Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Jo Aleksander Johansen

Techniques for signal reconstruction from sparsely compressed vibration measurements

Master's thesis in Cybernetics and Robotics
Supervisor: Frank Ove Westad
Co-supervisor: Johannes Holm Gjeraker
June 2022

NTNU
Norwegian University of
Science and Technology

Jo Aleksander Johansen

Techniques for signal reconstruction from sparsely compressed vibration measurements

Master's thesis in Cybernetics and Robotics
Supervisor: Frank Ove Westad
Co-supervisor: Johannes Holm Gjeraker
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Amidst the mist, ignorance can be persuasive

-unknown

Abstract

As the world advances, measuring our surroundings becomes increasingly important. As the wireless sensors become smaller, they are utilized more places. The information has a wide range of uses, from monitoring the health of a population to predicting failures in machinery. This thesis explores how to efficiently capture and reconstruct data from wireless Internet-of-Things (IoT)-sensors. Lowering the transmitted data size will increase the longevity of the sensor itself whilst still providing accurate measurements.

The context of the compression is maintenance prediction on bearings using vibration sensors from Disruptive Technologies (DT). Time-series from vibration measurements are used to test multiple techniques for compression and reconstruction. Currently, DT samples a signal and only selects the 5 frequencies of the highest magnitude to be transmitted. Their method was recreated and compared to three other methods; autoencoders, decoder networks, and compressed sensing. Multiple different configurations were tested for each method. Autoencoders are trained to find some small set of latent features in the data, then reconstruct the original signal from this set of features. The results from this were not adequate for any predictions. Decoder networks were tested with a random subset of samples as input to recreate the whole signal. This also showed underwhelming results.

Two solvers were tested for compressed sensing; Embedded Conic Solver (ECOS) and Orthant-Wise Limited-memory Quasi-Newton (OWL-QN). ECOS was deemed too slow to be viable, but OWL-QN showed good results. Both prediction algorithms discussed might be able to predict with the recreated signals when the sample size is between 10% and 50% of the whole sequence. The Root-Mean-Square (RMS) over time is surprisingly accurate for a sample size as low as 1%.

Sammendrag

Verden er i stadig digitalisering og vi lener oss mer og mer på teknologi for å fortsette utviklingen av omverdenen. En stor del av dette innebærer å samle informasjon. Ettersom trådløse sensorer blir mindre plasseres de flere steder. Informasjonen de samler brukes til alt fra å monitorere menneskers helse til å forutse feil i industrielt utstyr.

I denne oppgaven diskuteres kompresjons- og rekonstruksjons-metoder i kontekst av å predikere feil på kulelager. Vibrasjonsmålinger i form av 1 sekund lange sekvenser med tidsserie blir brukt til å teste 4 forskjellige metoder. Ønsket er å holde informasjonen som sendes til et minimum og holde prossesringen på en sensor lav. Den første metoden brukes av Disruptive Technologies (DT) i en prototype. De taster i 1 sekund med en tastetid på 1100Hz før de fem frekvensene med størst magnitude blir valgt og sendt. Dette sammenliknes med tre andre metoder; autoenkodernettverk, dekodernettverk og komprimert sansing.

Autoenkoderen og dekoderen er begge testet med tre forskjellige arkitekturer som hver endrer dybden og kompresjonen de oppnår. Ingen av akitekturene oppnår resultater som kan brukes til predikering i noen av de to metodene. Det er vanskelig å peke på en grunn til dette da spesielt autoenkodere har vist gode resultater i flere publikasjoner.

Komprimert sansing går ut på å taste et signal tilfeldig, altså med varierende tasteperiode, for så å rekonstruere signalet med konveks optimalisering. To løsere ble testet; Embedded Conic Solver (ECOS) og Orthant-Wise Limited-memory Quasi-Newton (OWL-QN). ECOS viste seg å være for treg til å brukes med så store datamengder. Noen få sekvenser ble testet før den ble vurdert som ubrukelig. OWL-QN ble så testet og gav gode resultater. De rekonstruerte signalene med taste størrelserstørre enn 10% har en spektral omhylling som viser sterke likheter med det opprinnelige signalets spektrale omhylling. Det gjør komprimert sansing mulig å bruke i mer avanserte predikasjonsalgoritmer. En enkel måte å predikere feil på er å forutse Root-Mean-Square (RMS) på fremtidige sekvenser. Testene viser at en tastestørrelse på bare 1% gir en RMS-kurve som utvikler seg likt med det opprinnelige signalets RMS-kurve. Det ser derfor ut til at denne predikasjonsmetoden er effektiv med tastestørrelse helt ned i 1%.

Hvorfor komprimert sansing fungerer så mye bedre enn de andre metodene er ikke utforsket i denne oppgaven. Noen betraktninger bør allikevel gis. For å predikere noe må man ha nok av den riktige informasjonen tilstedet. DT sin metode

gir, etter alt å dømme, for lite informasjon. Autoenkoderen og dekoderen prøver å rekonstruere utifra dataene de er gitt, det kan se ut som metodene ikke bringer mer informasjon til bordet og da blir det for lite å basere predikasjoner på. Det er her man kan spekulere i om forskjellene ligger. Komprimert sansing prøver å optimalisere seg frem til mer informasjon, informasjon som matcher det den alt er gitt. Hvis rekonstruksjonen stemmer bedre overens med det opprinnelige signalet vil, ikke overaskende, en tilsvarende prediksjon også være mulig.

Preface

My interest in learning, exploring, and making things has led my path throughout my life. Eventually, it brought me to an engineering degree. Looking back to the start of this 5-year long journey, I must admit I did not comprehend what was coming. The amount of work required to get through all the assignments, the lab exercises, and the accompanying discussions late at night to grasp the abstract theories has been overwhelming, to say the least. Equally overwhelming is the joy of understanding those abstract theories, finding solutions, or having made something. Equally powerful are the friendships, not only forged by the tears of not knowing but also by knowing.

I am grateful for everyone that I have befriended through studies, group work, and too many a beer. Our conversations have been helpful, inspiring, and educational, both personally and at the academic level.

Working with this master thesis forced me to deep dive into the intricacies of optimization from a math-oriented point of view. Although much of the theory was taught in classes, I cannot say I had a good understanding of what it all meant. Nor good knowledge of how different subjects are connected. The exploration I carried out this final semester has significantly developed a mathematical intuition for multiple concepts in optimization and related domains. For that, I am grateful.

I want to thank my supervisor Frank Ove Westad for challenging my experiments and my way of thought. Your free reins approach to my thesis subject is something I value. I hope my work is deserving of the trust you have given me.

I thank my other supervisor Johannes Holm Gjeraker from Disruptive Technologies. Multiple times you have listened to my, not always so coherent, line of thought as I found my way through this task. Your straightforwardness and attention to detail during discussions and proofreading are much appreciated.

I thank Bendik Austnes for proofreading. Without your moral support through the years, our discussions, and our numerous batches of brews, the thesis I am about to deliver might not have existed.

Finally, I thank my family and the people closest to me. I always get deeply engrossed in the things I do, and your patience with me has not gone unnoticed.

Contents

Figures

Tables

Acronyms

ADAM Adaptive Moment Estimation. 18, 19

AM Amplitude Modulated. 33

AR Auto Regressive. 7, 33

ARIMA Auto Regressive Integrating Moving Average. 8

ARMA Auto Regressive Moving Average. 7, 8, 33

BFGS Broyden–Fletcher–Goldfarb–Shanno. 30, 31, 44, 45

CFD Computational Fluid Dynamics. 34

CNN Convolutional Neural Network. xiii, 21–23, 34

CPU Central Processing Unit. 37

CUDA Compute Unified Device Architecture. 37

DCT Discrete Cosine Transform. 3, 45

DL Deep Learning. 14, 19

DNN Deep Neural Network. 34

DT Disruptive Technologies. v, vii, xii–xiv, 1, 37, 38, 41, 44, 48, 52, 54, 55, 59–61, 67, 68

DWT Discrete Wavelet Transform. 34

ECOS Embedded Conic Solver. v, vii, xiv, 29, 30, 44, 50, 62, 67

EWT Empirical Wavelet Transform. 33

FCNN Fully Connected Neural Network. 20, 21, 23, 25, 33

FFT Fast Fourier Transform. 38, 41, 42, 68

FM Frequency Modulated. 33

FT Fourier Transform. 3, 4

GP Gaussian Process. 13

GPR Gaussian Process Regression. xiii, xxii, 1, 13, 14, 34, 59–62, 67

GPU Graphical Processing Unit. 37

GRU Gated Recurrent Unite. 24

HT Hilbert Transform. 3, 4, 33

IMS Intelligent Maintenance Systems. 38

IoT Internet-of-Things. v, 59, 67, 68

KL Divergence Kullback-Leibler Divergence. 18

kNN k-nearest neighbors. xiii, 9, 10, 33

L-BFGS Limited Memory BFGS. 30, 31

LR linear regression. xiii, 12, 34

LS Least Square. xiii, 6, 12

LSTM Long-Short Term Memory. xiii, 24

MA Moving Average. 7, 19

MEMS Micro-Electro-Mechanical Systems. 38

ML Machine Learning. 8, 9, 13, 14, 18, 27, 29, 31, 34, 59

MLE Maximum Likelihood Estimation. 6

MPN Multilayer Perceptron Network. 34

MRI Magnetic Resonance Imaging. 26

MSE Mean Square Error. xiv, 42, 44, 46, 54–58, 60–62

NN Neural Network. xxii, 13, 14, 18, 19, 29, 31, 33, 34, 44, 59, 61–63

OWL-QN Orthant-Wise Limited-memory Quasi-Newton. v, vii, xiv, 31, 45, 50, 51, 53, 62, 67, 68

PC Principal Component. 5, 6, 9, 25, 63

PCA Principal Component Analysis. 5, 6, 9, 33, 61, 63

PDF Probability Density Function. xxii, 6, 7, 33

RAM Random Access Memory. 37

RBF Radial Basis Function. xiii, 11, 13

ReLU Rectified Linear Unit. 19, 20, 42

ResNet Residual Neural Network. 23, 34

RIP The Restricted Isometry Property. 32

RMS Root-Mean-Square. v, vii, xiv, 1, 34, 46, 52, 53, 59–62, 67

RNN Recurrent Neural Network. 24, 25, 34

RPCA Recursive PCA. 33

RPM Revolutions Per Minute. 38

SGD Stochastic Gradient Descent. 15, 18, 19

SVM Support Vector Machine. xiii, 11, 12, 33, 34

WNN Wavelet Neural Network. 33

Glossary

conjugate Swapping of the sign. Often used with complex numbers where ex. $2+i3$ has the conjugate $2-i3$ [1]. 5

convex set A set S is convex if the straight line connecting any two points in the set lies inside S [2]. 29

entity embedding When working with categorical data, one would usually one-hot encodes them. But another way of representing categorical data is *entity embedding*, where items with higher relation can be placed closer together in the embedded space. Further information are given in the original paper by Guo and Berkhahn [3]. 34

feasible set A set of points that satisfies the constraints given in optimization problems [2]. 27, 29

flatten The process of manipulating a matrix into an array. 21

gaussian noise Statistical noise adhering to the normal distribution described by $N(\mu, \sigma)$, [4]. 9

generalizability A models ability to cover variance it has not seen before; generalizability is high if the model has high accuracy on data it has not seen before [5]. xxii, 16

gradient Taking the derivative of a function in n-space will generate a hyperplane, this is the *graident* of the function. The gradient is used to find local- and absolute minima in optimization problems [1]. 14, 15, 17

homotopy Two functions that can be continuously transformed into each other are called *homotopic* [6, 7]. 30

hyperplane A line and a plane are easy to visualize in 3d space, but planes in a higher dimension is an abstract concept hard (if not impossible) to visualize. These higher dimensional planes are called hyperplanes [1]. 11, 14

isometry When mapping a set of points from one space to another, the mapping is an isometry if the inter-point distances (shape and size) remain the same in the new space [8]. 32

kernel trick A set of data that is not linearly separable might become linearly separable if mapped to a higher dimension, this mapping and splitting is called *the kernel trick* and enables effective classification of the data points [9]. 11

Kriging Another word for Gaussian Process Regression (GPR), based on the name of a pioneer in the field of *geostatistics*[10]. 13

kurtosis A measure of the "tailedness" of a PDF [11]. 33

learning rate When optimizing a Neural Network over its gradient the optimizer has to take steps, the "length" of the step is called *learning rate*. 18

Nyquist-Shannon sampling theorem To accurately measure a signal one must sample two times faster than the highest frequency that is measured. 38

orthant A generalization of quadrants and octants into n-dimentional Euclidian spaces [12]. 31

periodic If a signal repeats it self, in a finite time space or to infinity, it is called periodic for the given time span [9]. 3

regularization Any method that avoids overfitting is a regularization technique. Regularization enhances generalizability [5]. 10, 16

skewness A measure of asymmetry of a PDF [11]. 33

spectral envelope A smooth curve defined by the upper boundaries of the spectral density of a time-series. Can be thought of as an enclosing function on the spectral density of a signal [13]. 1, 45, 60–62, 67, 68

stationary Stationarity is used to describe a statistical property of system generating a signal. A stationary signal has constant mean and variance [9]. 7–9

super position For a linear system, the principle of superposition says that the net response from multiple stimuli is equal to the sum of each individual stimuli [1]. 3

Chapter 1

Introduction

Bringing small and cheap sensors to end-users in households, office buildings and industries might be essential to further modernize the world. At the same time, current wireless IoT sensors have the flaw of draining the batteries in a relatively short amount of time. At large scales, frequent battery changes in sensors impose a large amount of waste on nature. The staff needed to change all these batteries must also be paid, thus limiting the economical value of the whole system.

To enable such large quantities of sensors to report important and detailed information for years upon years without the need for battery change can be looked upon as the pinnacle of sensor technology. Therefore it is a need for optimized hardware as well as processing that uses cutting edge techniques to limit the power consumption. The theme of this paper will be signal reconstruction. The focus will be on limiting the content of wireless transmissions as well as limiting the processing required on the sensor. It is building upon already optimized hardware from Disruptive Technologies (DT) in the form of a prototype vibration sensor. Also, the project will be built upon the work done by myself in my project thesis [14]. Some interesting techniques discussed in that paper will be considered with regards to the results in this paper.

Multiple methods of reconstruction will be implemented, tested, and compared; the original method implemented on the DT sensors, auto-encoders, a decoder network and compressed sensing. The results from the reconstructions will be evaluated and a discussion of how the loss will impact maintenance prediction algorithms will be given. There are two maintenance algorithms of interest. (1) A method by 'Wavelet filter-based weak signature detection method and its application on rolling element bearing prognostics' that tries to enhance weak signatuers of a failure in a signal. The accuracy of the recreated spectral envelope is essential for such an approach to work. (2) Predicting the n next Root-Mean-Square (RMS) values using Gaussian Process Regression (GPR), a method presented by Hong and Zhou [16]. Implementing and testing the said prediction algorithms is not in the scope of this thesis.

In chapter 2 the thesis starts by presenting the theory needed to understand the previous work and the methods tested and discussed. Some previous work on both

maintenance prediction algorithms and compression algorithms will be presented in chapter 3 as well as some reasoning as to why the methods tested is chosen. Then a description of the methods used is given in chapter 4 before the results are presented in chapter 5. Some discussion takes place in chapter 6. Comments on possible work to be executed going forward are mentioned in chapter 7, before a conclusion is given in chapter 8.

Chapter 2

Theory

2.1 Transforms

2.1.1 Fourier Transform

Given a periodic signal, the signal can be decomposed into a sum of sines and cosines with a limited set of frequencies ω. The technique for doing so is called a Fourier Transform (FT), essentially mapping the signal form the time domain to the frequency domain. Mathematically it is described as

$$\hat{f}(\omega) = \mathcal{F}(f(x)) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x} dx \tag{2.1a}$$

$$f(x) = \mathcal{F}^{-1}(\hat{f}(\omega)) = \int_{-\infty}^{\infty} \hat{f}(\omega) e^{-i\omega x} d\omega \tag{2.1b}$$

where (2.1a) is the transformation from the time domain to the frequency domain, and (2.1b) is the inverse transform [17].

The FT is a cornerstone to digital signal processing and compression. JPEG use a slightly modified version of FT called Discrete Cosine Transform (DCT) to decompose a picture and remove weights that are negligible before reconstructing the image [18]. DCT is essentially Fourier series with only cosines.

2.1.2 Hilbert Transform

Although the Hilbert Transform (HT) and FT are closely related, they differ in the assumptions of the signal. FT assumes the incoming signal as a super position of sines and cosines. The HT, on the other hand, demodulates the signal only using a single sine which is modulated. Feldman [19] states that vibration signals are "exactly of that model", which is convenient for the topic of this paper.

The HT is a phase-shifted transform which has a linear relation to FT. For a

signal, u, it is given by

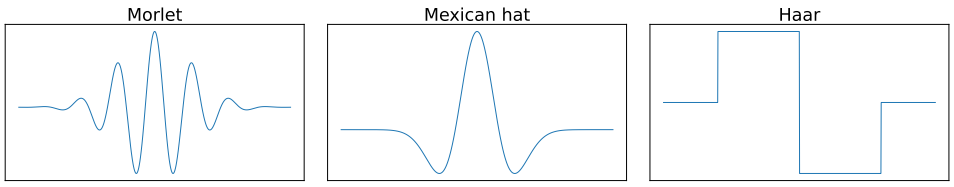
$$\mathcal{F}(H(u)) = \sigma_H \mathcal{F}(u) \tag{2.2a}$$

$$\sigma = \begin{cases} i, & x < 0 \\ 0, & x = 0 \\ -i, & x > 0 \end{cases} \tag{2.2b}$$

where $\mathcal{F}$ is the FT and H is the HT. Thus a way of calculating the HT is to (1) Fourier Transform the signal, (2) shift the phase by 90° then (3) inverse FT [19].

2.1.3 Wavelet Transforms

Wavelet transform is a generalization of the Fourier Transform described in subsection 2.1.1, letting functions other than sines and cosines represent the input signal. These functions are called wavelets, hence the name, and can be scaled and translated to form a set of wavelets. As the wavelets are convolved with the input, one can visualize the transforms as a filter, the wavelet, moved across the input. This generates a set of corresponding weights. A chosen wavelet function is called the *mother wavelet* [17]. One highly used and known wavelet is the *morlet wavelet*, but others are frequently used as well. There is no scientific way to determine if one wavelet is better than another; it is a matter of trial and error as it also depends on the input signal [20]. An illustration of some common wavelets, including morlet, is shown in Figure 2.1, and a visualization of the transformation is shown in Figure 2.2.



(a) **(b)** **(c)**

Figure 2.1: Figure 2.1a show the morlet wavelet, probably the most known of the three. In Figure 2.1b, a "Mexican hat" wavelet is shown, also called a Ricket wavelet. At last, a wavelet named after one of the first persons to explore wavelets; the Haar wavelet

As described by Brunton and Kutz [17], the wavelets are given by

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}\psi(\frac{t-b}{a})}, \tag{2.3}$$

where a and b scales and translates the wavelet. The transform is then described as

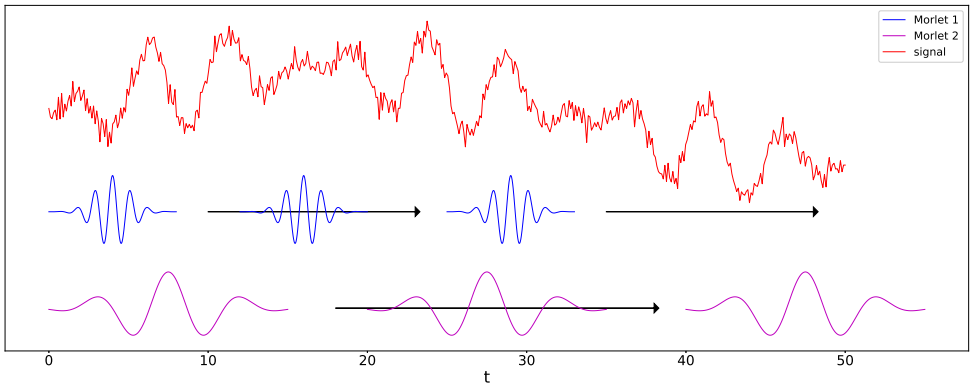


Figure 2.2: The *mother wavelet* is scaled multiple times to form a set of wavelets that are convolved with the input signal. Such operations can be thought of as filters sweeping over the input as illustrated with the arrows, thus generating a set of weights. These weights combined with the wavelets form a decomposed representation of the data.

$$\mathcal{W}_\psi(f)(a,b) = \langle f, \psi_{a,b} \rangle = \int_{-\infty}^{\infty} f(t)\bar{\psi}_{a,b}dt, \tag{2.4}$$

where $\bar{\psi}_{a,b}$ is the conjugate of $\psi_{a,b}$. It's inverse is

$$f(t) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathcal{W}_\psi(f)(a,b)\psi_{a,b}(t)\frac{1}{a^2}dadb. \tag{2.5}$$

2.1.4 Principal Component Analysis

Processing multidimensional data can be computationally expensive and the results hard to interpret. With many variables in a dataset, it might be hard to know if some are highly correlated or the significance or their effect on a measured response. A common method for analyzing such data, finding the correlated or insignificant factors, is called Principal Component Analysis (PCA).

PCA assumes factors of high variance are more important to the output than factors of low variance. Thus the method will generate a new basis for the data, where the axes maximize the variance. The first Principal Component, PC1, maximizes the variance without constraints and the following n PCs have to be orthogonal to the previous PCs. By adding one more Principal Component, more of the variance in the original data is captured/explained [21].

This leads to multiple uses, where the main ones are; looking for some hidden relationships by analyzing on the new basis, performing regression on the new basis, and dimensional reduction [21]. If 5 PCs can explain 90% of the variance in an $n = 20$-dimensional dataset, further processing or modeling might be more effective due to the reduced set of dimensions.

Given a set of data $\boldsymbol{X}$ where the samples are the row vectors the data must be normalized before executing Principal Component Analysis [17]. The normalized data, $\boldsymbol{B}$, is calculated by

$$\boldsymbol{B} = \boldsymbol{X} - \bar{\boldsymbol{X}}, \tag{2.6}$$

where $\bar{\boldsymbol{X}}$ is the mean of the set. With $\boldsymbol{B}$, Brunton and Kutz [17] describe the PCA process by first calculating the covariance matrix of the rows by

$$\boldsymbol{C} = \frac{1}{n-1}\boldsymbol{B} \circledast \boldsymbol{B} \tag{2.7}$$

and so the first PC, $\boldsymbol{u}_1$, is given as

$$\boldsymbol{u}_1 = \arg\max_{||u_1||=1} \boldsymbol{u}_1 \circledast \boldsymbol{B} \circledast \boldsymbol{B}\boldsymbol{u}_1. \tag{2.8}$$

2.2 Statistical Prediction

2.2.1 Weibull Distribution

According to Walpole *et al.* [4] the Weibull distribution has been popular among engineers in the last decade to describe and predict the lifetime of components. The distribution is defined as

$$f(x;\alpha;\beta) = \begin{cases} \alpha\beta x^{\beta-1} e^{-\alpha x^{\beta}}, & x > 0 \\ 0, & \text{elsewhere} \end{cases} \tag{2.9a}$$

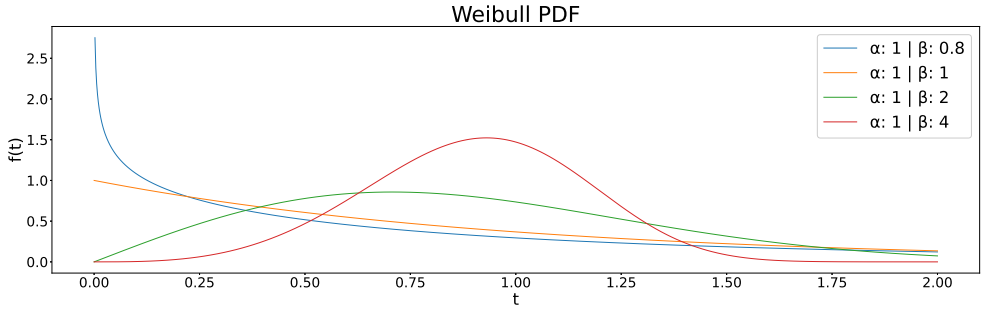
$$F(x) = 1 - e^{-\alpha x^{\beta}}, \qquad x > 0 \tag{2.9b}$$

$$Z(t) = \frac{f(t)}{1-F(t)} \tag{2.9c}$$

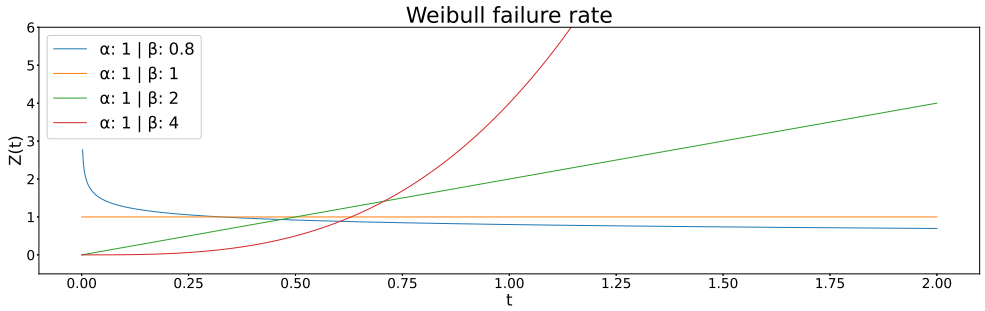
$$= \alpha\beta t^{\beta-1}, \qquad t > 0 \tag{2.9d}$$

where (2.9a) is the density function, (2.9b) is the cumulative distribution function and (2.9d) is the failure rate, also called the hazard function. β is the shape parameter and α is the scale parameter.

