University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Acoustic Anomaly Detection in Hydropower Plants

Dominic Tobler
Zurich, Switzerland
Student ID: 21-705-223

Supervisor: Prof. Dr. Bruno Rodrigues, Jan von der Assen, Prof. Dr. Burkhard Stiller
Date of Submission: April 14, 2025

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zuerich, 11.04.2025

Signature of student

Zusammenfassung

Diese Bachelorarbeit untersucht die Erkennung von Anomalien im Audiobereich als Grundlage fuer die Umsetzung von vorausschauender Wartung (predictive maintenance) in Wasserkraftwerken. Im Turbinenhaus des Pumpspeicherkraftwerks Rodundwerk II in Vorarlberg, Oesterreich, wurden Tonaufnahmen gemacht, die zur Auswertung von drei nicht ueberwachten (unsupervised) Modellen des maschinellen Lernens benutzt werden, namentlich der dense Autoencoder, der convolutional Autoencoder und die Self-Organizing-Map. Fuer alle Modelle wurde ein Design erstellt, das dann mit Python umgesetzt wurde. Ziel dieser Modelle ist die Erkennung von Anomalien, die auf mechanische Unregelmaessigkeiten in den Komponenten des Wasserkraftwerks hindeuten. Die Genauigkeit der Modelle wurde mit der Flaeche unter der receiver operating characteristic Kurve gemessen. Mittels Optimierung der Hyperparameter wird eine signifikante Verbesserung der Klassifikations-Genauigkeit erreicht. Unter den untersuchten Modellen zeigt die Self-Organizing-Map die beste Leistung in der Anomalieerkennung. Obwohl das Training des convolutional Autoencoder ressourcenintensiver ist ,verglichen mit dem dense Autoencoder, zahlt sich dieser erhoehte Aufwand nicht in einer besseren Leistung aus. Diese Arbeit unterstreicht das Potenzial akustischer Analyseverfahren zur Frueherkennung von Fehlern in kritischer Infrastruktur von Wasserkraftwerken und schafft eine Grundlage fuer zukuenftige, in Echtzeit operierende Ueberwachhungssysteme.

Abstract

This thesis investigates sound-based anomaly detection as a tool for predictive maintenance in hydropower plants. In the pumped storage hydropower plant Rodundwerk II in Austria, acoustic data was collected in the turbine house and used to evaluate three state of the art unsupervised machine learning approaches: a dense autoencoder, a convolutional autoencoder, and a Self-Organizing Map. Each model was designed and implemented to identify anomalies in recorded sound signals that may indicate mechanical irregularities. The evaluation considers classification performance on induced anomalies in terms of the area under curve of the receiver operating characteristic. The performed hyperparameter tuning led to a significant increase in classification accuracy. Results show that the among the evaluated models, the Self-Organizing Map has the best anomaly detection performance for the used datasets, namely for both the audio recorded in the hydropower plant as well as for the external MIMII dataset. Even though the convolutional autoencoder required more computational resources during training, this was not justified with a better anomaly detection accuracy. This work demonstrates the potential of acoustic analysis for early fault detection in critical hydropower infrastructure and provides a foundation for future real-time monitoring systems.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor, Prof. Dr. Bruno Rodrigues, for his valuable guidance, insightful feedback, and continuous support throughout the course of this thesis.

I am also very thankful to Dr. Ing. Arne Schacht, director of the Rodundwerk II hydropower plant, for the opportunity to visit the facility and, most importantly, to record sound samples in the turbine hall.

My sincere thanks go to my co-supervisor Jan von der Assen, as well as to Prof. Dr. Burkhard Stiller and the entire Communication Systems Group for enabling me to explore the fascinating field of sound-based anomaly detection within the scope of this Bachelor thesis.

Last but not least, I would like to thank my friend Remo Wiget for his thoughtful proofreading and the valuable suggestions that emerged from it.

Contents

Chapter 1

Introduction

This bachelor thesis explores state of the art approaches to anomaly detection and applies these principles to concrete prototypes that are evaluated using sound signals collected in a hydropower plant.

1.1 Motivation

> ***"Hydropower is the most important global renewable source of electricity."*** [7]
>
> Paul Breeze - Hydropower

Pumped storage hydropower plants (PSHPP) not only produce cheap and climate friendly electricity in large quantities, but they also play a vital role in grid stabilization. In times of electricity surplus, PSHPP use power to pump water from their lower reservoir up to their upper reservoir, turning it into a battery that is storing potential energy. When electricity is scarce, the water is flowing back into the lower reservoir while releasing the potential energy that is then used to produce electricity. In 2017, 97% of the global active electricity storage capacity for grids was provided by PSHPPs [7].

Hydropower plant equipment like turbines and generators are critical to plant operations. Any failures in these mechanical parts can result in costly downtime. The exact number of missed income in an hour of downtime is highly fluctuating and mainly dependent on the current energy price on the market. To make an example, the PSHPP Rodundwerk II in Vorarlberg, Austria is used. It has a full-load capacity of 295 MW leading to an electricity production of 295 MWh in an hour. With a wholesale price of 97,30 Euro/MWh [2] the expected downtime cost is over 28'000 Euros per hour. Since PSHPP are used as peak load power plants, they sell their electricity mainly in times of high prices, leading to even higher downtime costs.

Power plant operators need to find an efficient maintenance strategy. Waiting too long increases the the risk of unplanned downtime. Replacing parts of their machinery too often is inefficient and also leads to maintenance downtime. Many old hydropower plants do

review their machinery in a scheduled maintenance process to ensure a reliable operation. Industry standards are established as guidelines to help power plant operators in deciding which maintenance schedule is appropriate.

The advancements in machine learning and available computational power form an opportunity to better target the maintenance process [6]. By leveraging data analytics and machine learning, operators can gain insights into plant performance. [23] The ultimate goal is to achieve predictive maintenance to know how the different parts degrade over time. With this information available the maintenance planning can be optimized, downtime can be reduced and ultimately maintenance costs are lowered.

Machines emit signals in different domains, giving hints about their condition. A fault inside a machine may not lead to a externally visible failure directly but only after a certain period of time. Ultrasonic sound and vibrations are signals present early, while audible sound or even heat are generally observable later in time [37]. Sound-based anomaly detection investigates the audible and depending on the microphone also the ultrasonic signals of machines. It can be a powerful tool towards achieving predictive maintenance. By using acoustic signals, potential issues with mechanical parts can be identified before they lead to operationally relevant faults [14].

1.2 Thesis Goals

This Bachelor Thesis will give an overview of available state of the art approaches to anomaly detection with a focus on the sound domain. The main contribution is the implementation of three anomaly detection methods and the comparison regarding their capabilities in detecting real-world anomalies in sound signals in a hydropower plant. There are four computer science students from university of St.Gallen (HSG) who write their master project on a similar topic. Together with them, sound samples are recorded in the PSHPP Rodundwerk II, located in Vorarlberg, Austria. Their resulting approach to anomaly detection based on a dense autoencoder architecture will be one of the implemented methods.

The results of this thesis will be used to design and deploy a real-time anomaly detection system in the PSHPP Rodundwerk II.

- Overview of sound based anomaly detection techniques
- Design and Implementation of three anomaly detection approaches based on the state of the art
- Comparison of anomaly detection effectiveness with real-world data
- Exploration of hyperparameter tuning effects

The practical relevance of this bachelor thesis lies in its contribution to the development of an anomaly detection system for hydropower plants. The insights and results obtained are applied in the design of a system that is deployed at the PSHPP Rodundwerk II.

1.3 Thesis Outline

In order to provide an overview for anomaly detection techniques, an analysis of background topics and available related work is performed in Chapter 2.

The Design Chapter 3 provides a comparison of different methods to choose two suitable approaches, along with the dense autoencoder, that can be implemented in this bachelor thesis. As a next step, a high-level overview regarding the designs for the chosen approaches is given. These designs are then implemented into concrete python prototypes described in more detail in the Implementation chapter 4. Finally in the Evaluation chapter 5, the approaches are compared against each other in terms of classification results and runtimes. The question here is, how well the different methods are able to detect artificially induced and real anomalies present in the sound signals that were previously recorded.

Chapter 2

Fundamentals

2.1 Background

The background section introduces the fundamental concepts that are relevant for understanding the following chapters.

2.1.1 Anomaly Detection

Anomalies are rare events, deviations from the standard pattern, or significant variations from the norm or normal behaviour [35].

The definition implies that anomalies are coupled to a specific problem or application, since the notion of normal behaviour is domain-specific [22]. In order for detection algorithms to be useful, they still need to be applicable to a variety of problems.

In [35] different fields are listed for which anomaly detection is relevant and used in applications. Recognizing spam emails and credit card fraud, diagnosis of cancer or quality control in manufacturing are examples where anomaly detection is put into practice.

Different methodologies exist for the detection of anomalies:

- Statistical methods based on probability theory
- Machine learning methods that recognize patterns in existing data
- Rule-based systems that make use of domain specific knowledge

Some statistical methods such as the z-score are straight-forward to implement. The audio signal is split into frames and for each frame features are extracted. The deviation from the mean over all frames for each feature can be seen as a measure of anomaly. In [22] a more sophisticated statistical method was used. It is based on Tukey's test that was able to recognize anomalies well in a energy consumption dataset . Machine learning is

expected to perform well in anomaly detection tasks. Different authors used variants of autoencoders to find unusual patterns in various domains including the sound domain [1] [13]. Rule-based systems require much more domain specific knowledge than the previous two methodologies and are therefore more tedious to implement. In the literature rule-based approaches are mainly used in combination with other mechanisms such as deep learning models [8].

2.1.2 Sound Analysis

Sound waves have different characteristics, such as frequency, speed, and amplitude. They can be seen as the alternation of compressions and relaxations of particles. This alternation is propagated through a medium [46].

Frequency is the number of compression and relaxation alternations that particles experience in a specific time interval. The standard unit uses a time interval of one second and is called Hertz (Hz). High frequencies correspond to a high pitch, whereas low frequencies are perceived as a low pitch. Amplitude describes the maximum displacement of particles from a specified starting point. It is measured in decibel (dB). Amplitude can be viewed as intensity of a sound wave and corresponds to loudness [26].

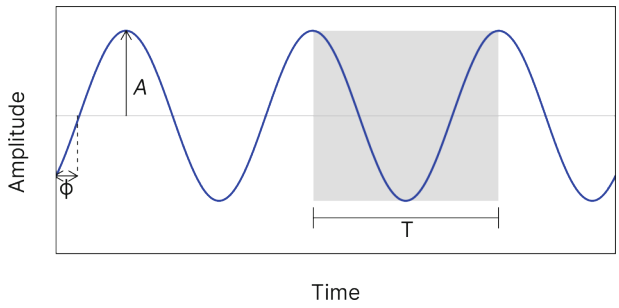

Figure 2.1: Time-Amplitude Graph. Adapted from [46]

Figure 2.1 shows an example of a Time-Amplitude graph with time interval T, Amplitude A and phase Φ.

2.1.2.1 Digital Representation of Sound waves

The first stage in sound analysis using a computer is the conversion of the audio signal into a digital format. This analogue digital conversion involves two key processes: sampling and quantization [46].

a) Sampling: To capture sound, a sensor such as a microphone records the perceived amplitude at regular intervals. The frequency at which these samples are taken is referred to as the sample rate, measured in Hertz (Hz). Common sample rates are 44'100 and 22'050 Hz.

b) Quantization: Each sample is assigned a numerical value corresponding to its amplitude. Most audio systems use an integer-based scale, typically with a 16-bit depth, allowing for 65,536 distinct values.

The output of these processes is a discrete, finite set of data points that represent the original sound, and can be stored and analysed by a computer. Higher sample rates and bit depths provide a more accurate digital representation of the sound, capturing finer details of the original audio.

2.1.2.2 Fourier Analysis

There are different operations that are summarized under the term Fourier Analysis [45]. Fourier series decompose a periodic signal into a sum of a constant term and an infinite series of sine and cosine functions. These sinusoidal functions represent specific frequencies, allowing the Fourier series to expose the frequency content of a periodic signal.

In practice, captured sound signals are neither periodic nor infinite and are measured in discrete time intervals. An adaptation of the Fourier series for such cases is the Discrete Fourier Transform (DFT), which operates on discrete and time-limited signals [27]. The result of a DFT has a limited frequency resolution that depends on the signal length. For shorter signals, it is harder to distinguish between similar frequencies, since they may fall into the same frequency bin. In addition to that, the DFT operation implicitly assumes the time limited signal to repeat itself. This can introduce frequency artefacts, known as spectral leakage in the obtained result. The DFT is invertible, meaning that the original signal can be recovered from the transform without the introduction of information loss.

