresholds collapsing, it seems reasonable to select the code size right at this point as a starting point for further investigation. From Table 3.2, this code size would be 18. Since the encoder/decoder size of (144, 96, 63) was ideal for the nearby code size of 20, it seemed reasonable to hold this fixed moving forward.

Codes size	Both	Difference
15	0.2-0.3	0.4-0.5
17	0.2-0.3	0.3-0.4
19	0.2-0.3	0.2-0.3
20	0.2-0.3	0.2-0.3
25	0.1-0.2	0.1-0.2

Table 3.2: Thresholds for when the input starts affecting the training and validation errors of the models. Column 'Both' represents a threshold after which both the training and validation errors start to increase. Column 'Difference' corresponds to a threshold after which the training and validation errors, in addition to increasing, also diverge from each other. Note how the thresholds in the two columns are equal for the high code sizes but different for the low ones.

Chapter 4

Validation study

This chapter describes a large validation study. Autoencoder models with different code sizes and different level of input noise were compared using a large, labeled dataset containing actual fault cases.

4.1 Dataset description

Data from 19 WTs and four wind farms was used to form a labeled dataset. All turbines were located in Europe and of the same type with a rated capacity of 2 MW. Their available signals can be found in Table 3.1 as the turbines were of the same type as the ones used in the previous selection stage. For each turbine, the time period from the first of July 2015 to the last of June 2016 was used as training data and the period from the first of July 2016 to the last of December 2016 was used as validation data.

The turbines had service logs available and these served as a basis for labeling test cases. A test case refers to a time period ranging from three to six months where one turbine was either considered healthy or showed some anomalous behaviour.

For each turbine the logs were read thoroughly and all anomalies that the models should be able to detect were recorded. Next, the logs were compared with the output of some models. During this comparison, additional anomalies were detected that had previously been overlooked in the logs. If an anomaly existed in the logs it was added to the list of anomalies. Sometimes it was not clear whether an event would induce a fault in the model or not. This was the case for e.g. storms and grid instabilities. In these cases, the data was excluded from the test cases as it could not be labeled. Furthermore, some anomalies occurred multiple times. If all of them had been used the results would have been biased toward the model most capable of detecting this kind. Therefore, for each kind of anomaly, only one test case was included.

In Tables 4.1, 4.2, 4.3 and 4.4 all different test cases and the excluded

data is shown. In total six different anomalies; a yaw encoder malfunction, a rotor sensor malfunction, high generator temperatures due to a malfunctioning ventilation duct, a preventive maintenance that changed the hydraulic oil temperature, high VCP temperatures, and a grid curtailment were considered.

Wind farm A – test cases		
Turbine	**Timespan**	**Description**
1	2017-01-01:2017-07-01	Normal behaviour
1	2017-07-01:2017-12-31	Normal behaviour
1	2018-02-01:2018-07-01	Normal behaviour
1	2018-07-01:2018-12-31	Normal behaviour
4	2017-01-01:2017-07-01	Normal behaviour
4	2017-07-01:2017-11-30	Normal behaviour
4	2018-01-01:2018-07-01	Malfunctioning yaw encoder, replaced in 2018-04
5	2017-01-01:2017-07-01	Normal behaviour
5	2017-07-01:2017-11-30	Normal behaviour
Wind farm A – excluded data		
Turbine	**Timespan**	**Reason**
All	2018-01-01-2018-02-01	Large storms highly affected the wind speed. Since these are abnormal conditions this time period was excluded for all turbines.
2, 3	2017-01-01:	All models detected a change in the operations caused by a preventive maintenance during the validation period. Since the reason of the faults is unknown, these turbines were ignored.
4	2017-11-30:2017-12-31	As it was unclear when the fault fixed 2018-04 started, an additional month before was excluded.
4	2018-10-08:2018-10-09	The turbine was paused but still produced a low power. Thus, the power filter was not triggered but the model showed an abnormal behaviour.
5	2017-12-15:	After a preventive maintenance, the Hub controller temperature is consistently lower than expected. Most likely a setting was changed or something was clean but since the reason cannot be determined from the logs the remaining time period is ignored.

Table 4.1: Description of test cases and excluded data for the wind farm labeled A.

Wind farm B – test cases		
Turbine	**Timespan**	**Description**
6	2017-01-01:2017-07-01	Normal behaviour
6	2017-07-01:2017-12-31	Normal behaviour
6	2018-02-01:2018-07-01	Normal behaviour
6	2018-07-01:2018-12-31	Normal behaviour
8	2017-01-01:2017-07-01	Normal behaviour
8	2017-07-01:2017-12-31	High generator temperatures, replacing an extraction duct fixes it.
8	2018-02-01:2018-07-01	Normal behaviour
8	2018-07-01:2018-12-31	Normal behaviour
9	2017-01-01:2017-07-01	Normal behaviour
9	2017-07-01:2017-12-31	Normal behaviour
9	2018-02-01:2018-07-01	Normal behaviour
9	2018-07-01:2018-12-31	Normal behaviour
Wind farm B – excluded data		
Turbine	**Timespan**	**Reason**
All	2018-01-01-2018-02-01	Large storms highly affected the wind speed. Since these are abnormal conditions this time period was excluded for all turbines.
7	2017-01-01:	

Table 4.2: Description of test cases and excluded data for the wind farm labeled B.

Wind farm C – test cases		
Turbine	**Timespan**	**Description**
10	2017-01-01:2017-07-01	Normal behaviour
10	2017-07-01:2017-12-31	Normal behaviour
10	2018-02-01:2018-07-01	Normal behaviour
10	2018-07-01:2018-12-31	Preventive maintenance changes the hydraulic oil temperature, most likely a setting.
11	2017-01-01:2017-07-01	Normal behaviour
11	2017-07-01:2017-12-31	Normal behaviour
11	2018-02-01:2018-07-01	Normal behaviour
11	2018-07-01:2018-12-31	Normal behaviour
12	2017-01-01:2017-07-01	Normal behaviour
12	2017-07-01:2017-12-31	High VCP temperatures from 2018-07-19 and onward.
13	2017-01-01:2017-07-01	Normal behaviour
13	2017-07-01:2017-12-31	Normal behaviour
13	2018-02-01:2018-07-01	Normal behaviour
13	2018-07-01:2018-12-31	Normal behaviour
14	2017-01-01:2017-07-01	Normal behaviour
14	2017-07-01:2017-12-31	Normal behaviour
14	2018-02-01:2018-07-01	Normal behaviour
14	2018-07-01:2018-12-31	Normal behaviour
Wind farm C – excluded data		
Turbine	**Timespan**	**Reason**
All	2018-01-01-2018-02-01	Large storms highly affected the wind speed. Since these are abnormal conditions this time period was excluded for all turbines.
12	2018-01-01:	A fault occurred in 2017-07-19 and was never fixed. To not overrepresent this fault only the first time period where it was present was used and the following were dropped.

Table 4.3: Description of test cases and excluded data for the wind farm labeled C.

Wind farm D – test cases		
Turbine	**Timespan**	**Description**
15	2017-07-01:2017-12-31	Normal behaviour
15	2018-02-01:2018-06-01	Normal behaviour
15	2018-07-01:2018-12-31	Grid curtailment between 2018-11-06 and 2018-11-08
16	2017-07-01:2017-12-31	Normal behaviour
16	2018-02-01:2018-06-01	Normal behaviour
16	2018-07-01:2018-11-01	Normal behaviour
17	2017-07-01:2017-12-31	Normal behaviour
17	2018-02-01:2018-06-01	Multiple rotor sensor errors and strange recorded rotor speeds.
17	2018-07-01:2018-11-01	Normal behaviour
18	2017-07-01:2017-12-31	Normal behaviour
18	2018-02-01:2018-06-01	Normal behaviour
18	2018-07-01:2018-11-01	Normal behaviour
Wind farm D – excluded data		
Turbine	**Timespan**	**Reason**
All	2018-01-01-2018-02-01	Large storms highly affected the wind speed. Since these are abnormal conditions this time period was excluded for all turbines.
All	2017-04-12:2017-04-13	A possible grid curtailment was noticed in the data but not in the logs. To avoid overrepresenting grid curtailments, this time period was excluded.
All	2017-04-29:2017-05-01	A possible grid curtailment was noticed in the data but not in the logs. To avoid overrepresenting grid curtailments, this time period was excluded.
All	2017-06-06:2017-06-07	A possible grid curtailment was noticed in the data but not in the logs. To avoid overrepresenting grid curtailments, this time period was excluded.
All	2018-06-04:2018-06-06	A grid curtailment was noticed both in the data and in the logs. To avoid overrepresenting grid curtailments, this time period was excluded.
All except 15	2018-11-06:2018-11-08	A grid curtailment was noticed both in the data and in the logs. To avoid overrepresenting grid curtailments, this time period was excluded for all but one wind turbine.

Table 4.4: Description of test cases and excluded data for the wind farm labeled D.

4.2 Postprocessing of the reconstruction error

The reconstruction error can be noisy even for healthy WT data and a well-fitted autoencoder model. Wind gusts, grid instabilities, and other environmental anomalies can cause high spikes that would trigger a threshold-based alarm system, even if the WT is still functioning well. Hence, it is necessary to process the reconstruction error in some way so that the alarm system doesn't trigger for isolated errors.

For a real malfunction, it is expected that the errors have a larger amplitude on average and occur more frequently. It is desirable to capture even small shifts in the average error and not trigger alarms for big but rarely occurring spikes. The exponentially weighted moving average (EWMA) is commonly used to construct a control chart for such cases. It smoothens the input data and accumulates a history of previous inputs, giving it exactly the desired properties. EWMAs are described further in Section 2.3.

For each of the test cases described in Section 4.1 an EWMA was calculated according to equation (2.14), using the Mahalanobis distance of the reconstruction error as the underlying time series. The smoothing parameter λ was set to give a reasonable trade-off between smoothing spikes and having a short response time. By visual inspection of calculated EWMAs, it was found that values between $\lambda = 0.001$ and $\lambda = 0.01$ were appropriate. To have a physical interpretation, the value $\lambda = 1/144 \approx 0.007$ was chosen. This corresponds to a response time of approximately 24h or one day.

In Figures 4.1 and 4.2 EWMA are compared with the original data for normal SCADA data and SCADA data from a turbine which experienced a grid curtailment. It can be seen how the EWMA removes all spikes for the normal data but doesn't completely remove the spikes from the grid curtailment as these were larger and continuously occurring.