The Probability Density Function (PDF) is plotted in Figure 2.3a with different β values. One reason for the popularity of the Weibull distribution is its flexibility; as can be seen from Figure 2.3b when α is constant and $\beta = 1$ it becomes memoryless, meaning the probability of something happening at time t_1 given it survived to t_0 is the same as the probability for surviving to t_1. This is not the case when $\beta \neq 1$. From the plot in 2.3b it is clear that $\beta < 1$ makes an event less probable over time, while $\beta > 1$ makes an event more probable over time. Because of this flexibility, the distribution parameters can be estimated to fit a wider range of systems using, for example, the Maximum Likelihood Estimation (MLE) or Least Square (LS) method [22].



(a) PDF for the Weibull distribution, using different values for β. As the x-axis is time the PDF describes the probability of an event after the time t. The event might be an electrical component or a mechanical part where the distribution explains the probability of a failure within a certain time.



(b) The failure rate for the Weibull distribution, using different values for β. From this plot, it should be clear that Weibull distribution can represent both memoryless systems, and systems of wear or strengthening with respect to time.

Figure 2.3: Weibull distribution

2.2.2 ARMA & ARIMA Models

Auto Regressive Moving Average (ARMA) and its perturbations model a time-series based on its previous steps. The assumptions for ARMA to work are (1) the system generating the signal, say X, is stationary, and (2) there is a linear relationship between some previous step(s) and the next. Before looking at mathematics, let us review the name. Auto Regressive (AR) tells us that the model assumes a relationship with p previous steps with *auto* meaning *self* and *regression* is the technique of fitting a line to some points. Mathematically, it can be formulated as

$$\hat{x}_{t+1} = x_t + \alpha_p x_{t-p}, \tag{2.10}$$

where $\hat{x}_{t+1}$ is the next predicted value, α_p are the regression coefficients, and x_{t-p} are the steps previously observed [9].

Moving Average (MA) is an averaging of the errors that move along the time-series data. MA uses the q previous steps to compute the average at any point in

time, thus

$$\hat{\epsilon} = \frac{1}{q}\sum_{i=-q}^{t}(x_i - \hat{x_i}). \tag{2.11}$$

Knowing this, it should make sense to call this method ARMA(p, q). Combining Equation 2.10 and Equation 2.11 gives

$$x_{t+1} = x_t + \alpha_p x_{t-p} + \beta_q \hat{\epsilon}_{t-q} + \epsilon, \tag{2.12}$$

where x_{t+1} is the accurate prediction of the next step, since ϵ is the error. ϵ, however, is unknown, thus it is removed to give the estimate of the next value. The final formulation is

$$\hat{x}_{t+1} = x_t + \alpha_p x_{t-p} + \beta_q \hat{\epsilon}_{t-q}. \tag{2.13}$$

[9]

Auto Regressive Integrating Moving Average (ARIMA) is only different from ARMA in that it only requires a system with stationary variance. The changing mean is first compensated for by the integration part, moving the time-series X to a time-series Z. The ARMA model is now predicting the new time-series Z which *is stationary*. ARIMA models are also noted as ARIMA(p, d, q), where d is the number of integrations. Mathematically, the new time-series for $d = 1$ is described as

$$z_t = x_{t+1} - x_t, \tag{2.14}$$

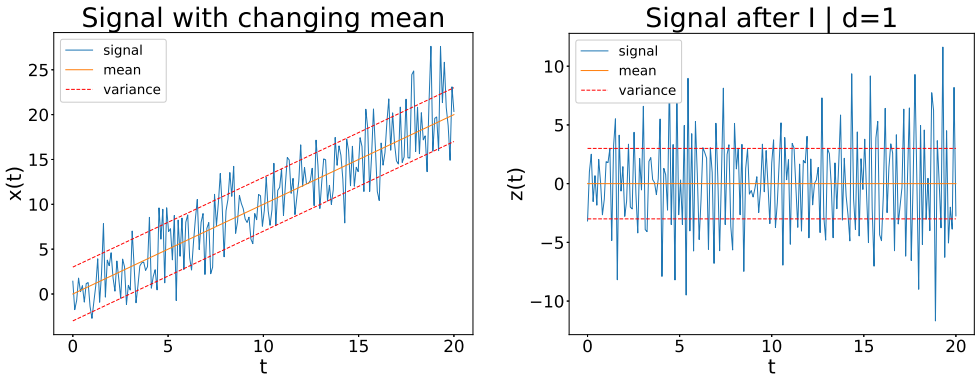
and for $d = 2$ the procedure is done once more, thus

$$w_t = z_{t+1} - z_t. \tag{2.15}$$

The pattern continues for $d = n$ where $n \in 0, N$. An illustration of a time-series before and after the procedure described in Equation 2.15 is shown in Figure 2.4 [23].

2.3 Machine Learning and Deep Neural Networks

Since its first successful deployment by Samuel [24] in the 1950-s, Machine Learning (ML) has been subject to extensive research. The early models used manmade algorithms for prediction and classification, which led to huge advancements in automation in industries as well as in everyday lives. Modern ML uses optimized hardware to process models with incredible complexity. Where traditional methods were explainable, modern models prove extremely hard to comprehend by humans. These models now control an ever-increasing portfolio of problems spanning everything from facial identification for security purposes to the regulation and control of big oil platforms. A rundown of different machine learning principles, new and traditional, and techniques are given below.



(a) **(b)**

Figure 2.4: Figure 2.4a show a signal with added gaussian noise. The mean is increasing with time, t, thus it is not a stationary signal. After the first step of ARIMA, the signal is moved to have constant mean as shown in Figure 2.4b The signal is now stationary.

2.3.1 K-nearest Neighbors

A simple and effective method for supervised classification of data samples is called k-nearest neighbors (kNN) [17]. Given a set of labeled samples with n classes, which preferably show some clustering on a given basis, kNN will use the k number of closest labeled data points to vote on a predicted class. The basis used to represent the data can be the raw measurements of the samples, the Principal Components from a PCA, or some other representation that is meaningful. Popular frameworks for ML and data science in multiple languages have implemented effective algorithms for this technique making it easy to utilize. [1] [2] An illustration of kNN is given in Figure 2.5.

2.3.2 Random Forests

Before taking a look at random forests, an explanation of its building blocks, *decision trees*, shall be given. A set of data points from n measured variables, say x_n, are used to build a decision tree. Visually the tree consists of nodes, some nodes have a criterion that splits the data and forwards the splits to other nodes. *Leaf nodes* are endpoints of the data where a cluster of points is gathered as a class. On top of the tree is the first node which will use the variable x_1 to find a threshold that best splits the data. The two sets are forwarded to the next nodes where one might be a leaf node representing a predicted class, and the other node might be a new criterion splitting its share of the first split. The splitting will go on like this until the data are classified; thus, the domain space is sectioned to fit the classes.

[1] `https://scikit-learn.org/stable/modules/neighbors.html`
[2] `https://www.geeksforgeeks.org/k-nn-classifier-in-r-programming/`

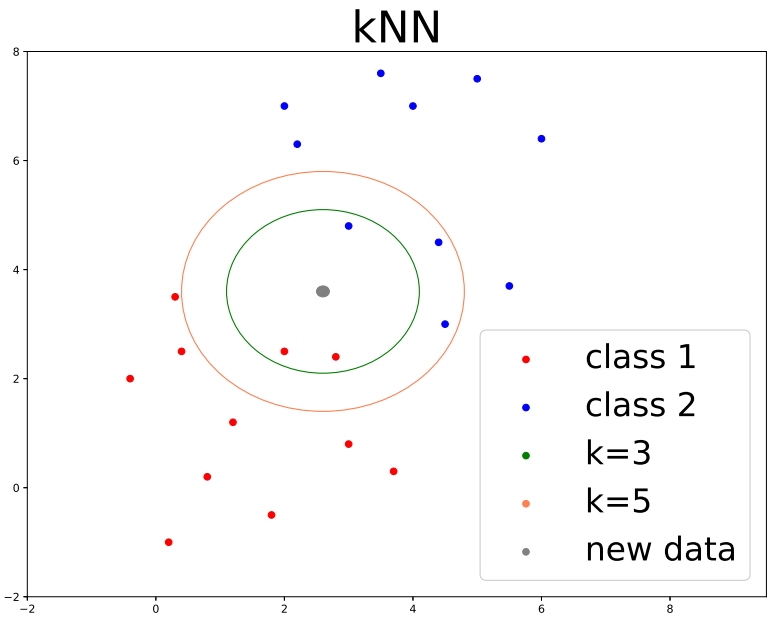


Figure 2.5: This example shows how the k-nearest neighbors algorithm classifies new datapoints given the k number of nearest neighbors. It also displays how the number k affect the classification.

A tree might look like what is shown in Figure 2.6. Such a tree is highly sensitive to the starting variable and the order in which it propagates through decisions.

Multiple splits can be chosen for a given node, but some measure of what is most effective is necessary. There are multiple ways of computing this measure, and one of them is to compute the entropy of each choice similarly to what is described in subsection 2.3.7.

To combat the sensitivity a *random forest* is used. It is called a forest because it uses multiple trees to produce multiple predictions, then a final prediction is given from the highest vote. The randomness comes from *bootstrapping* the data set, that is producing new datasets from a random subset of the original. The data in these subsets are also of random order, meaning a new decision tree will start with a different variable and propagate through a new order of variables than the last. Random forests are thus a regularization method expanded from decision trees, making it less biased to a dataset and capturing more variance. Random forests are implemented in coding languages often used by data scientists, making the method easy to explore. [3] [4] [25]

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`

[4] `https://www.rdocumentation.org/packages/randomForest/versions/4.6-14/topics/randomForest`

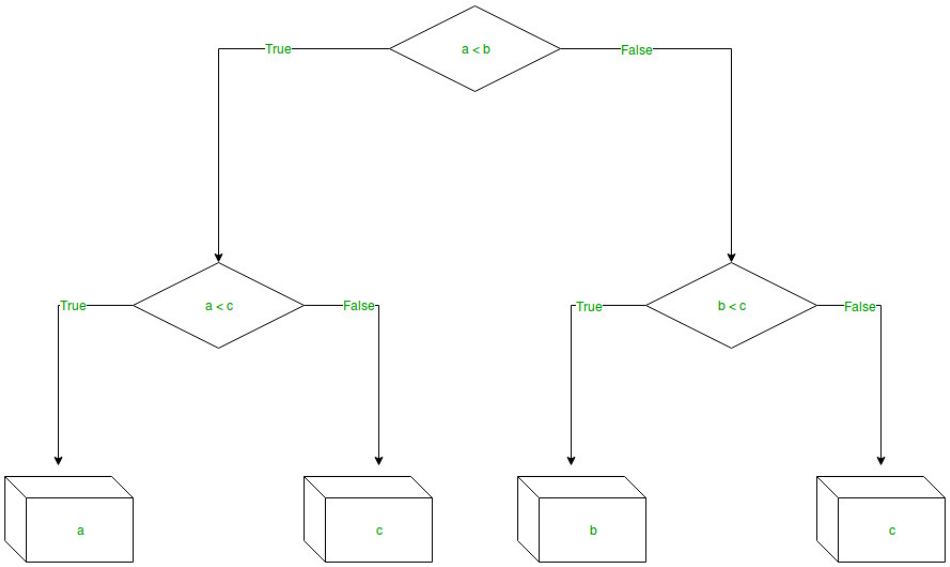


Figure 2.6: A visualization of a simple decision tree

Source: https://www.geeksforgeeks.org/python-decision-tree-regression-using-sklearn/

2.3.3 Support Vector Machines

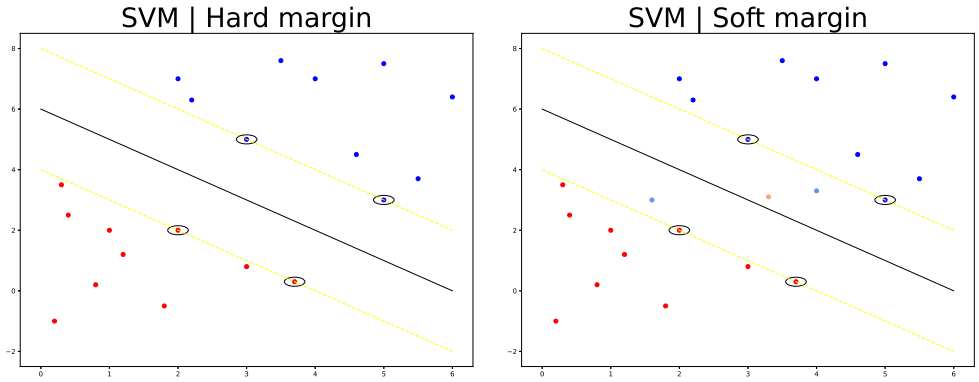
Support Vector Machine (SVM) is another way of classifying high-dimensional data. Based on some kernel function this algorithm will make hyperplanes that effectively split the data so that a hyperplane has the longest distance to the clusters it is separating. Hyperplanes that tangent the clusters are called the margins. The kernels deciding how a hyperplane is placed can vary in accordance with the problem-domain. A linear kernel will only be able to classify problems that are linearly separable while a non-linear kernel can split more complex problems. Many non-linear problems use the *kernel trick* to capture the discriminant features of the data [9].

For a lot of classification problems, the classes overlap, meaning a *hard margin* will not yield a solution. Therefore *soft margin* SVMs exist, where some points are allowed to cross the margins. Such a solution address the *bias-variance tradeoff* in SVMs [9]. A simple illustration of linear classification using both hard margin and soft margin SVMs is shown in Figure 2.7.

One of the popular kernels is the Radial Basis Function (RBF) given by

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \exp(-\gamma||\boldsymbol{x} - \boldsymbol{x'}||), \tag{2.16}$$

which enables non-linear classification [25].



(a) **(b)**

Figure 2.7: Two illustrations of Support Vector Machines, one with hard margin 2.7a and one with soft margin 2.7b. SVMs try to fit hyperplanes so that they split clusters of data by maximizing the distance to the margins (yellow lines). Many problems are not linearly separable, thus a soft margin is implemented such that some overlapping data points are allowed.

2.3.4 Linear Regression

Regression is a different form of machine learning. Instead of classifying data points, regression is trying to fit a function to the set of points by minimizing their distance to the function. There are multiple choices of distances that can be minimized, but the most known might be Least Square (LS) [9].

With linear regression (LR) one assumes the data points hold a linearity, thus a line can be fitted to the points. When the data has one variable x and one response y it is called a *simple linear regression*. When the data has more than one variable it is called *multiple linear regression*. An illustration of linear regression with LS is presented in Figure 2.8.

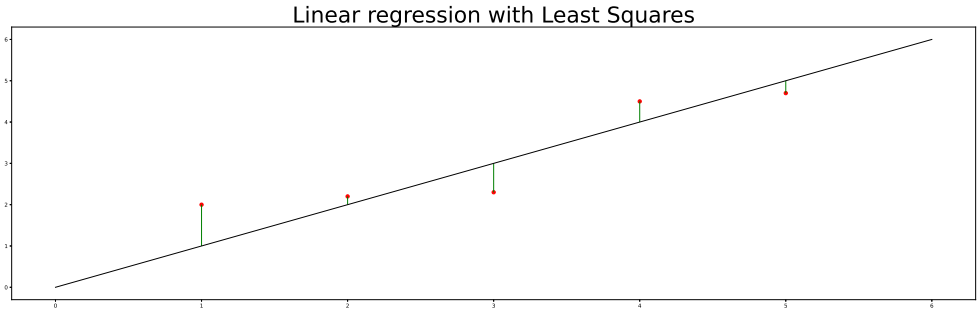


Figure 2.8: A simple illustration of linear regression (black line) on some data points (red dots) using Least Square (green lines). The total distance from the data points to the line is minimized to make an optimal, linear, model. The technique can be expanded to functions of higher order as well as systems of multiple dimensions.

2.3.5 Gaussian Process Regression

Gaussian Process Regression (GPR), or Kriging, is the technique of calculating a distribution over functions, rather than the function parameters.

First, given a set of stochastic variables x_n for $n \in \{1, N\}$. For those to be a Gaussian Process (GP) it is assumed that the probability $p(f(x_1), ..., f(x_n))$ is jointly normally distributed with a mean, μ, and a variance given by the kernel, or covariance, $\Sigma(x) = K(x_i, x_j)$.

Then, for GPR to work the idea is that for two similar x a correspondingly similar output y will be observed. A confidence interval can be computed on the regression and from the idea of similarity it should be clear that the interval is smaller where similar samples are observed [9].

$K(x_i, x_j)$ denotes the kernel function used to compute the similarity of the inputs, thus it represents all possible parameter combinations of the kernel given the inputs it has seen. How suitable the kernel is to the data is the only variable affecting a GPR's performance. A popular kernel used in Gaussian Process Regression is the previously described Radial Basis Function (2.3.3). Given $\mu = 0$ and kernel K, the general formulation of the predictive density is

$$p(f_*|\boldsymbol{X}_*, \boldsymbol{X}, y) = \mathcal{N}(f_*|\mu_*, \Sigma_*) \tag{2.17a}$$

$$\mu_* = \boldsymbol{K}_*^T \boldsymbol{K}^{-1} y \tag{2.17b}$$

$$\Sigma_* = \boldsymbol{K}_{**} - \boldsymbol{K}_*^T \boldsymbol{K}^{-1} \boldsymbol{K}_*. \tag{2.17c}$$

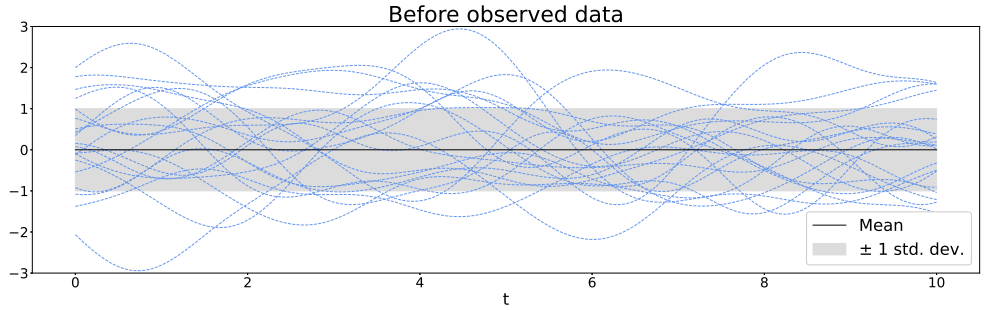
where the distribution is calculated over the function f_*, given possible inputs $\boldsymbol{X}_*$ and the previously observed inputs $\boldsymbol{X}$ with corresponding responses y. A powerful trait of Kriging is that it gives a confidence interval describing the certainty of the model in unobserved areas. An example showing the distribution over functions before and after the model has seen data are displayed in Figure 2.9.

There are implementations of Gaussian Process Regression in multiple frameworks for multiple languages, but one power full tool worth mentioning is the GPyTorch for Python which utilizes modern parallelized hardware for accelerated processing. [5]

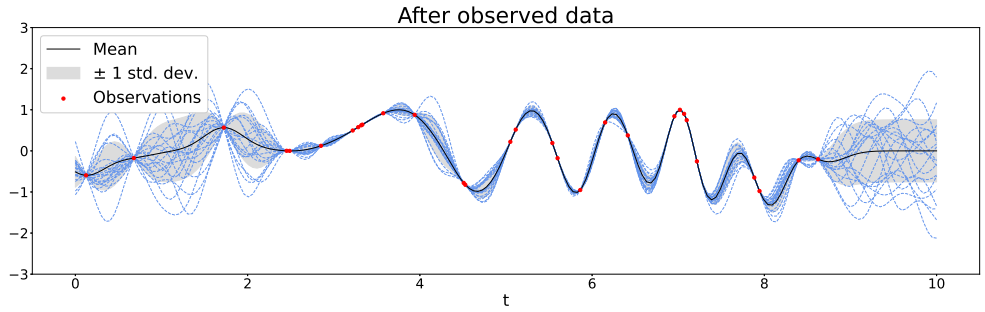
2.3.6 Training Neural Networks

Before talking about different modern ML architectures one should first understand what ML solutions are on a meta-level. ML is not a magic box that learns like a human, it is more useful to think of it as an optimization problem. The goal of every Neural Network (NN) is to regress or classify some form of information subject to its statistical distributions. A good analogy for this is fitting a line to some measured samples. To do this accurately a need for *optimization* quickly occurs, the goal is to optimize the fitted line subject to some measure of error. A simple illustration can be seen in Figure 2.10.

[5] `gpytorch.ai`



(a)



(b)

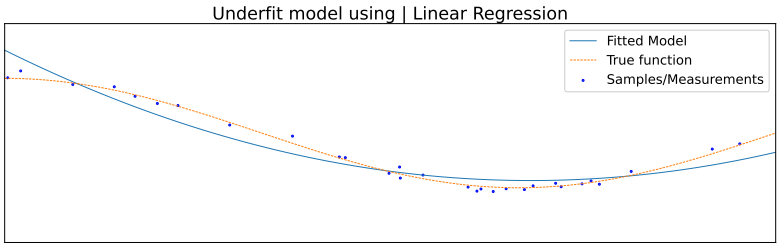
Figure 2.9: A Gaussian Process Regression model before 2.9a and after 2.9b some observed data. Before any observed data the confidence interval is huge as illustrated by the subset of plotted possible functions, thus the model is of little use. After some points are observed a model is formed and the subset of possible functions are "drawn" into the points. The black line is the fitted model and the blue lines represents the uncertainty in the model at each point. Such a feature is power full as it can, for any input, give both the predicted output as well as a measure of certainty based on the statistical properties of the kernel.

Such problems have been addressed by traditional Machine Learning for a long time, but the last decade has brought us Deep Learning (DL). New techniques and architectures that inhabit large parameter counts allow us to capture more variance in complex tasks. Every NN consists of several layers of which each layer has a number of nodes. A node represents some weighted mathematical operation connecting the inputs to the output. Think of each layer as a matrix for now. When optimizing a NN, the weights in each node in each layer must be changed incrementally to train the network. That is; enhancing the performance of the network, by minimizing some objective function. The training starts by feeding the network with some labeled information. Since the information is labeled a comparison between the output of the network and the label can be made. The result from the comparison triggers a change in the weights of the network. With the results from the comparisons at the output, one works backward through the network, computing the gradient. The gradient is a hyperplane that locally approximates a description of how changes in the weights affect the performance

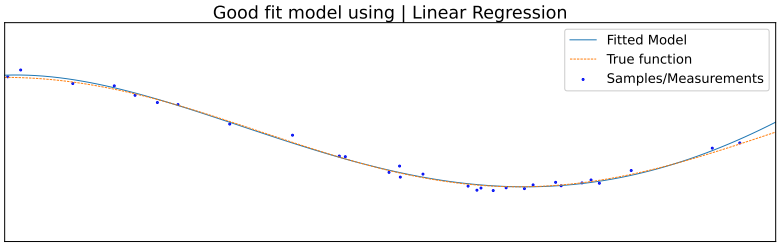
of the network. The method of calculating the gradient is called *backpropagation*, or "backprop". Each iteration in the optimization process takes a step on the gradient, meaning it will find a direction of the steepest descent and move all the weights in that direction. An illustration of a gradient and the steps taken during optimization is shown in Figure 2.11.