Conceptually, the DFT operation transforms a signal from the time domain to the frequency domain, making the previously hidden frequency information observable. This process can be compared to a prism that separates white light into its colour spectrum.

The direct computation of a DFT has a computational complexity of $O(n^2)$, which becomes infeasible for large signals. The Fast Fourier Transform (FFT) is an algorithm that applies a divide-and-conquer strategy and exploits certain properties that are incorporated in the problem due to the periodicity of sinusoids. This approach is able to reduce the complexity, leading to a number of operations in the order of $n * log(n)$, making it efficient for large signals [46].

The result of a FFT consists of complex numbers corresponding to different frequencies. The magnitude and phase of each frequency component can be derived from the real and imaginary parts of the complex number, respectively. The concept of phase is visualized in 2.1. In the context of an FFT result, it shows how much each frequency component is phase-shifted.

The range of frequencies that can be analysed depends on the sampling rate. The Nyquist-Shannon sampling theorem states that a signal can be fully captured by samples only if the sampling rate is at least twice the maximum frequency [41]. The interpretation of the theorem can be stated as following. Frequencies present in the environment that are higher than half the sampling rate cannot be captured reliably and should therefore not be used in analysis.

2.1.3 Clustering

Clustering is an unsupervised machine learning approach that partitions the data into different parts, so-called clusters [9]. The underlying assumption is that points in the same cluster are of the same class. In the case of anomaly detection the two classes are normal data and anomalies. Some clustering algorithms allow for data points to lie outside any clusters, in that case these points are candidates for an anomaly [35].

Since anomalies are rare events, labeled data would be unbalanced with the normal case being heavily overrepresented. This asymmetric distribution discourage the use of traditional supervised learning methods, such as classical feed-forward neural networks or Support Vector Machines that are built on the assumption of balanced training data [6, 34].

An unsupervised learning method requires no labeled data during training. There exist clustering metrics such as the silhouette score or the Dunn index that give information about the quality of formed clusters [52]. But to evaluate how well an approach is able to discriminate between the normal case and anomalies, it is still highly useful to have knowledge about the existing anomalies.

A variety of clustering approaches exist. Examples are the k-means or the spectral clustering algorithms [35].

2.1.4 Autoencoder

Autoencoders are a type of unsupervised deep neural network architecture that have the same number of output units as input units. They are trained to minimize the difference between its output y and its input x [18]. An autoencoder consists of two parts [34], as visualized in Figure 2.2.

- Encoder: The input x is mapped to a representation *z(x)*
- Decoder: The representation *z(x)* is mapped to the output *y(z)*

To force the model to learn the underlying patterns of the data, a restriction for the representation z(x) is introduced. In undercomplete autoencoders, the restriction is present in the form of a bottleneck. The dimension of z is restricted to be lower than the dimension of the input, leading the encoder to perform a dimensionality reduction [18]

In the context of anomaly detection, autoencoders are trained with data that is representing the normal case to learn its underlying patterns. When the trained autoencoder is used to reconstruct new data, the difference between the model's reconstruction and its input can be used as a measure of anomaly. [5] A threshold θ is defined that is used to discriminate between cases based on the reconstruction error σ:

$$\sigma \leq \theta \longrightarrow normal$$

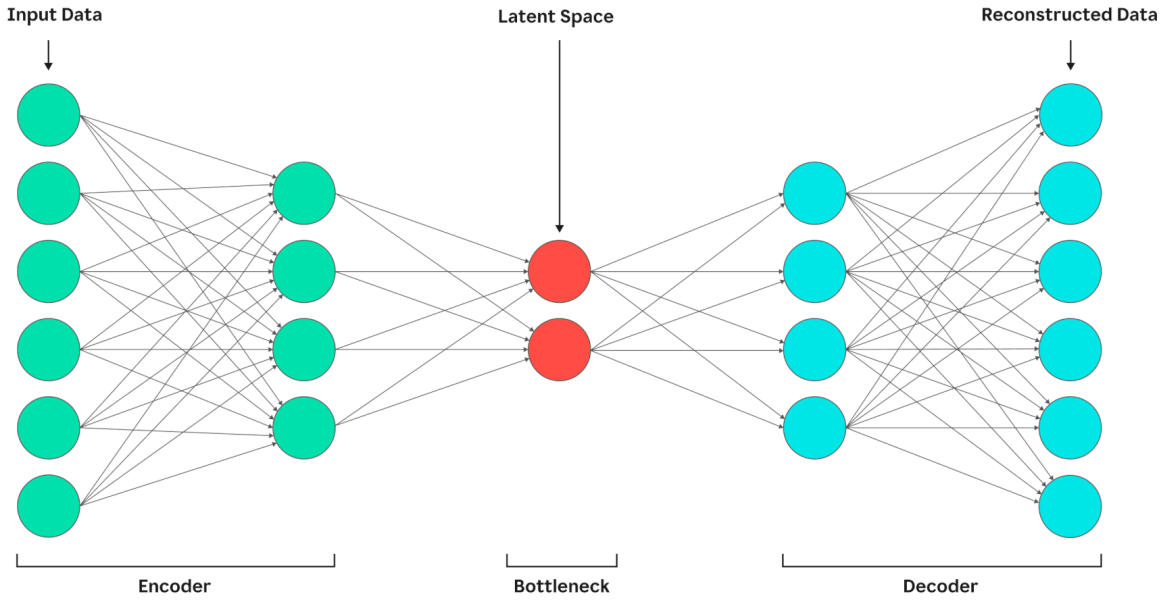

Figure 2.2: General autoencoder architecture

$$\sigma > \theta \longrightarrow anomaly$$

The underlying assumption is that the autoencoder can reconstruct patterns similar to what it was trained with well, leading to a low reconstruction error. These low reconstruction errors should lie below the defined threshold and are thus classified as normal sound. Anomalous sound contains patterns that are not known to the model, it should therefore be harder for the model to reconstruct these patterns, which will increase the difference between reconstruction and original input. If this difference is bigger than the defined threshold, it is classified as an anomaly

2.2 Related Work

This section explores the work of other authors in the field of anomaly detection and sound analysis.

In [34] the authors tried out different unsupervised machine learning approaches to find anomalies in multivariate data collected in hydropower plants. The autoencoder and the Isolation Forest model performed the best, the authors preferred the autoencoder because its output is easier to interpret. Additionally, they used SHAP values for identifying the root cause of present anomalies. Even though the task of finding the reason for an anomaly is interesting, it is out of the scope of this thesis. The fact that autoencoders and Isolation Forests work well for multivariate data, makes it appealing to apply these models in acoustic anomaly detection and test their capabilities in finding anomalies in this signal domain.

The authors of [53] preprocessed sound signals by applying mel filters and the constant Q transform. These two sound representations are used as an input matrix for a convolution neural network to classify different sound events in a supervised fashion. The idea of feeding convolutional layers with sound spectrograms is not straight-forward and is potentially useful for the design of an unsupervised anomaly detection method.

There is a paper [31] using spectral clustering for detecting anomalies in power current data in the context of photovoltaic power plants. Here, the cluster with the largest current is seen as the normal case, all others are classified as anomalies. The anomaly criterion for a cluster cannot be generalized from power current to sound signals. However, the authors mention the excellent performance for low-dimensional data, making it interesting to explore relevant audio features that are capable of representing the whole audio signal.

A deep neural network is proposed in [10] to detect anomalies in multivariate time series data in an unsupervised fashion. It combines the capabilities of convolutional layers to capture spatial dependencies and long short-term memory layer for capturing temporal dependencies. Since audio data contains temporal dependencies and if the audio is processed to a spectrogram also spatial dependencies occur, this advanced machine learning model should be further investigated.

In [50] a sparse autoencoder and a support vector machine are compared against each other in their capability of detecting faults in acoustic signals. 87 features from the time, frequency and wavelet domain are extracted. The sparse autoencoder performed well and is recommended by the authors for fault diagnosis of motors.

Paper [14] transforms the collected audio signal into mel spectrograms that are then fed into a conditional convolutional autoencoder and a conditional long short-term memory autoencoder. The reconstruction error threshold with maximal Youden's index is used to discriminate between normal and anomalous sound. The choice of the threshold is important when using autoencoders for anomaly detection, the definition of an optimal threshold via Youden's index can be useful when designing an approach.

Also [16] uses Youden's index to find a suitable threshold for the proposed convolutional autoencoder that is trained with sound from various industrial machines. The area under the receiver operating curve (ROC) and the F1-score are used as evaluation metrics. They can be useful in evaluating the results of anomaly detection approaches implemented in this bachelor thesis.

The authors of [6] uses a self-organizing map (SOM) to reduce the dimensionality of multivariate hydropower data in an unsupervised fashion. The SOM is trained to fit normal data. When confronted with new data, a key performance index (KPI) based on the distortion measure is defined. KPI values below the threshold of three standard deviations away from the KPI average are considered as anomalies. While autoencoders and SOMs both try to identify unusual patterns through dimensionality reduction, their approach is different and may be interesting to be compared against each other.

In [22], a statistical method based on Tuckeys test is used to self-label energy consumption data. Through the calculation of the probability density function of a Gaussian distribution, identified unlikely patterns are considered anomalous. In comparison to an

undercomplete sparse autoencoder, the statistical model performed better in detecting anomalies.

The k-means clustering algorithm is used in [40] for audio classification. After using the FFT to transform the audio signal to the frequency domain, k-means establishes clusters. The Rand index is used as an evaluation metric of the clusters, where low numbers indicate a better separation between clusters.

[1] compares unsupervised anomaly detection algorithms. The used models were the Support Vector Machine, the Isolation Forest, the Local Outlier Factory and the Robust Covariance. A synthetic dataset where anomalies were introduced was created to test the anomaly detection capabilities of these models. Accuracy, Precision, Recall and the weighted average of the latter two, creating the F1 score, was used to evaluate the classification results. It would be interesting to see if the results obtained by classifying the synthetic dataset can be observed with real data particularly within the sound domain.

MIMII [42] is a dataset containing over 26'000 sound samples of functioning valves, pumps, fans, and slide rails and some samples that contain anomalies. While the audio signals recorded in the Hydropower plant Rodundwerk II are the main interest of this thesis, this additional dataset can be useful to explore if the results obtained with the hydropower data can be generalized to other machine sounds.

Table 2.1: Related Work Summary

Source	Type of Data	Application Domain	Anomaly Detection Approach
[31]	Electrical current	Photovoltaic Power Plant	Spectral Clustering
[6]	Data from multiple sensors	Hydropower plant	Self-organizing map
[50]	Acoustic signals	Rotating machines	Sparse Autoencoders
[53]	Acoustic signals	Power plants	Convolutional Neural Network
[34]	Data from multiple sensors	Hydropower plant	Autoencoder, Isolation Forest
[22]	Power consumption	Grocery stores	Undercomplete Autoencoder, Statistical method based on Tuckeys test
[40]	Acoustic signals	Milling machinery	K-Means Clustering
[10]	Data from multiple sensors	Power plants	Convolutional Recurrent Autoencoder
[14]	Acoustic signals	General machinery	Conditioned Autoencoder
[1]	Synthetic dataset	-	One-Class Support Vector Machine, Isolation Forest, Local Outlier Factor, Robust Covariance

Chapter 3

Design

The first section outlines the rationale behind selecting suitable anomaly detection models. The data preprocessing section explores different signal representations and provides a high-level design for extracting spectrograms and features. Subsequent sections detail the architectures of both the dense and convolutional autoencoders. Finally, the last section introduces the design of the Self-Organizing Map.

3.1 Choice of Models

The autoencoder model was used in multiple papers that analysed anomalies within the sound domain. The results suggest that this model is adequate for the purpose of this thesis and is therefore chosen to be implemented. It will be interesting to see if it is possible to come up with a design for an autoencoder variation that outperforms the results achieved in the baseline implementation from the HSG Master project [21], which uses a dense autoencoder. Therefore their proposed architecture is implemented and evaluated against the other models.

The Self-Organizing Map performs a dimensionality reduction and is, in this sense, similar to the autoencoder. However, the process of reducing the input dimensionality and using the resulting information is different. This model is chosen to be implemented, to assess its suitability for detecting anomalies in audio data, which has not been found in related work so far.