4.3 Alarm system

To determine when the alarm system should trigger, a threshold was used for the EWMA. Typically, the so-called *control limits* (2.17) and (2.18) are used as an upper and lower threshold respectively. However, these control limits are derived for a process that has a normal distribution with mean μ. With the Mahalanobis distance, this is not the case. Positive or negative does not make sense when talking about a summary of multiple reconstruction errors and the Mahalanobis distance is always positive. Therefore, it is not reasonable to use its mean and standard deviation as the target and width of the control chart as is done in equations (2.17) and (2.18). As a target, zero is reasonable since we are aiming for a perfect reconstruction on normal data. For the width, it helps to imagine a hypothetical "signed Mahalanobis distance" which is symmetric around zero. For a single signal, the Mahalanobis

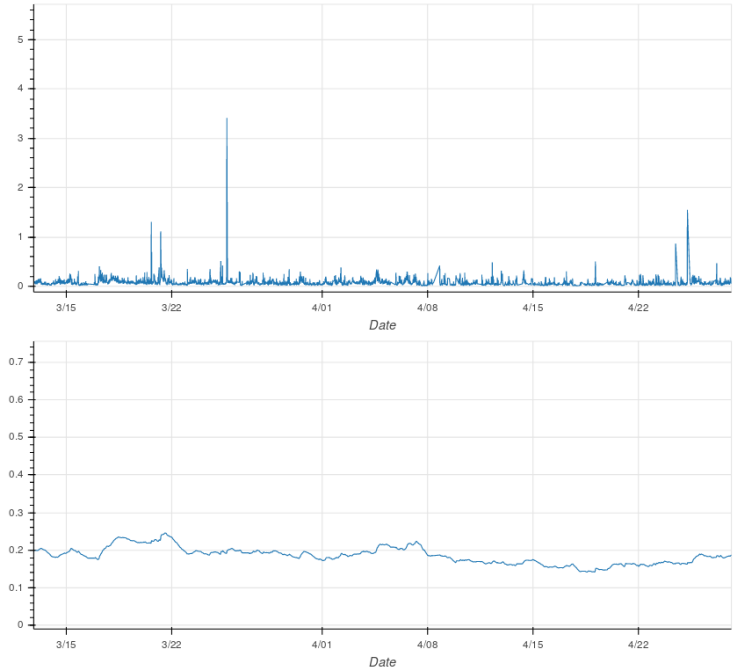

Figure 4.1: The difference between the Mahalanobis distance and an EWMA calculated on this distance. The data is from normal conditions.

distance corresponds to the absolute value of the reconstruction error and this "signed Mahalanobis distance" would hence be the actual signed reconstruction error. The standard deviation of a single, signed, reconstruction error r is estimated as its root mean square (RMS):

$$\sigma_r = \frac{1}{n}\sum_{i=1}^{n} r^2.$$

Similarly, it makes sense to estimate the width of the actual control chart with the root mean square of the Mahalanobis distance and the only control limit is therefore set to

$$\mathrm{CL}_{\mathrm{MD}} = 0 + L \times \mathrm{RMS}(\mathrm{MD}) \times \sqrt{\frac{\lambda}{2-\lambda}}. \tag{4.1}$$

Note that this is a *steady-state* control limit and therefore more resemblant to the control limits (2.19) and (2.20) than to (2.17) and (2.18). With the selected λ, the difference between the actual control limits and the steady-state control limits less than 0.0001% after the EWMAs had been running for one week. Since all test cases ranged for months and all anomalies occurred at least one week after the EWMAs were initiated it was justified to use this simplified control limit.

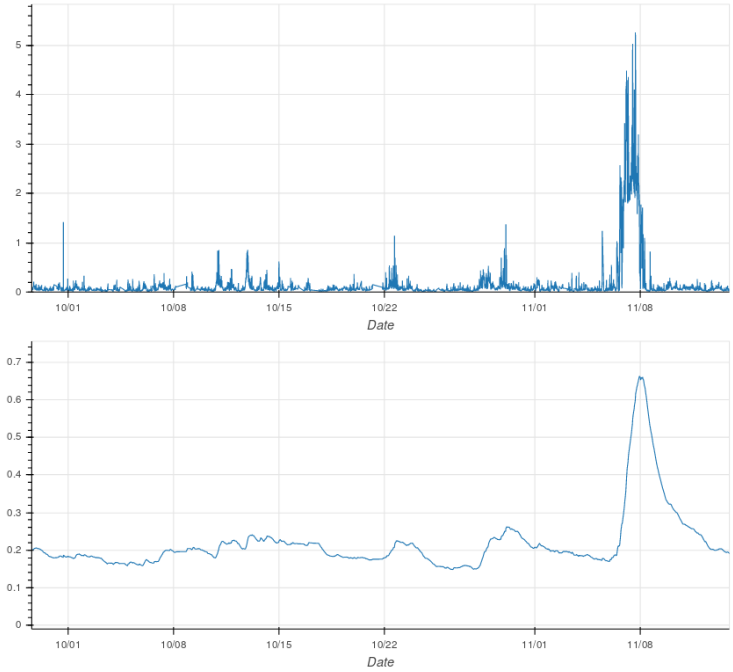

Figure 4.2: The difference between the Mahalanobis distance and an EWMA calculated on this distance. Included in the data is a grid curtailment at around the sixth of November.

If the control limit (4.1) was passed at any time in the test case this was considered a positive verdict by the model. The value of L was varied from zero to the maximum value required to yield zero positives. Based on this, precision-recall curves where calculated for all models.

4.4 Results

Using the test cases listed in Tables 4.1, 4.2, 4.3 and 4.4 precision-recall curves were calculated for different code-sizes and noise levels. To minimize the effect of the stochasticity in training five networks were trained for each configuration and their resulting precision-recall curves were averaged.

From Section 3.4.2 we expect code size 18 to be the optimal capacity. For this code size, there existed one distinct threshold for the level of input noise. Before this threshold, training and validation errors were barely affected by the noise, but after they both increased. Based on this threshold, four different noise levels were chosen. These were 0, 0.2, 0.3 and 0.5 which, respectively, correspond to no input noise, noise slightly lower than the threshold, noise slightly higher than the threshold and a high level of input noise.

The code sizes ranged from low to high capacity, covering the optimal capacity. They were 6, 12, 18, 24, 30 and 36 and the four previously chosen noise levels were tested for each one.

In the following sections, precision-recall curves and their interpretations are presented. First, for each code size, the effect of input noise is investigated. Then, using the best performing model for each case, a comparison between the code sizes is shown.

4.4.1 Code size 6

Figure 4.3 shows a comparison between the different levels of input noise for code size 6. All curves show similar behaviour of detecting most of the anomalies but struggling to reach a recall above 83%, only doing so at very low precision. This corresponds to failure at detecting one of the anomalies and, upon closer inspection, it was found that the change in hydraulic oil temperature did not give any significantly increased Mahalanobis distance. This case is presented in more detail in Appendix B. From the figure, it also appears that the input noise had an ever so slightly worsening effect.

4.4.2 Code size 12

Figure 4.4 shows the same comparison for code size 12. Once again the models struggle to detect the final anomaly and once again this corresponds to the change in hydraulic oil temperature. Now, however, the noise has a clear effect, giving increasingly better models up to input noise 0.3 and then a worse model for highest level of input noise 0.5.

4.4.3 Code size 18

For code size 18, shown in Figure 4.5 the results are a bit different. The models still struggle to detect all the anomalies but they do manage at a higher precision than previously. Also, they reach a higher recall of around 90% before the precision gets too low. Upon closer inspection it was found that the rotor sensor errors and the change in hydraulic oil temperature were both difficult, albeit not impossible, to detect. The odd number of 90% comes from the averaging over iterations. For some iterations, the model detected all anomalies; for some, the model missed the sensor errors; and, for some, it missed the hydraulic oil change. For the effect of input noise, it is hard to give a concise verdict, 0.5 gives the best recall for a high precision while the others reach perfect, unity, recall without losing as much precision.

4.4.4 Code size 24

In Figure 4.6, the models with code size 24 show similar behaviour to those with code size 18. Once again, when investigating the Mahalanobis distances it was found that the rotor sensor error and hydraulic oil change where the most difficult to detect. In this case, the contribution of noise is more clear, the high levels of 0.3 and 0.5 show a relatively high recall for a high precision but then fail to detect the final anomalies. On the other hand, noise of 0 and 0.2 show a quick drop in precision but then manage to reach perfect recall sooner.

4.4.5 Code size 30

Code size 30, shown in Figure 4.7 shows similar behaviour to code sizes 6 and 12. For all noise levels, the models quickly detect all but one anomaly, the hydraulic oil temperature change. There is a slight variation between noise levels but not a lot.

4.4.6 Code size 36

The *overcomplete* models with code size 36, Figure 4.8, show similar behaviour to those with code size 30 of quickly detecting all but one anomaly, once again the change in hydraulic oil temperature. It is surprising that these models are able to detect anything as with a code size of 36 and only 33 inputs they should be able to perfectly model the identity mapping and thus yield zero reconstruction error.

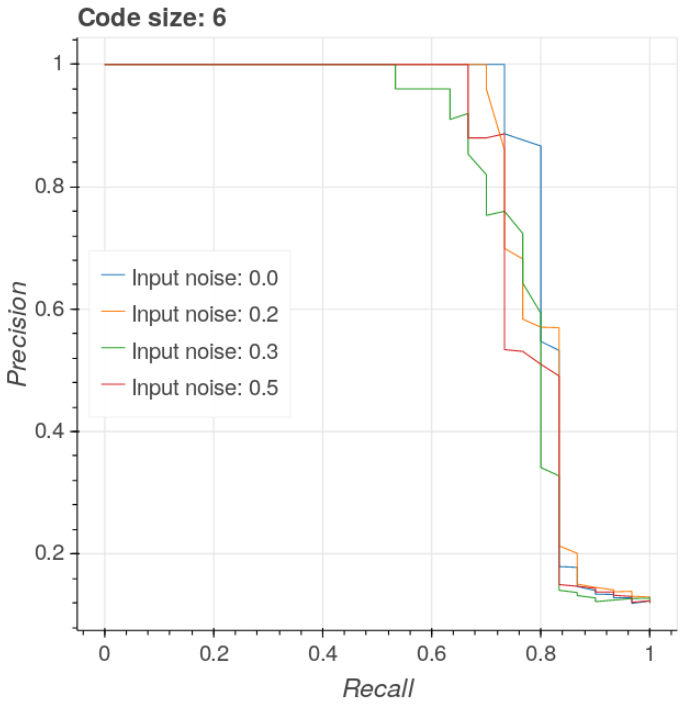

Figure 4.3: Precision-recall curves for Autoencoder models with code size 6 but different levels of input noise. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