Taking these steps can generate some problems, where the two main ones are an *exploding gradient* and a *vanishing gradient*. When calculating the gradient from the output side of a deep network the parameters on the input side can be subject to their gradient zeroing out, i.e. *vanishing*. Therefore the network becomes hard to train and the benefit of many parameters is not utilized. Different solutions are implemented depending on the architecture.

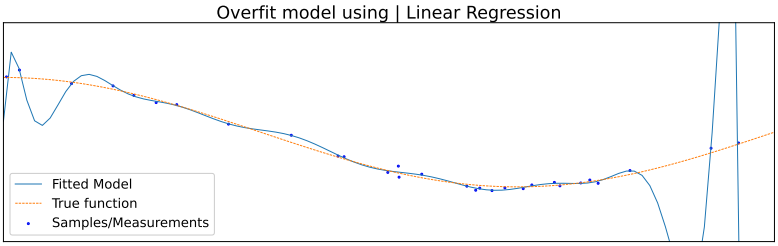
Paying too much attention to certain parts of the information might generate huge weights at certain nodes in the network. The gradient around these nodes will get disproportionately steep and potentially encourage a big step in the optimization. Big steps might lead to more big steps and the network becomes unstable as described by Pascanu *et al.* [26]. This phenomenon is called *exploding gradient*. Preventing this is usually done by clipping the gradient, thus only using the directional information in the step, not the step size. The popular Stochastic Gradient Descent (SGD) optimizer also solves the problem in many instances [5].



(a) Underfitted model. The model is too simplistic, not being able to explain the variance of the given data.



(b) A model with good fit, the least complex model that covers the variance of the given data.



(c) Overfitted model. It covers variance which is too specific for the given data, thus it is not going to perform well on new data. Regularization methods is needed to generalize the model.

Figure 2.10: A simple illustration with samples (blue dots) from a system (orange line) showing the principals of overfitting and underfitting a model (blue line). In 2.10a) it is clear that the model does not capture the variance of the measured samples, which is opposite to 2.10c) that captures to much variance. Both situations is a problem since they cannot represent the true model and thus it has low generalizability. Figure 2.10b is closest to the true function, it is the least complex model that fits the data points at hand.

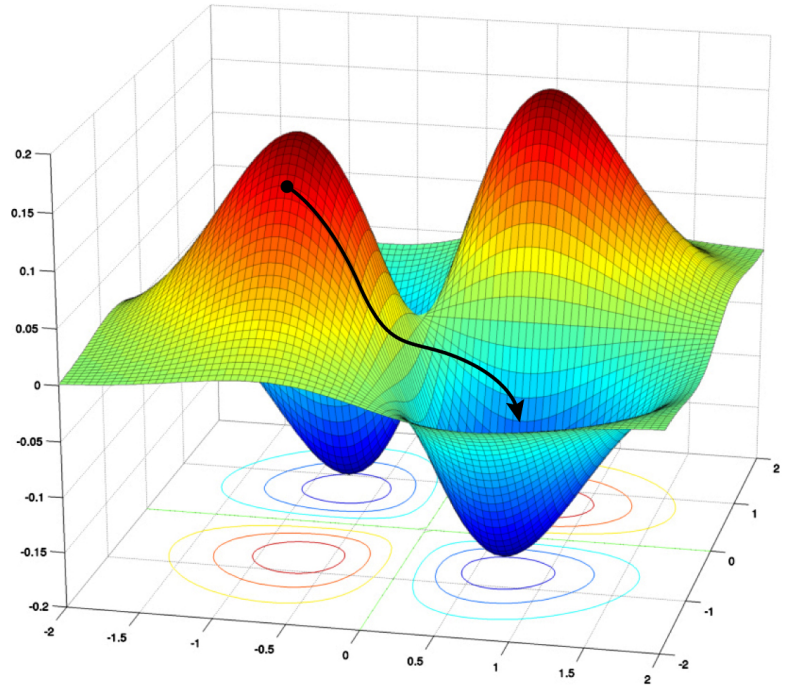


Figure 2.11: This illustrates a simplified gradient computed during the optimization of a neural network. Steps taken during optimization should follow the black line as it is the direction of steepest descent.

Source: https://sciencesprings.wordpress.com/tag/stochastic-gradient-descent/

2.3.7 Cross-entropy Loss

To optimize a model there is a need for a function to minimize. This function is called the *loss function* in the ML community, and it describes how well the model performs.

Cross-entropy loss is a way to quantify the distance between two probability distributions. In the case of supervised machine learning the labeled data are one-hot encoded to describe the true class. Similarly, the output of the model is an array with a score given to each class. These two arrays can be interpreted as two probability distributions, thus the distance between them can be calculated. Mathematically, cross-entropy is closely related to Kullback-Leibler Divergence (KL Divergence):

$$H(P,Q) = -\mathbf{E}_{X\sim P} log(Q(x)), \tag{2.18}$$

where P and Q are the two probability distributions, and $\boldsymbol{E}$ is the expected value of some distribution [5].

2.3.8 Optimizer Algorithms

As described briefly in subsection 2.3.6, optimizing a Neural Network involves computing a gradient; a hyperplane describing how a change in parameters affects the objective. When optimizing on this plane the algorithm has to take one step at a time to find a minimum. How big these steps are is called the *learning rate*. For fast training of the model, it is beneficial to have a large learning rate, but this could make for problems when closing in on a minimum. Looking at Figure 2.12b it is clear that the model is missing the optimal point by "jumping over it", this could be avoided by choosing a smaller learning rate at the cost of training time. To get the best of both worlds the learning rate should be *adaptive*, decreasing the size of the step as the optimizer gets closer to the minima. A demonstration of a too short and too large learning rate, where *adaptive learning rate* would be a solution, is shown in Figure 2.12. There are several methods that utilize this principle, for example, Adagrad [27], Stochastic Gradient Descent [28], and Adaptive Moment Estimation (ADAM) [29].

All the mentioned optimizers are gradient descent methods, but SGD is an important and highly used technique. When datasets becomes big the normal *gradient descent* method is computationally expensive as it computes the gradient across all the data in the dataset. To mitigate this cost of optimization, while still achieving comparable accuracy, stochastic behavior was introduced by only calculating the gradient in a uniformly chosen subset of the data, a *mini-batch*. This does not ensure that the next step is the *true optimal* direction, but rather a step close to it. As a result, the set of steps will be uneven, but heading towards the minima [5].

A challenge with SGD and ADAM is that the uneven steps might jump back and forth excessively. To compensate for this, both algorithms use previous steps to "guide" the new steps; this is called *momentum*. One difference between SGD and

(a) **(b)**

Figure 2.12: In Figure 2.12a the learning rate is small and converges to the minima, but, with such small steps, it might take some time to get there. And so, in Figure 2.12b, the step size is increased in the hopes of converging faster, but what can happen is the model steps over the minima and might also diverge. Both problems can be solved by adapting the step size to its current place in the gradient.

Source: `https://laptrinhx.com/understanding-optimization-algorithms-3\818430905/`

ADAM is the implementation of momentum. Where SGD uses the actual gradient to compute the momentum, ADAM uses a Moving Average (MA) of the gradient [29]

2.3.9 Activation Functions

Between layers in a NN it is normal to use *activation functions*. Activation functions can be any function whose purpose is to map an input to an output. It is preferably a function whose derivative is continuous, which makes optimization with the gradient easier. Non-linear activation functions make a linear system of layers non-linear. Without a non-linear activation function, a NN would simply be a linear combination of factors, not being able to capture the more complex dependencies in a system. Rectified Linear Unit (ReLU) is an activation function given by

$$y(x) = \begin{cases} 0 & ,x < 0, \\ x & ,x > 0 \end{cases} \tag{2.19}$$

for $x \in \{-\infty, \infty\}$. It is probably the most used activation function between the hidden layers in modern DL. One reason might be that the derivative of ReLU is 1 for all $x > 0$, which mitigates the problem of vanishing gradients; multiplying with one does not contribute to a change in the parameters. A second reason might be its low computational complexity, as opposed to, for example, *sigmoid*, which is also a common activation function.

A sigmoid is given by

$$y(x) = \frac{1}{1+e^{-x}}. \tag{2.20}$$

The sigmoid function will map any input $x \in \{-\infty, \infty\}$ to an output $y \in \{0, 1\}$. It is used as the last layer activation function in binary classification to ensure an output contained within the boundaries 0 and 1. For classification with more than two classes, the most common function is Softmax. Softmax looks and behaves similar to a sigmoid activation, but it also ensures that the sum of the scores across all classes equals to one. It is given by

$$y(x)_i = \frac{e_i^x}{\sum_{j=1}^{K} e^{x_j}} \tag{2.21}$$

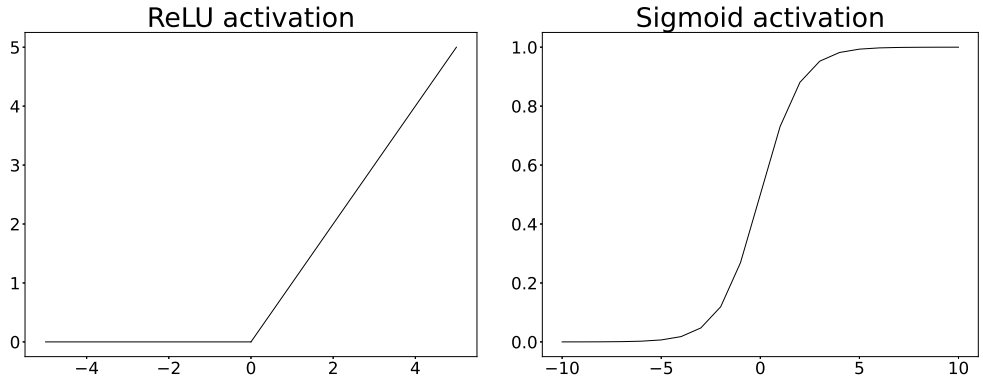


Figure 2.13: The ReLU activation is shown in Figure 2.13a. ReLU will zero out any $x \in \{-\infty, 0\rangle$ and pass through all values $x \in [0, \infty\}$. The sigmoid function, shown in Figure 2.13b, will take any $x \in \{-\infty, \infty\}$ and map it to a value $y \in \{0, 1\}$.

2.3.10 Fully Connected Neural Networks

The most basic neural network is called a Fully Connected Neural Network (FCNN). FCNNs are built by stacking multiple layers of nodes, connecting all nodes between each layer. FCNNs are usually made with one or more layers between the input and the output. These layers are called *hidden layers*, and the total number of layers is referred to as the *depth* of the network. An input of size n is fed through the layers of the network. The output of the network corresponds to the k number of classes, so FCNN maps an input to a class [5, 30].

Mathematically, it is convenient to look at a shallow network with only one hidden layer. The first layer in the network is the input layer, thus it equates to **x**. As explained by Goodfellow *et al.* [5], a set of weights is then multiplied by **x**

to map the input to the hidden layer. These first weights can be denoted as the $\boldsymbol{\omega}_0$ matrix. Adding some bias, $\mathbf{b}$, and assigning $\mathbf{h}$ to the output yields the linear system

$$f(\mathbf{x}) = \mathbf{h} = \boldsymbol{\omega}_0^\intercal \mathbf{x} + \mathbf{b}. \tag{2.22}$$

This maps the input to the hidden layer, and the same procedure is done to map the hidden layer to the output classes:

$$f(\mathbf{x}) = \mathbf{y} = \boldsymbol{\omega}_1^\intercal \mathbf{h} + \mathbf{b} = \boldsymbol{\omega}_1^\intercal (\boldsymbol{\omega}_0^\intercal \mathbf{x} + \mathbf{b}) + \mathbf{b} \tag{2.23}$$

Continuing this pattern results in a chained system of functions where

$$f(\mathbf{x}) = f^N(f^{N-1}(f^{\cdots}(f^0(\mathbf{x})))). \tag{2.24}$$

This forwarding of the data is the reason why this architecture is sometimes called feed forward networks. An FCNNs depth and width determines its complexity thus the variance it can capture when trained. FCNNs can be given any input vector, but usually one would feed the network features from the system being modeled [5].

2.3.11 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a different form of network architecture where the input matrix is convolved with one or multiple kernels of a predetermined size, k. The depth of this network is determined by how many layers of convolution is being done, in addition to this CNNs often use *pooling layers* between convolutions. The pooling layers use methods such as maximum value or average value to extract features and downsize the information, thus reducing the computational need for the next layer. This technique can also be used dynamically to ensure that different input sizes result in the same number of features, which is important when the features are put into an FCNN for classification. FCNNs use pure matrix multiplication, as described in subsection 2.3.10, which is dependent on matching matrix sizes. CNNs works in multiple dimensions, but an illustration of an architecture for time-series can be seen in Figure 2.14 [30].

CNNs are often used to extract features, especially in the field of computer vision. A trained network is extremely effective in finding the most basic structures in an input, resulting in a set of feature maps corresponding to these structures. These feature maps are the result of optimizing the kernels during training; feature maps are normally flattened and fed through a FCNN for classification.

As exemplified by Goodfellow *et al.* [5], convolution for discrete problems is mathematically noted as

$$f(x) = (x * \omega)(t) = \sum_n x(a)\omega(t-a), \tag{2.25}$$

where x is an input and ω is the kernel. Visually, one can think of the kernel ω as a filter that slides over the input. The numbers held by the kernel determine the

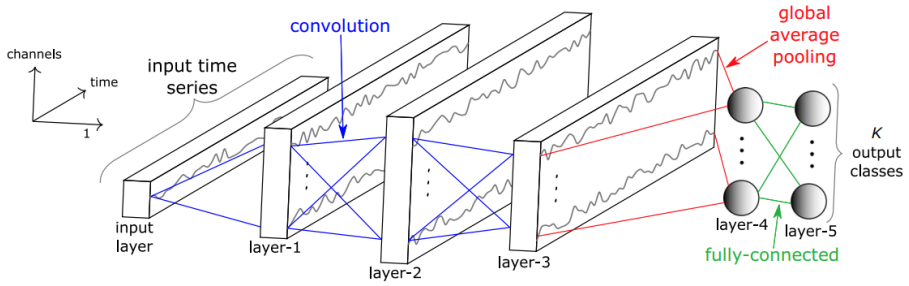


Figure 2.14: An illustration of a single channel CNN. For the context of time-series, one would call it one dimentional, then time will be the second dimension. As illustrated above the the convolutions will generate multiple feature maps according to the architecture perameters, before they are all flattened and fed through the fully connected part of the ntework.

Source: [31]

feature it extracts. Kernels can be edge extractors by emphasizing different lines, they can be derivatives emphasizing areas of rapid change, or even shapes that resemble specific or profound features. These filters are interesting to visualize in the shallow layers, since the feature they extract is visible and interpretable. As we convolve into the network, the features become more and more abstract. An example of the convolution procedure is visualized in Equation 2.26, where the colors show how the kernel "slides" over the input matrix producing the elements in the resulting matrix.

$$\left[\begin{array}{c|ccccc|c} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & \textcircled{1} & 0 & 2 & 3 & 1 & 0 \\ 0 & 3 & 2 & 0 & 7 & 0 & 0 \\ 0 & 0 & 6 & 1 & 1 & 4 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \circledast \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} \textcircled{2} & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \end{bmatrix} \tag{2.26a}$$

$$\left[\begin{array}{c|ccccc|c} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 2 & 3 & 1 & 0 \\ 0 & 3 & 2 & \textcircled{0} & 7 & 0 & 0 \\ 0 & 0 & 6 & 1 & 1 & 4 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \circledast \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & -1 & 11 & -2 & -13 \\ 10 & -4 & \textcircled{8} & ? & ? \\ ? & ? & ? & ? & ? \end{bmatrix} \tag{2.26b}$$

A key feature of CNNs with pooling layers is the loss of spatial information, which enables the detection of known distributions without them appearing in the same part of the input [30]. For example, a network classifying cats and dogs would achieve the same accuracy no matter where the animal is placed in the picture; for time-series the network would observe the same pattern no matter when it happens.

The biggest breakthrough of Convolutional Neural Networks is their relatively low computational complexity. They are easy to train, tune and test which makes them practical and powerful to deploy in real-world applications. Convolution and pooling are, by design, the main reason for this computational efficiency [5].

2.3.12 Residual Neural Networks

Residual Neural Networks (ResNets) is based upon the normal Fully Connected Neural Network and/or Convolutional Neural Network. In a paper by He *et al.* [32] a residual block is defined as one or more weight operations with a ReLU activation layer between them. They add a shortcut connection from the input to the output of the block, thus forwarding the information to the input of the next block. An illustration of the residual block can be seen in Figure 2.15

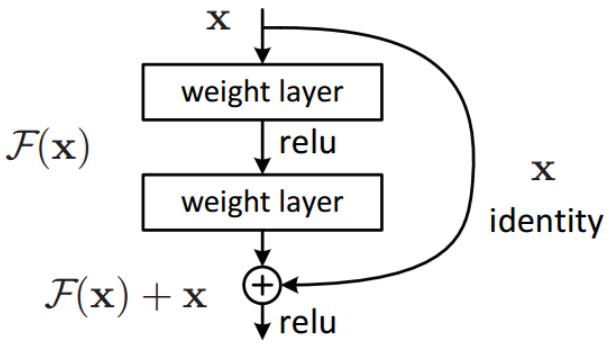


Figure 2.15: An illustration of a single residual block used in Residual Neural Networks.

Source: `https://neurohive.io/en/popular-networks/resnet/`

Mathematically, the operation can be described as

$$output = F(x) + x \tag{2.27}$$

where F(x) is the weight operations in one node as illustrated in Figure 2.15. This expands to

$$output = F(x, W_s) + W_i * x \tag{2.28}$$

where W project the output and the input to equal dimensions. W is necessary when residual blocks convolve the input, as convolution downsizes the data.

Although the shortcut connection does not add computational complexity to the network it inhabits two major advantages compared to the previously mentioned FCNN and CNN [32]. It enables *simple optimization* of extremely deep neural networks while achieving a significantly lower training error than its counterparts. Deeper networks have more parameters allowing them to *capture more variance*, therefore this is a powerful feature [31].

2.3.13 Recurrent Neural Networks

There is another group of networks that tackle sequential problems; they are called Recurrent Neural Networks (RNNs). In general, a hidden *state* is implemented in the model. When a sequence of inputs is given, a state will receive the previous state and one point in the sequence as input. The information is processed, and the state is updated and fed into the next state. In its most basic form, RNN is formulated as

$$h_t = f(h_{t-1}, x_t), \tag{2.29}$$

where h is the hidden state [5]. This concept is expanded to solve problems in different domains, where some might have a sequence of inputs and a single output and others might have a sequence of inputs with a sequence of outputs.

RNNs suffers from the exploding and vanishing gradients explained in subsection 2.3.6. Exploding gradients is solved by scaling down the update values[5], but vanishing gradients are solved by controlling the error/gradient flow [33]. Solutions are implemented in Long-Short Term Memory (LSTM) and Gated Recurrent Unite (GRU) networks [5]. They both add an internal state, c, alongside the hidden units, h, and implement slightly different versions of gated units. At every point in the sequence, these gated units choose which information to include or forget in the internal state, as well as its impact. Gated units can be described as

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}, \tag{2.30}$$

where W is the weights as a function of the previous hidden units h_{t-1} and the current input x_t. i, g , f , and o is the gates respectively deciding if a cell should be written to, how much to write, if a cell should be deleted, and how much of a cell should contribute to the output [33, 34]. The middle parenthesis describes the activation function for each gate.

The internal state is updated by

$$c_t = f \cdot c_{t-1} + i \cdot g, \tag{2.31}$$

before c_t are used to compute the hidden units which are propagated forward in the network. h_t is given by

$$h_t = o \cdot \tanh(c_t). \tag{2.32}$$

Equation 2.31, the internal state, is what solves the majority of the vanishing gradient problem. When computing the gradient, the output of the weights in each unit is contained in the internal state, which avoids the problem of multiplying the same small numbers over and over. An illustration of a gated LSTM, implementing Equation 2.30, 2.31, and 2.32, are shown in Figure 2.16.

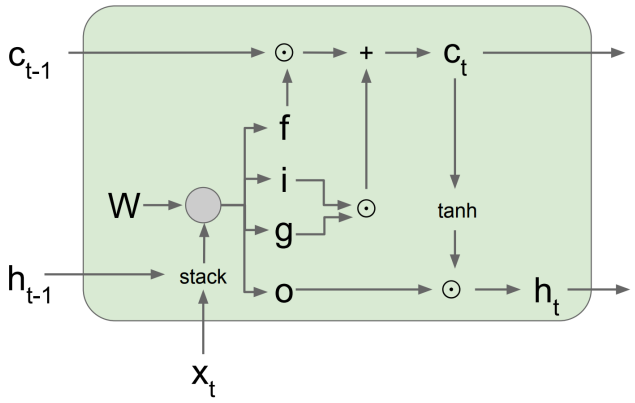


Figure 2.16: The previously hidden unit h_{t-1} and the current point in a sequence $\boldsymbol{X}$ are inputs to the RNN unit where the internal state is also accessible. The internal state and the hidden units are computed according to Equation 2.30, 2.31 and 2.32. The gradient will be computed on the internal state c_t from which W is updated.

Source: [34]

2.3.14 Autoencoder Networks

A method for dimensional reduction that can be exploited for the purpose of compression is called an *autoencoder*. Autoencoders are described by Goodfellow *et al.* [5] as a special case of FCNN, where the purpose of the model, h, is to recreate the input:

$$h(x) \approx x \tag{2.33}$$

A generic illustration of an autoencoder is shown in Figure 2.17, displaying the three parts of the network; the *encoder*, the *code*, and the *decoder*. The encoder inputs the full-size data before extracting some form of meaningful representation which is the code. The code is also called, by some researchers, the *latent representation* of the data. This code is fed into the decoder which will process it back to a lossy copy of the input.

An autoencoder could be a network of equally sized hidden layers, but reducing the hidden layer sizes as the data are propagated through the network will restrict its capacity and might make the network discover some prominent features in the data. Such "bottlenecked" networks are called *undercomplete*, and linear activations between the layers yield results closely linked to the Principal Components of the data [5].

Instead of analyzing the reduced space, one can separate the *encoder* and *decoder* using the code as a compressed version of the data [35].

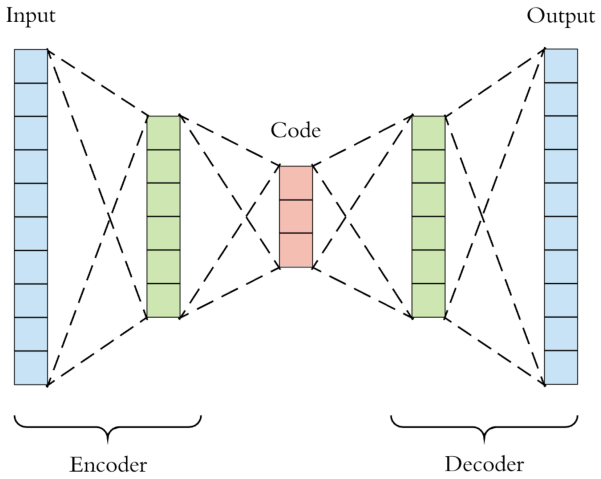


Figure 2.17: An illustration of a general autoencoder architecture. The truncating side is called the *encoder*, it takes the full input size and processes it into a smaller set of values. This set of values is called the code and hopefully contains some prominent features from the input. The expanding side is called the *decoder*. It tries to recreate the input data from the code if trained to do so.

Source: `https://blog.paperspace.com/content/images/2020/01/1_oUbsOnYKX5DEpMOK3pH_lg.png`

2.4 Convex Optimization and Compressed Sensing

Compression of data has traditionally been done by transforming the data onto a new basis, usually some form of frequency space, where the original signal can be represented as a sparse vector of coefficients. The sparsity comes from the removal of coefficients of low significance [17]. The amount of information removed in this process is so big that most of the information we capture is excessive when it comes to reconstructing the data [36]. This knowledge leads to the thought that one might only capture the data of interest in the first place, eliminating the need for compression. Only the captured data should then be used in the reconstruction of the measured variable. *Compressed sensing* is an attempt at doing exactly such a maneuver.