The spectral clustering algorithm faces performances limitations when dealing with high-dimensional data. However, to achieve a reasonable anomaly detection result, various sound features, such as a range of mel-frequency cepstral coefficients (MFCC) need to be taken into account. K-Means, on the other hand, can handle input vectors with many features but is limited in recognizing arbitrary cluster shapes. Because of these limitations, it is expected that clustering based methods are less suited to sound based anomaly detection tasks compared to the approaches chosen to be implemented.

3.2 Data Preprocessing

For a sound signal to be used as input to anomaly detection algorithms, data preprocessing is necessary. The analogue audio signal present in the environment is converted to its digital representation during the recording, which includes sampling and quantization, as described in 2.1.2.1. The digitalized signal is present as amplitude as a function of time. In other words, for every sample taken at a specific time, there is an associated amplitude. The visualization of this function is called waveform. To extract information regarding the frequencies contained within the waveform, a Fourier Transform is applied, which results in a magnitude value for each frequency. While this transformation allows us to analyse the intensity of various frequencies in an audio sample, the time-varying information gets lost.

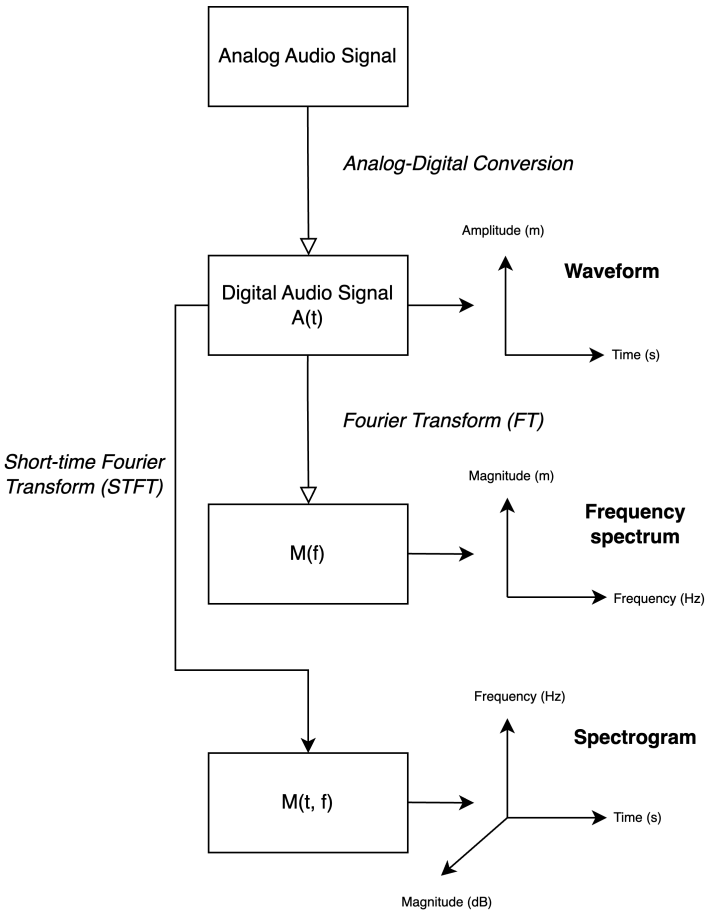

Figure 3.1: Signal Transformations

To compromise the time and frequency domains and get information from both, the Short-time Fourier Transform (STFT) is used. The basic idea is to split the original waveform into frames and apply the Fourier transform in a sliding window operation to the audio frames. The resulting function describes the magnitude for a given time and a given frequency. It is visualized as a spectrogram, a three dimensional plot with time and frequency as spatial dimensions and the corresponding magnitude values as a colour dimension. Figure 3.1 summarizes the described signal transformations as a part of the

preprocessing.

The further preprocessing steps are different depending on the specific model. For the autoencoder approaches, mel spectrograms are used as input and the approach using a SOM requires sound features as input.

3.2.1 Spectrograms

The spectrogram obtained after applying the STFT is further processed to a mel spectrogram. The goal of this step is to obtain a perceptually-relevant frequency representation. Humans perceive sound not in linear but rather in logarithmic fashion [43]. With a mel spectrogram the distance between two frequency bands are perceived as equal. Since the anomalies recorded in the hydropower plant are audible, the perceptually relevant representation is expected to be better suited when detecting the captured anomalies. Other sound applications that include the training of a deep neural network such as [13, 15] use the mel spectrogram as well. Before the mel spectrogram can be used as input for the autoencoder model, it is normalized with a min-max scaling such that its values are in the range of [0, 1]. This preprocessing step is recommended by [25].

3.2.2 Features

For models that take features as input, Figure 3.2 illustrates the different steps involved in the extraction of spectral sound features.

While in principle it would be possible to extract a feature from a longer signal, this leads to a low time resolution, not useful for an analysis. Therefore, framing is applied in order to obtain a time resolution that reveals details present in the signal. Framing is the process of dividing the signal into small, equal-length fractions. In the case of the waveform, for every frame, the features of interest can be calculated directly. In the domains where frequency is present, the additional operation of windowing is needed to prevent the introduction of high-frequency artefacts into the results. A windowing function has the characteristics of putting maximum weight onto its middle and small weight for larger or lower values. Every frame is multiplied with a given window function, leading the frame to fade-in and fade-out. As an unwanted effect of the windowing, information loss at the beginning and end of a frame is introduced. This loss is compensated by using overlapping frames. The choice of specific feature combinations is discussed in the implementation chapter.

3.3 Dense Autoencoder

The baseline model proposed by Huber and Krummenacher [21] is an undercomplete autoencoder with fully connected layers. This section provides a brief overview of its architecture.

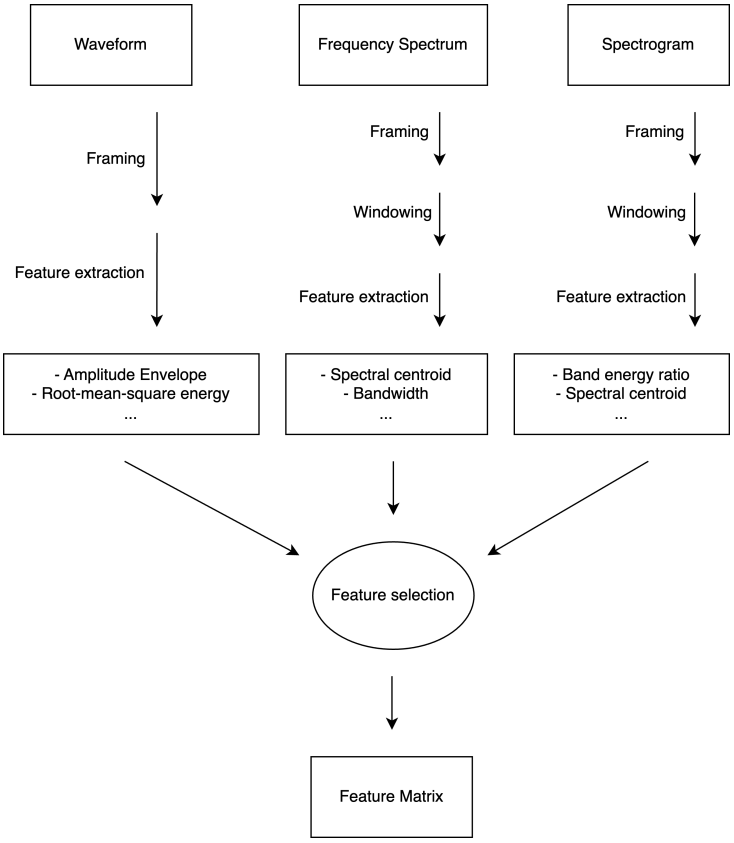

Figure 3.2: Feature extraction pipeline

The input spectrogram, originally a two-dimensional representation, is first flattened into a one-dimensional sequence. The encoder consists of three fully connected layers, each using the rectified Linear unit (ReLU) activation function 3.1. The first layer reduces the input dimension to 64 neurons, followed by a second layer with 32 neurons and a third with 16. At the core of the autoencoder is the bottleneck layer, a fully connected layer with eight neurons and no activation function.

The decoder mirrors the encoder's structure in reverse. It takes the bottleneck representation as input and progressively reconstructs the data using three fully connected layers, each with ReLU activation, expanding the representation back to 64 neurons. Finally, the data is mapped back to the original flattened spectrogram size, where a sigmoid activation function is applied before reshaping it into its original two-dimensional form.

The model is trained to minimize the reconstruction error, measured as the pixel-wise mean squared error 3.2, ensuring that the output spectrogram closely resembles the input.

3.4 Convolutional Autoencoder

It is expected that a model leveraging convolutional layers performs better compared to the dense autoencoder. The reason is that such convolutional layers better take into account spatial dependencies present in the data. It is not straight-forward that a convolutional

network, primarily used for image processing, can take sound signals as input. In the implementation part of this thesis, it is described how a spectrogram can be used as input to a convolutional network.

The convolutional layers in Figure 3.3 consist of three building blocks, with the two dimensional convolution itself being the first one. As a second block, the Rectified Linear unit (ReLu) as activation function according to Equation 3.1, defined in source [17], is applied. And additionally, batch normalization [24] is the third building block.

$$ReLU(x) = max(0, x) \tag{3.1}$$

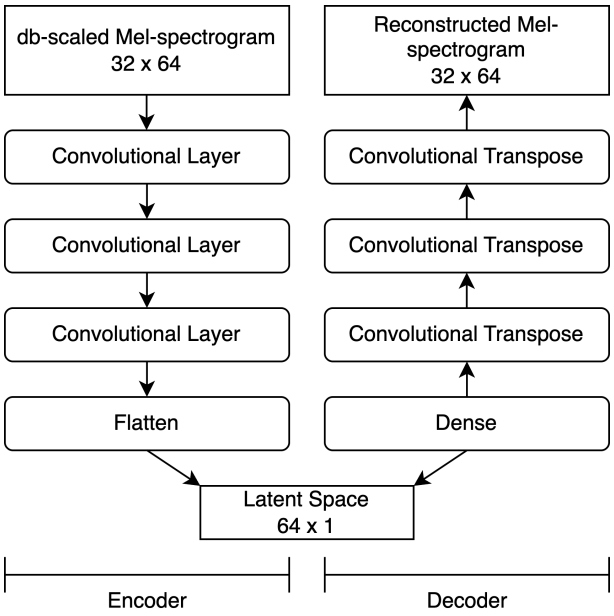

Figure 3.3: Convolutional autoencoder design

An important decision to be made regarding the architecture of an autoencoder is the number of layers. More layers build a more complex model that is capable of learning a good representation of the underlying patterns in the sound signal. On the other hand, not only the computational effort increases, but also the risk of overfitting to the training data while loosing generalization abilities grows with the model being more complex. Another unwanted effect that might be introduced is that the model could be able to reconstruct anomalous sound very well so that it will be hard to distinguish between normal and anomalous cases. An architecture that is expected to perform well uses three convolutional layers in the encoder and symmetrically three layers in the decoder. Each convolution layer is additionally defined by a set of hyperparameters that are a core part to the architecture of the overall model. These hyperparameters are the number of convolutional filters, so called kernels, the size of these kernels and the stride, defining the number of shifts between the convolutional operations.

The size of a convolutional kernel is defined by its side-lengths. In this design the kernels are constructed with equal side-lengths m, resulting in $m \cdot m$ weights per kernel. For it to

have a well defined centre, m should be an odd number. In general, m is a small number from three to seven. According to [49], there are two schools of thought when it comes to deciding the kernel size. One argues that m should be as small as possible such that small features can be learnt. The other uses slightly larger sizes initially and decrease it in deeper layers. In this thesis, the first school of thought is followed, using a kernel size of three by three in all layers. Since the encoder compresses the spectrogram in each layer, the same kernel size already processes a bigger fraction of the original data. This renders the additional decrease of the kernel size as redundant.

In this thesis, the number of kernels n increases for deeper layers. With 16 as the initial kernel number of the first layer, the model is expected to have a reasonable complexity, while staying at an acceptable computational effort level.

In order for the model to compress the spectrogram, a convolutional stride of two is used at each layer. This will have the effect of halving both the height and the width of the input. Leading to an overall reduction by a factor of 4 per layer.

Another architectural decision is the size of the latent space. In contrast to the convolutional layers which process two-dimensional data, the latent space is a one-dimensional dense layer. It is fully connected to the last layer of the encoder and the first layer of the decoder. For very small latent space sizes, a lot of information is lost due to the compression, making it hard for the model to reconstruct even the normal case. On the other hand, large sizes will lead the model to reconstruct anomalies too precise, not being able to discriminate between the cases. A latent space consisting of 64 neurons should be a good balance.