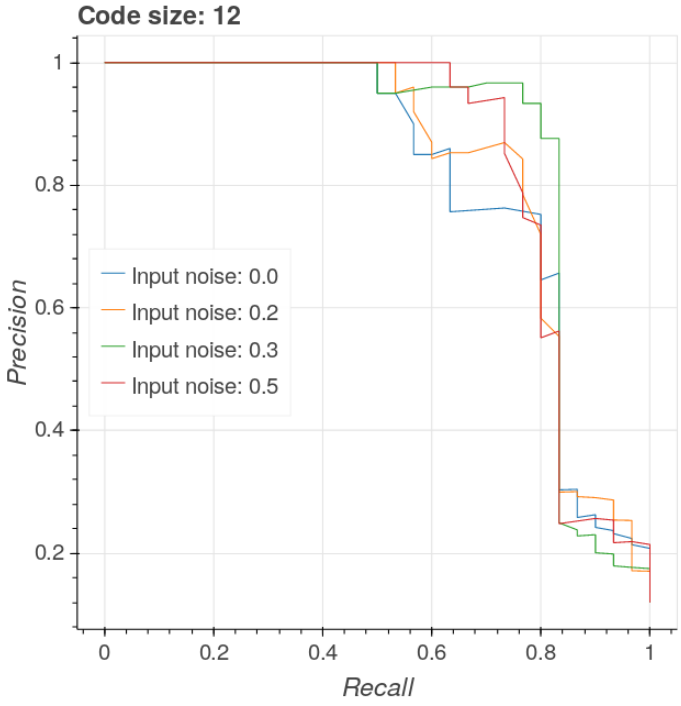

Figure 4.4: Precision-recall curves for Autoencoder models with code size 12 but different levels of input noise. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

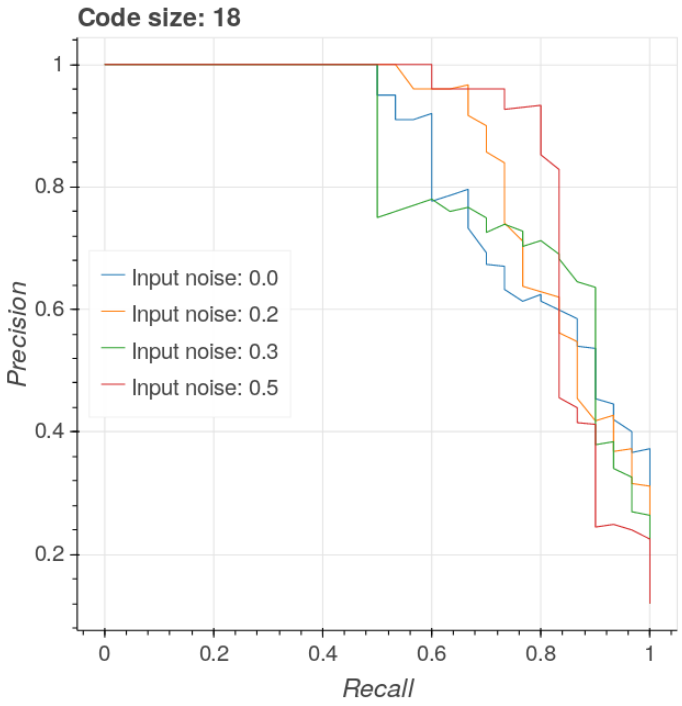

Figure 4.5: Precision-recall curves for Autoencoder models with code size 18 but different levels of input noise. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

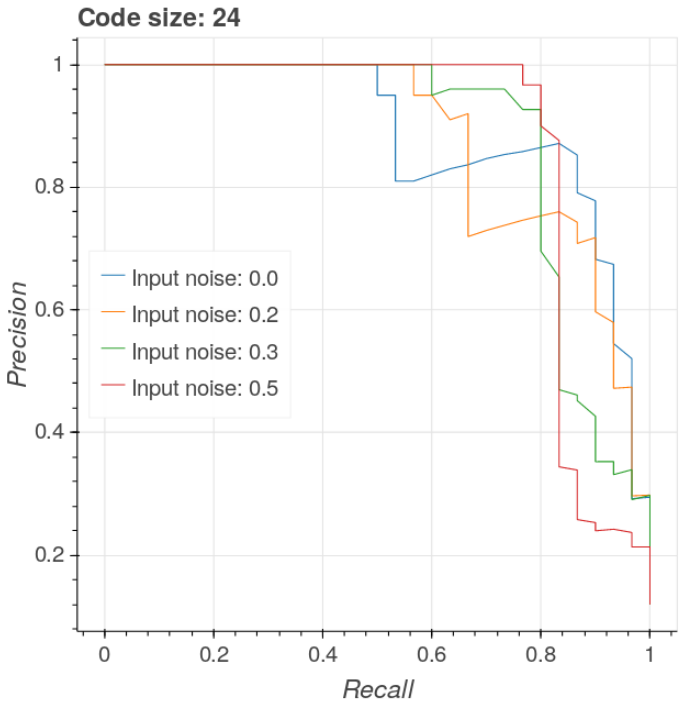

Figure 4.6: Precision-recall curves for Autoencoder models with code size 24 but different levels of input noise. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

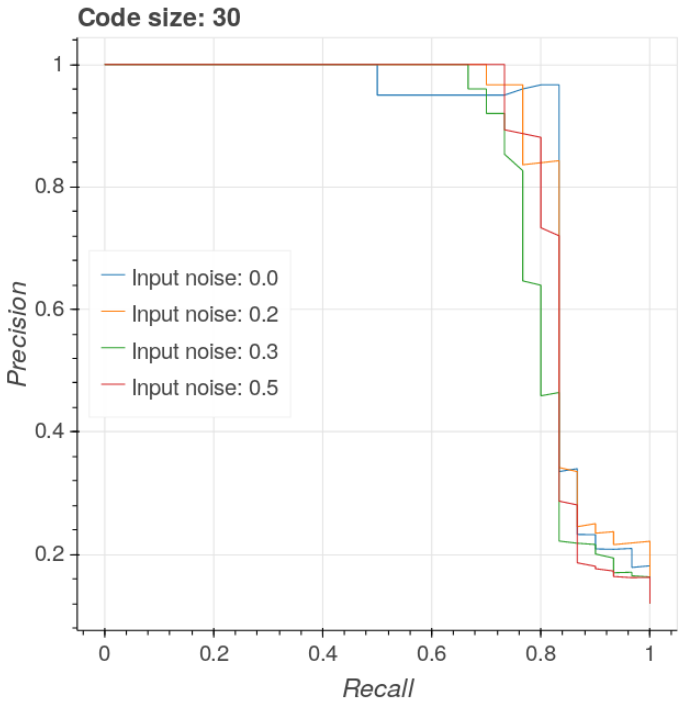

Figure 4.7: Precision-recall curves for Autoencoder models with code size 30 but different levels of input noise. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

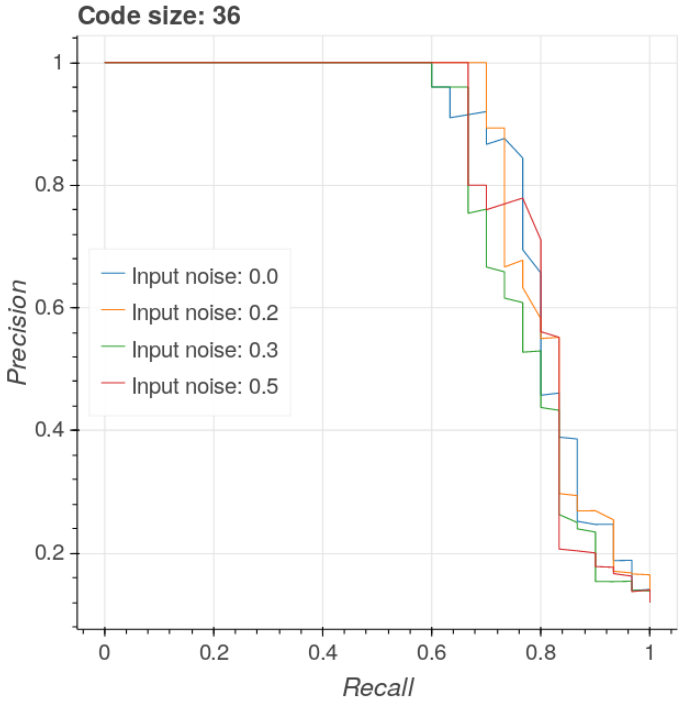

Figure 4.8: Precision-recall curves for Autoencoder models with code size 36 but different levels of input noise. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

4.4.7 Comparison between different code sizes

In Figure 4.9 a comparison of the best autoencoder models for each code size is shown. In some cases, there was no obvious best model as some curves showed a high precision early on but had troubles surpassing a certain recall while others quickly dropped in precision but then reached a higher recall. It is possible that precision can be improved with more effective post-processing. If, on the other hand, a model fails to detect an anomaly, post-processing cannot help. Therefore, the latter models were chosen in these ambiguous cases.

From the graph, code size 24 is the clear winner with code size 18 being the second best. These were the only two code sizes for which the models could reliably detect all anomalies. Hence, the perfect model capacity should be somewhere in this region.

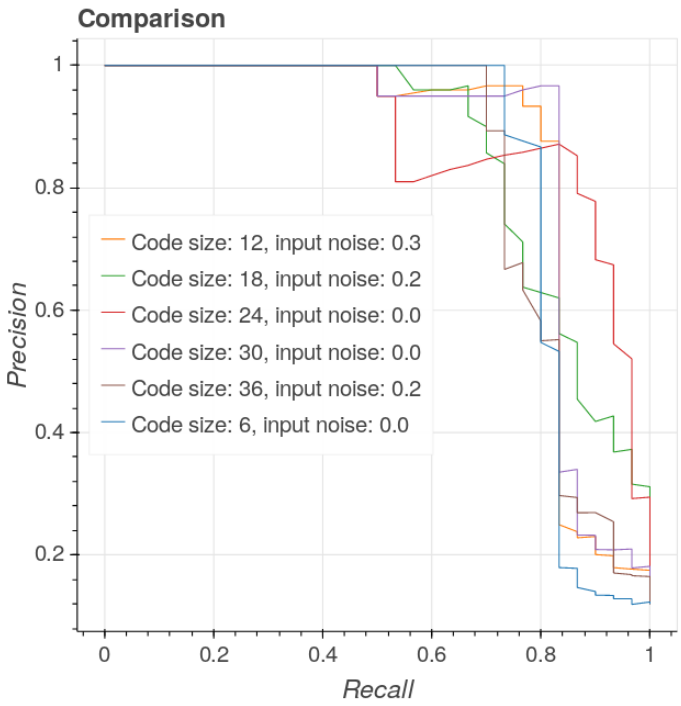

Figure 4.9: Precision-recall curves for the best performing Autoencoder models for each code size. For each setting, five models were trained and the results averaged. This was done to account for the stochasticity in training.

Chapter 5

Closure

This chapter discusses the various parts of the thesis, suggests future work and finally summarizes the thesis.

5.1 The importance of code size

Out of all the models compared in Chapter 4, the ones with code size 24 and no or low input noise performed the best, being able to detect all anomalies in the dataset. The models with code size 18 were also able to detect all faults but at lower precision. For all the other code sizes, the precision was very low when the last two faults were detected. It cannot be said that these models actually found anything as pure random guessing would come close to the same precision.

All of this indicates that the optimal code size is somewhere close to 24, although it appears that the exact choice is not crucial since code size 18 also performed well. There seems to be a rather wide range of code sizes that perform well and as long as one of these is selected, an autoencoder works fine for anomaly detection. This is good news for a practitioner since one often wants to model different data distributions, each requiring different model capacities, Performing a study such as this for each case would require a lot of work and is not always feasible.