Other names for compressed sensing often used in the literature are compressive sensing [37], compressive sampling [38], and sparse sampling [39]. Faster Magnetic Resonance Imaging (MRI) scans are one of the most known uses of compressed sensing [40], but other uses include high-resolution radar imaging [41] and image up-scaling (super-resolution) [42]. First a look at convex optimization which is the workhorse of compressed sensing.

2.4.1 Norms

A useful measure to talk of within linear algebra is the length of a vector and the size of a matrix. Different measures for quantifying the length and size exist and the most known are norms.

The l_p-norm is defined as

$$||x||_p = (x_0^p + x_1^p + \cdots + x_n^p)^{\frac{1}{p}} \tag{2.34}$$

for any $x \in \mathbb{R}^n$ [43]. Let $p = 2$ and Equation 2.34 yields the widely known *Euclidian norm*, or l_2-norm:

$$||x||_2 = (x_0^2 + x_1^2 + \cdots + x_n^2)^{\frac{1}{2}} = \sqrt{\sum_{k=0}^{n}(x_k^2)} \tag{2.35}$$

The definition allows for the l_1-norm which is the sum of absolute values:

$$||x||_1 = (|x_0| + |x_1| + \cdots + |x_n|) = \sum_{k=0}^{n}|x_k|, \tag{2.36}$$

and the *max*-norm, l_∞-norm, given by

$$||x||_\infty = max\{x_0, x_1, \ldots, x_n\} \tag{2.37}$$

In terms of ML and optimization; norms are often used to regularize the solution, meaning a penalty is given if too many parameters are activated. Different norms yields different results, for example; the l_2-norm penalizes less the closer a parameter gets to zero. It is unlikely that any parameter will be pushed to zero, which again leads to lots of small parameters. l_1 on the other hand penalizes equally even though a parameter is close to zero, so it yields solutions with few active parameters and many zeroed out parameters. A simple example of a penalty is when given a vector where $x_0 = 2$ and $x_1 = 2$ the norm will be

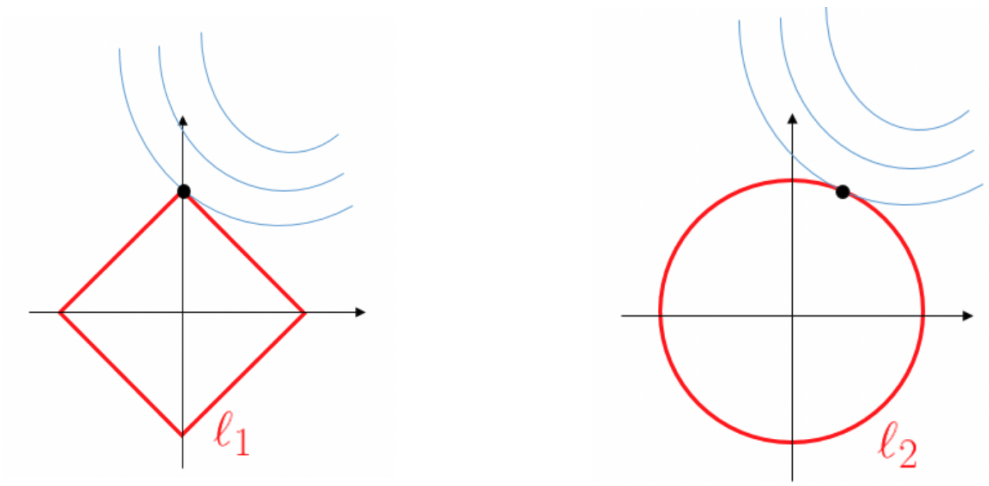
$$||x||_1 = (|2| + |2|) = 4, \tag{2.38}$$

and

$$||x||_2 = (2^2 + 2^2)^{\frac{1}{2}} \approx 2,828 \tag{2.39}$$

meaning an algorithm generating a solution $x_0 = 2, x_1 = 2$ will get penalized harder if the l_1-norm is used rather than l_2-norm.

Visually one can look at the illustrations in Figure 2.18, where the red lines represent the norms. When a norm intersects a solution in the feasible set a black point is made, the figure illustrates two different points of intersection. The l_2-norm in Figure 2.18b intersects between the two axes meaning both axes are used when providing a solution, but the l_1-norm in Figure 2.18a intersects on an axis leaving the other axis zeroed out. For this reason, optimizing with the l_1-norm often results in sparse solutions, which is useful in many fields of science and engineering [17]. This is also the key to why l_1-norm is used for compressed sensing as will be further explained in subsection 2.4.2 and subsection 2.4.3.



(a) **(b)**

Figure 2.18: In Figure 2.18a an illustration are shown of where the L_1-norm might intersect a feasible solution. The black dot shows that the intersection is found on an axis, resulting in a solution where one axis is zeroed out and the other is active. Figure 2.18b shows the L_2-norm intersecting feasible solution between the two axes, resulting in a solution where both axes are active.

Source: `https://freakonometrics.hypotheses.org/57790`

2.4.2 Convex Optimization

Although NNs are an example of convex-like optimization and a high-level explanation of how they are trained has been given in subsection 2.3.6, a more mathematically intense overview of convex optimization will be given here. From Nocedal and Wright [2] a definition of a convex function, f, is when its domain S is a convex set and any two points in S satisfy the property given in Equation 2.40 [2].

$$f(\alpha x + (1-\alpha)) \leq \alpha f(x) + (1-\alpha) f(y), \qquad \forall \quad \alpha \in [0,1] \tag{2.40}$$

In its *standard form*, convex problems are written as

$$\begin{aligned} \text{minimize} \quad & f_0(x) && (2.41)\\ \text{subject to} \quad & f_i(x) \leq 0, \quad \text{for} \quad i = 1,\dots,m && (2.42)\\ & a_i^T x = b, \quad \text{for} \quad i = 1,\dots,p && (2.43) \end{aligned}$$

where the *objective function*, f_0, is convex, the *inequality functions*, f_i, are convex and the *equality constraints*, $a_i^T x - b = 0$, must be affine. There are multiple ways to numerically calculate a solution for convex problems, while some that are often used for ML are mentioned in subsection 2.3.8. A couple more will now be touched upon.

A rigorous rundown of different optimization techniques is outside the scope of this thesis, but some details should be mentioned. Starting with the well known Newtons method; given a twice differentiable function to minimize, f, the *Newton step* is given by [43]

$$\Delta x_{nt} = -\Delta^2 f(x)^{-1} \Delta f(x) \tag{2.44}$$

Equation 2.44 says that $x + \Delta x_{nt}$ is the step needed to minimize f. If f is quadratic the step is an exact minimizer, if f is close to quadratic; an approximation, $\hat{f}$, is made. For the approximation a second-order Taylor series is used:

$$\hat{f} = f(x) + \Delta f(x)^T v \; \frac{1}{2} v^T \Delta^2 f(x) v, \tag{2.45}$$

where $v = \Delta x_{nt}$. A step that minimizes the Taylor approximation should also minimize the objective function.

Embedded Conic Solver (ECOS)[6] is one solver based on Newtons method and was presented by Domahidi *et al.* [44]. It uses a *barrier* method, an *interior point* method; a rewrite of the standard form problem shown in (2.43) to shift the unconstrained minima into the feasible set. The problem (2.43) can then generally be written in the form

$$\begin{aligned} \text{minimize} \quad & f_0(x) - \mu \sum_{i=1}^{m} f_i(x) && (2.46)\\ \text{subject to} \quad & c(x) = 0 && (2.47) \end{aligned}$$

[6] ECOS is available in Python through the package CVXPY

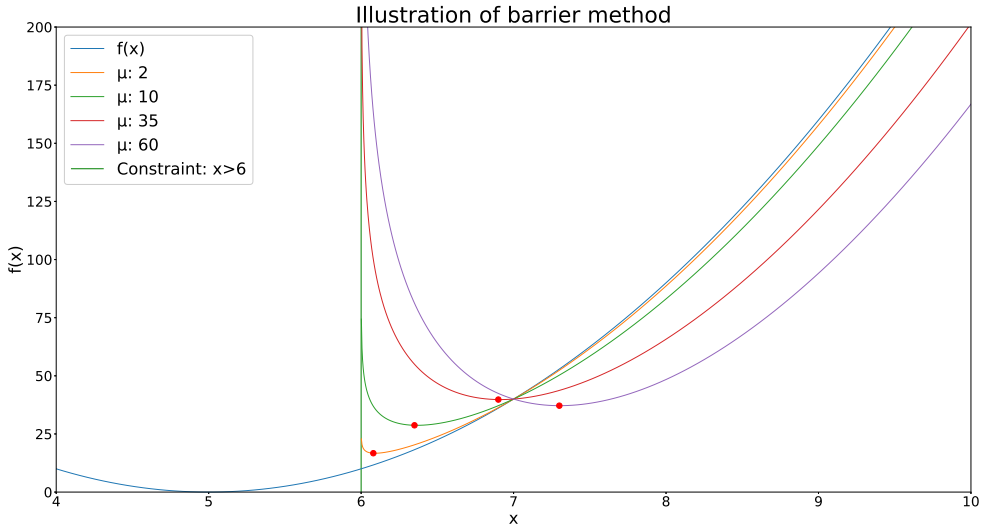


Figure 2.19: In this plot, the function $f(x) = (x-5)^2$ is minimized subject to the constraint $x > 6$. With a barrier method the problem becomes to minimize $f(x) = (x-5)^2 - \mu ln(x-6)$. One can see that the new problem moves the unconstrained minima into the feasible set and that decreasing μ brings the approximated optimum point closer to the actual solution.

where μ is the variable controlling how shifted the function is; the homotopy variable. During optimization μ is iteratively pushed to 0; consequently moving the unconstrained minima towards the constrained minima [45]. Figure 2.19 illustrates a greatly simplified barrier optimization. The reformulated problem can then be solved using Newtons method, per iteration of μ whilst checking for convergence for each solution.

Equation 2.44 makes it clear that it is necessary to store the Hessian matrix, $H = \Delta^2 f$, and calculate its inverse [43]. For bigger problems, this results in costly computations requiring expensive hardware, a lot of time, or both. Further readings on ECOS are found in [46] and [44]. Bypassing this problem of calculating and storing the Hessian is done with *quasi-newton methods* [43]. One popular algorithm is Broyden–Fletcher–Goldfarb–Shanno (BFGS)[7] which avoids computing the Hessian inverse by replacing it with an approximated matrix, $\boldsymbol{M}$, given by

$$\boldsymbol{M}_{k+1} = \boldsymbol{M}_k - \frac{\boldsymbol{M}_k s_k s_k^T \boldsymbol{M}_k}{s_k^T \boldsymbol{M}_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{2.48}$$

The important point to make from Equation 2.48 is that to calculate the approximated matrix $\boldsymbol{M}_{k+1}$ the previous approximation is needed; thus it must still be stored.

An even less memory-intensive version was proposed and is called Limited Memory BFGS (L-BFGS)[8]. L-BFGS does not store the approximated $\boldsymbol{M}$ between

[7] BFGS is available in Python through the package SciPy

[8] L-BFGS is available in Python through the package PyTorch

each step; it uses the identity matrix, $\boldsymbol{I}$, as a replacement for the previous approximation [5]. Also, Andrew and Gao [47] clarify that L-BFGS does not really estimate the $\boldsymbol{M}$, but merely the search direction $-\boldsymbol{H}_k J(\Theta)$. This results in a much faster numerical optimization algorithm suited for large-scale problems.

There is one major problem with BFGSs and L-BFGS that prevents them from being used in compressed sensing. To successfully generate a sparse solution to a linear problem, one will have to use the l_1-norm (2.36) of which derivative is not defined at $x = 0$ [47]. Andrew and Gao [47] provide a solution with their algorithm Orthant-Wise Limited-memory Quasi-Newton (OWL-QN). By realizing that the l_1-norm is differentiable and linear in any orthant, they made minor changes to the L-BFGS algorithm, limiting the search area to one *orthant* at a time. Through the optimization iterations, the active orthant is re-selected, thereby its name, and so the algorithm converges to a minimum. OWL-QN minimizes objective functions on the form

$$f(x) = l(x) + C|x|_1 \quad \text{for any} \quad C > 0, \tag{2.49}$$

where $l(x)$ is an arbitrary convex and differentiable function and C is a constant. It is difficult to illustrate the algorithm due to the nature of orthants, but further information on algorithmic details can be found in the original paper by Andrew and Gao [47].

Finally, some comments on the difference between optimizing Neural Networks and optimizing convex problems should be given. The reason optimizing NN is stated as "convex-like" at the start of this chapter is that although the problem is non-linear, it is not necessarily convex. As stated by Boyd *et al.* [43], convex optimization plays an important role in non-linear optimization as "...there is value in finding a good point, if not the very best." [43]. On the contrary, Goodfellow *et al.* [5] states that there might not be any critical point at all. Furthermore, Goodfellow *et al.* [5] point to a difference in the objective functions; when performing convex optimization one usually wants to directly optimize some performance measure P, whereas in Machine Learning some other objective function, J, is optimized of which one hopes will affect P. As the reader will understand in the next section, the optimization techniques for Compressed Sensing tend more towards that of ML as the objective function is the l_1-norm, optimizing for some sparse solution that can recreate a given signal.

2.4.3 Compressed Sensing

Given some data, $\boldsymbol{x}$, of length n, the goal is to make p measurements where $p \ll n$ and still recover the signal. Assuming x to be compressible it can be represented by a sparse vector $\boldsymbol{s}$ in a transform basis Ψ, thus

$$\boldsymbol{x} = \Psi \boldsymbol{s}. \tag{2.50}$$

Measuring $\boldsymbol{x}$ so that $y \in \mathbb{R}^p$, gives

$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} = \boldsymbol{C}\Psi\boldsymbol{s}, \tag{2.51}$$

where $\boldsymbol{y}$ is the captured information and $\boldsymbol{C}$ is the measurement matrix. At this point, the signal could be reconstructed if the sparse vector of coefficients $\boldsymbol{s}$ is known, and therefore the core of compressed sensing is to find the sparsest vector $\boldsymbol{s}$ that satisfy the measurements in Equation 2.51 [17].

For compressed sensing to work, two criteria must be met. The measurement matrix $\boldsymbol{C}$ must be *incoherent* to the transformed basis, meaning the rows of $\boldsymbol{C}$ should be close to normal to the columns of $\boldsymbol{\Psi}$. This effectively tests (or excites) more of the transformed space, widening the range of signals it can reproduce. Randomly sampled matrices are often good measurement strategies to ensure incoherence [17]. Uniformly sampled measurement matrices do not work, as it leads to multiple solutions with equal probability of being correct, a behavior correlating to the aliasing seen in undersampled signals [40]. Secondly, for a K-sparse signal in the basis $\boldsymbol{\Psi}$, the number of samples, p, needed to recreate the signal is

$$p \sim k_1 K \log(\frac{n}{K}), \tag{2.52}$$

where n is the number of samples in the original signal and K is the sparsity of the signal on the basis $\boldsymbol{\Psi}$. k_1 is a constant that describes the quality of the measurement, which is a measure of the incoherence of $\boldsymbol{C}$ and $\boldsymbol{\Psi}$. A better measurement matrix $\boldsymbol{C}$ means that fewer samples are needed. When these two conditions are satisfied, $\boldsymbol{C\Psi}$ has The Restricted Isometry Property (RIP), thus $\boldsymbol{C\Psi}$ acts like an *isometry* on sparse vectors [17]. This property is important as the geometry between the sparse samples taken from the data must be contained in the reconstructed signal.

Finding this sparsest vector $\boldsymbol{s}$ is a matter of optimization. Assuming $\boldsymbol{C\Phi}$ to hold RIP, Equation 2.51 is solvable with convex l_1 minimization. In standard form, the minimization problem is formulated as

$$\text{minimize} \quad ||\boldsymbol{s}||_1 \tag{2.53}$$

$$\text{subject to} \quad \boldsymbol{y} = \boldsymbol{C\Psi s} \tag{2.54}$$

where $\boldsymbol{s}$ is a vector of coefficients and $\boldsymbol{y}$ is the captured information [17].

Chapter 3

Previous Work

Although the subject of this thesis is signal reconstruction, the end goal is to use compressed signals for predictive maintenance. Therefore a look at the work regarding predictive maintenance is given alongside the work on signal reconstruction.

In the field of predictive maintenance, there have been multiple attempts at making accurate algorithms, and the techniques vary equally as much as the results. First off is Wang and Vachtsevanos [48] who try to establish a framework, or a set of metrics, for testing future predictive maintenance algorithms. After reading multiple papers on the subject it does not seem their metrics became industry standard. In the same paper, they also compared two methods for maintenance prediction; Wavelet Neural Network (WNN) and an AR model. WNN is an FCNN with a wavelet decomposed signal as input. Great results were achieved with the WNN model and other attempts of using wavelets in different ways have also been proposed in [15, 49, 50].

Instead of using wavelets with a Neural Network, Qiu *et al.* [15] use wavelets to both filter out noise from signals and enhance weak periodic signals. Enhancing weak periodic signals enables earlier detection of faults, which their approach is successful at. Deng *et al.* [49] use wavelets in the form of Empirical Wavelet Transform (EWT) to extract AM-FM components of a signal. The method is used as a first step in the prepossessing of the signal before predicting using SVM. Finally, Rezamand *et al.* [50] propose a method where the first step is to feed raw data through an ensemble of Neural Networks for regression, then the output is fed through an online PCA called Recursive PCA (RPCA). Wavelets are then used to estimate the Probability Density Functions (PDFs) of a signal state, thus predicting faults as a likelihood threshold. The methods using wavelets all seem promising.

More traditional methods have also been proposed including feature extraction from a HT as proposed by Durbhaka and Selvaraj [51]. The features extracted are classified using different techniques including SVM and kNN, where the latter achieve 87% accuracy on fault detection. Baptista *et al.* [52] use features generated by an ARMA model alongside other statistical features, like kurtosis and skewness, as input to a data-driven model. They also test different classification

models; NN, random forests, SVM, and LR. Their baseline is the Weibull distribution which all methods beat except the NN. A method as interesting as it is simple is explored by Hong and Zhou [16], who simply predict the future Root-Mean-Square (RMS) values using GPR. The authors show good results only by thresholding the RMS value, but it is evident that this approach is highly fitted to their chosen data set, thus a dynamic way of selecting a threshold should be explored.

Looking at modern methods two papers should be mentioned. Firstly, Amihai *et al.* [53] are using RNN for computing time-series data in a bidirectional manner. Feeding data forward through one RNN model, then back through another captures more information than traditional RNNs, they claim. Their technique uses entity embedding and categorical metadata in parallel and they claim good results, although hard to verify. An industrial case study was also done by Amihai *et al.* [54] claiming good results, but proprietary algorithms make it hard to verify. For a more black-box approach, Ismail Fawaz *et al.* [31] present multiple known ML methods modified to classify time-series data. It is maybe the most interesting paper as it tests and compares modern architectures like CNN, ResNet, and MPN. All the architectures and corresponding theories are thoroughly explained and illustrated as well as the experiment results. The code used for all their experiments is also publicly available [1] making it easy to verify. Their paper was the basis for the literature study leading up to this master thesis. A full rundown of all the theories and a more in-depth rundown of the papers are given there [14].

Compression of signals has also been a topic of high interest for many decades. The goal is to remove redundancy and only represent information necessary to recreate the data. Recent papers on compression include Krishnaraj *et al.* [55] who try to compress underwater pictures using CNNs in combination with Discrete Wavelet Transform (DWT). Their goal is to extract the most prominent features of an image before finding a sparse representation of those features using DWT. Transmission of the signal can then be done with low energy consumption. Reconstruction on the signal is done with the same setup in reverse and their results are promising. Azar *et al.* [56] use DWT and a compressor they call *squeeze* to compress multivariate time-series data. Reconstruction is done with the same system in reverse except for a DNN at the end. They compare the results to compressed sensing, among others, and claim good results. Glaws *et al.* [57] are using deep convolutional autoencoder networks with skip-connections as described in subsection 2.3.12, to compress and reconstruct data during Computational Fluid Dynamics (CFD) simulations. The performance of their model is good and they argue autoencoders are a good choice of compression compared to optimization-based techniques. When autoencoders are trained the computation time and the compression size is known, thus one can guarantee a result to some criteria. Such attributes might not be certain when compression and reconstruction are done using optimization. Helal *et al.* [58] also use a version of convolutional autoencoders to compress seismic data in the same manner as Glaws *et al.* [57] with CFD

[1] https://github.com/hfawaz/dl-4-tsc

simulations. Helal *et al.* [58] also report good compression ratios.

Chapter 4

Methodology

A description of the different methods and processes executed during the experiments will follow. All computation was done on a computer with the technical specifications given in Table 4.1.

Hardware	**Type**
CPU	AMD Ryzen 1700x
RAM	DDR4 16GB
GPU	NVIDIA GTX1070 6GB
Software	**Version**
Ubuntu OS	20.04.3 LTS
Python	3.7.11
GPU-driver	460.91.03
CUDA	11.3
cudatoolkit	11.3.1
cvxpy	1.1.8
pylbfgs	0.1.3

Table 4.1: Relevant hardware and software specifications

4.1 Disruptive Technologies Sensors

DT sensor products are made in a form factor of 19x19x2.5mm with a dual-sided sticker to fasten them to objects. A picture of some sensors is shown in Figure 4.1. They use a proprietary ultra-low power microprocessor developed in-house to control the sampling of a sensor and communication with the cloud. Different DT sensors can measure variables like temperature, humidity, and presence among others and their product line is ever increasing. All sensors will, by default, make one transmission of measurement(s) at 15-minute intervals. With their innovative technology and techniques, DT sensors sip $20nA$ at idle providing a battery life of approximately 15 years. Use cases for such a product are endless, but some

worth mentioning are remote monitoring of the transmission grid [1], monitoring cold storage for food safety and preventing waste [2], and desk occupancy for free seating office spaces [3].

Figure 4.1: An illustration of some sensors from the Disruptive Technologies product line

Source: `https://www.disruptive-technologies.com/`

There is a sensor prototype from Disruptive Technologies which can measure vibrations. Although not directly used in this thesis, all experiments are carried out with their use cases in mind. From internal documentation, one can read that the product uses a three-axis Micro-Electro-Mechanical Systems (MEMS) accelerometer to measure the frequency of the vibrations of an object. Its sample rate is 1100 hz which implies, at least in terms of Nyquist-Shannon sampling theorem, ≈550hz is the highest measurable frequency. A wireless vibration sensor with a battery life of 15 years should be of high value for monitoring and predicting faults in old machinery.

The size of data sent to the cloud is kept to a minimum for two reasons; transmission is the most power-consuming action in the sensor, and storage of data is limited. For the vibration sensor, this means that samples are taken for 1 second at a time every 15. minutes. A spectrum is generated using FFT of which the 5 frequencies with the highest magnitude are sent to the cloud.

4.2 Dataset

A bearing dataset from Intelligent Maintenance Systems[4] is used to test the compression methods. Four force lubricated double row Rexnord ZA2115 bearings were placed individually in a row of housings. A shaft was placed through all the four bearings and coupled to an AC motor, spinning at a constant 2000RPM. Force was applied to the shaft as a radial load of 2721 kg.

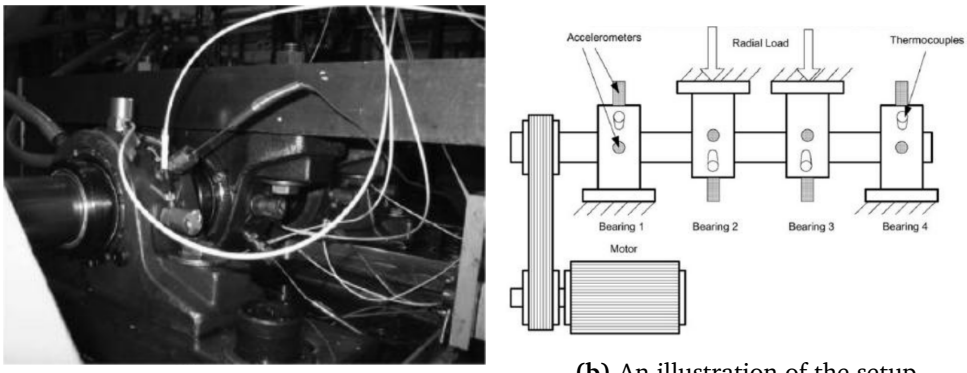
[1] `https://datascience.statnett.no/tag/disruptive-technologies/`

[2] `https://www.disruptive-technologies.com/blog/global-restaurant-chain-saved-1.25m-in-food-with-sensor-technology`

[3] `https://www.disruptive-technologies.com/blog/how-sensor-technology-is-helping-design-the-workspace-of-th`

[4] `https://ti.arc.nasa.gov/c/3/`

PCB 353B33 Quartz accelerometers are placed orthogonally on the bearing housing, capturing vibration measurements with a sampling frequency of 20kHz. Each sequence is a continuous measurement for 1s and is taken at 10-minute intervals. Both a picture and an illustration of the setup are shown in Figure 4.2.