Besides the architectural hyperparameters, there are also decisions to be made regarding the training phase. The goal of a neural network is to minimize its loss function. In the case of an autoencoder, we want to define a loss function that measure the reconstruction difference between input and output spectrogram. The mean squared error MSE is frequently used as loss function and is the choice for this autoencoder as well. Since it measures how different the input and its reconstruction are, the MSE is called reconstruction error in this context. Equation 3.2, defined in source [32], states the formula to compute the MSE with x_i being a pixel in the input spectrogram, y_i the corresponding pixel in the reconstructed output respectively and n the overall number of pixels.

$$MSE = \frac{1}{n} \cdot \sum_{i=1}^{n} (x_i - y_i)^2 \tag{3.2}$$

The weights update during the training process is guided by the optimizer. The adaptive moment estimation optimizer, known as Adam is a standard for deep neural networks in general and is used in this design [36].

After the autoencoder network is trained on the normal case it is used to classify unseen spectrograms. The two classes are normal patterns seen in the training data, and unusual, anomalous patterns. Based on the difference between input and the autoencoders output, which is a reconstruction of the input, the classes are assigned. Again, the reconstruction error is used to measure this difference. Depending on this error and the chosen threshold

θ, the cases are distinguished according to the decision criterion in 3.3, where x is the original input and y its reconstruction.

$$\begin{aligned} MSE(x, y) < \theta &\longrightarrow normal \\ MSE(x, y) \geq \theta &\longrightarrow anomaly \end{aligned} \tag{3.3}$$

The choice of a suitable threshold θ is important for the model to perform well. Before the results of the inference phase are available, it is not practical to come up with a threshold, since it is highly dependent on the given data. Different thresholds will be subject to the evaluation chapter.

3.5 Self-Organizing Map

The Self-Organizing Map (SOM) is primarily used for visualizing high-dimensional data. It consists of a grid of evenly spaced neurons, each associated with a weight vector that is iteratively adjusted during the competitive learning process. When a training sample is presented, the best matching unit (BMU) is identified as the neuron with the smallest distance to the sample. The weight vector of this neuron, and to a certain extent those of its neighbouring neurons, are then updated to better represent the training sample. A trained SOM captures the underlying patterns in the data and is expected to map similar high-dimensional data points to nearby neurons in the two-dimensional grid [28].

There are authors that apply the dimensionality reduction capabilities of SOMs to anomaly detection tasks [19, 20, 29]. This is done by using the quantization error as an anomaly score [44].

The quantization error for an input sample x_i is calculated as the distance, according to some distance function defined later in the design, of the input sample to its best matching unit (BMU) in the SOM [3] as stated in Equation 3.4, where x_i is a data sample. The underlying assumption is that for a SOM trained with normal data, the quantization error for anomalies is significantly higher compared to normal data.

$$qe = dist(x_i, bmu(x_i)) \tag{3.4}$$

The neurons in a SOM can be arranged in different topologies, affecting the number of neurons in their neighbourhood. Hexagonal architectures as in Figure 3.5 lead to six neurons in the close neighbourhood. Whereas in rectangular arrangement of neurons, the close neighbourhood contains eight other neurons as visualized in Figure 3.4. This design uses a rectangular architecture to place the neurons.

An important decision when designing a SOM is the size of the map, defining the number of neurons it contains. The parameters to decide on are x for the length and y for the height. In the literature, there is no theoretic argument that helps to choose a suitable

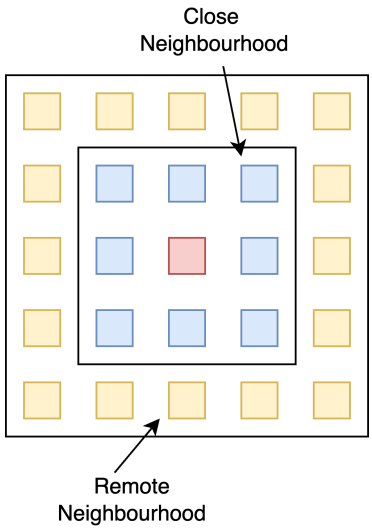

Figure 3.4: Rectangular SOM architecture with indicated neighbourhoods

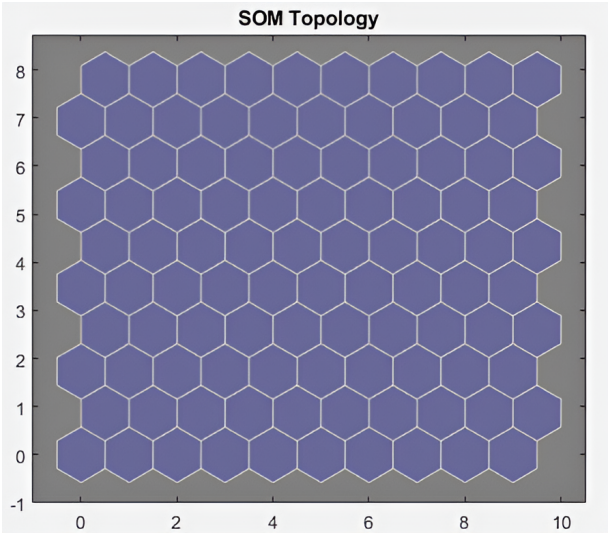

Figure 3.5: Hexagonal architecture for a Self-Organizing Map [12]

number of neurons. However, the documentation to the MiniSOM [51] package used in the implementation chapter, recommends to set the number of neurons around $5 \cdot \sqrt{N}$ with N being the number of samples that the model is trained on. This guideline for the number of neurons is followed as an orientation. The neurons are equally distributed across the two sidelengths x and y, leading to a square map. The trade-off in deciding the number of neurons is similar to complexity vs simplicity in the autoencoder models. Too many neurons involved will lead to low quantization errors even for anomalies. While too few neurons limit the models capacity to learn normal patterns in the data.

A definition of neighbourhood is needed in order for the model to know which surrounding neurons are affected by updates in which magnitude. The gaussian neighbourhood function is expected to provide a reasonable definition, putting high weight onto the close neighbourhood and quickly declining the weight for remote neurons. The spread of the gaussian function is set around $\sqrt{x^2 + y^2}$ initially, with x and y being the sidelenghts of

the map. This spread value is recommended by the MiniSom documentation [51]. With further iterations, the spread σ decreases to one according to the decay function in Equation 3.5, where σ_0 is the initial spread, t is the current iteration and n the total number of map updates.

$$\sigma_t = \sigma_0 + t * (1 - \sigma_0)/n \tag{3.5}$$

To calculate the BMU and the quantization error for a given input sample, the SOM needs a notion of distance between two points in the feature space. In this implementation the often used standard of the euclidean distance is used. It is defined in Equation 3.6, according to [47], where a and b are two vectors with a_i respectively b_i representing the value for feature i in samples a and b.

$$dist(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2} \tag{3.6}$$

Besides the architectural hyperparameters, there are decisions to be taken for the training phase. The learning rate defines how strong each weight update affects the map. In order for the model to converge, it is necessary that the learning rate decreases with subsequent training iterations [44]. The learning rate at a given iteration t, abbreviated as α_t, will linearly decrease to one according to the decay function in Equation 3.7, where α_0 is the initial learning rate and n is the total number of map updates.

$$\alpha_t = \alpha_0 * (1 - t/n) \tag{3.7}$$

In a training iteration, also called epoch, the best matching unit for each training sample is calculated and the neighbourhood is updated depending on the learning rate and its proximity to the BMU. The number of epochs when training a SOM is usually low since they are able to capture the topology of the data space quickly, making further training epochs unnecessary. This tendency is in contrast to the autoencoder model, optimizing a loss function through gradients, requiring considerably more epochs. In this SOM design, 5 epochs are used in the training phase, which is the same number of iterations used in [3].

A SOM takes multivariate data as input. To represent the sound signal in this multi-dimensional form, various spectral features are extracted from the time, frequency and time-frequency representations of sound. Since the derived features vary in the value range that they occupy, the SOM is biased towards giving more weight to the high-valued features. To put equal weight onto all features, feature-wise min-max normalization has to be applied during preprocessing.

The normalized feature vectors, extracted from the normal sound samples are split into 70% training, 15% validation and 15% testing sets. The training set is the SOM input during training. It is expected that the SOM learns to fit these normal features well, leading to a low quantization error for the training data. The validation set is used for hyperparameter tuning and consists of normal samples and anomalies. The test set is

used to check how well the fitted SOM is able to generalize the learned pattern to unseen sound samples.

After being trained during the defined number of epochs, the model is ready for inference. To classify the inference results, a threshold for the quantization error θ needs to be defined. Equation 3.8 provides a decision criterion based on the quantization error.

$$\begin{aligned} qe(x_i) < \theta &\longrightarrow normal \\ qe(x_i) \geq &\longrightarrow anomaly \end{aligned} \tag{3.8}$$

As stated in the design of the convolutional autoencoder model, it is not practical to define a threshold before the results of the inference are available. In the evaluation chapter, inference will be applied and the classification results depending on different thresholds will be evaluated.

Chapter 4

Implementation

This Chapter presents and discusses the implementation of the preprocessing pipelines and the designs introduced in Chapter 3. The focus is laid on the models tied to the hydropower dataset. The deviation for models adjusted to the additional MIMII dataset [42] are stated. The code snippets outline major parts of the implementation, written in Python. The libraries used are stated.

4.1 Preprocessing

To process audio files, a Python package for music and audio analysis, called Librosa [30], is used. It provides various functionalities related to sound preprocessing. Numpy [38] is another Python package for scientific computing. The main use of Numpy is to provide utility functions that ease the handling of data in arrays or matrices. The third package used is called Matplotlib.pyplot [33], providing an interface for data visualizations. In the following snippets Numpy is referred to as np and Matplotlib.pyplot as plt.

The process of recording sound includes mainly two steps, sampling and quantization. This is done by the analogue-digital converter built into the used microphone.

The code snippet in Listing 4.1 is used to visualize the content of a sound file as a waveform. First, a file is loaded with the sample rate of 22'050. The resulting signal is an array containing an amplitude value for each recorded sample. This will be the y-axis in the result plot. To obtain an x-axis representing time in milliseconds, Numpy's *linspace* function is used. It creates an evenly spaced array from the start to the end argument with the third argument being the number of desired values. The plot function takes two lists of equal length as input, where the first defines the x values and the second the y values. A title is added and the axes are labeled. Finally, the created waveform is saved to disk with a resolution of 300 dots per inch. Figure 4.1 shows the resulting visualization of a normal sound file that was recorded in the Hydropower plant.

```
1 signal, sr = librosa.load(filepath, sr=22050)
2 time = np.linspace(start=0, stop=len(signal)*1000 / sr, num=len(signal))
3
```

```
4 plt.plot(time, signal)
5 plt.title('Waveform')
6 plt.ylabel('Amplitude')
7 plt.xlabel('Time (ms)')
8 plt.savefig('waveform-normal.png', dpi=300)
```