Surprisingly, even the overcomplete autoencoders with code size 36 were able to detect a large number of the anomalies indicating that they did not just learn the identity mapping. Potential reasons for this are the depth of the neural network, the use of ReLU as an activation function, the stochasticity in training or the noise induced by batch normalization. For a concrete explanation, however, a deeper understanding of autoencoders and neural networks is required. What can be said though, is that one need not be too concerned about an autoencoder designed this way learning the identity function.

5.2 The effect of normally distributed input noise

In this study, denoising autoencoders were used to construct anomaly detection models. The input noise applied during training was normally distributed and various scales were tested. It was believed that this would help the autoencoder to model the true data distribution rather than the identity function, thus rendering it more capable as an anomaly detection model. For the models compared in Chapter 4, however, the effect of noise was not clear. For some code sizes, noise increased the model performance, while for others it decreased it. Even where noise had a positive effect there was no clear pattern as the performance went fluctuated with increasing noise.

The zero noise models were among, if not the best, performing models for almost all code sizes. Since no clear benefits of noise could be found, and it adds an additional hyperparameter to be selected, it is the author's recommendation to not use normally distributed noise for anomaly detection in WTs. However, this need not be the case for all types of input noise. As is described in the literature review in Appendix A, Jiang et. al [26] used masking noise to train autoencoders for anomaly detection. During training, masking noise replaces some of the inputs with a zero value, forcing the autoencoder to disregard information from this signal for the reconstruction. Compared to normally distributed noise this likely has a completely different effect on what the autoencoder learns, and it appears that it was beneficial for anomaly detection. Further studies of the effects of this kind of noise would be interesting.

In Chapter 3 the effect of input noise was investigated by the means of so-called "noise curves": plots of training and validation errors versus the input noise. Two thresholds were distinguished: one for when training and validation errors started to increase, and one for when they started to diverge from each other. It was found that as the code size was increased from low to high, the two thresholds came closer and collapsed around code size 18. Why this happened remains unclear, but code size 18 was considered a plausible candidate for a model with optimal capacity. In the validation study, however, the models with code size 18 were proven inferior to those with code size 24. Nevertheless, the proposed code size does lie within the region of good performance and the investigation appears to have had some merit. If the effect of normally distributed input noise was to be studied further, it could be possible to devise a method to select an autoencoder's optimal capacity based purely on training and validation errors which could prove helpful for hyperparameter selection.

5.3 Discussion of the validation study

The ultimate goal of this field of research is to maximize a condition monitoring system's ability to detect and prevent faults in a wind turbine. To achieve this, it is important to distinguish different components of a CMS and test each component according to its responsibilities. For the autoencoder, the task is to detect anomalies; it is not to distinguish between anomalies that will lead to critical failures and anomalies that can be safely ignored. Such filtering would require the incorporation of additional information, e.g. domain knowledge or environmental conditions, and this is probably easier done in the postprocessing step. Since this thesis focused in particular on the autoencoder, all kinds of anomalies were included in the validation study presented in Chapter 4. Neither the grid curtailment nor the shift in hydraulic oil temperature will lead to a failure in the wind turbine, but, they have the same characteristics as a real impending fault in that they constitute a persistent change to some of the signals. Thus, from the autoencoder's perspective, they look the same and an autoencoder that detects these will also detect a real impending fault.

For the validation study, service logs were utilized to create a labeled dataset. This, however, was not without difficulties. A test case was only labeled as positive if the logs described a clear event. Since these were all detectable by the models they appear to have been correctly labeled. The validity of the negatively labeled cases is not as certain. There were multiple cases where almost all models showed errors at the same time despite there being nothing in the logs. Whether such cases should have been included in the validation study or not is up for debate. Here, they were, and this might have skewed the comparison between different models. In Figure 4.9 it can be seen that some models quickly drop in precision as one lowers the threshold. The first drop comes as the models pick up these uncertainly labeled cases as false positives. If these cases correspond to actual anomalies and consequently should have been labeled as positives, they would instead have *increased* both the precision and recall of these models. Nevertheless, labeling these cases as positives, or even removing them from the test suite, would have introduced a selection bias since such a choice would be based solely on the results from some models.

Because of the difficulties in correctly labeling test cases, only data which had service logs was utilized. This limited the number of anomalies in the dataset; perhaps the models would have been more distinguishable if exposed to a larger variety of anomalies. Hence, if more data with more detailed service logs was available, the results of a validation study could be improved.

As a basis for an alarm system, the Mahalanobis distance was used in this work. The MD summarizes all reconstruction errors into one representative error signal. When analysing the results, it was found that, for a given fault, the reconstruction errors that contributed the most to the MD were

the ones related to the fault. This was the case for e.g. the shift in hydraulic oil temperature studied in Appendix B. The shift caused a reconstruction error for this temperature but the other signals were barely affected. Consequently, the shift was difficult to detect by just looking at the Mahalanobis distance. If an alarm system instead monitored each reconstruction error individually, this case would have been easily detected. Such an alarm system could, therefore, have narrower control limits and be more sensitive to anomalies. Furthermore, a CMS that monitored each reconstruction error individually could, on top of saying *when* an anomaly occurred, also give some insight to *why* it occurred. This could prove valuable to the industry and suggests a topic for further investigation.

5.4 Concluding remarks

In this thesis, a condition monitoring system for wind turbines was developed. The system, which was based on deep autoencoders, was able to detect a variety of faults and other anomalous behaviours in the wind turbines. This suggests that system-wide condition monitoring is possible. Moreover, the system developed could well be applied at an industrial scale.

From a more academic standpoint, the work here provides an investigation of autoencoders in light of their effectiveness as anomaly detection models. Different hyperparameters are tested and their influence on performance is evaluated. This can, potentially, lead to a better understanding of this unsupervised learning technique.

To improve upon these results, two main areas of future investigations can be identified. First, to further benefit the development of autoencoders, it would be interesting to do a more thorough investigation of the connection between input noise and training and validation errors. Perhaps, by looking at something similar to the noise curves here, a method could be developed that automated hyperparameter selection for autoencoders. To perform such a study, it would probably be beneficial to use synthetic data for which the retrieval of labeled cases would be trivial.

Second, to further develop a condition monitoring system for wind turbines, it would be useful to monitor the individual reconstruction errors as opposed to summarizing them to a single value as was done here. This could, in addition to providing a more sensitive anomaly detection, make it possible to identify which subsystem that is responsible for a given anomaly.

Bibliography

[1] "United Nations, climate change." http://www.un.org/en/sections/issues-depth/climate-change/index.html. Accessed: 2018-08-31.

[2] "Paris agreement." https://treaties.un.org/pages/ViewDetails.aspx?src=TREATY&mtdsg_no=XXVII-7-d&chapter=27&clang=_en. Accessed: 2018-11-08.

[3] I. (2018), "Renewable capacity statistics 2018," tech. rep., International Renewable Energy Agency (IRENA), 2018.

[4] I. (2018), "Renewable power generation costs in 2017," tech. rep., International Renewable Energy Agency (IRENA), 2018.

[5] Z. Hameed, Y. Hong, Y. Cho, S. Ahn, and C. Song, "Condition monitoring and fault detection of wind turbines and related algorithms: A review," *Renewable and Sustainable Energy Reviews*, vol. 13, no. 1, pp. 1–39, 2009. cited By 636.

[6] W. Yang, R. Court, and J. Jiang, "Wind turbine condition monitoring by the approach of scada data analysis," *Renewable Energy*, vol. 53, pp. 365–376, 2013. cited By 120.

[7] K. Kim, G. Parthasarathy, O. Uluyol, W. Foslien, S. Sheng, and P. Fleming, "Use of scada data for failure detection in wind turbines," *ASME 2011 5th International Conference on Energy Sustainability, ES 2011*, pp. 2071–2079, 2011. cited By 45.

[8] Y. Wang and D. Infield, "Supervisory control and data acquisition data-based non-linear state estimation technique for wind turbine gearbox condition monitoring," *IET Renewable Power Generation*, vol. 7, no. 4, pp. 350–358, 2013. cited By 22.

[9] Y. Zhao, D. Li, A. Dong, D. Kang, Q. Lv, and L. Shang, "Fault prediction and diagnosis of wind turbine generators using scada data," *Energies*, vol. 10, no. 8, 2017. cited By 0.

[10] M. Wilkinson, B. Darnell, T. Van Delft, and K. Harman, "Comparison of methods for wind turbine condition monitoring with scada data," *IET Renewable Power Generation*, vol. 8, no. 4, pp. 390–397, 2014. cited By 28.

[11] W. Yang, R. Court, and J. Jiang, "Wind turbine condition monitoring by the approach of scada data analysis," *Renewable Energy*, vol. 53, pp. 365–376, 2013. cited By 105.

[12] L. Wang, Z. Zhang, H. Long, J. Xu, and R. Liu, "Wind turbine gearbox failure identification with deep neural networks," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 3, pp. 1360–1368, 2017. cited By 9.

[13] P. Dao, W. Staszewski, T. Barszcz, and T. Uhl, "Condition monitoring and fault detection in wind turbines based on cointegration analysis of scada data," *Renewable Energy*, vol. 116, pp. 107–122, 2018. cited By 6.

[14] P. Cross and X. Ma, "Model-based and fuzzy logic approaches to condition monitoring of operational wind turbines," *International Journal of Automation and Computing*, vol. 12, no. 1, pp. 25–34, 2015. cited By 6.

[15] H. Long, L. Wang, Z. Zhang, Z. Song, and J. Xu, "Data-driven wind turbine power generation performance monitoring," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 10, pp. 6627–6635, 2015. cited By 14.

[16] J. Tautz-Weinert and S. Watson, "Comparison of different modelling approaches of drive train temperature for the purposes of wind turbine failure detection," *Journal of Physics: Conference Series*, vol. 753, 2016. cited By 3.

[17] E. Papatheou, N. Dervilis, A. Maguire, I. Antoniadou, and K. Worden, "A performance monitoring approach for the novel lillgrund offshore wind farm," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 10, pp. 6636–6644, 2015. cited By 13.

[18] M. Schlechtingen, I. Santos, and S. Achiche, "Wind turbine condition monitoring based on scada data using normal behavior models. part 1: System description," *Applied Soft Computing Journal*, vol. 13, no. 1, pp. 259–270, 2013. cited By 90.

[19] M. Schlechtingen and I. Santos, "Wind turbine condition monitoring based on scada data using normal behavior models. part 2: Application examples," *Applied Soft Computing Journal*, vol. 14, no. PART C, pp. 447–460, 2014. cited By 35.

[20] M. Garcia, M. Sanz-Bobi, and J. del Pico, "Simap: Intelligent system for predictive maintenance. application to the health condition monitoring of a windturbine gearbox," *Computers in Industry*, vol. 57, no. 6, pp. 552–568, 2006. cited By 169.

[21] A. Zaher, S. McArthur, D. Infield, and Y. Patel, "Online wind turbine fault detection through automated scada data analysis," *Wind Energy*, vol. 12, no. 6, pp. 574–593, 2009. cited By 196.