(a) A picture of the setup

(b) An illustration of the setup

Figure 4.2: Setup for measuring bearing vibrations

Source: [15]

Thus, each bearing has two channels of vibration data stacked. All the 2 channels$*$ 4 bearings $=$ 8 sequences are stacked column-wise in one file per 1s measurement. There are three sets of sequences corresponding to three distinct runs of the experiment, each a run-to-failure experiment. Three different failures were detected during the runs; inner race failure, outer race failure, and roller element failure. The different parts of a roller bearing is shown in Figure 4.3. Multiple faults were detected during the tests:

- Set 1: Inner race defect in bearing 3 and roller element defect in bearing 4
- Set 2: Outer race failure in bearing 1
- Set 3: Outer race failure in bearing 3

A preview of the data is shown in Figure 4.4. The vibrations from bearing 4, set 1 are shown at a healthy state and with severe failure, both raw measurements and a spectrum from Fourier Transform are displayed.

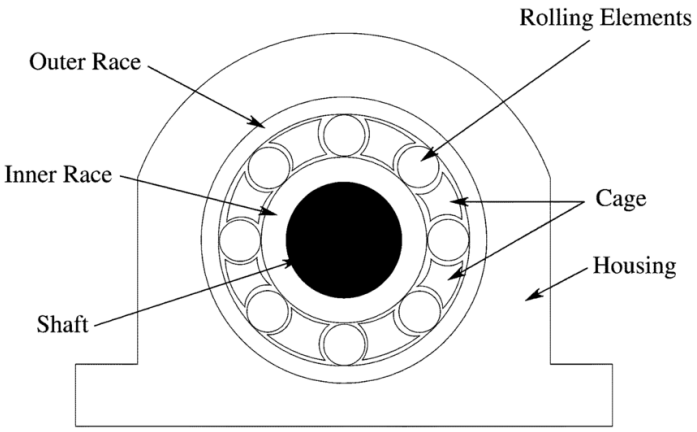


Figure 4.3: An illustration of the different parts of a roller bearing. In this figure, the outer race is fastened to the housing. The rotating part is the axle which is held by the inner race.

Source: `https://www.researchgate.net/figure/Typical-roller-bearing-showing-different-component-parts_fig1_3413987`

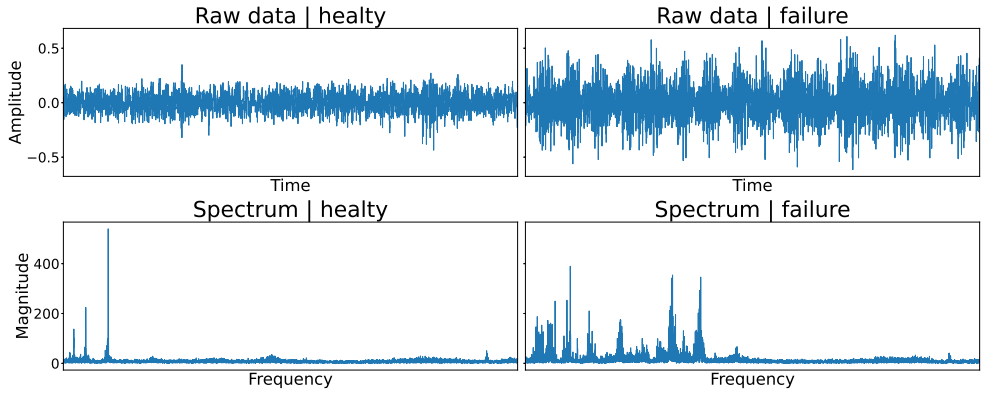


Figure 4.4: The plots are from set 1 bearing 4, showing raw measurement data over time with the corresponding frequency spectrum. The left sample column displays a healthy state and the right displays a severe fault in the bearing.

4.2.1 Formatting the Data

To generate the dataset, each file is loaded and its content separated by channels. Each channel is stored in separate files which are named corresponding to the test set, the time of sampling, and the channel it came from. For example, *1st_test_2003.10.22.12.09.13_CH5.csv* is the first experiment, at the specified date and time, and it is from the fifth channel. All the files were shuffled and split into a training set containing 80% of the data, or 30525 files, and a test set containing the remaining 20% of the data, that is 7700 files.

4.2.2 Recreating the Existing DT method

As touched upon in section 4.1, the current sensor implementation samples for one second. The samples are then Fourier transformed to make a spectrum, from which the 5 frequencies with the biggest magnitude are selected. These frequencies are transmitted to the cloud where they can be accessed by the user. An illustration of the procedure is shown in Figure 4.5.

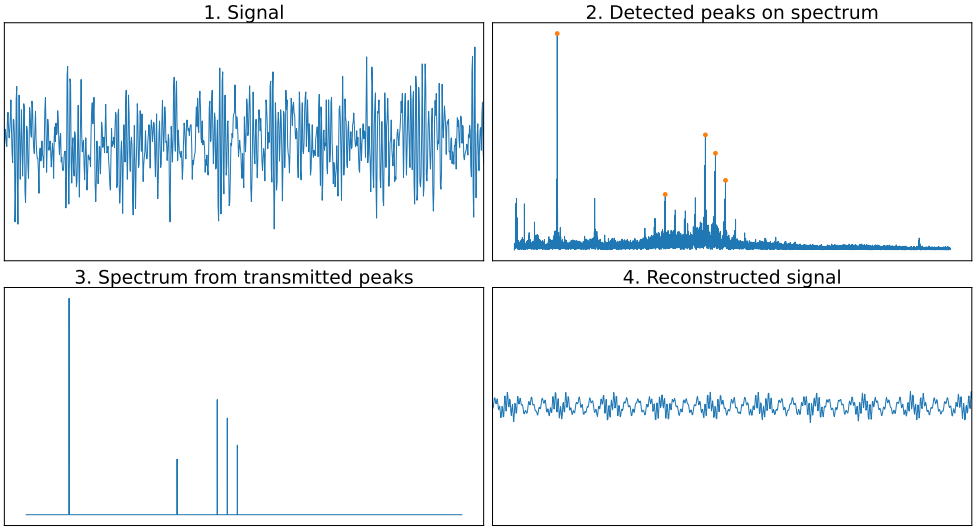


Figure 4.5: In the upper-left plot, a vibration signal has been loaded. The spectrum of the signal is calculated with FFT(upper-right) before the peaks are located and selected (orange dots). The peaks are transmitted and projected onto an array of zeros, illustrated in the lower-left plot, generating a new array of Fourier coefficients. Inverse FFT on the new coefficient array yields a reconstructed signal (lower-right)

To compare the usefulness of the information given by the DT method with the other methods, the procedure had to be recreated. Since the sensor only samples at 1100hz the 5 peaks from the sensor must be in the range of 0 to 550hz, but for the sake of experimenting two more tests were executed. Selecting 5 peaks from the full 10000hz-range might make for a more comparable experiment regarding the

signal reconstruction. With that said, 5 points is a lot less information to send than the smallest set of points tested with the other methods. Therefore, the third test was selecting 2000 points from the full 10000hz-range, which is a more realistic comparison in terms of the information transmitted, $\approx$ 10% of the samples. The three tests are:

1. Selecting 5 peaks in the range 0hz to 550hz
2. Selecting 5 peaks from the full 10000hz-range
3. Selecting 2000 peaks from the full 10000hz-range

The dataset is already consisting of time-series data spanning 1 second each, so the following pseudo-code illustrates the steps implemented for a signal x:

1. fft_x = FFT of x
2. numpy.find_peaks of fft_x, which yields an array of indices pointing to peaks
3. sort the indices with respect to peak magnitude in declining order
4. copy n first indecies with corresponding magnitudes
5. allocate an array of zeros with the size of fft_x
6. project the selected points onto the array of zeros
7. inverse FFT if necessary

4.2.3 Autoencoder Architecture

A simple network architecture was used consisting of fully connected layers with ReLU activation in between. The implementation was made so that the depth of the network could be changed making it easier to test multiple compression ratios. Each model compression ratio is displayed in Table 4.3. The architechture of model 1 is shown in Table 4.4, model 2 in Table 4.5, and model 3 in Table 4.6.

The training dataset described in subsection 4.2.1 was used for training the models and the final evaluation was then done using the test set. All the architectures was trained using the specifications shown in Table 4.2.

Specification	**Name**	**Parameters**
Optimizer	Adam	Learning rate = 0.1, weight decay = 0.8
Loss function	MSE	n/a

Table 4.2: The training specifications for the autoencoders

Layer #	**Compression ratio**
1	$16384/8192 = 2:1$
2	$16384/2048 = 5:1$
3	$16384/256 = 64:1$

Table 4.3: The compression ratios for the autoencoders

Layer	Layer Type	In features	Out features	Comment
1	Linear	16384	8192	-
2	Linear	8192	16384	-

Table 4.4: Autoencoder with one layer, which essentially is linear combinations as there are no activation layers

Layer	Layer Type	In features	Out features	Comment
1	Linear	16384	8192	-
2	Activation	-		ReLU
3	Linear	8192	2048	-
4	Linear	2048	8192	-
5	Activation	-		ReLU
6	Linear	8192	16384	-

Table 4.5: Autoencoder with two layers

Layer	Layer Type	In features	Out features	Comment
1	Linear	16384	8192	-
2	Activation	-		ReLU
3	Linear	8192	2048	-
4	Activation	-		ReLU
5	Linear	2048	256	-
6	Linear	256	2048	-
7	Activation	-		ReLU
8	Linear	2048	8192	-
9	Activation	-		ReLU
10	Linear	8192	16384	-

Table 4.6: Autoencoder with three layers

4.2.4 Decoder Architecture

During the experiments the decision was made to only decode a set of random samples. The reasoning for this choice is that IoT sensors, DTs in particular, has extremely limited computing power. To make a more relevant comparison between Neural Networks and compressed sensing the architectures specified in Table 4.8, Table 4.9, and Table 4.10 were implemented. For each 1-second sequence, a random subset of samples was used as input to the model. The size of this subset corresponds to the size of the input layer of the model in question, thus the compression ratios are the same as shown in Table 4.3. All the architectures was trained using the specifications shown in Table 4.7.

Specification	Name	Parameters
Optimizer	Adam	Learning rate = 0.1, weight decay = 0.8
Loss function	MSE	n/a

Table 4.7: The training specifications for the decoders

Layer	Layer Type	In features	Out features	Comment
1	Linear	8192	16384	-

Table 4.8: Decoder with one layer
Decoder with one layer, which essentially is linear combinations as there are no activation layers

Layer	Layer Type	In features	Out features	Comment
1	Linear	2048	8192	-
2	Activation	-		ReLU
3	Linear	8192	16384	-

Table 4.9: Decoder with two layers

4.2.5 Compressed Sensing Implementation

Since there is no training involved in compressed sensing only the test set was used in this part. Experiments were done using two different packages in Python, CVXPY and PyBFGS, and all with comparable compression ratios to what is described in subsection 4.2.3; 100 : 1, 10 : 1 and 2 : 1. Both algorithms were used with an orthogonal cosine basis.

First, the CVXPY pack was used to do l_1 optimization with the ECOS solver described in subsection 2.4.2. During the implementation, it became evident that the solver is too slow to process all the 7700 test samples in a reasonable time. To illustrate the problem; it took $\approx$ 1.1 minute to converge for one sequence with a sample size of 1% and almost 1 hour for a sequence with a sample size of 10% to

Layer	Layer Type	In features	Out features	Comment
1	Linear	256	2048	-
2	Activation	-		ReLU
3	Linear	2048	8192	-
4	Activation	-		ReLU
5	Linear	8192	16384	-

Table 4.10: Decoder with three layers

converge. A subset containing 4 test samples was made just to get some results, the same files have also been tested separately on the other models for comparison. When starting to reconstruct from a 50% sample size the optimization stopped due to insufficient RAM. In the chapter 5 only sample sizes of 1% and 10% will be shown for that reason.

The other Python package, PyBFGS, containing the OWL-QN was used to speed up the processing. With this method, convergance was achieved in 1 to 2 seconds, depending on the sample size. It took 11.7 hours to compute all the data; 7700 test sequences × 3 different sample sizes = 23100 optimizations. An *evaluation function* is required by the solver which must return the computed loss $l(x)$ and its gradient. After some trial and error (partially due to non-existent documentation) an evaluation function was implemented with a least squares-approach. A cosine space was chosen as a sparsifying basis due to its widespread uses in multiple fields of engineering. The Discrete Cosine Transform (DCT) is highly related to Fourier Transform, using only the real numbers. Given a solution x from the solver, a gradient vector g, and the random samples b, the two functions calculated are

$$l(x) = ||\boldsymbol{A}x - b||_2^2, \tag{4.1}$$

$$\Delta l(x) = 2\boldsymbol{A}^T(\boldsymbol{A}x - b). \tag{4.2}$$

To be clear; the sum of squares in Equation 4.1 will be regularized by the l_1-norm (2.49) in the solver. An implementation was made with the following steps.

1. Inverse-DCT(x), which yields $\boldsymbol{A}x$
2. Compute the residual $r = \boldsymbol{A}x - b$, the result is of same size as b
3. Compute the sum of squares of the r vector, the value is returned as the evaluation
4. Make an array of zeros, Axb, with the same dimensions as x
5. Project r onto the empty matrix Axb
6. The gradient (4.2) is calculated by DCT(Axb) which yields $\boldsymbol{A}^T(\boldsymbol{A}x - b)$
7. Multiply the result in *5.* by 2 and add the result to the gradient vector g

4.2.6 Evaluation Metrics

A set of metrics will be used to evaluate the effectiveness of the different methods. Some qualitative considerations of the spectral envelope will also be discussed

alongside the RMS over time. The metrics are known and considered useful for this purpose [59]. For the qualitative evaluation, a difference in RMS of each time-series sample will be considered as well as the difference in spectrum.

The selected metrics are *covariance* and *MSE*. Covariance quantifies the relationship between two variables and is calculated as

$$cov(x, y) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})(y_i - \bar{y}) \tag{4.3}$$

where $\bar{x}, \bar{y}$ is the mean of the variables. The result can be any number between $-\infty$ and ∞, a greater number means a stronger relationship.

MSE measures the difference between the original signal and the reconstructed signal. MSE is calculated as

$$MSE(x, y) = \frac{\sum_{i=1}^{N} (x_i - y_i)}{N}, \tag{4.4}$$

Chapter 5

Results

The test results are presented in different ways to enable a proper evaluation. First, a qualitative approach to representing the results are given before some quantitative results will be presented.

A qualitative illustration of the test results is given with plots from Figure 5.1 to Figure 5.15. The two same samples have been used throughout these plots; test 1, channel 8 with the healthy sample *2003.10.22 12:39:13* and the end-of-life/failure sample *2003.11.25 23:39:56*. To the left in each plot is the same window of the time-series data of the healthy sample, then comes the spectrum of the healthy sample in the middle and the failure sample to the right. Figure 5.1 is the original signals from the dataset, thus all methods are compared to this.

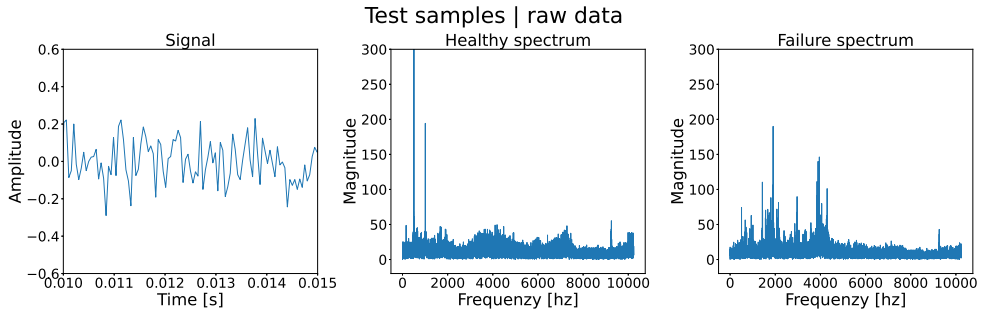


Figure 5.1: The original signals before any compression

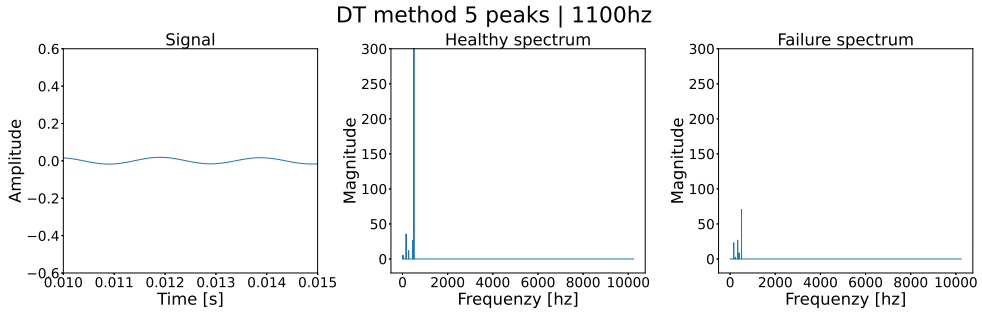


Figure 5.2: The reconstructed signals using a DT method with 5 peaks at 1100hz

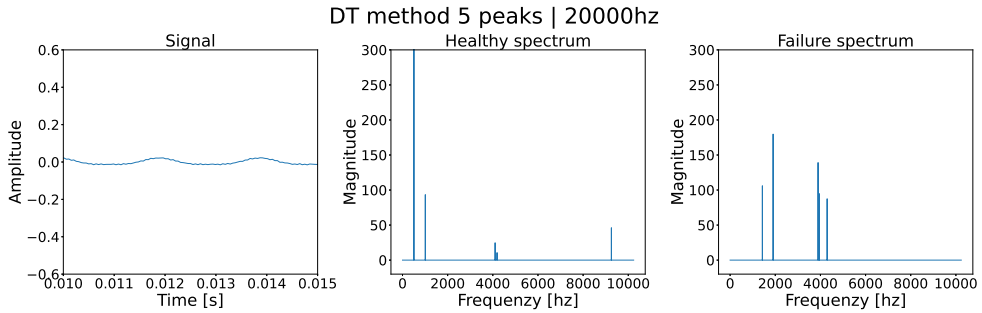


Figure 5.3: The reconstructed signals using a DT method with 5 peaks at 10000hz

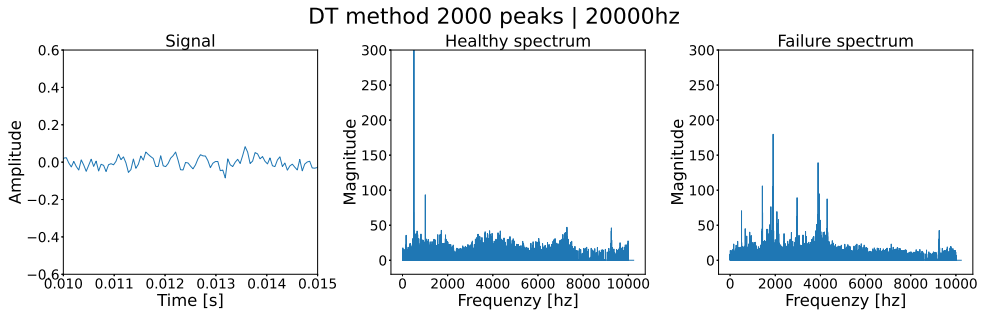


Figure 5.4: The reconstructed signals using a DT method with 2000 peaks at 10000hz