Listing 4.1: Code to visualize a sound signal as a waveform

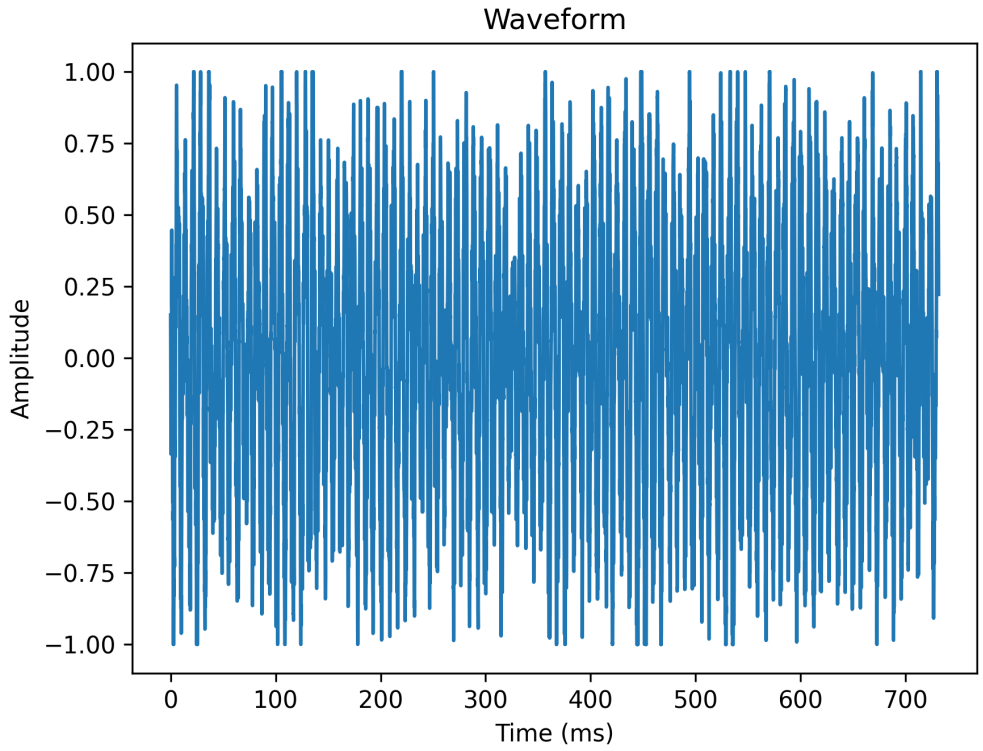

Figure 4.1: Waveform of a normal sound file recorded in the HPP

To get an overview of the frequencies present in the signal, the Fourier Transform is applied. Listing 4.2 shows the code to obtain a frequency spectrum. As a first step, the signal is loaded again from a sound file with the sample rate of 22'050. The Fourier transform is then applied to the signal using the *fft* function contained in the Numpy package. Since only the magnitude information is relevant for the frequency spectrum, the phase information, which is the imaginary part of the *fft* result, is removed. Similar to the time axis in the waveform visualization, there is a frequency axis built with the *linspace* function. The Nyquist-Shannon sampling theorem states that only the frequencies until half the sampling rate can be captured reliably [41]. For this reason, half of the frequencies and their corresponding magnitude values cannot be used in analysis and are therefore truncated in the visualization. When applying the code to a normal sound file, the result is show in Figure 4.2. It is observable that one low frequency around 128 HZ is very dominant while high frequencies are absent.

```
signal, sr = librosa.load(filepath, sr=22050)

fourier_transform = np.fft.fft(signal)
magnitudes = np.abs(fourier_transform)
frequencies = np.linspace(0, sr, sr)

num_usable_frequencies = int(sr * 0.5)

plt.plot(frequencies[:num_usable_frequencies], magnitudes[:
    num_usable_frequencies])
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Frequency Spectrum')
plt.savefig('frequency_spectrum-normal.png', dpi=300)
```

Listing 4.2: Code to visualize the frequency spectrum of a sound signal

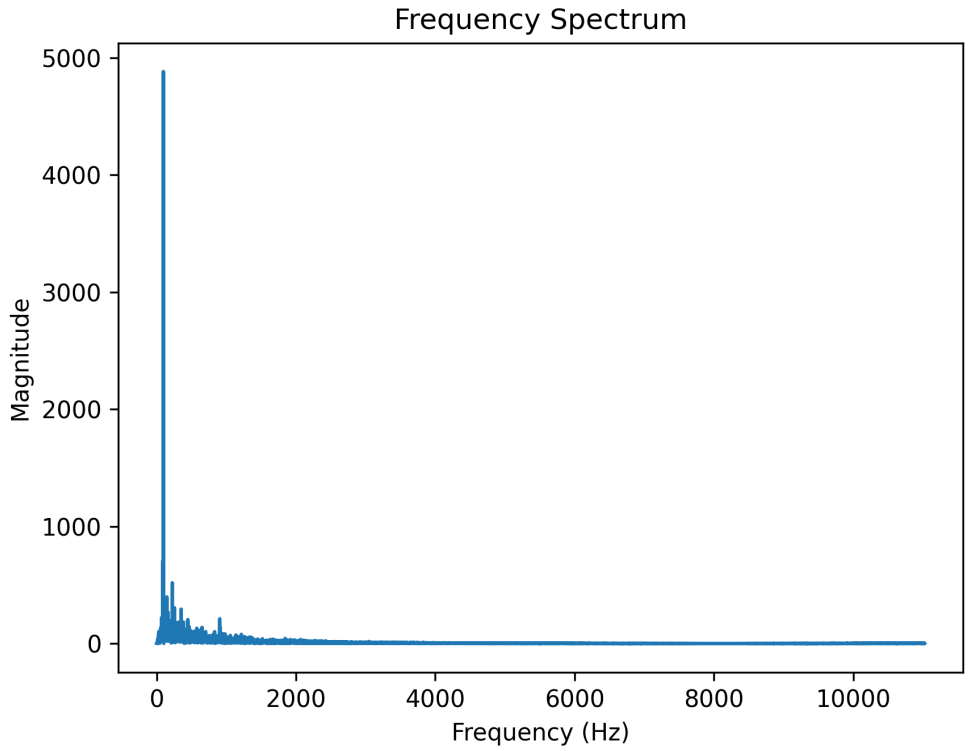

Figure 4.2: Frequency spectrum of a normal sound file recorded in the HPP

In order to get information about both the time and frequency domain, the Short-Time Fourier Transform (STFT) is applied in the code snippet 4.3. *Librosa* has the function *stft* to perform this step, it requires to define the frame lengths in the parameter *n_fft*. The documentation of the Librosa package recommends to use a power of two for the frame length, since this will lead to speed-up in the FFT algorithm. The bigger the frame length, the better the frequency resolution and the lower the time resolution gets. Setting 1'024 samples per frame creates 513 frequency bands that can be distinguished. It is

assumed in this thesis that this frequency resolution is fine-grained enough for the task of anomaly detection. The time resolution is not only dependent on the frame size. The number of time bands is highly coupled to the hop length, which is the step size between overlapping frames. By choosing a hop length that is a quarter of the frame size, the resulting spectrogram differs between 64 time bands, each having a duration of 11.5 ms. The *hann* windowing function is applied to the full length of each frame to reduce spectral leakage.

The result of the short time Fourier transformation is a two dimensional array containing a complex number for each frequency and time band combination. The phase information is not needed in this context and is removed with the call to *np.abs*, which transforms the complex numbers to real numbers by eliminating their imaginary part. The spectrogram is scaled to decibels to compress the value range and make low amplitude frequencies better visible.

Librosa provides the function *specshow* to visualize a spectrogram. It takes the sample rate and the hop length used earlier as input to calculate the scale of the time axis. Additionally, the information that the frequency scale was converted to decibel is provided through the 'log' argument to the *y_axis* parameter.

```
1  signal, sr = librosa.load(filepath, sr=22050)
2
3  short_time_fourier_transform = librosa.stft(signal, n_fft=1024,
       hop_length=256, window='hann', win_length=1024)
4  short_time_magnitudes = np.abs(short_time_fourier_transform)
5  db_magnitudes = librosa.amplitude_to_db(short_time_magnitudes)
6
7  plt.figure()
8  librosa.display.specshow(db_magnitudes, sr=sr, hop_length=256, x_axis='
       time', y_axis='log')
9  plt.colorbar()
10 plt.title('Spectrogram')
11 plt.savefig('spectrogram-normal.png', dpi=300)
```

Listing 4.3: Code to apply stft to a signal and visualize it as a spectrogram

In the following subsections, the preprocessing steps coupled to the specific models are explained. The autoencoders use a type of spectrograms as input, while the Self-Organizing Map expects various sound features.

4.1.1 Spectrograms

The code in Listing 4.4 walks through the files in a given directory using the OS package which is part of the Python standard library. It is assumed that the directory contains sound files in the wav format of equal length that are named with subsequent integers. The files are sorted in ascending order according to their file name to ensure they are processed in the correct order.

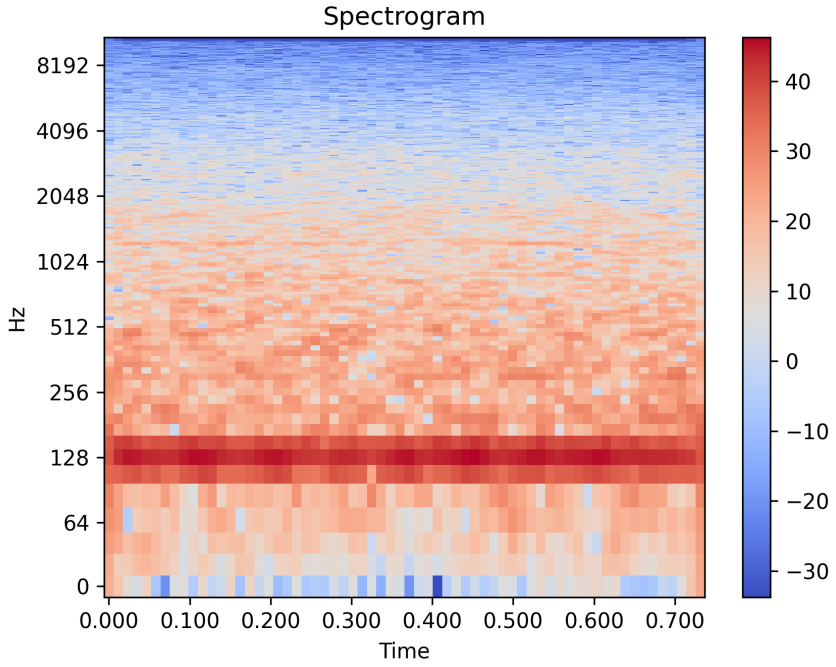

Figure 4.3: Spectrogram of a normal sound file recorded in the HPP

Librosa loads the content of a file with the sample rate of 22'050. There exists a shortcut function in Librosa to directly extract the mel spectrogram of a given signal. Instead of pre-computing the spectrogram with the short time Fourier transform, this function does that internally. The arguments that need to be given are again the frame length, the hop length, and the windowing function which are chosen analogue to the preprocessing steps in Listing 4.3, where the trade-offs in choosing these parameter values are explained. In addition to the already seen arguments, the number of mel bands has to be specified. In related work, the number of mel bands varies between 32 and 128. In this thesis, 64 mel bands are used to balance frequency resolution and resources needed to compute the mel spectrogram.

The result of the mel spectrogram function is a power spectrum. To convert it to a decibel representation, *power_to_db* function is used.

```
1 spectrograms = []
2     for _, _, files in os.walk(input_dir):
3         for file in sorted(files, key=lambda x: int(x.strip('.wav'))):
4             input_path = os.path.join(input_dir, file)
5             signal, sr = librosa.load(input_path, sr=22050)
6             melspectrogram = librosa.feature.melspectrogram(y=signal,
                  n_fft=1024, hop_length=256, n_mels=64, window='hann',
                  win_length=1024)
7             db_melspectrogram = librosa.power_to_db(melspectrogram)
8
9             spectrograms.append(db_melspectrogram)
```

Listing 4.4: Python code to build melspectrograms out of sound files

As a next preprocessing step, the dimensions of the spectrograms are mapped to those

of a greyscale image. This is needed since convolutional neural networks expect images as input. The digital representation of an image is a three dimensional array with one axis representing the horizontal and another the vertical pixels. The third dimension is the channel. In colour images three channels exist for the colour values of red, green and blue. Whereas in greyscale images, there is only one channel for brightness. Currently the produced mel spectrograms are stored as two dimensional arrays, with a time and a frequency axis. By wrapping each magnitude value into its own array, the third dimension is added and the dimensions match with a greyscale image. The code snippet 4.5 first changes the data type from a Python standard list to a Numpy array in order to support the following operation. To keep all the existing dimensions, the spread operator is used. Then the new axis is added.

```
1 spectrograms = np.array(spectrograms)
2 spectrograms = spectrograms[..., np.newaxis]
```

Listing 4.5: Transformation of the spectrogram to match the dimensions of a greyscale image

To prepare the data for training, a split into training, validation and test set is performed. The Python package Sklearn [39] offers a solution for randomized splitting. As a first step in Listing 4.6, the spectrograms are split between training data and rest, where the rest occupies 30% of the original data. Secondly, the rest is split again into test set and validation set which are of equal length. Resulting in 70% of the data reserved for training purpose and 15% each for validation and testing.

```
1 normal_train, normal_rest = sklearn.model_selection.train_test_split(
    normal_spectrograms, test_size=0.3)
2 normal_validation, normal_test = sklearn.model_selection.
    train_test_split(normal_rest, test_size=0.5)
```

Listing 4.6: Splitting of the data into training, validation and test sets

The spectrograms for all three datasets are normalized using the Equation 4.1 according to [11]. Listing 4.7 defines a function in Python code that uses this formula. It is then applied to the datasets. The normalized mel spectrograms are ready to be used as input to train the autoencoder. The obtained result is saved to disk with the Numpy save function, such that the preprocessing steps don't have to be repeated when training different model specification during the hyperparameter tuning described in the evaluation chapter.

$$x_{normalized} = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

```
1 def normalize_spectrograms(spectrograms):
2     normalized_spectrograms = []
3     for spectrogram in spectrograms:
4         norm_melspectrogram = (spectrogram - spectrogram.min()) / (
              spectrogram.max() - spectrogram.min())
5         normalized_spectrograms.append(norm_melspectrogram)
6     return np.array(normalized_spectrograms)
7
8 normal_train_scaled = normalize_spectrograms(normal_train)
9 normal_validation_scaled = normalize_spectrograms(normal_validation)
```

```
normal_test_scaled = normalize_spectrograms(normal_test)

np.save(output_path, normal_train_scaled)
np.save(output_path, normal_validation_scaled)
np.save(output_path, normal_test_scaled)
```

Listing 4.7: Normalization of the spectrograms

Since the MIMII files have a longer duration than those in the Hydropower dataset, the resulting spectrograms have different dimensions. This poses a problem if the number of mel bands or time bands is not consistently divisible by two. More specifically, in the context of the autoencoder designs, both dimensions must be divisible by 2^n where n is the number of downsampling layers in the encoder. In this case, with three downsampling layers, both the mel band and time band numbers must be divisible by $2^3 = 8$.