[22] M. Schlechtingen and I. Ferreira Santos, "Comparative analysis of neural network and regression based condition monitoring approaches for wind turbine fault detection," *Mechanical Systems and Signal Processing*, vol. 25, no. 5, pp. 1849–1875, 2011. cited By 95.

[23] P. Bangalore and L. Tjernberg, "An artificial neural network approach for early fault detection of gearbox bearings," *IEEE Transactions on Smart Grid*, vol. 6, no. 2, pp. 980–987, 2015. cited By 44.

[24] P. Qian, X. Ma, and P. Cross, "Integrated data-driven model-based approach to condition monitoring of the wind turbine gearbox," *IET Renewable Power Generation*, vol. 11, no. 9, pp. 1177–1185, 2017. cited By 1.

[25] L. Wang, Z. Zhang, J. Xu, and R. Liu, "Wind turbine blade breakage monitoring with deep autoencoders," *IEEE Transactions on Smart Grid*, vol. 9, no. 4, pp. 2824–2833, 2018. cited By 1.

[26] G. Jiang, P. Xie, H. He, and J. Yan, "Wind turbine fault detection using a denoising autoencoder with temporal information," *IEEE/ASME Transactions on Mechatronics*, vol. 23, no. 1, pp. 89–100, 2018. cited By 1.

[27] "Intel's new chip puts a teraflop in your desktop. here's what that means." https://www.popsci.com/intel-teraflop-chip. Accessed: 2018-09-12.

[28] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[29] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. cited By 5127.

[30] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[31] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. cited By 8459.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[33] J. A. Rice, *Mathematical Statistics and Data Analysis.* Belmont, CA: Duxbury Press., third ed., 2006.

[34] G. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006. cited By 5660.

[35] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006. cited By 5277.

[36] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research*, vol. 9, pp. 249–256, 2010. cited By 1656.

[37] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in *2009 IEEE 12th International Conference on Computer Vision*, pp. 2146–2153, Sept 2009.

[38] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines vinod nair," *Proceedings of ICML*, vol. 27, pp. 807–814, 06 2010.

[39] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *Journal of Machine Learning Research*, vol. 15, 01 2010.

[40] D. Sussillo, "Random walks: Training very deep nonlinear feed-forward networks with smart initialization," *CoRR*, 12 2014.

[41] P. C. Mahalanobis, "On the generalised distance in statistics," in *Proceedings National Institute of Science, India*, vol. 2, pp. 49–55, Apr. 1936.

[42] A. Kessy, A. Lewin, and K. Strimmer, "Optimal whitening and decorrelation," *The American Statistician*, 12 2015.

[43] D. Montgomery, *Introduction to Statistical Quality Control.* Wiley, 2009.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 01 2012.

[45] D. P. B., S. W. J., and K. Andrzej, "Stationarity-based approach for the selection of lag length in cointegration analysis used for structural damage detection," *Computer-Aided Civil and Infrastructure Engineering*, vol. 32, no. 2, pp. 138–153, 2017.

[46] S. Johansen, "Statistical analysis of cointegration vectors," *Journal of Economic Dynamics and Control*, vol. 12, no. 2, pp. 231 – 254, 1988.

[47] J.-S. Jang, "Anfis: Adaptive-network-based fuzzy inference system," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 3, pp. 665–685, 1993. cited By 8734.

[48] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861 – 874, 2006. ROC Analysis in Pattern Recognition.

Appendix A

Literature review

Condition monitoring systems for wind turbines based on SCADA-data is an active field of research. Many different methods have been investigated and some have proven quite successful. Studies differ mainly in what anomaly detection model they use and in general these can be classified into one of three categories. The first kind of model summarizes training data and then calculates how similar new data points are to this summary. These models are referred to as *similarity-based models* and typically unsupervised machine learning techniques such as clustering are used.

The second kind of model reconstructs one or all available signals from a combination of the other available signals. The hypothesis is that, since the parametrized reconstruction model is configured using healthy training data, it will only give correct results for data coming from the same distribution. Consequently, a fault in the WT and a change in its dynamics will lead to detectable errors the reconstruction. These models are called *residual-based models* and various techniques have been employed including artificial neural networks (ANN) and polynomial fitting.

The third category of models monitor the power curves of WTs. These models are typically not as sophisticated as models in the other categories and may not be as sensitive to impending failures. Nevertheless, they have the advantage of it being simple to compare different WTs with each other and monitoring can be done on a wind farm level as opposed to at a wind turbine level. Therefore, the models are more robust to abnormal weather conditions and other environmental effects because these will affect all turbines equally. Comparing the results for different turbines is not as straightforward for the other kinds of models as these tend to be more individualized. For the same reasons, this category of models can also be effective for a more general performance assessment.

The following subsections present a review of the scientific work on the three different kinds of models. Some models could be categorized into several categories, in such case the study is presented where it makes the

most sense with regards to the surrounding text in order to enhance the readability of this chapter.

A.1 Similarity-based models

Similarity-based models detect anomalies by comparing the properties of new data to data which represents normal operating conditions.

Kim et. al [7] investigated two methods for detecting a gearbox fault in a research turbine in Colorado. The first method was based on deep autoencoders (DAE) which is a neural network that takes its input data, compresses it to fewer nodes and then decompresses it back to a model of the original input data, see Section 2.1.9 for further detail. The model is trained with healthy WT data and then the values at the middle layer were used to form two statistical measures. Thresholds for these statistics were constructed from the training data and these were then used to determine whether a new sample was normal or abnormal. The model produced alarms both when tested on healthy data and data leading up to the gearbox fault showing that this threshold system was unsuccessful. Nevertheless, the statistics exceeded the thresholds a lot more on the day of the failure than on the days of healthy data, indicating that it could be possible to get better results using a different monitoring method.

In a second method the authors used a self-organizing map (SOM) to create clusters in the data. A SOM is a neural network in which a given number of neurons are randomly placed in the data space. During a training process these are then iteratively moved toward input data points in a competitive process. Eventually, the neurons stabilize at center points of different clusters of the training data. They thus provide a compressed representation of the data to which new signals can be compared. The authors used both faulty and healthy data for their training and found that some clusters only contained healthy data, some only faulty data and some a mixture of both. Therefore, the results were inconclusive.

In [8] Infield et. al employed a nonlinear state estimation technique (NSET) which was originally developed for condition monitoring in nuclear power plants to compute a similarity measure between historical and new data. NSET is a rather simple method in which a so-called state memory matrix is constructed by putting together historic data vectors in a matrix. Next, the new data is projected onto the space spanned by this matrix. The distance between the data and its projection is finally used to determine whether the data comes from healthy or faulty conditions. A case study was performed with two months of data from ten turbines. Seven of these were used for training, one for validation and the two others that had experienced gearbox faults were used for testing. The model was able to detect the failures one month in advance.

Zhao et. al [9] applied density-based clustering (DBScan) to predict the remaining useful life (RUL) of a generator. The intuition goes as follows: most of the time the wind turbine is functioning properly and therefore this data should be much more numerous and dense than data coming from an unhealthy state. The largest cluster should, therefore, be the one corresponding to healthy data. For a healthy turbine the other clusters will come from special weather conditions and other abnormal working states but for a turbine with a gradually progressing fault smaller clusters would correspond to worsening states of the turbine. Based upon this, the authors introduced an anomaly operation index equal to the proportion of data instances not in the biggest cluster during a time period t. They set a threshold for what was considered a normal and abnormal value of this index and used this to calculate the RUL by weighting with the life expectancy of a healthy WT. The authors had access to 18 months of data from 150 WTs and they knew when faults had occurred for these. They looked at five WTs that experienced generator faults and took data from 30, 40, 50 and 60 days backwards in time. Based upon this they were able to estimate the RUL with around 80% accuracy up to 18 days in advance.

A.2 Residual-based models

Residual-based models recreate some part of the SCADA data and use the recreation error to detect if a fault has occured. For this to work the model has to be accurate for healthy data and a number of different approaches have been investigated.

Some authors have looked into the physics of a WT and from this constructed parametrized functions (typically polynomial) which have been fitted on data [10]. Although this approach has shown some promise, the models often have to be adjusted from case to case since two WTs do not have the exact same behaviour and different wind farms often have different weather conditions. Therefore, this approach is not applicable for large scale condition monitoring and only one work is presented here. Other authors have also attempted to fit polynomials to the data but instead of physical reasoning they have applied various statistical methods to derive the shape of the polynomials [11], [13]. These methods would be easier to apply at scale and the results show some promise. All polynomial models are presented in Section A.2.1.

Common supervised learning methods have also been applied to create residual-based models including linear models (LM), k-nearest neighbours (kNN), support vector machines (SVM), Gaussian process regression (GPR) and artificial neural networks (ANN). In comparative studies [22], [14], [16] ANNs have often been the winner and a lot of studies have shown the feasability of ANN-based fault detection. ANNs are able to

capture the nonlinear relations between SCADA-signals and are also general enough to be applied on a large scale. Different network structures have been employed, including single hidden-layer feedforward networks (SLFN), deep feedforward networks (DFN), layer-recurrent networks (LRN), adaptive neuro-fuzzy inference systems (ANFIS) and autoencoders (AE).

To alert the operator that a fault has occurred different error measures have been employed including the Euclidean distance from the true signal, the Mahalanobis distance (MD) and T^2-statistics. An alarm is then produced when the error has been large enough over a long enough time. Schemes to determine this has included simple and more elaborate threshold models and, more recently, exponentially weighted moving average (EWMA) where old errors contribute a little and newer more.

A.2.1 Polynomial models

Wilkinson et. al [10] showed three different methods for fault prediction in a WT. First, they compared the temperature of a component of one WT to the temperature of the same component in five other WTs in the same farm. They showed that such a method could indicate a fault but ultimately they dismissed this method since the fault detection was not very reliable. Next, they investigated the use of SOMs and measured the distance between input data and its closest node in the trained SOM. Based upon this they were also able to detect impending failures. But, since a SOM cannot say which component that had generated the fault this method was also dismissed. Finally, they looked into the physics of a WT and used correlation analysis to fit polynomial models to various component temperatures. This method was then pursued in a large case study on different wind farms with an operational range from two and a half to seven years. In total 472 WT-years of data was used and the models were able to predict 24 out of 36 various generator and main bearing faults one month to two years in advance and only produced three false positives.

Yang et. al [11] took a statistical based approach to create a anomaly detection model. They filtered the SCADA-data to interesting ranges of operations and removed standby data. Next, they binned the data according to wind speed and used the expected value for each bin. From this preprocessed data they fitted pairs of signals to each other with a fourth-degree polynomial and least squares. The pairs where chosen so that they would correlate to the subsystem in which the fault took place. E.g. for detecting a blade breakage they looked at the power versus wind speed and torque versus wind speed. They compared a historic fit to a present fit and correlation statistic on which they determined if a fault was on its way. Their method was able to detect an impending blade breakage and a generator bearing fault months ahead.