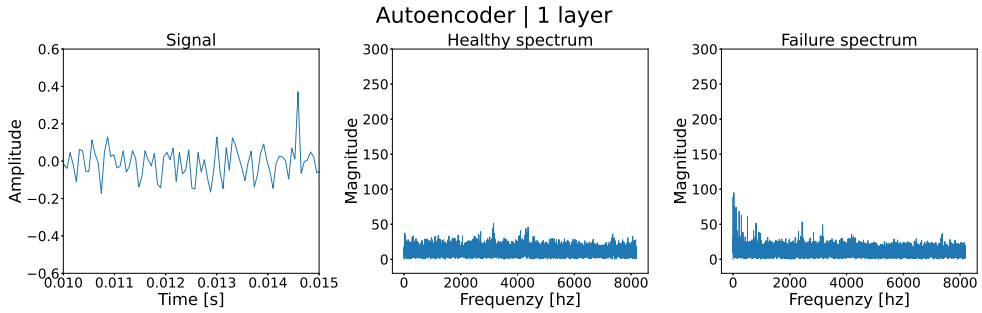


Figure 5.5: The reconstructed signals using a single-layer autoencoder

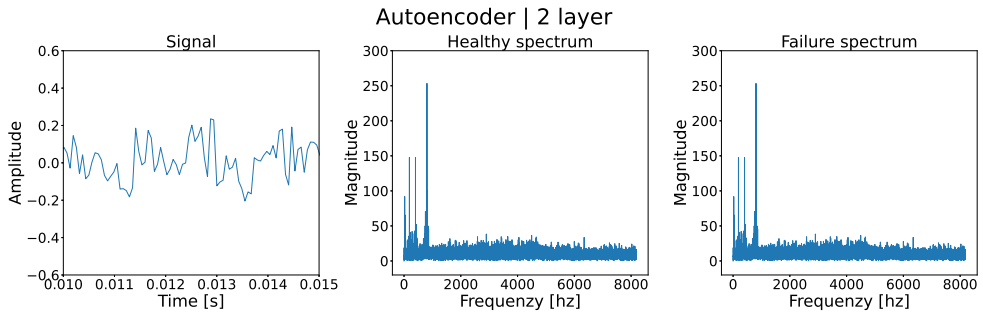


Figure 5.6: The reconstructed signals using a two-layer autoencoder

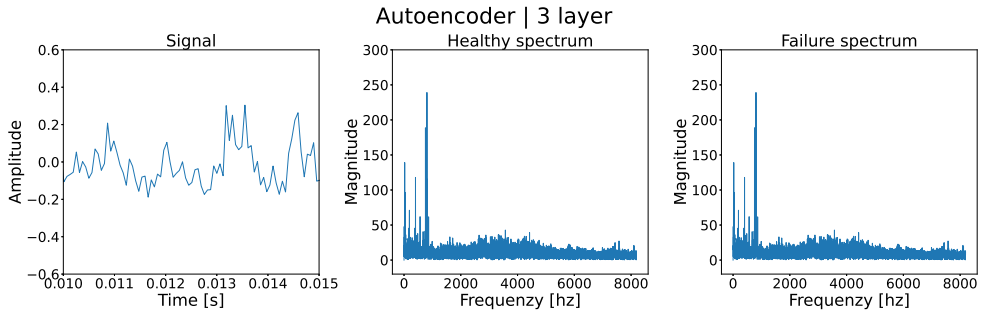


Figure 5.7: The reconstructed signals using a three-layer autoencoder

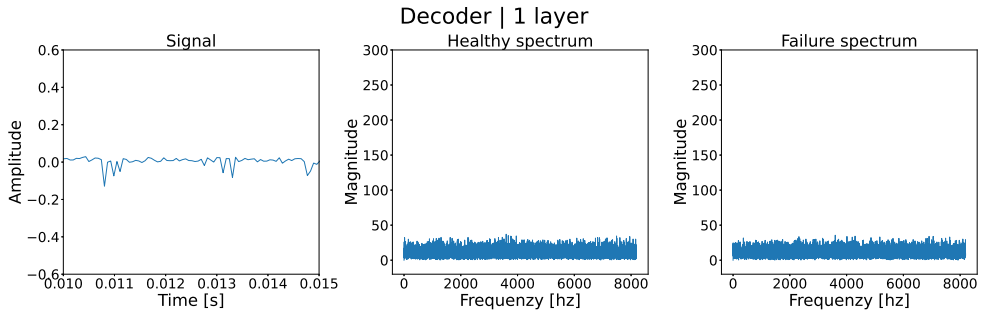


Figure 5.8: The reconstructed signals using a single-layer decoder

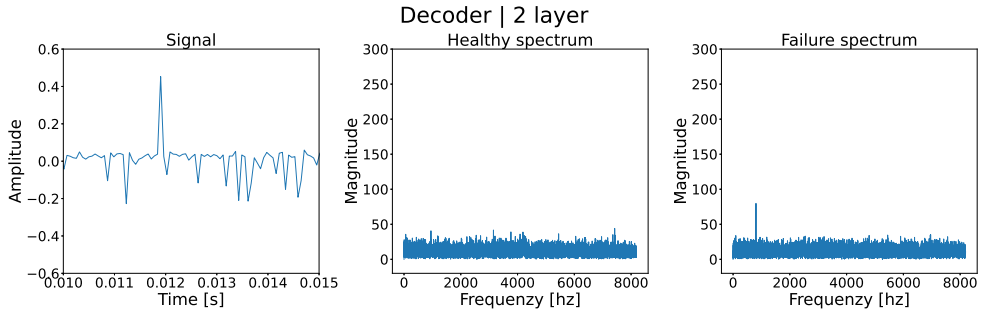


Figure 5.9: The reconstructed signals using a two-layer decoder

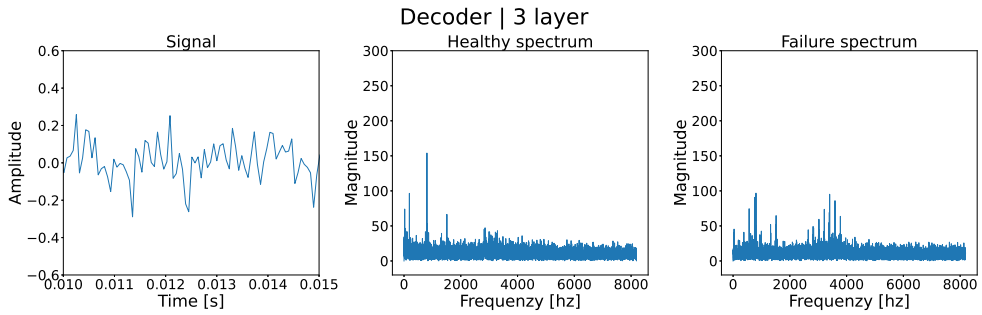


Figure 5.10: The reconstructed signals using a three-layer decoder

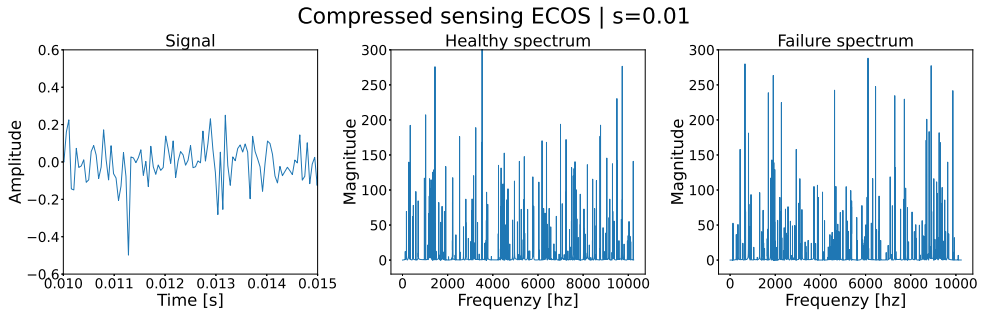


Figure 5.11: The reconstructed signals using ECOS with a sample size of 1%

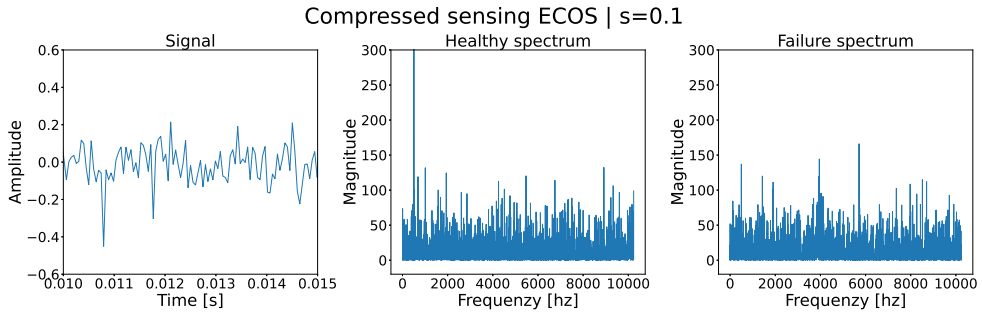


Figure 5.12: The reconstructed signals using ECOS with a sample size of 10%

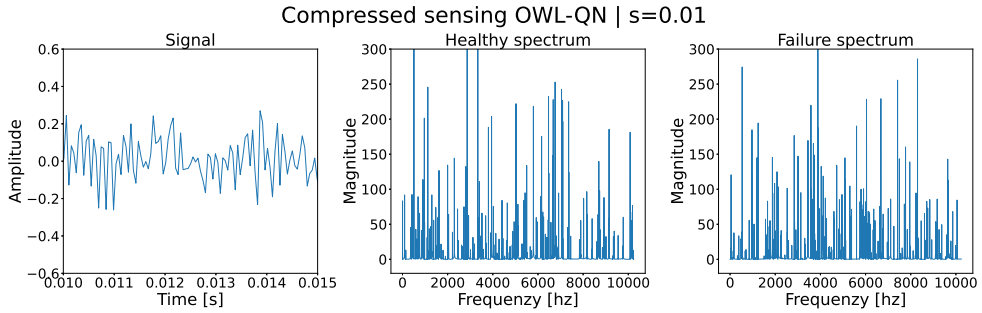


Figure 5.13: The reconstructed signals using OWL-QN with a sample size of 1%

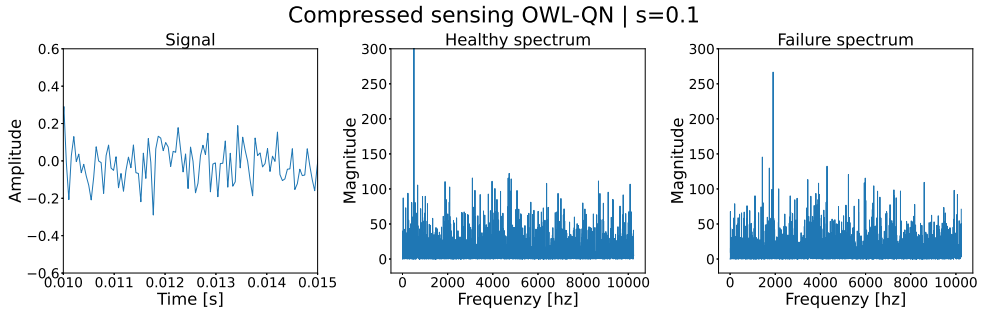


Figure 5.14: The reconstructed signals using OWL-QN with a sample size of 10%

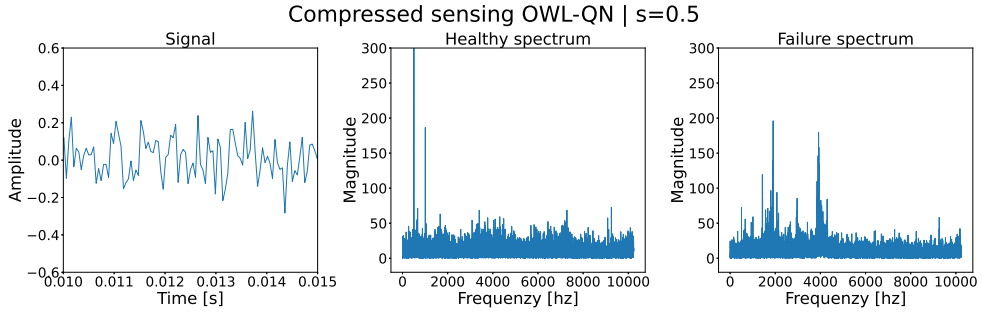


Figure 5.15: The reconstructed signals using OWL-QN with a sample size of 50%

The quantitative representation of the results are illustrated by computing the RMS of each test-sample during a test and plotting them over time. The same is done with each compression method. Test 1 and 2 is used for this with channel 8 and channel 1 respectively. Channel 1 in test 2 shows sever degradation of the bearing, thus it is a useful example for evaluation. The plots are shown in Figure 5.16 to Figure 5.19.

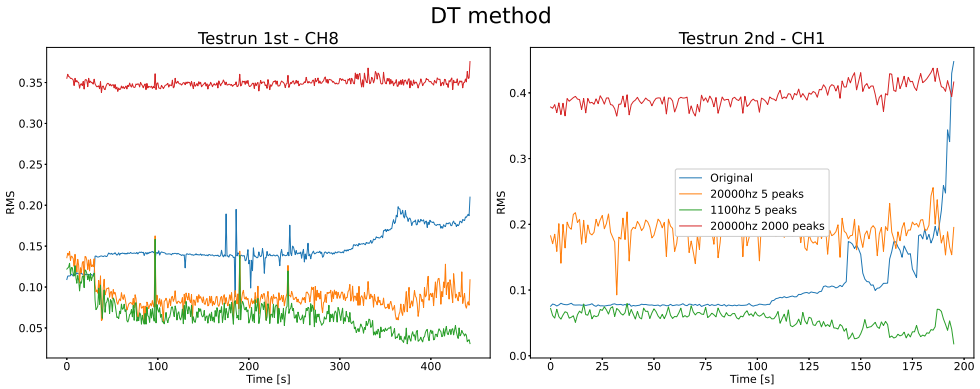


Figure 5.16: RMS over time using DT methods

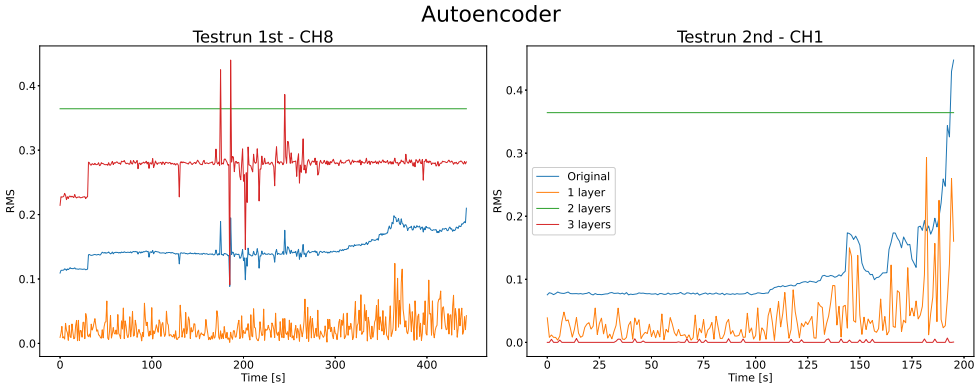


Figure 5.17: RMS over time using autoencoders

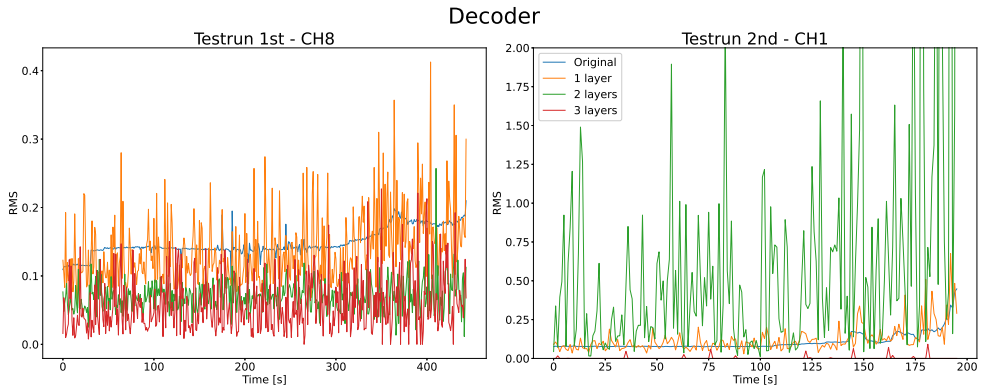


Figure 5.18: RMS over time using decoders

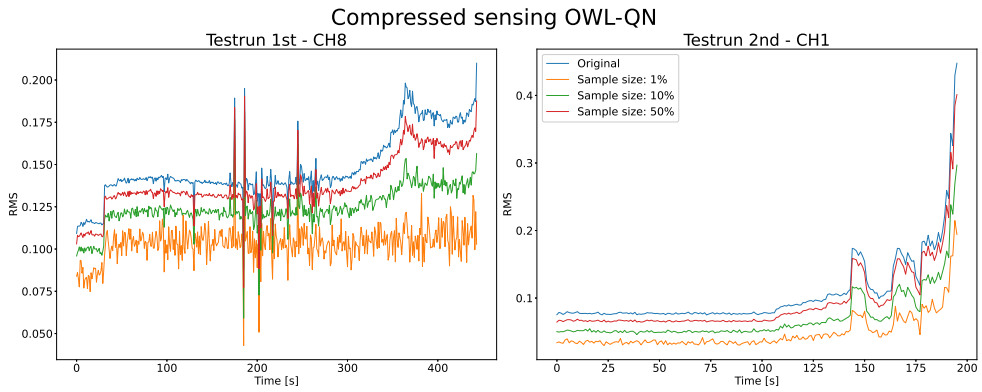


Figure 5.19: RMS over time using compressed sensing with OWL-QN

MSE and covariance were calculated for each sample. The channel-wise mean of the values was then stored for every combination of method, method specification, and test from the dataset. The results are plotted for illustration in Figure 5.20 to Figure 5.31. Two points should be known to the reader before continuing; (1) some bars climb outside of the plotted range, which has little to no impact on the discussion of the results. Should there be a need to raise the results beyond any shadow of a doubt, the exact numbers can be found in Appendix A. (2) The y-axis is scaled logarithmically.