During the encoding process, each downsampling step halves the spectrogram's spatial dimensions. While Keras can handle non-divisible dimensions by applying floor division during downsampling, this creates a mismatch during upsampling in the decoder. Since upsampling doubles the dimensions, the reconstructed spectrogram ends up larger than the original input, leading to misalignment.

For the MIMII dataset, the number of time bands in the spectrograms is 862, which does not meet the divisibility requirement. To address this, an additional preprocessing step is applied, where two time bands are zero-padded, extending the time-band dimension to 864. This adjustment ensures that the spectrogram shape remains compatible with the architecture of the autoencoder, preserving spatial integrity throughout the encoding and decoding processes.

The code in Listing 4.8 implements the additional preprocessing step that is necessary for MIMII spectrograms only. After loading the spectrograms from a given file, they are converted from a Numpy array to a Python list, since the dimensions of the data will be temporarily inhomogeneous, which is not allowed for Numpy arrays. Then the function loops for each spectrogram through all frequency bands and adds two zero valued time bands. Finally, the spectrograms are converted back to a Numpy array and are stored in the same file again.

```
def zero_padding_time_bands_file(file_path):
    spectrograms = np.load(file_path)
    spectrograms = spectrograms.tolist()
    number_of_spectrograms = len(spectrograms)
    number_of_frequency_bands = len(spectrograms[0])
    for i in range(0, number_of_spectrograms):
        for j in range(0, number_of_frequency_bands):
            spectrograms[i][j].append([0])
            spectrograms[i][j].append([0])

    spectrograms = np.array(spectrograms)
    np.save(file_path, spectrograms)
```

Listing 4.8: Function that adjusts the MIMII spectrograms to the required number of time bands by zero padding

4.1.2 Features

Table 4.1 describes the selected features in terms of the domain where they are extracted and the concepts that they measure. The Mel frequency cesptral coefficients are a special type of feature since they produce multiple values for a single frame in contrast to the other features, producing exactly one value per frame.

Table 4.1: Selected features

Name	Domain	Measurement
Root mean square energy	Time	Indicates the power of a signal
Zero crossing rate	Time	Distinguishes if more high or low frequencies are present
Spectral centroid	Time-Frequency	Indicates the centre of mass, perceived as brightness of the sound
Spectral bandwidth	Time-Frequency	Difference between upper and lower frequencies
Spectral contrast	Time-Frequency	Indicates at which frequency the energy is centred upon
Spectral flatness	Time-Frequency	Measures how uniformly energy is distributed
Spectral roll-off	Time-Frequency	The frequency below which 85% of the energy lies
Mel-frequency cepstral coefficients	Time-Frequency	Captures the timbre of a sound signal

All the selected features can be extracted with Librosa. The code in Listing 4.9 defines a function to extract the features of a given sound file. The root mean square energy and the zero crossing rate can be extracted from the signal directly. In order to obtain a useful resolution, the functions *rms* and *zero_crossing_rate* split the signal into frames. The frame lengths and the hop length are the parameters to guide this split. They are set to 1'024 samples per frame and a distance of 256 samples between two frames. This results in overlapping frames which would not be needed if all the features were in the time domain, since spectral leakage only occurs when applying the Fourier Transform. However, to match the dimensions of all obtained feature values, the frame and hop lengths need to be the same across all feature extractions. After extraction, each feature is stored in a Numpy array. Since Librosa not only returns the feature values but also their data type, only the first part of the result is used.

The STFT is applied to the signal using the same arguments to guide the framing process. Again the phase information is discarded with *np.abs*. Now all the time-frequency features, except the mel frequency cesptral coefficients can be extracted using the obtained

spectrogram as input. All the features obtained so far are stacked together to form a two dimensional array. The outer part contains the different time slots and the inner part the feature values for a specific slot.

As preparation to the extraction of the mel frequency cepstral coefficients, the mel spectrogram is produced out of the signal with the same frame and hop lengths used in the other feature extractions. To balance resulting precision and required computational resources, the number of mel bands is chosen as 64. It is the same number as used to create mel spectrograms for the autoencoder models. The *mfcc* function provided by Librosa expects a mel spectrogram scaled to decibel, which is obtained using the *power_to_db* function. As a result, the *mfcc* function returns the 64 specified coefficients for each frame in the mel spectrogram. Each of these coefficients is stacked onto the feature array.

```
1  def extract_features_from_file(audio_path):
2      signal, sr = librosa.load(audio_path, sr=22050)
3
4      rms_energy = librosa.feature.rms(y=signal, frame_length=1024,
           hop_length=256)
5      rms_energy = np.array(rms_energy[0])
6
7      zero_crossing_rate = librosa.feature.zero_crossing_rate(y=signal,
           frame_length=1024, hop_length=256)
8      zero_crossing_rate = np.array(zero_crossing_rate[0])
9
10     stft = librosa.stft(signal, n_fft=1024, hop_length=256)
11     stft = np.abs(stft)
12
13     spectral_centroid = librosa.feature.spectral_centroid(S=stft)
14     spectral_centroid = np.array(spectral_centroid[0])
15
16     spectral_bandwidth = librosa.feature.spectral_bandwidth(S=stft)
17     spectral_bandwidth = np.array(spectral_bandwidth[0])
18
19     spectral_contrast = librosa.feature.spectral_contrast(S=stft)
20     spectral_contrast = np.array(spectral_contrast[0])
21
22     spectral_flatness = librosa.feature.spectral_flatness(S=stft)
23     spectral_flatness = np.array(spectral_flatness[0])
24
25     spectral_rolloff = librosa.feature.spectral_rolloff(S=stft)
26     spectral_rolloff = np.array(spectral_rolloff[0])
27
28     feature_vectors = np.column_stack((rms_energy, spectral_centroid,
           spectral_bandwidth, spectral_contrast, spectral_flatness,
           spectral_rolloff, zero_crossing_rate))
29
30     melspectrogram = librosa.feature.melspectrogram(y=signal, sr=sr,
           n_fft=1024, hop_length=256, n_mels=64)
31     mfccs = librosa.feature.mfcc(S=librosa.power_to_db(melspectrogram),
           sr=sr, n_mfcc=64)
32     mfccs = np.array(mfccs)
33
34     for mfcc in mfccs:
35         feature_vectors = np.column_stack((mfcc, feature_vectors))
36
```

```
37      return feature_vectors
```

Listing 4.9: Python function to extract the selected features from a given sound file

As a next step, feature normalization is applied to each feature according to 4.1, such that the values of every feature lie between zero and one, while the underlying relationship per feature is preserved. In Listing 4.10 the feature vectors are transposed such that the outer array represents the different features while the inner array contains the feature values for a specific time interval. Each feature is now normalized with respect to the other values within that feature. The result is transposed again to transform it to the shape of the original feature vectors.

```
1 def feature_normalization(feature_vectors):
2     vectors_t = feature_vectors.T
3     for i in range(len(vectors_t)):
4         vectors_t[i] = (vectors_t[i] - vectors_t[i].min()) / (vectors_t[
              i].max() - vectors_t[i].min())
5
6     return vectors_t.T
```

Listing 4.10: Feature wise min max normalization

The feature vector is now split into training, validation and test sets. The involved steps are equal to the process of data splitting described in the code snippet 4.6 used to split spectrogram into the various datasets.

4.2 Dense Autoencoder

To implement the dense autoencoder model, the Tensorflow Keras package for Python [48] is used. It provides a functional API to combine the different layers of the autoencoder.

In Listing 4.11 the Python class *Autoencoder_Dense* is created. Based on the given arguments, the spectrogram shape, the output sizes of the layers and the bottleneck size, a dense autoencoder is constructed. All the arguments are stored in private variables. Additionally the layer number and the size of the flattened input are stored for convenience. The build method is called, which sequentially builds the encoder, the decoder and combines them together in a third step.

```
1  class AutoencoderDense:
2      def __init__(self, input_shape, layer_output_sizes, bottleneck_size)
           :
3          self._input_shape = input_shape
4          self._layer_output_sizes = layer_output_sizes
5          self._bottleneck_size = bottleneck_size
6
7          self._number_of_layers = len(self._layer_output_sizes)
8          self._flattened_input_size = input_shape[0] * input_shape[1]
9
10         self._encoder = None
11         self._decoder = None
12         self._model = None
```

```
13
14          self._build()
15
16      def _build(self):
17          self._build_encoder()
18          self._build_decoder()
19          self._build_autoencoder()
```

Listing 4.11: Definition of the Autoencoder$_{D}enseclass$

Listing 4.12 presents the code used to construct the individual components of the dense autoencoder. The encoder begins with a two-dimensional input layer matching the spectrogram's shape. This input is flattened into a one-dimensional sequence before being processed. For each specified layer, a fully connected (Dense) layer is created with the corresponding output size and sequentially added to the model. Following these layers, the bottleneck is introduced as another Dense layer with a reduced number of neurons. A Keras model is then instantiated, mapping the input spectrogram to progressively lower-dimensional representations until reaching the bottleneck. This model is stored in a private variable for later access.

The decoder reverses this process, starting from the bottleneck-sized input and reconstructing the data by applying the layer configurations in reverse order. Dense layers are used to upsample the data according to the predefined output sizes. A final Dense layer restores the data to its original flattened dimension. A sigmoid activation function is applied to ensure the output remains within the [0,1] range, matching the normalized spectrogram input. Lastly, a Reshape layer reconstructs the one-dimensional output back into its original two-dimensional spectrogram form. The resulting Keras model, which takes bottleneck-sized input and processes it through the decoder layers, is stored in a local variable.

The final build method integrates the encoder and decoder into a complete autoencoder. The input data first passes through the encoder, reducing its dimensionality, before being reconstructed by the decoder through the upsampling layers.

```
1   def _build_encoder(self):
2           encoder_input = Input(shape=self._input_shape, name='
                encoder_input')
3           flatten = Flatten()
4           layers = flatten(encoder_input)
5           for layer_index in range(self._number_of_layers):
6               dense_layer = Dense(self._layer_output_sizes[layer_index])
7               layers = dense_layer(layers)
8
9           bottleneck = Dense(self._bottleneck_size, name='bottleneck')(
                layers)
10
11          self._encoder = Model(encoder_input, bottleneck, name='encoder')
12
13      def _build_decoder(self):
14          decoder_input = Input(shape=(self._bottleneck_size, ), name='
                decoder_input')
15          layers = decoder_input
16          for layer_index in reversed(range(0, self._number_of_layers)):
```

```
17              dense_layer = Dense(self._layer_output_sizes[layer_index])
18              layers = dense_layer(layers)
19
20          dense_layer = Dense(self._flattened_input_size)
21          layers = dense_layer(layers)
22
23          sigmoid = Activation("sigmoid", name="sigmoid_layer")
24          layers = sigmoid(layers)
25
26          output_layer = Reshape(self._input_shape)(layers)
27          self._decoder = Model(decoder_input, output_layer, name='decoder
                ')
28
29
30      def _build_autoencoder(self):
31          input = Input(shape=self._input_shape, name='autoencoder_input')
32          layers = self._decoder(self._encoder(input))
33          self._model = Model(input, layers, name='autoencoder')
```

Listing 4.12: Building process of the different Autoencoder parts

4.3 Convolutional Autoencoder

Again, the Tensorflow Keras package is used to build the convolutional autoencoder. The Python class *Autoencoder_Convolutional* is created in Listing 4.13. As instantiation arguments the input shape in the form of a tuple and the number of neurons in the latent space as an integer need to be passed. Additionally, the size and the number of filters as well as the stride have to be defined with a list each, containing the values to specify the convolutional layers. The class expects all lists defining the layers to be of equal length.