Dao et. al [13] used a statistical method called cointegration to create a

simple polynomial model. Cointegration is a technique originally developed in the field of Econometrics which is based on the concept of stationary time series. A stationary time series is one which has a distribution that doesn't change with a shift in time, see Figure A.1. Usually, time series occuring in engineering problems are not stationary, but, there may be stationary patterns within them. Cointegration is one way of finding such a pattern. Formally a set of non-stationary time series are said to be cointegrated if one can create a stationary time series by taking the linear combination of them. This happens when there are some long-run correlations between the time series and there are various methods to test for this. The authors first used methods in [45] to establish optimal time delay between the signals. Then they used Johansen's cointegration method [46], a sequential procedure of maximum likelihood estimates, to find a set of cointegrated stationary time series from the SCADA data. The method determines which signals that should be combined and calculates the coefficients between them. Given this combination one of the signals can be expressed as a linear combination of the others and given that there indeed exists some stationary pattern the residual should be a white-noise signal. The method can thus be summarized as a method based on maximum likelihood estimates to create linear mappings with optimal time delay between different SCADA-signals. The proposed method was tested on 30 days of data from a 2 MW WT. During this time period the data exhibited two anomalies: one sudden wind speed drop that caused the rotor to stop and one temporary gearbox fault. Using a threshold-based alarm system the cointegration residual alerted for both of these anomalies without giving any false alarms.

A.2.2 Single output neural networks

One of the first attempts to use neural networks for anomaly detection was done by Garcia et. al in 2006 [20]. They developed a complete system for predictive maintenance called SIMAP and tested it on a gearbox fault in a WT. They modeled three characteristic temperature properties of the gearbox with single hidden layer feedforward neural networks (SLFN) and selected a different set of inputs for each. These inputs included signals such as generator power and nacelle temperatures and also time-delayed values of the modeled signal. The inputs and outputs for the different models are summarized in table A.1. Next, they used a fuzzy system which, based on the reconstruction error of the models, determined when a fault had occurred. For the gearbox fault investigated, the system was able to produce a warning roughly 26 hours in advance.

Zaher, McArthur, and Infield [21] created a similar model to the SLFNs in SIMAP by using three months of SCADA data from 26 WTs. Next, the model was tested on two years of data from the wind farm. Two gearbox faults were detected, one six months in advance and one one month in ad-

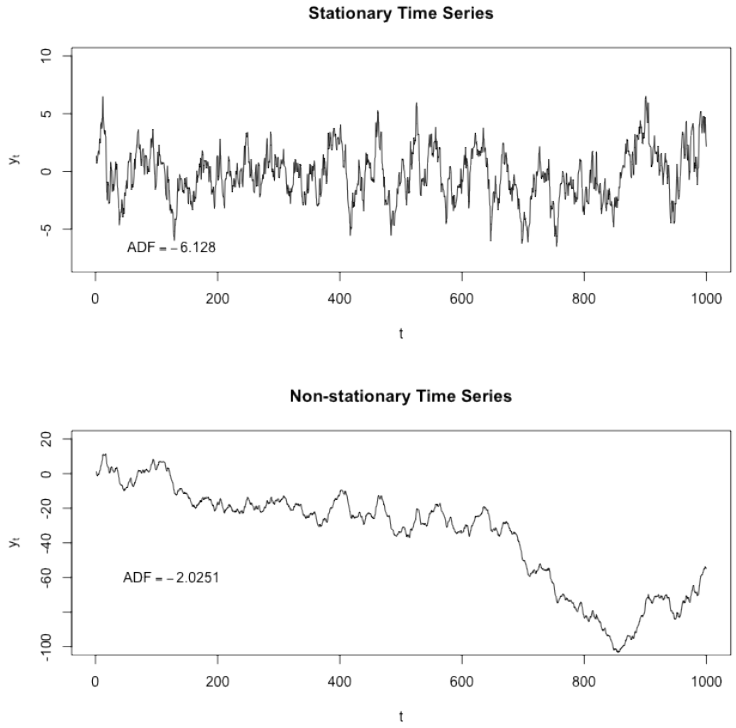

Figure A.1: A comparison between a stationary (top) time series and a nonstationary (bottom).

vance. No reports of false positives or missed faults (false negatives) were given. The authors also created a similar model for a generator and this model was able to detect faults up to one year in advance.

Schlechtingen and Santos [22] compared two different SLFNs with a linear model. One of the SLFNs only used current values of the signals while the other also used time-delayed values of the output signal. All models were trained on three months of data from a turbine with a newly replaced bearing and were used to model generator stator temperature. Next, the models were tested on the same turbine to detect a second bearing fault. Both the SLFNs and the linear models were able to detect the second fault well ahead in time (18, 25 and 25 days ahead). Next, they performed a number of similar studies investigating two bearing damages and two stator temperature anomalies in other WTs. The authors argued that a linear model was hard to scale to many WTs since important parameters and correlations had to be determined for each individual WT. This was, however, automized with the neural networks. Therefore, in these studies, they only compared SLFNs with and without time-delayed signals. The results showed similar performance of both approaches.

Output	**Inputs**
Gearbox bearing temperature (t)	Gearbox bearing temperature $(t-1)$ Gearbox bearing temperature $(t-2)$ Generated power $(t-3)$ Nacelle temperature (t) Cooler fan slow run $(t-2)$ Cooler fan fast run $(t-2)$
Gearbox thermal difference (t)	Gearbox thermal difference $(t-1)$ Generated power $(t-2)$ Nacelle temperature (t) Cooler fan slow run $(t-2)$ Cooler fan fast run $(t-2)$
Cooling oil temperature (t)	Cooling oil temperature $(t-1)$ Generated power $(t-2)$ Nacelle temperature (t) Cooler fan slow run $(t-2)$ Cooler fan fast run $(t-2)$

Table A.1: The various input and output combinations used by Garcia et. al in [20].

In a different study [18] Schlechtingen and Santos investigated the usage of adaptive neuro-fuzzy interface system (ANFIS) as a residual-based model. ANFIS aims to combine the easy interpretation of a fuzzy inference system with the learning capabilities of a neural network. In short, the model learns how to assign membership functions to the input and how to construct rules given a membership. For a more detailed explanation on fuzzy design and ANFIS see [47]. For each of 18 WTs, the authors constructed 45 models using different combinations of SCADA signals. Inputs to output were chosen with the help of genetic algorithms and expert knowledge. The models were trained on nine months of data. SLFNs with the same inputs and outputs were also created and the reconstruction errors were similar. Compared to ANFIS's, however, SLFNs have a relatively long learning time which becomes important when many models have to be trained.

Next, the authors proposed a fuzzy inference system that, based on expert knowledge, could correlate the reconstruction errors to faults in various subsystems of the WT. In a second paper [19] the whole system was tested on two and a half years of additional data from the same 18 WTs. The system was able to detect a variety of anomalies including a hydraulic oil leakage, cooling system filter pollutions, converter fan malfunctions, anemometer offsets and a turbine performance decrease. For many of these cases service was performed months after and related components were replaced.

Bangalore et. al [23] created five SLFNs with time-delayed inputs to

model five different gearbox bearing temperatures. It was found that the size of the reconstruction error was dependent on the actual, operating temperature of the monitored component. To take this into account, their covariance matrix was calculated on the training dataset and the Mahalanobis distance (MD) with respect to this was used for a threshold-based alarm system. The models were trained on two different WTs using data from one year of operation where there were no logged faults. During the following year, a bearing fault happened for one of the WTs while the other remained healthy. For the healthy WT none of the models gave any alarm but for the one with a fault all of the models gave alarms around three months before the replacement and one week before a warning issued by a vibration based CMS. Further, the model corresponding to the faulty bearing produced the largest error which indicates that it could be possible to detect which bearing that was damaged. Next, the authors showed that the MD gave a much better signal to noise ratio than the typical root-mean-square error.

In a comparative study from 2015, Cross et. al [14] compared linear models, SLFNs and so-called state-dependent parameter (SDP) models. The SDP can be seen as a linear model in the inputs but for which the coefficients are themselves functions of the inputs (state). To get the coefficient-functions the authors used a Kalman filter and a fixed-interval smoothing algorithm. They used data from 26 turbines and 16 months and modeled gearbox bearing temperature with old temperatures and one or two extra SCADA-parameters. For the models with the single extra input the SLFN gave the best fit on the validation set, but, for the ones with two extra inputs, the SDP gave better results which was interpreted as that the SLFN had overfitted on the data. They applied their SDP with two extra inputs to a turbine which had shown a gearbox fault and used fuzzy rules to establish when the residuals had broken a threshold for too long. With this, they were able to give an alarm 20 hours before the fault occurred.

In a different comparative study, Watson et. al [16] compared linear models, linear models which also had terms with products of the inputs, SLFNs, Layer recurrent neural networks in which the output of layers was sent back as input to include inertial effects, GPR, SVMs and ANFIS. To select the best inputs they used crosscorrelation and also included lag of the different signals. They did not include previous values of the output in order to be able to predict faults. Bearing temperature and generator wind temperature was modeled for 100 WTs individually in a wind farm in the US with 52 days of data. Next, the models were tested on a remaining four months of data and compared by how well they were able to predict the temperatures. They found that most models performed better the more inputs that were used except for the ANFIS and the GPR. Most models also gave good precision with mean absolute errors around 2 °C. The SVMs and GPRs were, however, often outperformed.

Qian et. al [24] applied the extreme learning machine (ELM) in com-

bination with a genetic algorithm to set up a SLFN for various gearbox parameters. The ELM works by randomly initializing the input weights of the network and then solving for the best weights in the output using a matrix method. This is much faster than gradient descent based training agorithms and has been proven to have universal approximation capability just as normally trained SLFNs. Since the training is so fast multiple trainings can be tested using different input weights. The authors took advantage of this and applied a genetic algorithm to search for good input weights and biases. They modeled three different gearbox temperatures and trained the models with SCADA data from one year of data from one WT which later had a gearbox failure. The Mahalanobis distance was used as a measure of the size of the residual and the models were combined to give a better fault detection. The fault was detected in advance and the warning time was dependent on which threshold sensitivity that was chosen since it was seen that the residual errors started rising around 400 hours before the fault occurred.

Wang et.al [12] compared a DFN to kNN, two linear models, SVM and SLFN to model gearbox lubricant pressure in order to detect gearbox failures. The authors argued that this would be better than modelling for gearbox temperature since the temperature is more strongly affected by ambient temperatures and that it, therefore, would be more difficult to detect faults. The DFN model used had three hidden layers with 100 nodes each and was trained using stochastic gradient descent and dropout to avoid overfitting. Six different wind farms were analysed and data from all healthy WTs in a farm were used to train the models for that farm. The training data ranged from 3641 to 76500 datapoints per farm, corresponding to approximately 1 - 22 WT-months per farm. The models were compared using mean absolute percentage errors (MAPE) on a validation set and it was seen that the DFN outperformed the others with a MAPE of 6.01%. The SLFN came closest with 7.13% and the others had around 9%, see table A.2. Next, an EWMA was set up as an alarm system and the DFN was run on all WTs in the farms. It successfully identified all gearbox failures for the WTs affected and produced no false alarms. A similar model was then created to model the gearbox temperature and this one showed one true alarm, missed four of the failures and produced three false alarms. This clearly indicates that the lubricant pressure is a more reflecting property.