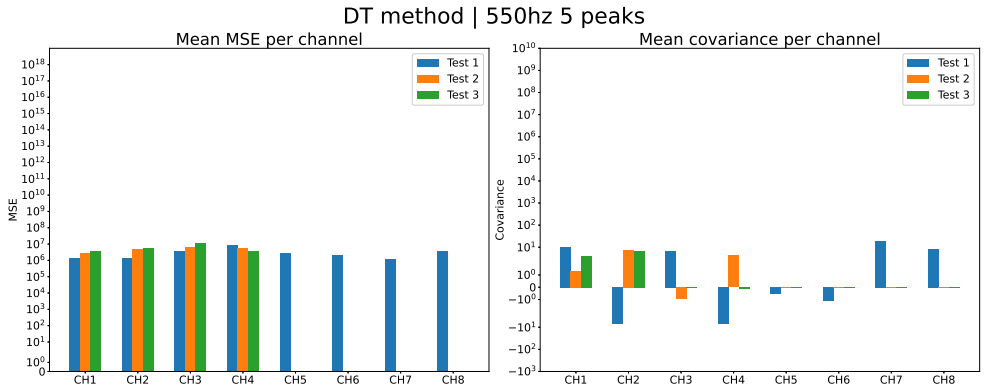


Figure 5.20: MSE and covariance for DT method with 5 peaks at 550hz

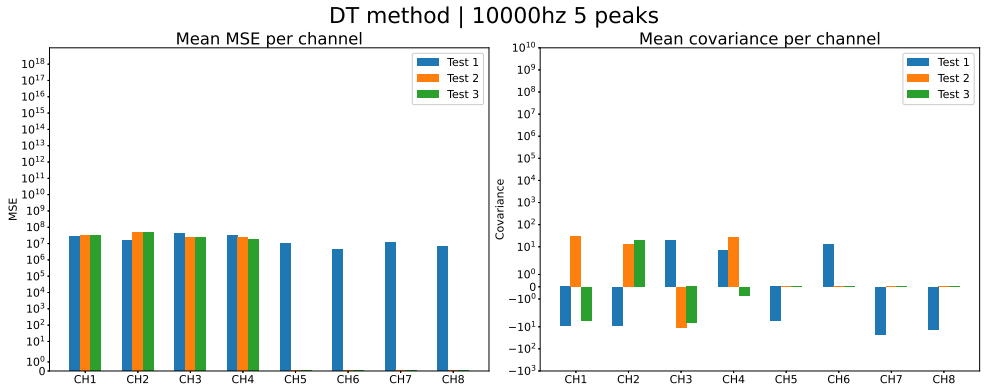


Figure 5.21: MSE and covariance for DT method with 5 peaks at 10000hz

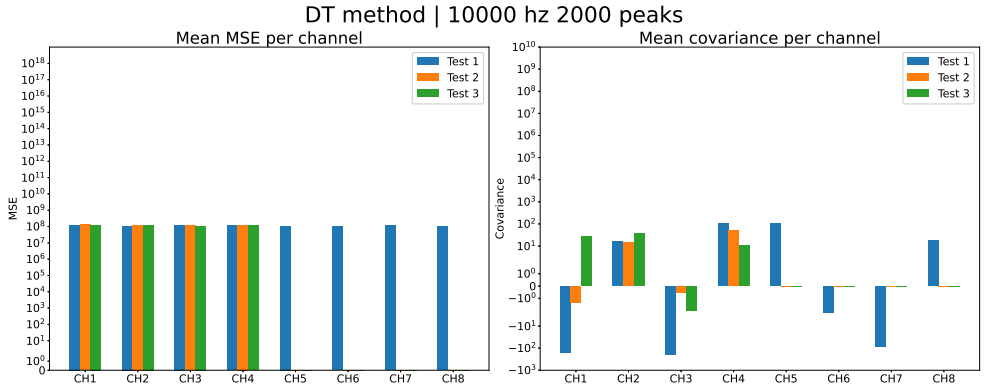


Figure 5.22: MSE and covariance for DT method with 2000 peaks at 10000hz

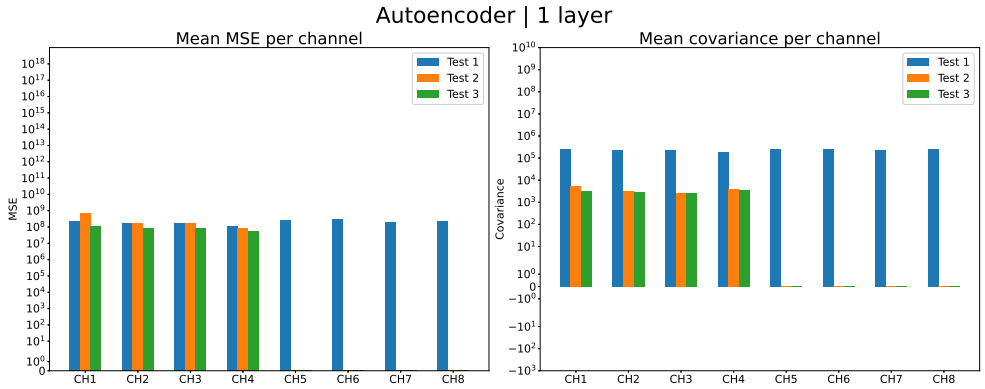


Figure 5.23: MSE and covariance for autoencoder with 1 layer

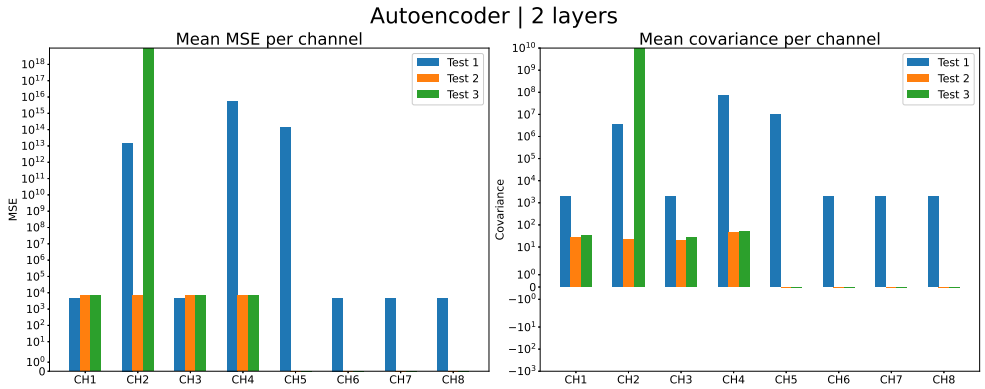


Figure 5.24: MSE and covariance for autoencoder with 2 layers

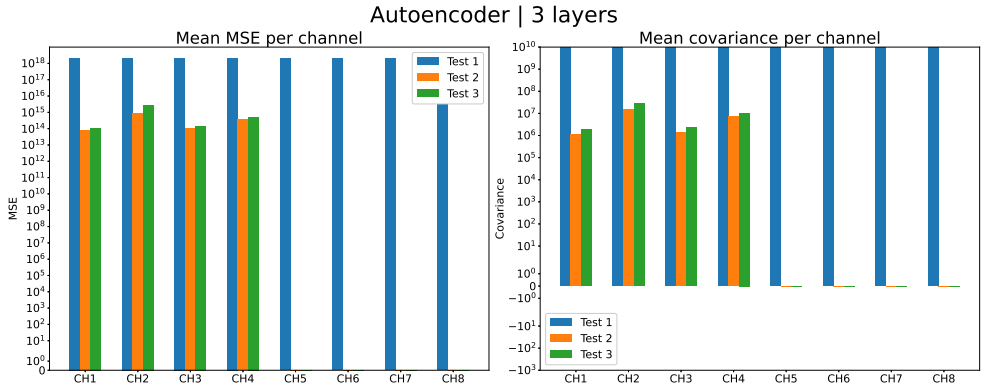


Figure 5.25: MSE and covariance for autoencoder with 3 layers

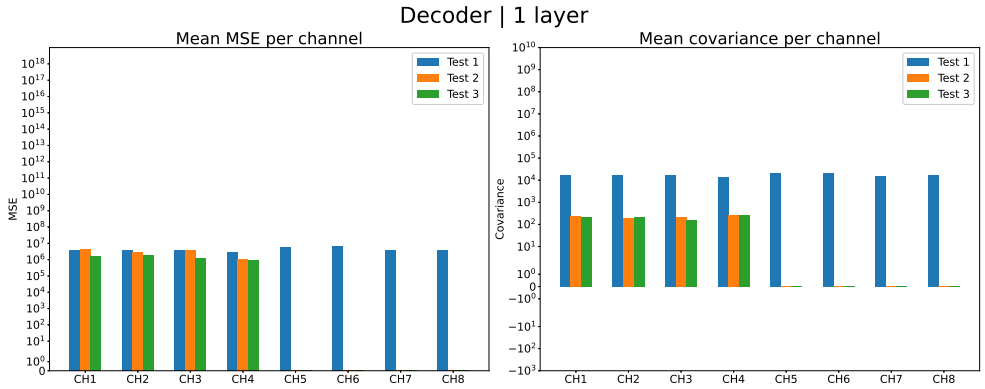


Figure 5.26: MSE and covariance for decoder with 1 layer

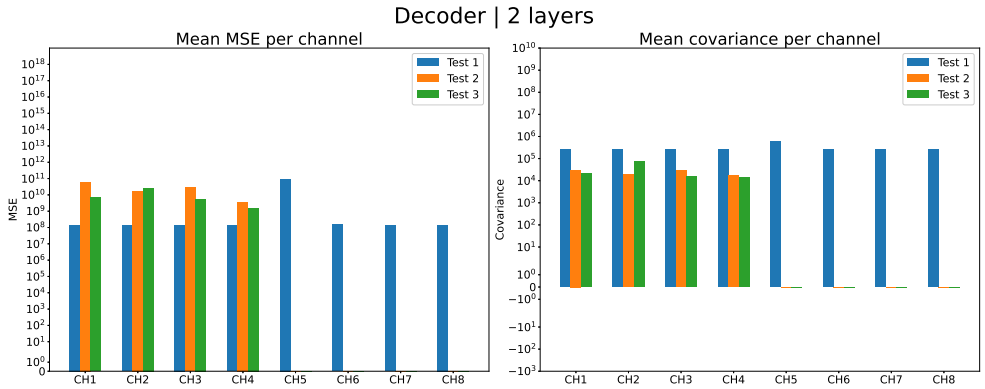


Figure 5.27: MSE and covariance for decoder with 2 layers

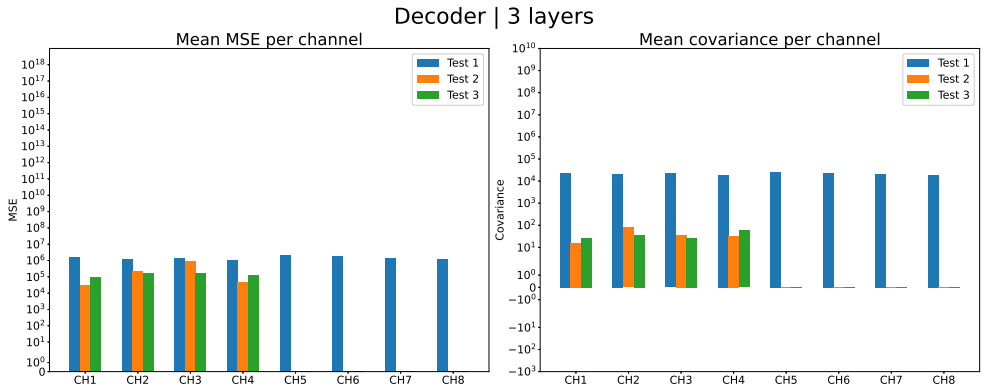


Figure 5.28: MSE and covariance for decoder with 3 layers

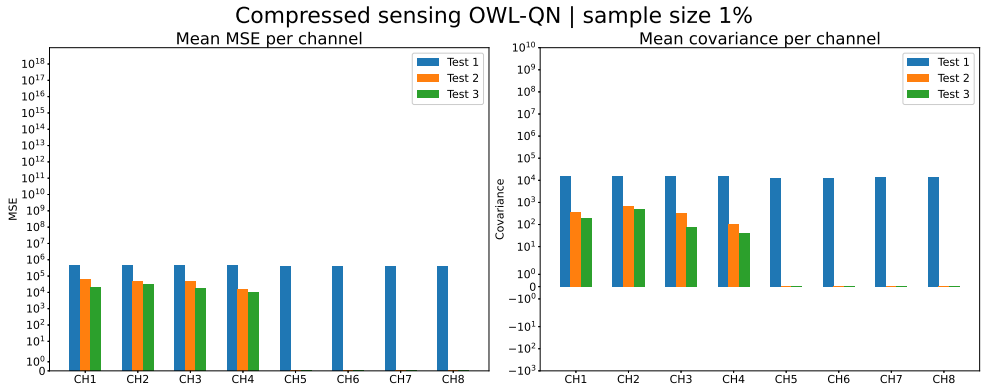


Figure 5.29: MSE and covariance for compressed sensing with a sample size of 1%

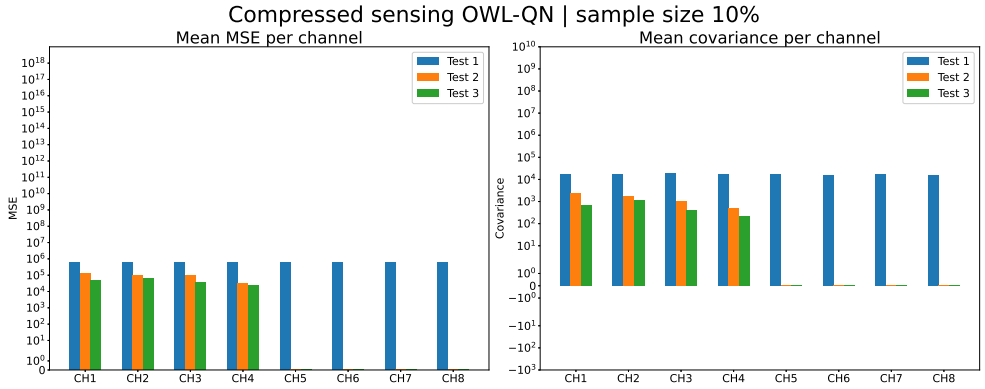


Figure 5.30: MSE and covariance for compressed sensing with a sample size of 10%

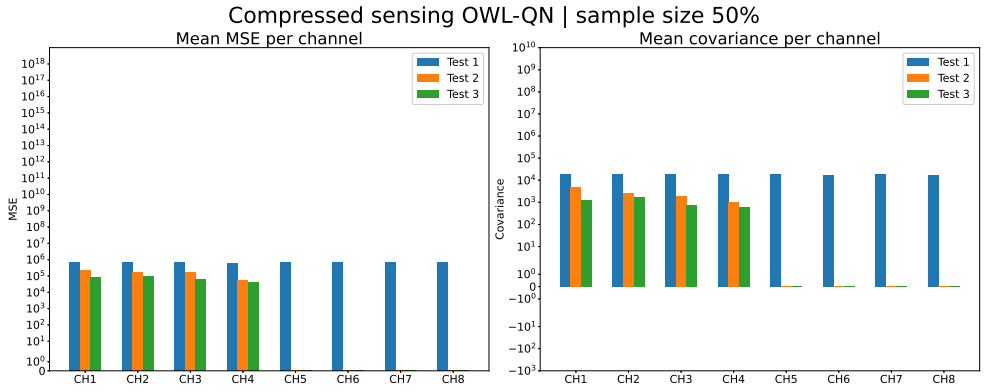


Figure 5.31: MSE and covariance for compressed sensing with a sample size of 50%

Chapter 6

Discussion

There has been a lot of work on both predictive maintenance and compression of signals, as described in chapter 3. Detecting early signs of failure in industrial equipment is important to increase revenue and safety, and some methods already work adequately. Although research in this area must be continued, there is another problem emerging from the literature related to measuring the world: all the sensors needed to collect and transmit the measurements must be operated. To reduce the cost of installation and operation, both the sensor itself and the transmission of signals must be done in an energy-efficient way. Disruptive Technologies have already developed a miniature platform for ultra-low power wireless IoT devices. What is left then is to find better ways of transmitting the data. This leads up to the use of compression, which also has been an area of research for many years.

Advancements in computational hardware and optimization algorithms enable new ways of compression, and some recent papers have been mentioned. As the world is currently surfing the Machine Learning wave it is no surprise that all of them use a Neural Network of some sort. From these papers, it is clear that some form of autoencoder should be tested on the time-series data used in this thesis, but another highly relevant technique is mentioned by most of the mentioned papers; compressed sensing. Although many of them discuss compressed sensing, they all have one thing in common; they capture all the data points before compression. Thus, the most powerful feature of compressed sensing is not utilized; a sparse sampling of the data. Regarding an energy-limited wireless sensor, it should give two advantages; less energy being used to sample data and less information to transmit. Effectively moving most of the computation to a receiver. This is why experiments on different compression techniques, including compressed sensing, are explored in this thesis.

The results from the experiments are all considered in regards to two maintenance prediction algorithms touched upon in chapter 3. (1) Firstly, the novel method of Gaussian Process Regression presented by Hong and Zhou [16] only predicts the future path of the RMS values. A method that effectively yields a probability of the RMS exceeding a threshold. The RMS calculated from a received sig-

nal must contain some form of information that make it possible to calculate an RMS that at least follows a similar path to the actual RMS, and is preferably close in value. (2) The second maintenance prediction algorithm considered is presented by Qiu *et al.* [15]; using wavelets to enhance weak signatures corresponding to a failure, *the wavelet method* from now on. For this approach to work the received signal must contain some information that closely represents the actual signal. If a failure corresponds to some arbitrary frequency, that frequency must be seen in the reconstructed signal.

The current method implemented by Disruptive Technologies in their vibration sensor prototype might spark some initial concerns. Only selecting the 5 biggest peaks in a spectrum builds upon an assumption that the relevant features of a measured object will present themselves as frequencies of the highest magnitudes. A rather bold assumption to make on a general basis. Adding to this the small range of measurable frequencies, 0 to 550hz, one is entitled to question its usefulness, at least regarding the prediction of failure in bearings. Simply illustrated; if a bearing spins at 100 rounds per second, an outer race fault might show as a peak around 100hz. If it spins at 3000 rounds per second the fault might appear as a peak at around 3000hz. If the fault was in a roller element on the bearing is spinning at 100 rounds per second the peaks might appear at the rolling speed of the particular element. Intuitively, the range 0hz to 550 hz will not be able to capture such variance.

From the original signal in Figure 5.1, it seems like the progression from a healthy state to an end-of-life state is manifested as an increase in some frequencies that eventually dominate. Comparing this to the DT method in Figure 5.3 to Figure 5.4, one might think that the approach will work with the full frequency range. The frequencies of the highest amplitude are indeed preserved.

The biggest problem with only selecting 5 peaks comes with the wavelet method. As it tries to enhance the early signatures of a defect, the early signatures must be captured in the first place. One cannot rely on the DT method to extract the peaks representing these signatures early on, or at all. The DT method is not deterministic in what it extracts; a random set of frequencies that happened to have large magnitude at the time of sampling. This observation is substantiated when considering the MSE and covariance shown in Figure 5.21. An evenly large MSE can be observed across all channels. The relationship between the spectrum of the raw data and the spectrum from DT method has, at times, negative values. Two spectral envelopes that should resemble the same signal are far from resemblant when the mean covariance is negative. This underpins the claim of the method being non-deterministic, thus it is expected to not work with the wavelet method. When 2000 peaks are selected it might be plausible for the algorithm to work better as the spectral envelope is better represented. Again, looking at the covariance showing negative relationships; it might seem like no such predictions are possible.

When it comes to GPR, Figure 5.16 shows that a few Fourier coefficients are not enough to calculate the correct RMS values. An explanation is that the recon-

structed signals in Figure 5.2 to Figure 5.4 all have small amplitudes. All those small frequencies observed in the whole spectrum add up when removed. What might be more surprising is that even with 2000 peaks in the spectrum, the RMS value still shows little correlation with the path of the RMS of the actual signal. Certainly, none of the tested DT method alterations will work for RMS prediction using GPR.

Of course, one could select the 5 frequencies transmitted more smartly. An example might be to do a PCA to find the relevant frequencies for a given setup. There are multiple drawbacks of such an approach. One is the loss of generality; the relevant frequencies when predicting bearing failure are probably different from wind turbines. Another related one is that the signatures of faults should, preferably, be known, which also translates to a loss of generality of the method.

Moving on to alternative methods for compression. Autoencoders were first implemented and tested because of the promising results in previous literature. During testing a realization was made that autoencoders violate a premise in this thesis; the sensor itself has limited processing power and energy capacity. Compressing a signal using a NN on-device does fit the problem description. While continuing to test with the autoencoders, an alternative approach was made. By just using the decoder part of the network to reconstruct from some random samples, all the processing is done at the receiver. The decoder method can be viewed as a hybrid between compressed sensing and autoencoders.

With autoencoders one hopes the network will find some relationship in the data which, in this case, enables reconstruction from a smaller set of code. Normally, a network needs a high amount of parameters to capture the variance presented in the classifying or regressing domain, and so the results of the single-layer autoencoder are as expected. From Figure 5.5 there is an indication that the output is noise, close to random. With the number of transmitted samples halved, the result shows an unusable architecture which is to be expected. The network is only matrix multiplication.

Upping the number of layers to two and three also adds activation layers which might make the autoencoder capture non-linearities in the data. Evidently, from Figure 5.6 and Figure 5.7 it seems like the reconstruction became a tiny bit better; the amplitude of the signals is comparable and the spectral envelope show some resemblance, albeit far from representative. Keep in mind that these signals were reconstructed from $\approx$ 10% and $\approx$ 1% data respectively. With that said, the spectrum of the reconstructed signals does not display the frequencies found in the original data, thus none of the architectures are likely to work with the wavelet method. When it comes to GPR, Figure 5.17 neither the two-layer architecture nor the three-layer architecture manages to follow the RMS path. The single-layer version shows some resemblance to the path although quite noisy. Looking at the evaluation matrics in Figure 5.23 to Figure 5.25 it is clear that the MSE is high and all over the place. Covariance is low and equally all over. None of the autoencoder setups tested would be viable for neither GPR nor the wavelet method.

For the decoder approach, the single layer and two-layer architectures seem to

produce nothing but noise as seen in Figure 5.8 and Figure 5.9. The spectral envelopes of both are generally flat which means they most likely do not contain the information needed to detect failure signatures with the wavelet method. A three-layered decoder seems to capture some latent relationships in the data. Both the healthy spectrum and the failure spectrum, seen in Figure 5.10, have clear similarities with the original spectrum in Figure 5.1. Although the magnitude differs, it seems like the spectral envelope is preserved. Comparing the evaluation metrics in Figure 5.26 to Figure 5.28 one can confirm that the reconstruction error (MSE) is substantially lower for the three-layered decoder. Keeping in mind that the three-layered decoder uses the least amount of information, $\approx 1\%$, this observation is interesting. Without making any conclusions, one might view this as an example of how deeper NNs can extract and learn possible non-linearities. Regarding RMS prediction, Figure 5.18 shows that the single layer and the two-layer decoder seem to follow some path similar to the actual RMS, albeit extremely noisy. Because of this noise, it is hard to imagine that this method is useful for predicting RMS with GPR.

Compressed sensing starts off underwhelming. In Figure 5.11 and Figure 5.12 results from the ECOS solver are shown. It does not seem to have converged to a good solution as the spectral envelope of both scenarios does not resemble the actual spectral envelope. Combining this with the computation time leaves these results rather uninteresting, with its only contribution being to switch the solver algorithm.

When using the more advanced OWL-QN algorithm, the results are unlike anything that has been seen in this paper so far. At 1% of the samples taken, Figure 5.14, a familiar spectrum starts forming. Going up to 50% of the samples, in Figure 5.15, the similarity in the reconstructed spectral envelope is striking, visually at least. With these lightweight qualitative observations in mind, a look at the evaluation matrics in Figure 5.29 to Figure 5.31 is reassuring. Although neither MSE nor the covariance is remarkably better than for example the three-layer decoder, they are consistent. For each sample size across all tests and channels, the MSE sees little variance. The MSE is still substantial, but the spectral envelope can still be close to correct. As far as detecting weak signatures with the wavelet method go, it is the envelope that is important. These tests indicate that compressed sensing with OWL-QN is capable of providing the necessary details for failure prediction with highly sparse samples. When it comes to predicting the RMS the results are still highly positive. Looking at Figure 5.19 it is clear that even with a sample size of 1% the RMS paths closely align with only an offset. The reconstruction becomes really useful when the sample size increases to 10%. As seen in Figure 5.19, a clear pattern emerges that certainly is usable when predicting RMS values with Gaussian Process Regression.

Why compressed sensing using OWL-QN outperforms the other methods have not been explored in the experiments. The autoencoder and decoder methods should perform better than what was achieved in this thesis. With that said, one could question what a decoder network would learn. Some research indicates

that a single-layered autoencoder learns to find features comparable to Principal Components from PCA. Might the decoder learn some kind of equivalent to compressed sensing? The answer is probably *no*, as a trained NN does not perform optimization. Compressed sensing might be better for that exact reason; it is actively trying to find more information that fits what was given. The others do not.

Chapter 7

Further Work

From this paper there is a lot of possible work one can undertake as there are many questions still to be answered. Knowing how many years the battery will last if 1% to 10% of the data was transmitted would make for an interesting evaluation of its economical use-cases. The value of 15 years of too little data versus 5 years of accurate data might be highly dependent on the use case, but instinctively the latter sounds better.

On a more technical note, further alterations of the decoder could be explored. Although the architectures in this paper show poor performance, there might be other versions of the architectures that perform better. Examples include using a fully convolutional decoder, other activation layers, and/or more hidden layers.

In this thesis, optimizing for a sparse solution was regularized using the l_1-norm. Another implementation to try is an approximation of the l_0-norm. Some experiments have been done by Wei *et al.* [60] on which another paper could be based. Also, the sparsifying basis used in these experiments is the cosine space, but any space where the signal has a sparse representation could work. Which space works best for vibration data has not been explored in this paper, but a wavelet basis might be a good place to start.

The technicalities of actually performing randomly sparse sampling with a discrete sensor are not touched upon in this thesis. It was assumed to be possible. If this is a challenge then it should be subject to further work.

Chapter 8

Conclusion

By now, it should be clear that compression is essential when the world is moving into the IoT space. This thesis has looked at methods of compressing data and sampling data to use with maintenance prediction.

A look at previous attempts at making maintenance prediction algorithms were first given. Although many of the papers lack the details to recreate their proposed algorithms, some ways of predicting faults seem promising. Two of the methods were brought into this thesis when an evaluation of the compression methods was given. The first method calculates the RMS of a signal and tries to predict the n next RMS values using GPR. The second method chosen focused on enhancing the early fault signatures using wavelet decomposition.

Then, some recent papers on compression was presented where the methods of interest to this thesis were brought to light; autoencoder and compressed sensing. Two more methods were also tested and discussed; the method currently implemented in vibration sensors from Disruptive Technologies (DT) and a decoder network.

The DT method was tested with three different configurations, but none could achieve any results usable for maintenance prediction. It provides too little information and, possibly, the wrong information to enable a representative "picture" of what it is measuring.

The autoencoder and the decoder show some more promising results, but the usefulness of the recreated signal is questionable anyhow. Results from these methods are noisy, the RMS values do not change correctly over time, and the spectral envelopes they provide is not resembling the measured signal. They are deemed useless in the context of this thesis.

Two different solvers were used for testing compressed sensing; ECOS and OWL-QN. ECOS was too slow and memory intensive to be practical with the large size of the data, and so the method was not tested for more than a couple of samples. OWL-QN was then used which computed signals quickly. The results from compressed sensing with OWL-QN were good. With only 1% of the data sampled, the RMS over time closely followed the original signal. This makes it useful for failure prediction with GPR. When the sample size was adjusted to 10%

and above, the spectral envelope show a clear resemblance to the original spectral envelope. When the recreated signals have captured the essence in the spectra, the weak signatures should be present. Thus, compressed sensing with OWL-QN should work with the data provided.

The results in this paper indicate multiple advantages of using compressed sensing when designing a low energy vibration IoT sensor. Where the original DT method fails to capture the important spectral envelope of the signal, compressed sensing can reconstruct it accurately. The computational load on the sensor-processor is also removed as it does not have to compute the FFT nor locate any peaks. By just random sampling in a fixed time window, the optimization running somewhere else will take care of the rest. This point is also valid for the decoder method, but as the results have shown the simple architectures tested were not able to provide adequate reconstructions. The biggest downfall of compressed sensing is that size of the data transmitted increases, even with a sample size of only 1%. With the technology provided by DT, a choice must be made; battery life versus useful information.

Making new technologies smart is obvious, retrofitting old technology with smart sensors; is not. With the Disruptive Technologies sensor platform and the techniques described in this paper, a step closer to a simple transition to a world of IoT is hopefully taken.

Bibliography

[1] E. Kreyszig, H. Kreyszig and E. J. Norminton, *Advanced Engineering Mathematics*, Tenth. Hoboken, NJ: Wiley, 2011, ISBN: 0470458364.

[2] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2e. New York, NY, USA: Springer, 2006.

[3] C. Guo and F. Berkhahn, 'Entity embeddings of categorical variables,' *arXiv preprint arXiv:1604.06737*, 2016.

[4] R. E. Walpole, R. H. Myers, S. L. Myers and K. Ye, *Probability and statistics for engineers and scientists*, 9th edition. Pearson, 2016, ISBN: 1292161361.

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

[6] A. Hatcher, *Algebraic topology*. Cambridge University Press, 2005.

[7] G. P. Collins, 'The shapes of space,' *Scientific American*, vol. 291, no. 1, pp. 94–103, 2004.

[8] H. S. M. Coxeter and S. L. Greitzer, *Geometry revisited*. Maa, 1967, vol. 19.

[9] K. P. Murphy, *Machine learning: a probabilistic perspective*, 1st ed. MIT press, 2012.

[10] *Memorial Tributes: Danie G. Krige*. 2015, vol. 19.

[11] H.-Y. Kim, 'Statistical notes for clinical researchers: assessing normal distribution (2) using skewness and kurtosis,' *Restorative dentistry & endodontics*, vol. 38, no. 1, pp. 52–54, 2013.

[12] S. Roman, S. Axler and F. Gehring, *Advanced linear algebra*. Springer, 2005, vol. 3.

[13] A. McDougall, D. Stoffer and D. Tyler, 'Optimal transformations and the spectral envelope for real-valued time series,' *Journal of statistical planning and inference*, vol. 57, no. 2, pp. 195–214, 1997.

[14] S. Wold, K. Esbensen and P. Geladi, 'A literature review of maintenance prediction techniques with preliminary tests using vibration data from bearings,' 2021.

[15] H. Qiu, J. Lee, J. Lin and G. Yu, 'Wavelet filter-based weak signature detection method and its application on rolling element bearing prognostics,' *Journal of sound and vibration*, vol. 289, no. 4-5, pp. 1066–1090, 2006.

[16] S. Hong and Z. Zhou, 'Application of Gaussian process regression for bearing degradation assessment,' in *2012 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012)*, IEEE, 2012, pp. 644–648.

[17] S. L. Brunton and J. N. Kutz, *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2019.

[18] G. K. Wallace, 'The JPEG still picture compression standard,' *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[19] M. Feldman, 'Hilbert transform in vibration analysis,' *Mechanical systems and signal processing*, vol. 25, no. 3, pp. 735–802, 2011.

[20] M. Misiti, Y. Misiti, G. Oppenheim and J.-M. Poggi, *Wavelets and their Applications*. Iste UK and EEUU, 2007, vol. 330.

[21] S. Wold, K. Esbensen and P. Geladi, 'Principal component analysis,' *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[22] F. N. Nwobi and C. A. Ugomma, 'A comparison of methods for the estimation of Weibull distribution parameters,' *Metodoloski zvezki*, vol. 11, no. 1, p. 65, 2014.

[23] J. D. Hamilton, *Time series analysis*. Princeton university press, 2020.

[24] A. L. Samuel, 'Some studies in machine learning using the game of checkers,' *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[25] J. Friedman, T. Hastie, R. Tibshirani *et al.*, *The elements of statistical learning*, 2nd ed. Springer series in statistics New York, 2009, ISBN: 978-0-387-84858-7. DOI: 10.1007/978-0-387-84858-7.

[26] R. Pascanu, T. Mikolov and Y. Bengio, 'On the difficulty of training recurrent neural networks,' in *International conference on machine learning*, PMLR, 2013, pp. 1310–1318.

[27] J. Duchi, E. Hazan and Y. Singer, 'Adaptive subgradient methods for online learning and stochastic optimization.,' *Journal of machine learning research*, vol. 12, no. 7, 2011.

[28] N. Ketkar, 'Stochastic gradient descent,' in *Deep learning with Python*, Springer, 2017, pp. 113–132.

[29] D. P. Kingma and J. Ba, 'Adam: A method for stochastic optimization,' *arXiv preprint arXiv:1412.6980*, 2014.

[30] S. Albawi, T. A. Mohammed and S. Al-Zawi, 'Understanding of a convolutional neural network,' in *2017 International Conference on Engineering and Technology (ICET)*, Ieee, 2017, pp. 1–6.

[31] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar and P.-A. Muller, 'Deep learning for time series classification: a review,' *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[32] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition,' in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[33] S. Hochreiter and J. Schmidhuber, 'Long short-term memory,' *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] F.-F. Li, J. Johnson and S. Yeung, *Lecture 10: Recurrent Neural Networks*, May 2017.

[35] G. E. Hinton and R. R. Salakhutdinov, 'Reducing the dimensionality of data with neural networks,' *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[36] D. L. Donoho, 'Compressed sensing,' *IEEE Transactions on information theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[37] M. Davenport, 'The fundamentals of compressive sensing,' *IEEE Signal Processing Society Online Tutorial Library*, vol. 12, 2013.

[38] E. J. Candès and M. B. Wakin, 'An introduction to compressive sampling,' *IEEE signal processing magazine*, vol. 25, no. 2, pp. 21–30, 2008.

[39] T. Blu, P.-L. Dragotti, M. Vetterli, P. Marziliano and L. Coulot, 'Sparse sampling of signal innovations,' *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 31–40, 2008.