All arguments are stored as private class variables. The number of layers is explicitly stored, since it will be needed at various places when building the convolutional autoencoder. Additionally, other private variables, which will be used in further steps, are initialized. The call to *self._build* starts the building process of the encoder, the decoder and as a last step, the whole autoencoder is constructed by putting the two pieces together.

```
1  class Autoencoder_Convolutional:
2      def __init__(self, input_shape, filter_sizes, numbers_of_filters,
           strides, latent_space_size):
3          self._input_shape = input_shape
4          self._filter_sizes = filter_sizes
5          self._numbers_of_filters = numbers_of_filters
6          self._strides = strides
7          self._latent_space_size = latent_space_size
8
9          self._number_of_layers = len(self._numbers_of_filters)
10         self._shape_before_bottleneck = None
11
12         self.encoder = None
13         self.decoder = None
14         self.model = None
15
```

```
16          self._build()
17
18      def _build(self):
19          self._build_encoder()
20          self._build_decoder()
21          self._build_autoencoder()
```

Listing 4.13: Definition of the Autoencoder_Convolutional class

The code shown in Listing 4.14 starts by creating the input of the previously defined shape. Keras uses an architectural style involving a chain of blocks. First a building block is defined which returns a function that is then called with the already built blocks as argument. The layers variable holds this chain that is growing step by step. For each layer specified when instantiating the autoencoder, a convolutional layer is created and then added to the chain of layers. The convolutional layers consist of the convolution itself, a ReLu activation function and a batch normalization.

The shape of the data prior to the bottleneck is stored to a variable, since this information will be needed in the decoder part. To create the latent space, the data is first flattened and then fully connected to a dense layer with the number of neurons specified in the late space size. The encoder is saved as a Keras Model with the specified input and the created latent space as output.

```
1   def _build_encoder(self):
2           encoder_input = Input(shape=self._input_shape, name='
               encoder_input')
3           layers = encoder_input
4           for layer_index in range(self._number_of_layers):
5               layers = self._append_convolutional_layer(layer_index,
                   layers)
6
7           self._shape_before_bottleneck = keras_backend.int_shape(layers)
               [1:]
8           bottleneck = Flatten()(layers)
9           bottleneck = Dense(self._latent_space_size, name='encoder_output
               ')(bottleneck)
10          self.encoder = Model(encoder_input, bottleneck, name='encoder')
11
12      def _append_convolutional_layer(self, layer_index, layers):
13          layer_number = layer_index + 1
14          conv_layer = Conv2D(
15              filters=self._numbers_of_filters[layer_index],
16              kernel_size=self._filter_sizes[layer_index],
17              strides=self._strides[layer_index],
18              padding='same',
19              name='encoder_conv_layer_' + str(layer_number)
20          )
21          layers = conv_layer(layers)
22          layers = ReLU(name='encoder_relu_'+str(layer_number))(layers)
23          layers = BatchNormalization(name='encoder_bn_' + str(
               layer_number))(layers)
24          return layers
```

Listing 4.14: Building process of the encoder

Listing 4.15 specifies the building process of the decoder part, which mirrors the building blocks of the encoder. The decoder uses the dimensions of the latent space as input size. Its input is then fully connected to a dense layer with the same number of neurons used in the encoder before downsizing to the latent space. The neurons are then reshaped, such that they match with the two dimensional structure that was present before the flattening was applied in the decoder.

The layer arguments specified at the instantiation of the autoencoder class are now used in reversed order. The inverse operation of a convolution is the convolutional transpose. Additionally to that building block, a ReLu activation function and batch normalization is added to each layer. The sigmoid activation function is used as last building block to ensure the output values are in the range between zero and one, like the normalized spectrograms used as input. The constructed decoder is stored in a private class variable as a Keras Model.

```
1       def _build_decoder(self):
2           decoder_input = Input(shape=(self._latent_space_size, ), name='
                decoder_input')
3           number_of_neurons = np.prod(self._shape_before_bottleneck)
4           layers = Dense(number_of_neurons, name='decoder_dense')(
                decoder_input)
5           layers = Reshape(target_shape=self._shape_before_bottleneck)(
                layers)
6
7           for layer_index in reversed(range(1, self._number_of_layers)):
8               layers = self._append_conv_transpose_layer(layer_index,
                    layers)
9
10          convolutional_transpose_layer = Conv2DTranspose(
11              filters=1,
12              kernel_size=self._filter_sizes[0],
13              strides=self._strides[0],
14              padding='same',
15              name='decoder_conv_transpose_layer_'+str(self.
                    _number_of_layers))
16          layers = convolutional_transpose_layer(layers)
17          output_layer = Activation("sigmoid", name="sigmoid_layer")
18          layers = output_layer(layers)
19          self.decoder = Model(decoder_input, layers, name='decoder')
20
21      def _append_conv_transpose_layer(self, layer_index, layers):
22          layer_number = self._number_of_layers - layer_index
23          conv_transpose_layer = Conv2DTranspose(
24              filters=self._numbers_of_filters[layer_index],
25              kernel_size=self._filter_sizes[layer_index],
26              strides=self._strides[layer_index],
27              padding='same',
28              name='decoder_conv_transpose_layer_'+str(layer_number))
29          layers = conv_transpose_layer(layers)
30          layers = ReLU(name='decoder_relu_'+str(layer_number))(layers)
31          layers = BatchNormalization(name='decoder_bn_'+str(layer_number)
                )(layers)
32          return layers
```

Listing 4.15: Building process of the decoder

The third step of the build method is the combination of the encoder and decoder to construct the complete autoencoder model. In Listing 4.16, the input shape of the Autoencoder is defined, ensuring it matches the input shape of the encoder. The model is built by sequentially passing the input through the encoder, followed by the decoder. Finally, the assembled autoencoder is stored as a Keras model in a class variable for later use.

```
1 def _build_autoencoder(self):
2     input = Input(shape=self._input_shape, name='autoencoder_input')
3     layers = self.decoder(self.encoder(input))
4     self.model = Model(input, layers, name='autoencoder')
```

Listing 4.16: Combination of encoder and decoder

The architecture of the autoencoder is defined by its layers. For the model to be useful, it needs to be additionally configured for training. The code in Listing 4.17 defines two methods, compile and train, which are essential for training an autoencoder model using Keras. The compile method configures the model for training by setting an optimizer and a loss function. Specifically, it initializes the Adam optimizer with a default learning rate of 0.0001 and uses the Mean Squared Error (MSE) loss function. The train method is responsible for fitting the model to the training data. Since an autoencoder learns to reconstruct its input, both the input and target values are the same. The training process runs for a specified number of epochs with a given batch size, while validation data is used to monitor the model's performance on unseen data. Additionally, shuffling is enabled to improve generalization by preventing the model from learning patterns based on the order of the training samples.

```
1     def compile(self, learning_rate=0.0001, loss_function=
          MeanSquaredError()):
2         optimizer = Adam(learning_rate=learning_rate)
3         self._model.compile(optimizer=optimizer, loss=loss_function)
4
5     def train(self, train_data, validation_data, batch_size, epochs):
6         return self._model.fit(
7             train_data, train_data,
8             batch_size=batch_size,
9             epochs=epochs,
10            validation_data=(validation_data, validation_data),
11            shuffle=True
12        )
```

Listing 4.17: Training configuration of the Autoencoder

4.4 Self-Organizing Map

The Python package MiniSom [51] provides a minimalistic implementation of a SOM. The configuration of this implementation, as shown in Listing 4.18, adheres to the design principles outlined in Chapter 3. The SOM consists of 361 neurons arranged in a rectangular $19 \cdot 19$ grid, determined using the heuristic formula $5 \times \sqrt{N}$, where $N = 5'063$, corresponding to the number of available training samples.

Each training frame consists of 71 extracted feature values, which define the dimensionality of the neuron weight vectors. Consequently, each neuron is initialized with a weight vector of the same dimensionality as the input feature space. The neighbourhood function is set to a Gaussian distribution with an initial spread (*sigma*) of 26, following the rule-of-thumb estimation $\sigma = \sqrt{x^2 + y^2}$. This value is gradually reduced over successive iterations using linear decay reaching a spread of one. The Euclidean distance function is used to measure activation distances between neurons and input samples. The learning rate is initialized to 0.5, the MiniSom default, and follows an asymptotic decay function to ensure it approaches zero over time.

The SOM is initialized with neuron weights randomly sampled from the training data. To achieve this, the training dataset is loaded as a Numpy array containing all feature values. The *random_weights_init* function iterates over the neuron grid, assigning each neuron the feature vector of a randomly chosen training sample, ensuring that initial weights reflect the input distribution.

The training process is specified within the train function. By setting *use_epochs* to true, each sample in the dataset is used exactly once per iteration, with the total number of iterations (*num_iteration*) set to five, effectively defining five epochs. Without this setting, the training process would perform a fixed number of weight updates equal to *num_iteration*, potentially underutilizing the dataset. Additionally, the training data is shuffled before each iteration, preventing the model from learning patterns based on the sequence of input samples.

```
som = MiniSom(
    topology='rectangular',
    x=19,
    y=19,
    input_len = 71,
    neighborhood_function='gaussian',
    sigma=26,
    sigma_decay_function='linear_decay_to_one',
    activation_distance='euclidean',
    learning_rate=0.5,
    decay_function='linear_decay_to_zero'
)

train_frames = np.load(train_frames_file)
som.random_weights_init(train_frames)

som.train(
    data=train_frames,
    use_epochs=True,
    num_iteration=5,
    random_order=True
)
```

Listing 4.18: Configuration of the MiniSom implementation

Chapter 5

Evaluation

The task of the anomaly detection system is to classify a given sound file as either normal or containing an anomaly. The evaluation of the implemented dense and convolutional autoencoders and the Self-Organizing Map, is conducted based on two datasets: the self-created dataset of hydropower sound signals and the external MIMII dataset [42], containing normal and anomalous pump machine sounds.

For both datasets, normal signals are split into training, validation, and test sets, as described in Section 4.1. Anomalous signals are based on the same preprocessing steps but are only divided into validation and test sets of equal size. A training subset for anomalies is, in this case, not required, as the unsupervised models are trained exclusively on normal data. The anomaly validation set is used during hyperparameter tuning to identify the best-performing model, while the anomaly test set serves as an independent benchmark for final evaluation, ensuring it remains separate from data used in training and tuning.

5.1 Data Acquisition

The hydropower sound was recorded in the PSHPP Rodundwerk II in Voralberg. Together with Huber and Krummenacher, the authors of the HSG Master Project [21], four microphones were placed in the turbine building around the rotating machinery according to Figure 5.1. When the HPP is operating, there is a loud noise level from the rotating turbine and generator. Anomalies were induced by a person repeatedly clapping their hands and by hitting a shovel. To capture the sound that is normal, a recording of around 90 minutes was captured, where the HPP was using its upper reservoir to produce electricity under full load.

In 5.2, the rotating connection between the turbine, where water flows through, and the generator, where rotational energy is converted into electrical energy, is visible. In the bottom left corner, one of the four microphones captures the ambient hydropower sound along with the anomaly introduced by striking a shovel with a hammer.