A.2.3 Autoencoders

In [25] Wang and Zhang used deep autoencoders (DAE) to predict blade breakages. DAEs use more hidden layers than normal AEs, in this study five hidden layers were used. Ten neurons were used in the central layer, a result based on a preliminary PCA. For the other layers different configurations were tested and the one selected had hidden layers with 500, 250, 10, 250

Models	MAPE
Lasso	9.51
Ridge	9.48
kNN	9.01
SVM	9.76
SLFN	7.13
DFN	6.01

Table A.2: Mean absolute percentage reconstruction errors of the models in [12] on data from normal conditions.

and 500 nodes. The DAEs were first pretrained using restricted Boltzmann machines (RBM) and then trained to recreate all the 25 SCADA signals available. In a first study, the authors selected the data from all healthy WTs in a wind farm for six months. From this data, they created one DAE according to the procedure described above, one DAE without temperature parameters, one without the RBM pretraining and one shallow AE with one hidden code layer. These were all compared studying one healthy WT and one with an impending blade breakage. The squared euclidian norm, i.e. the mean square, of the reconstruction errors was used and the only model able to detect the blade breakage was the proposed DAE with pretraining and temperature parameters. Next, the authors looked at two months of data from two new wind farms where it was known that two blade breakages had occurred but not when nor for which turbines. The authors trained one DAE for each wind farm using the data from all WTs in that farm. Next, they used an EWMA to construct a threshold-based alarm system. Despite the danger of overfitting, the models detected both blade breakages and did not produce any false alarms.

Similarly to [7] and [25] Jiang et. al [26] used an autoencoder model for anomaly detection. Their model was a shallow, sliding-window, denoising autoencoder. Denoising means that, during training, the input data is corrupted and the AE is taught to decrypt the corrupted data. For this work, the authors applied masking noise, which randomly masks some of the input signals with a zero value during training. Hopefully, this prevents the AE from purely learning an identity mapping and instead it should extract more representative features. Denoising autoencoders are described further in Section 2.1.9.

Sliding-window means that not only one but several timestamps, i.e. a window of time were used. In particular, for most cases, they modeled the six most recent timestamps. As a basis for an alarm system, they used a version of the MD which is more robust to outliers. A threshold was set by estimating the distribution of the MD on training data. Initially, the authors used data from a generic, well-established WT benchmark model. This

benchmark models an offshore 5 MW WT and is able to simulate a variety of realistic malfunctions in different parts of the WT. In this work, six fifferent sensor faults and two different actuator faults were simulated and different autoencoder models were compared with PCA-based methods. The data was sampled at a high frequency, 0.125 s, and individual timestamps were labeled as either normal or faulty. The comparisons were done with Receiver-operating-charactheristic (ROC) plots, which are similar to the precision-recall curves described in Section 2.4 but instead of precision, *false positive rate* is used. More information on ROC plots can be found in e.g. [48]. It was found that the autoencoder models outperformed the PCA-based methods. Among the autoencoder models, the proposed method was better than both denoising autoencoders without the sliding window and sliding window autoencoders without the denoising. Additionally, the effect of some of the hyperparameters was investigated. In particular, it was found that the number of nodes in the hidden layer did not have a significant effect on performance. After this study with simulated data, the method was validated on a Mongolian wind farm with over 100 WTs with a rated capacity of 1.5 MW. The data was acquired at 30 s intervals and, with the help of service logs, individual timestamps could be labeled as either normal or faulty. The authors selected two different types of faults, each containing 2000 normal samples and 2000 fault samples. Training was done with 8000 normal samples and the comparisons were done using ROC-plots for both fault cases. Once again, the proposed sliding-window denoising autoencoder model showed the best performance with both high recall and low false positive rate.

A.3 Power curve monitoring

By comparing power curves of different WTs in a wind farm it is possible to detect underperforming WTs. This has mainly been used for anomaly detection [17], [15], for a single WT where one, just as for the previously discussed residual-based models, detects a change in the PC and concludes that a fault has happened. However, these methods could also be applied to detect WTs that have always been underperforming; this could be due to a fault that the WT had from birth or due to influence from the other WTs such as wake. Since the previously discussed residual-based models are trained on the normal data they cannot possibly detect these faults since they have always existed and would, therefore, be categorized as "normal".

Papatheou et. al [17] modeled power curves for 48 offshore WTs in the Lillgrund wind farm where the WTs are spaced particularly close. They developed two different types of models, one based on a SLFN, and one based on GPR. The measured wind speed at each turbine was used as input. The data was collected from one year of service and bad data was filtered with

the help of SCADA statuses. Two models (one SLFN and one GPR) was trained for each WT and using wind data of that WT. Next, the models were tested with the wind speed data from the *other* WTs. Overall all models performed well despite being fed with the data from different turbines. Next, the authors developed an alarm system based on extreme value statistics. The system was able to detect some anomalies which were verified by the SCADA status data, however, many statuses remained unnoticed.

Long et. al [15] investigated the shape of PCs to determine if a WT was malfunctioning. They argued that PCs may shift with factors such as air density and to simply compare it with reference PCs may not give an accurate picture. Instead, the authors took wind speed and power data from one WT and divided it into sequential time periods. For each time period, they fitted two models, one linear and one Weibull cumulative distribution function. Both of these models had two parameters and in a first phase a Hotelling T-squared control chart was used to detect extreme outliers amongst the parameters for different time periods. The authors argued that this method is effective at detecting data with irregular curvature but perhaps not abnormally widely spread data. For this, the authors used a residual approach where the sums of residuals for each time period were compared against each other. If the sum exceeded a threshold this time period was classified as abnormal. Next, the authors performed two blind studies on WTs in the U.S. and China. SCADA data sampled at 10 s interval from two months of operations was provided together with fault logs. The data was divided into 177 time periods based on a fixed interval and out of these 27 were, according to visual inspection, abnormally shaped. Out of these, the combined methods were able to detect 22 abnormal cases. Additionally, fives cases were detected that appeared normal during the visual inspection, but, according to the fault logs responded to cases where the WT had experienced various problems like emergency stops and brake malfunctions. For the second study SCADA data sampled at a 10 minute interval from three months and 32 WTs was provided without fault logs. Two of these WTs had, however, been manually investigated by experts and were analyzed closer by the authors. The methods were able to detect a number of abnormal PCs which all corresponded to anomalies detected by the manual inspections. The authors concluded that the proposed methods could be implemented for online monitoring of a WT but suffered from not being able to predict faults as timely as methods based on data from single timestamps.

Appendix B

Case studies

In this chapter, two particular anomalies are studied. The first anomaly was an increased hydraulic oil temperature which occurred after a preventive maintenance. Most likely, this happened because of a settings change which was not noted in the service logs. Even if this is not a fault per se, it is interesting to study as it clearly shows how the autoencoder is able to identify which signal is abnormal.

The second anomaly was a malfunction in the yaw encoder system lasting for about one month. The system was later replaced and it is clear how the reconstruction errors of the autoencoder model immediately decrease.

B.1 Change in hydraulic oil temperature

On the 5th of November 2018, a preventive maintenance was performed on a 2 MW wind turbine. The wind turbine appears to have been functioning well both before and after the maintenance. As is clear from Figure B.1, however, the maintenance caused a consistent rise in the hydraulic oil temperature. Figure B.1 shows the reconstruction of this temperature and one can notice how the autoencoder model fails to reconstruct the temperature after the maintenance as its behaviour is different from the behaviour during the training period.

While the reconstruction of the hydraulic oil temperature got worse after the maintenance, other signals were not as affected. This is clear from Figure B.2 where the reconstruction of a gear bearing temperature is shown. For this signal, as well as many other signals, the autoencoder was still able to give a good reconstruction. Consequently, the monitored Mahalanobis distance did not increase significantly. Since the MD is a measure of all the reconstruction errors combined, a slight change in only one of these errors does not affect it much. It can be observed from Figure B.3 where the MD along with its inferred EWMA are presented. For this reason, it might be beneficial to construct an alarm system based on all *individual*

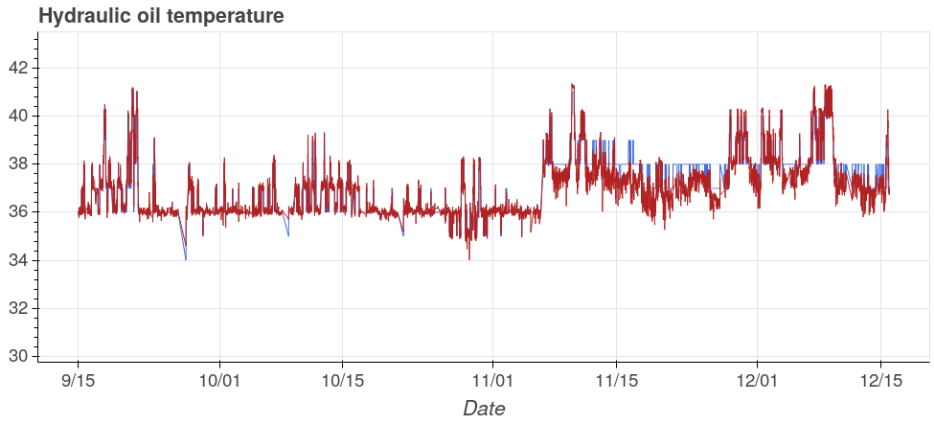

Figure B.1: Reconstruction of a turbine's hydraulic oil temperature, before and after a preventive maintenance on the 5th of November. The reconstructed signal is plotted in red and the original in blue. It is clear how the maintenance changed the temperature and how the reconstruction got worse.

reconstructions as opposed to combining them with something similar to the Mahalanobis distance, as discussed in Section 5.3.

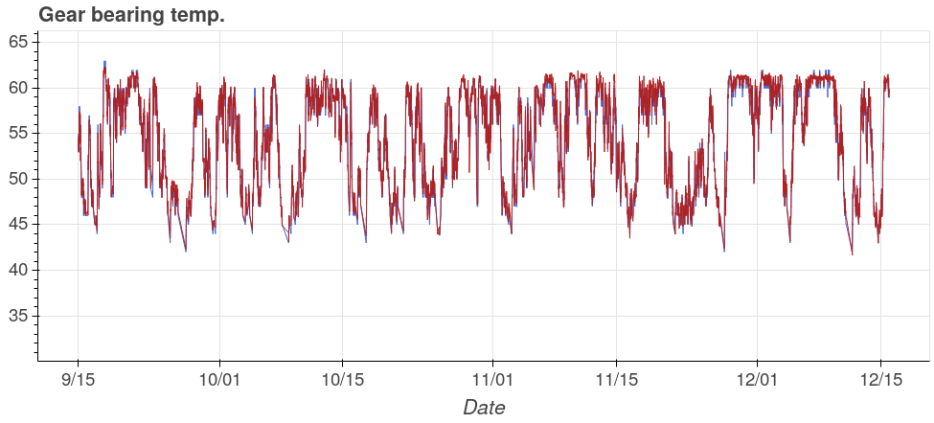

Figure B.2: Reconstruction of the same turbine's gear bearing temperature, before and after the preventive maintenance. The reconstructed signal is plotted in red and the original in blue. Clearly, the reconstruction error did not change by a lot for this signal.