[40] M. Lustig, D. L. Donoho, J. M. Santos and J. M. Pauly, 'Compressed sensing MRI,' *IEEE signal processing magazine*, vol. 25, no. 2, pp. 72–82, 2008.

[41] M. A. Herman and T. Strohmer, 'High-resolution radar via compressed sensing,' *IEEE transactions on signal processing*, vol. 57, no. 6, pp. 2275–2284, 2009.

[42] P. Sen and S. Darabi, 'Compressive image super-resolution,' in *2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, IEEE, 2009, pp. 1235–1242.

[43] S. Boyd, S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[44] A. Domahidi, E. Chu and S. Boyd, 'ECOS: An SOCP solver for embedded systems,' in *2013 European Control Conference (ECC)*, IEEE, 2013, pp. 3071–3076.

[45] A. Wächter and L. T. Biegler, 'On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming,' *Mathematical programming*, vol. 106, no. 1, pp. 25–57, 2006.

[46] L. Vandenberghe, 'The CVXOPT linear and quadratic cone program solvers,' *Online: http://cvxopt. org/documentation/coneprog. pdf*, 2010.

[47] G. Andrew and J. Gao, 'Scalable training of l 1-regularized log-linear models,' in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 33–40.

[48] P. Wang and G. Vachtsevanos, 'Fault prognostics using dynamic wavelet neural networks,' *AI EDAM*, vol. 15, no. 4, pp. 349–365, 2001.

[49] W. Deng, S. Zhang, H. Zhao and X. Yang, 'A novel fault diagnosis method based on integrating empirical wavelet transform and fuzzy entropy for motor bearing,' *IEEE Access*, vol. 6, pp. 35 042–35 056, 2018.

[50] M. Rezamand, M. Kordestani, R. Carriveau, D. S.-K. Ting and M. Saif, 'A new hybrid fault detection method for wind turbine blades using recursive PCA and wavelet-based PDF,' *IEEE Sensors Journal*, vol. 20, no. 4, pp. 2023–2033, 2019.

[51] G. K. Durbhaka and B. Selvaraj, 'Predictive maintenance for wind turbine diagnostics using vibration signal analysis based on collaborative recommendation approach,' in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, IEEE, 2016, pp. 1839–1842.

[52] M. Baptista, S. Sankararaman, I. P. de Medeiros, C. Nascimento Jr, H. Prendinger and E. M. Henriques, 'Forecasting fault events for predictive maintenance using data-driven techniques and ARMA modeling,' *Computers & Industrial Engineering*, vol. 115, pp. 41–53, 2018.

[53] I. Amihai, M. Chioua, R. Gitzel, A. M. Kotriwala, D. Pareschi, G. Sosale and S. Subbiah, 'Modeling machine health using gated recurrent units with entity embeddings and k-means clustering,' in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, IEEE, 2018, pp. 212–217.

[54] I. Amihai, R. Gitzel, A. M. Kotriwala, D. Pareschi, S. Subbiah and G. Sosale, 'An industrial case study using vibration data and machine learning to predict asset health,' in *2018 IEEE 20th Conference on Business Informatics (CBI)*, IEEE, vol. 1, 2018, pp. 178–185.

[55] N. Krishnaraj, M. Elhoseny, M. Thenmozhi, M. M. Selim and K. Shankar, 'Deep learning model for real-time image compression in Internet of Underwater Things (IoUT),' *Journal of Real-Time Image Processing*, vol. 17, no. 6, pp. 2097–2111, 2020.

[56] J. Azar, A. Makhoul, R. Couturier and J. Demerjian, 'Robust IoT time series classification with data compression and deep learning,' *Neurocomputing*, vol. 398, pp. 222–234, 2020.

[57] A. Glaws, R. King and M. Sprague, 'Deep learning for in situ data compression of large turbulent flow simulations,' *Physical Review Fluids*, vol. 5, no. 11, p. 114 602, 2020.

[58] E. B. Helal, O. M. Saad, A. G. Hafez, Y. Chen and G. M. Dousoky, 'Seismic Data Compression Using Deep Learning,' *IEEE Access*, vol. 9, pp. 58 161–58 169, 2021.

[59] F. Salahdine, E. Ghribi and N. Kaabouch, 'Metrics for evaluating the efficiency of compressing sensing techniques,' in *2020 International Conference on Information Networking (ICOIN)*, IEEE, 2020, pp. 562–567.

[60] Z. Wei, J. Zhang, Z. Xu, Y. Huang, Y. Liu and X. Fan, 'Gradient projection with approximate L0 norm minimization for sparse reconstruction in compressed sensing,' *Sensors*, vol. 18, no. 10, p. 3373, 2018.

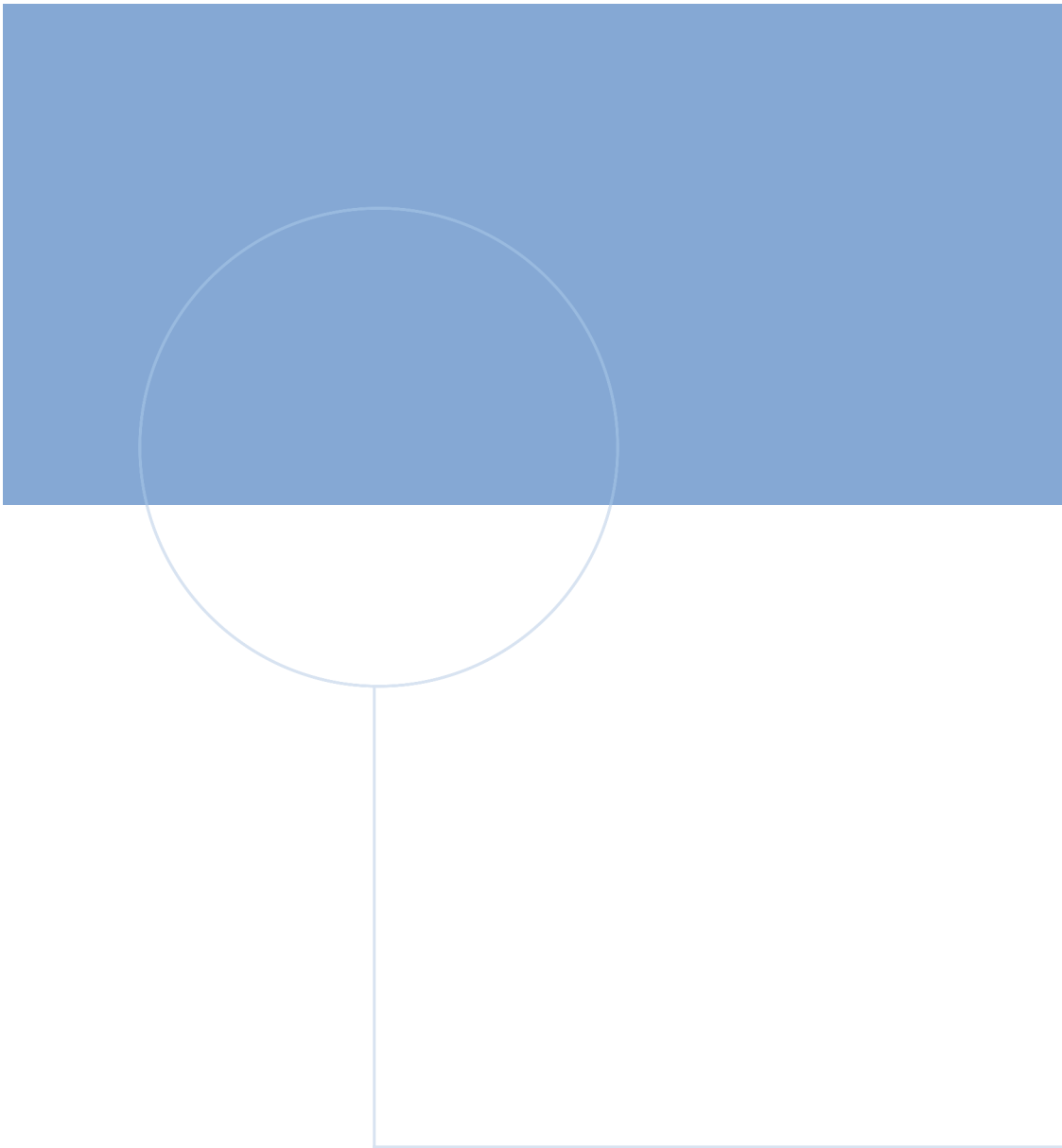
Appendix A

Additional Material

	DT method							
	5 peaks at 550							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	1.3936223281305884 01e+06	1.490427343406841625e+06	3.799117889712288976e+06	8.370553834597004578e+06	2.949474451870576944e+06	2.138229358071683440e+06	1.281928793872042326e+06	3.815165240059590898e+06
Test 1 | COV	1.048958154763419337e+01	-7.192958315990664708e+00	6.920395643336274105e+00	-6.785151234249296337e+00	-5.469000350879137384e-01	-1.105510529787310459e+00	1.920647110974886829e+01	8.201461400047323380e+00
Test 2 | MSE	2.643100138807475567e+06	5.280935545313508250e+06	6.384666437418897636e+06	5.370962326043351553e+06	n/a	n/a	n/a	n/a
Test 2 | COV	1.320024256365706306e+00	7.680759392850260880e+00	-9.097868060912582822e-01	4.204502397823662641e+00	n/a	n/a	n/a	n/a
Test 3 | MSE	3.855792790713205934e+06	6.030738239727609791e+06	1.149122910208127089e+07	3.627107929459680803e+06	n/a	n/a	n/a	n/a
Test 3 | COV	3.941525141271226662e+00	6.945233407371070200e+00	-7.699417000460968830e-02	-1.466539298207672204e-01	n/a	n/a	n/a	n/a
	DT method							
	5 peaks at 10000hz							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	2.844212802539634332e+07	1.528523684714080580e+07	4.177079685052900761e+07	3.141946202730499208e+07	1.062301203279210441e+07	4.244429920220618136e+06	1.185929736181650870e+07	6.778451027927610092e+06
Test 1 | COV	-9.298048236569474767e+00	-8.729805739624113770e+00	1.897342822530007567e+01	6.696004089813680338e+00	-5.554296438991456419e+00	1.343346671057613584e+01	-2.271565419509639128e+01	-1.317521588124516896e+01
Test 2 | MSE	3.023304432740109414e+07	4.462827864974663407e+07	2.219888556517611817e+07	2.429269520889344439e+07	n/a	n/a	n/a	n/a
Test 2 | COV	2.891888942481055125e+01	1.339867422244893902e+01	-1.069697544968732394e+01	2.752852637383163170e+01	n/a	n/a	n/a	n/a
Test 3 | MSE	3.136642182644686103e+07	4.623670257805746049e+07	2.388185999993267283e+07	1.897152760581874475e+07	n/a	n/a	n/a	n/a
Test 3 | COV	-5.201907160461962221e+00	1.924862656901208169e+01	-6.638528846014388307e+00	-7.070536786502399362e-01	n/a	n/a	n/a	n/a
	DT method							
	2000 peaks at 10000hz							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	1.209055411985414177e+08	1.052758696479572803e+08	1.253388639714757502e+08	1.168671899995569438e+08	1.064078566033818871e+08	9.726729182252615690e+07	1.146526908222461343e+08	1.023605888753243834e+08
Test 1 | COV	-1.530838191708251 77e+02	1.599684989736520180e+01	-1.913199219580759518e+02	1.056356013457559584e+02	1.020033668713685273e+02	-2.402412686633374594e+00	-8.546659800866235912e+01	1.790598754474592980e+01
Test 2 | MSE	1.312696949094050229e+08	1.261298041288526505e+08	1.126076715551437885e+08	1.235838361083655804e+08	n/a	n/a	n/a	n/a
Test 2 | COV	-1.325650605406139659e+00	1.377437656035934488e+01	-5.168783842587539867e-01	5.009403907206500861e+01	n/a	n/a	n/a	n/a
Test 3 | MSE	1.227763941270773709e+08	1.262246819095009416e+08	1.115094874356980473e+08	1.180236801442140341e+08	n/a	n/a	n/a	n/a
Test 3 | COV	2.731704965955462683e+01	3.678990039563744574e+01	-2.091159879062830296e+00	1.073703713328080767e+01	n/a	n/a	n/a	n/a
	Autoencoder							
	1 layer							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	2.408488473766756952e+08	1.648816600163049102e+08	1.712323135049584508e+08	1.163662061743448973e+08	2.573676785737839043e+08	2.820697436435102224e+08	2.084285465817244649e+08	2.325477646587332487e+08
Test 1 | COV	2.625015095839610440e+05	2.217369199902455439e+05	2.216446817630167352e+05	1.850247826773251581e+05	2.708564538395260461e+05	2.510158154601632268e+05	2.329600848608649394e+05	2.478705120838846778e+05
Test 2 | MSE	7.397122651588450670e+08	1.724831684752695858e+08	1.780876693219014108e+08	8.901507830018529296e+07	n/a	n/a	n/a	n/a
Test 2 | COV	5.225671918025631385e+03	3.121676259551717976e+03	2.553319823737099341e+03	3.857792491861408962e+03	n/a	n/a	n/a	n/a
Test 3 | MSE	1.076647560278687179e+08	9.014620243437382281e+07	8.389628857397617400e+07	5.594348448263374716e+07	n/a	n/a	n/a	n/a
Test 3 | COV	3.405219091170237789e+03	2.920124308718724023e+03	2.548595934780662446e+03	3.685969574816708246e+03	n/a	n/a	n/a	n/a
	Autoencoder							
	2 layer							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	4.528214194511554524e+03	1.522385880247892773e+13	4.527595709781394362e+03	5.775854345855247000e+15	1.348327821703363750e+14	4.626845618764898063e+03	4.581314266205911736e+03	4.591219864182956371e+03
Test 1 | COV	1.949759575662905490e+03	3.536571277768551838e+06	1.946200583402717712e+03	6.812789641794726253e+07	1.017703699389486946e+07	1.892875599724914309e+03	1.898356724857248537e+03	1.889558717973591229e+03
Test 2 | MSE	7.022286447863097237e+03	6.981764992355872891e+03	6.991009558139830006e+03	6.901778207591708451e+03	n/a	n/a	n/a	n/a
Test 2 | COV	2.641542065142652262e+01	2.289754458037775819e+01	1.969898938244218556e+01	4.575191052097166278e+01	n/a	n/a	n/a	n/a
Test 3 | MSE	6.927261270655274529e+03	9.278454290045237566e+22	6.926516665329490024e+03	6.884374568705996353e+03	n/a	n/a	n/a	n/a
Test 3 | COV	3.398468987664653440e+01	1.152111010160497665e+10	2.828149426095248131e+01	5.176925607510975169e+01	n/a	n/a	n/a	n/a
	Autoencoder							
	3 layer							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	2.126286889029704192e+18	2.120882080369323520e+18	2.118767474856602880e+18	2.117151758048523008e+18	2.029990802741289216e+18	2.018610760961132288e+18	2.022396936766479616e+18	2.014569987890757376e+18

Test 1 | COV	3.41540904944391745e+10	3.40162183784726524e+10	3.40210300497216758 7e+10	3.39758719420897216 8e+10	3.23911495009369316 1e+10	3.22964235437062835 7e+10	3.24546223362881240 8e+10	3.22764621937191543 6e+10
Test 2 | MSE	8.00923328048065937 5e+13	8.98515584347239375 0e+14	1.06669452492414078 1e+14	3.66530499017220562 5e+14	n/a	n/a	n/a	n/a
Test 2 | COV	1.13808081941534834 9e+06	1.51695050612493474 0e+07	1.32568298281107447 1e+06	6.97592736022196523 8e+06	n/a	n/a	n/a	n/a
Test 3 | MSE	1.05951215325616906 2e+14	2.73309523211495250 0e+15	1.40557608152636234 4e+14	5.30245047217502062 5e+14	n/a	n/a	n/a	n/a
Test 3 | COV	1.84157812716334220 0e+06	2.91876180741532072 4e+07	2.22083162652652198 4e+06	1.03959340998564623 3e+07	n/a	n/a	n/a	n/a
	Decoder							
	1 layer							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	3.94629402725635049 9e+06	3.64329090206788247 5e+06	3.80900283273943467 1e+06	2.88536801729319896 5e+06	5.65745057033645827 3e+06	6.24929832613057363 8e+06	3.84268479328568279 7e+06	3.83394440887920977 5e+06
Test 1 | COV	1.77165283102425128 2e+04	1.72299906241693897 8e+04	1.67303390392672107 4e+04	1.44359834397031791 0e+04	2.14778261397275782 7e+04	2.21388640957317584 3e+04	1.62148646542735696 1e+04	1.65356818957288996 9e+04
Test 2 | MSE	4.12760335355647141 1e+06	2.87750539022895833 5e+06	3.60019607390837464 5e+06	1.13645508112289151 2e+06	n/a	n/a	n/a	n/a
Test 2 | COV	2.44082998812819482 7e+02	2.01638443395056810 9e+02	2.25792500914365774 6e+02	2.69557902616537660 4e+02	n/a	n/a	n/a	n/a
Test 3 | MSE	1.65771258729856950 2e+06	1.85571246379759837 9e+06	1.21979106652795546 7e+06	8.97480751951596234 0e+05	n/a	n/a	n/a	n/a
Test 3 | COV	2.21354162647789280 5e+02	2.06455534877291654 5e+02	1.62396531269817870 7e+02	2.57065114156559957 3e+02	n/a	n/a	n/a	n/a
	Decoder							
	2 layer							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	1.41442479467880785 5e+08	1.36568848302934348 6e+08	1.41401247996825745 e+08	1.28855364741355136 0e+08	9.23083581041932830 8e+10	1.56085972037497818 5e+08	1.36656989008982271 0e+08	1.39536079969065398 0e+08
Test 1 | COV	2.61334149574401992 1e+05	2.57948230509453074 8e+05	2.60542191749775898 6e+05	2.53172157952107809 3e+05	6.23141042424595798 4e+05	2.60367349444437859 3e+05	2.49484771616805170 2e+05	2.50268611444251961 1e+05
Test 2 | MSE	5.96743682838095779 4e+10	1.69839684852945785 5e+10	2.85745511869199676 5e+10	3.65906838015852451 3e+09	n/a	n/a	n/a	n/a
Test 2 | COV	3.04210584524257938 0e+04	1.99261533186640117 5e+04	2.75924092440582971 9e+04	1.75808577155647362 8e+04	n/a	n/a	n/a	n/a
Test 3 | MSE	6.70514187072887802 1e+09	2.69521251214835166 9e+10	5.65077558109291839 6e+09	1.60425523157750892 6e+09	n/a	n/a	n/a	n/a
Test 3 | COV	2.17647088407807641 5e+04	7.37084391463804786 3e+04	1.59578918076494246 6e+04	1.40998996721554958 6e+04	n/a	n/a	n/a	n/a
	Decoder							
	3 layer							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	1.52300150179183832 4e+06	1.26096139751663594 5e+06	1.46349582337637338 8e+06	1.09764526959277782 6e+06	2.10798475570556847 4e+06	1.99140998979231156 4e+06	1.44800185806920146 6e+06	1.28316366784357582 2e+06
Test 1 | COV	2.29260112419295583 1e+04	2.06899571128423813 2e+04	2.19447519228552591 8e+04	1.90280570768058823 9e+04	2.55628947381208272 4e+04	2.34346978507881911 e+04	2.04249384056646231 3e+04	1.92256033224413731 7e+04
Test 2 | MSE	3.05502621884862346 6e+04	2.18980205488235573 3e+05	8.96609241301184985 8e+05	4.94443616739343197 0e+04	n/a	n/a	n/a	n/a
Test 2 | COV	1.56621455695592644 0e+01	7.87463461026492694 8e+01	3.74777029078821186 7e+01	3.30960283681596862 9e+01	n/a	n/a	n/a	n/a
Test 3 | MSE	9.06328904232311906 5e+04	1.64790877943252620 8e+05	1.62926052354994695 6e+05	1.35773853297798719 7e+05	n/a	n/a	n/a	n/a
Test 3 | COV	2.61765504405414191 1e+01	3.78758800894783789 2e+01	2.66645146292225980 7e+01	5.79272715780553042 5e+01	n/a	n/a	n/a	n/a
	Compressed sensing OWL-QN							
	Sample size: 1%							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	4.24978690121896506 8e+05	4.29104670169816759 9e+05	4.36853374091921781 7e+05	4.42207441429887781 9e+05	3.91408209320042806 2e+05	3.91506659860413579 7e+05	4.10141062278421130 0e+05	4.00583091324771638 0e+05
Test 1 | COV	1.47691227026076339 8e+04	1.47410611827381453 6e+04	1.52210583576560966 3e+04	1.52435619372489327 4e+04	1.30010835038069781 1e+04	1.29310372143561871 8e+04	1.38782201904896192 0e+04	1.35947868150714721 e+04
Test 2 | MSE	6.20784108801870534 1e+04	4.69940234644058582 5e+04	4.96173683875928181 8e+04	1.53371811895323917 3e+04	n/a	n/a	n/a	n/a
Test 2 | COV	3.48146499223032037 7e+02	6.48476049306838263 0e+02	3.20749655745196435 0e+02	1.08119720250234880 e+02	n/a	n/a	n/a	n/a
Test 3 | MSE	2.13575836496971060 1e+04	2.96285663320652784 0e+04	1.73987120351842058 9e+04	1.04274352734351323 3e+04	n/a	n/a	n/a	n/a
Test 3 | COV	1.97238428058358010 7e+02	4.79184115950479151 e+02	7.49466614496329413 0e+01	4.11011800463530576 6e+01	n/a	n/a	n/a	n/a
	Compressed sensing OWL-QN							
	Sample size: 10%							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	6.04442559719881159 3e+05	5.81004386956910020 7e+05	6.03430752691700356 1e+05	5.75410506317495484 8e+05	5.95670877983050653 7e+05	5.96566327258295728 8e+05	5.63448901884440216 2e+05	5.62379842083877651 0e+05
Test 1 | COV	1.79306699581040666 1e+04	1.72092204562165461 5e+04	1.82866711377594510 9e+04	1.75609806682608250 4e+04	1.63715103699015890 0e+04	1.60196311570038142 2e+04	1.65323869961552482 0e+04	1.61292051517964191 6e+04
Test 2 | MSE	1.30918400221107498 5e+05	9.96699242420744558 3e+04	9.86398258880592329 7e+04	3.26083991103959815 5e+04	n/a	n/a	n/a	n/a
Test 2 | COV	2.43079535407384264 5e+03	1.72327179782382040 7e+03	1.02896945798807496 4e+03	4.73754280452206842 3e+02	n/a	n/a	n/a	n/a

Test 3 | MSE	4.639734449963921361e+04	6.307647780176149536e+04	3.735161872284601850e+04	2.248058572121092948e+04	n/a	n/a	n/a	n/a
Test 3 | COV	6.504406695765134145e+02	1.120545458315941687e+03	3.918392642295011115e+02	2.249134958664542978e+02	n/a	n/a	n/a	n/a
	Compressed sensing OWL-QN							
	Sample size: 50%							
	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8
Test 1 | MSE	6.945118280397618655e+05	6.605421862763895188e+05	6.848875486169225769e+05	6.370581124609321123e+05	7.217064846299318597e+05	7.293680217009644257e+05	6.520174657145628007e+05	6.539568105125237489e+05
Test 1 | COV	1.950134194234678216e+04	1.825368574177053597e+04	1.968173983200711518e+04	1.845213022975073545e+04	1.809504619247519804e+04	1.759654868030479338e+04	1.820396485255123480e+04	1.733740334885967604e+04
Test 2 | MSE	2.330449658097256615e+05	1.605489098956142843e+05	1.702477207417858008e+05	5.792791759308025939e+04	n/a	n/a	n/a	n/a
Test 2 | COV	4.668558307261637310e+03	2.670141627229381356e+03	2.009061793751382993e+03	1.042121489420562057e+03	n/a	n/a	n/a	n/a
Test 3 | MSE	7.937541681848754524e+04	1.013819724984406203e+05	6.652648165061228792e+04	4.097000147177395411e+04	n/a	n/a	n/a	n/a
Test 3 | COV	1.274297258562139859e+03	1.733568418728000097e+03	7.670773285231294949e+02	5.888166647970214171e+02	n/a	n/a	n/a	n/a



NTNU

Norwegian University of
Science and Technology