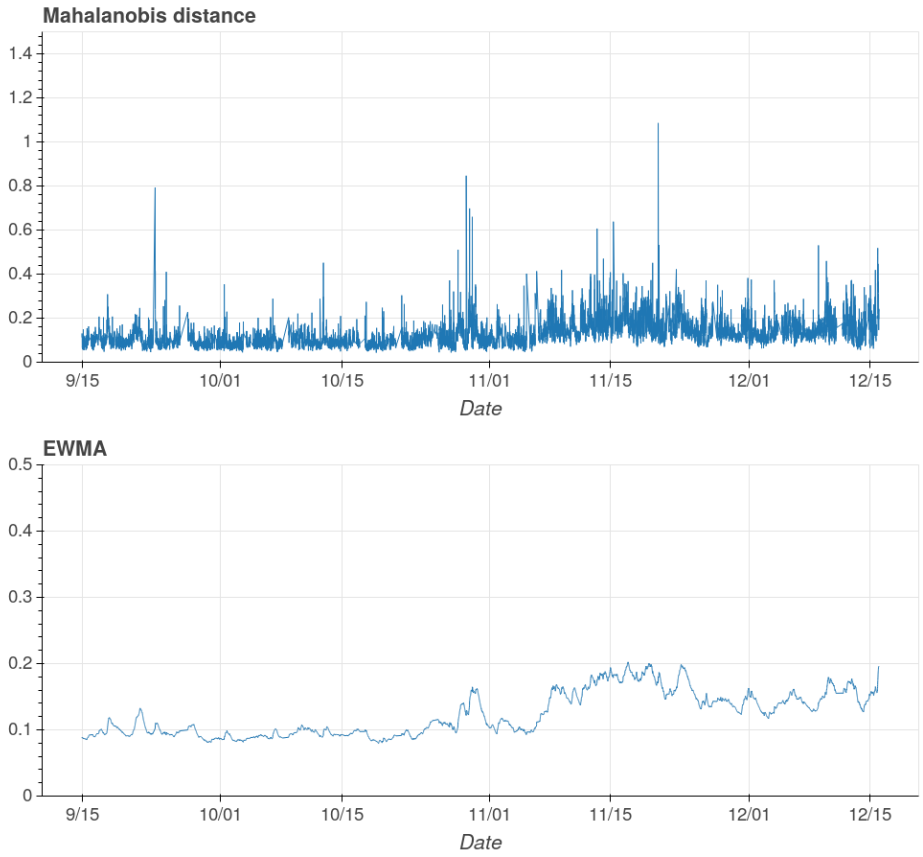

Figure B.3: The Mahalanobis distance along with its inferred exponentially weighted moving average from all the reconstructions of the different signals for the same time period as in Figure B.1. Only a slight increase is visible.

B.2 Yaw encoder malfunction

A different 2 MW turbine experienced a number of failures in the yaw encoder which were detected by the SCADA system beginning from the 15th of March 2018. On the 11th of April, the yaw encoder was replaced and the alarms seized. This fault was easily detected by the autoencoder models which showed errors as early as the 2nd of March. As is shown in Figure B.4, the errors went back down after the replacement.

The yaw encoder system determines how the turbine should turn to catch most of the wind. If this is malfunctioning, power production will be highly affected. This was caught by the autoencoder model, which had its largest reconstruction errors for the power signal and the blade angle signal. For other signals, e.g. a gear bearing temperature, the reconstruction errors were much lower. All of these reconstructions are shown in Figure B.5, were, for increased visibility, a shorter timespan than in Figure B.4 is covered. This

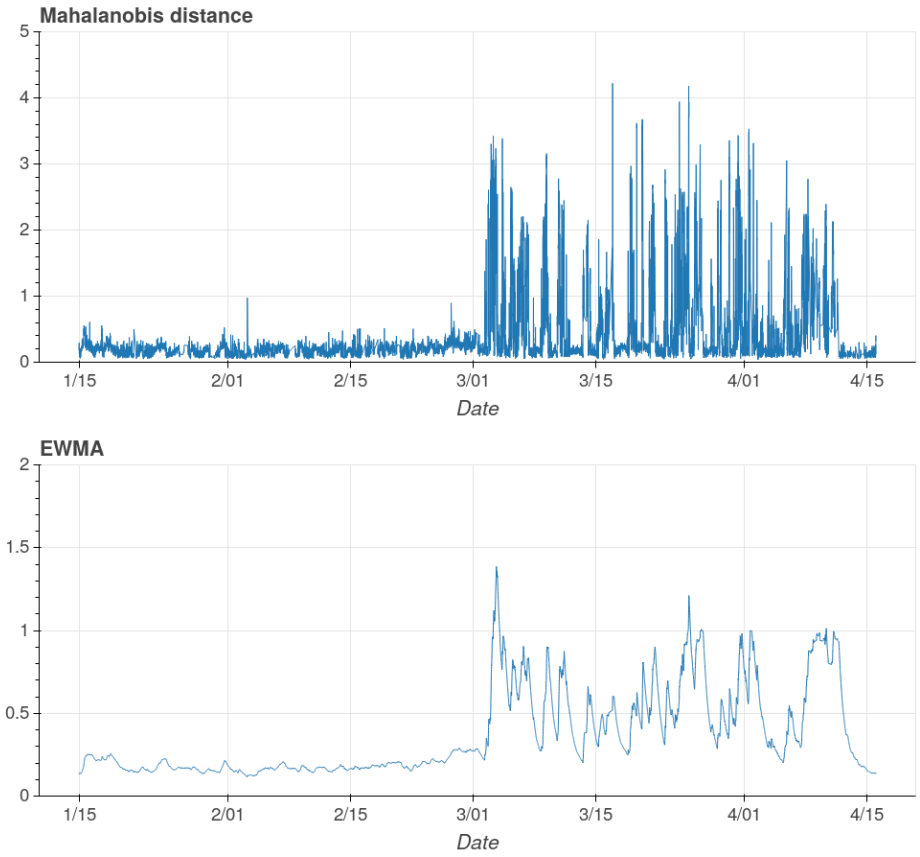

Figure B.4: Mahalanobis distance and exponentially weighted moving average of the reconstruction errors for a wind turbine experiencing faults in the yaw encoder system. It is clear when the faults start occurring and when the yaw encoder system is replaced.

behaviour suggests that the autoencoder model could be used to indicate *where* a fault is, in addition to just saying *when* it occurred and this is also discussed further in Section 5.3.

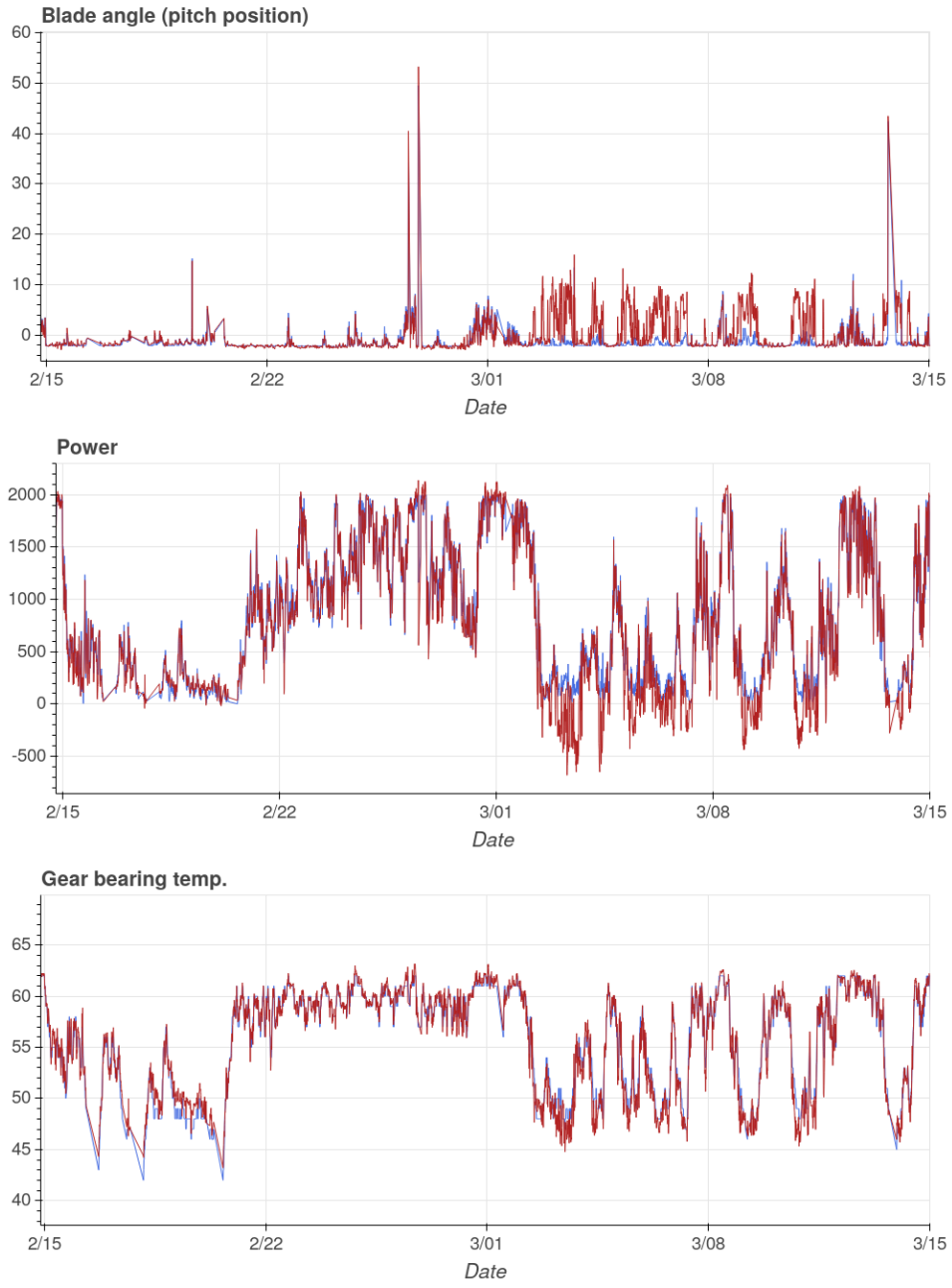

Figure B.5: Reconstructed (red) and original values (blue) for some different signals during the time of the yaw encoder malfunction. It is clear how the signals related to the yaw system (power and blade angle) are highly affected while the unrelated gear bearing temperature does not show any indication of a fault.