In order to train the unsupervised models, only the normal sound is needed. To know how well the classification into normal and anomalous case works during inference, labels

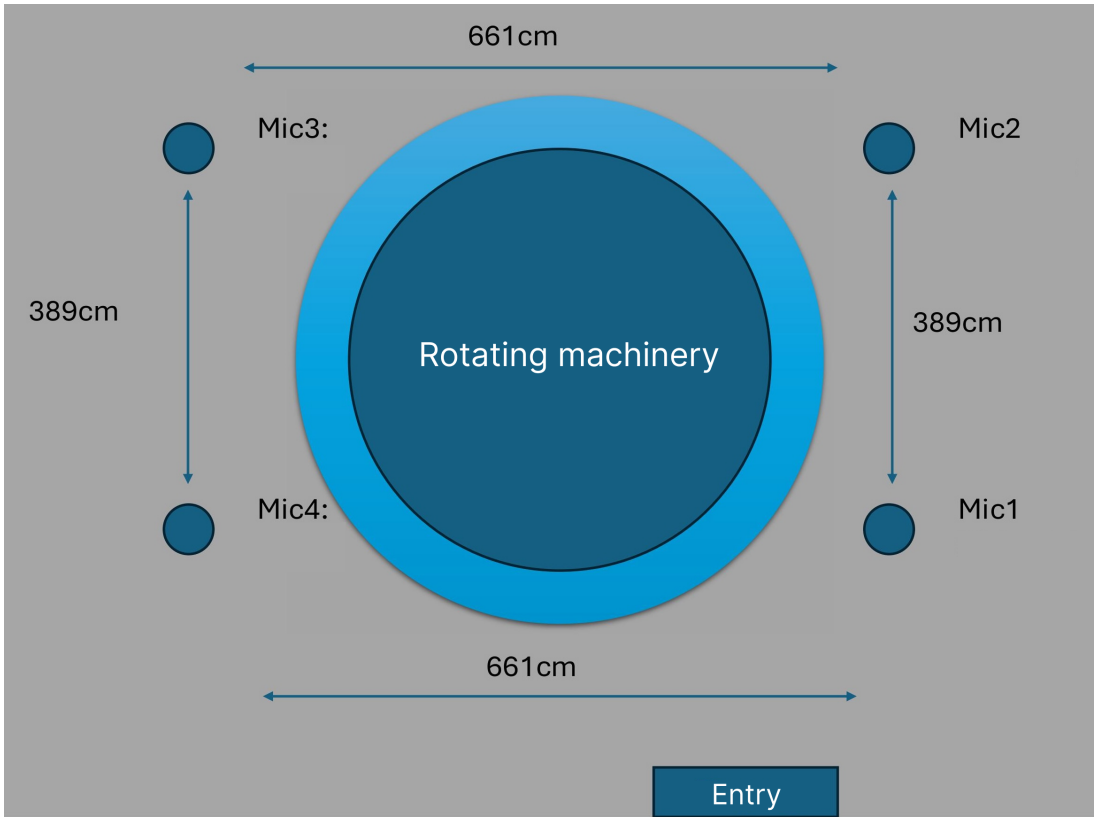

Figure 5.1: Placement of the four microphones within the turbine building

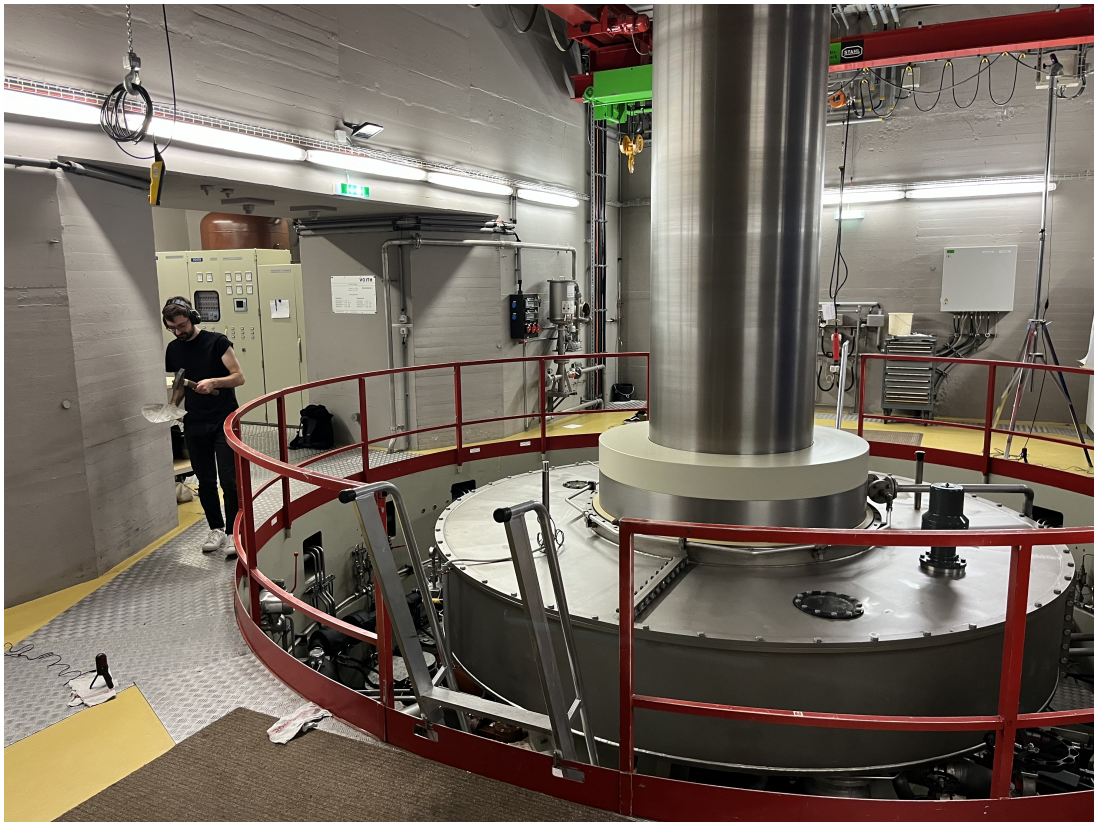

Figure 5.2: Photo of the sound capturing process

for the anomalies were created. Since the induced anomalies are audible for a short time only, the audio was split into files with a duration of around 700ms. Each sound file resulting from this split was listened to and the files containing an anomaly were isolated in a separate folder. The result of the labelling process is a dataset containing two folders where the bigger folder contains the normal samples. The other contains 80 induced anomalies from stationary clapping or walking around while clapping and stationary hitting of the shovel.

The Sound Dataset for Malfunctioning Industrial Machine Investigation and Inspection (MIMII) [42] is used to see if the results of the evaluation are generalizable to other machine sounds. It contains various machine sounds including pumps, valves and fans, where the normal and anomalous sound files are separated.

Table 5.1 summarizes how the available samples are used across the datasets. In the Hydropower dataset, each sample has a duration of approximately 700 ms, with a total of over 7,000 samples, only 1.1% of which contain anomalies. The MIMII dataset, while containing fewer samples overall, has a significantly higher anomaly proportion of 12.5%. With a sample duration of 10 seconds, the signals are longer in the MIMII dataset.

Table 5.1: Summary of datasets

	Hydropower	MIMII
Normal		
train	5063	704
validation	1085	151
test	1085	151
Anomaly		
validation	40	71
test	40	72
duration	731ms	10s

5.2 Autoencoder

This Section describes the evaluation process and gives visual insights on the example of the convolutional autoencoder that is trained and evaluated using the hydropower dataset. Figure 5.3 compares two normalized mel spectrograms from the Hydropower dataset, where the upper is representing the normal case while the lower contains an anomaly. A sudden burst of high-frequency components is observable in the anomaly spectrogram at around 270 milliseconds. In the normal spectrogram, there is a strong frequency around 200 Hz while the higher frequencies are generally declining in intensity.

The autoencoder models are trained on the training data representing the normal case. The validation data is used during training to check the generalization capabilities regarding other normal spectrograms. Figure 5.4 shows the progress of the convolutional

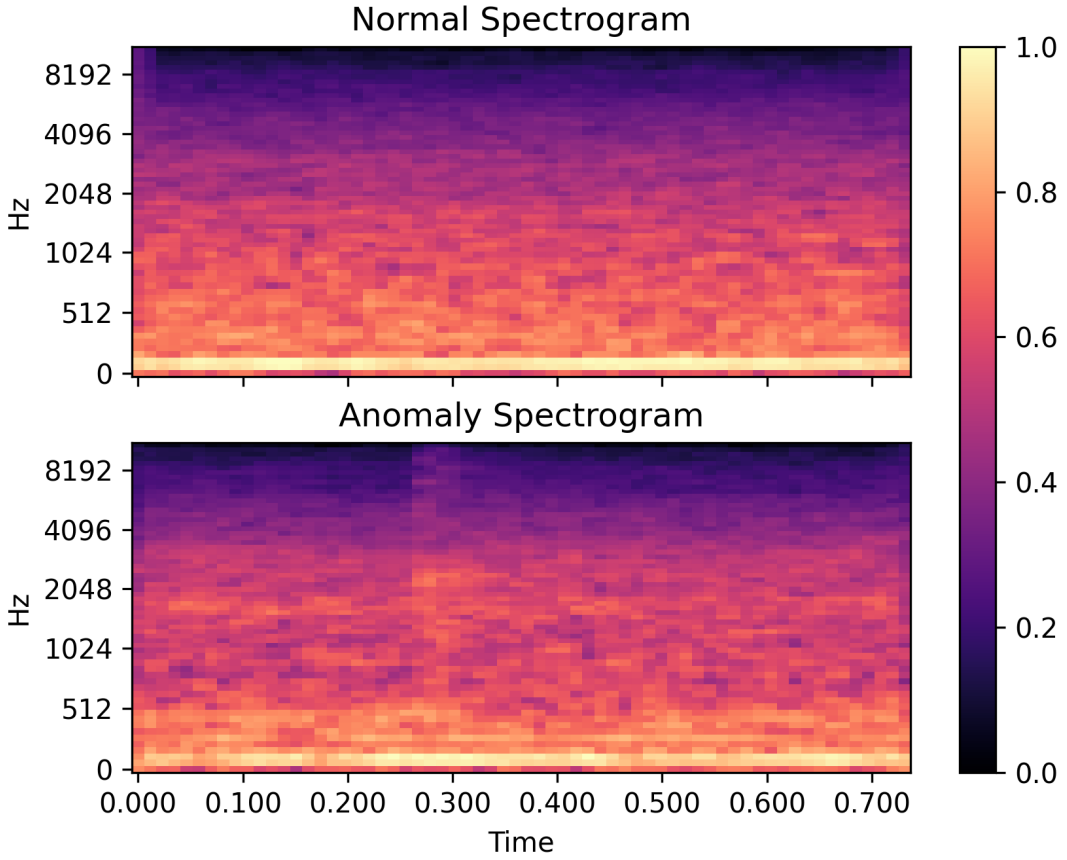

Figure 5.3: Comparison of normalized mel spectrograms with and without anomaly.

autoencoder training on hydroelectric data over 25 epochs with a batch size of 64. In blue, the reconstruction error between input and output is displayed for each epoch. The red line shows the difference between input and reconstructions for the validation data. The model shows no signs of overfitting during the training process, since both errors decline steadily with further epochs.

At inference time, a combination of the normal and anomaly test sets is used as input to the models. The reconstructed spectrograms were compared to the initial input by computing the reconstruction error as a mean squared error. Figure 5.5 visualizes the errors obtained by the convolutional autoencoder with hydropower data during inference in ascending order. The green dots represent a normal sample and the red dots contain an anomaly.

As expected, the model reconstructs the normal case better, resulting in low errors for these samples. However, one anomaly gets reconstructed well, leading to an overlap of the anomalous and normal samples regarding their reconstruction errors. This means that the model is not capable of perfectly discriminating between the two cases based on the obtained information.

Reconstruction errors between the best reconstructed anomaly and the worst reconstructed normal sample are candidates for a threshold. An ROC curve plots the false

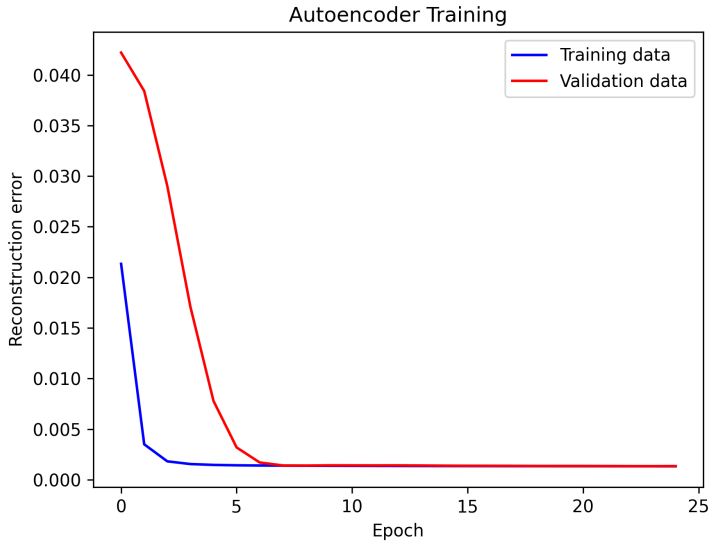

Figure 5.4: Reconstruction errors of the train and validation sets during the training of the convolutional autoencoder with hydropower audio.

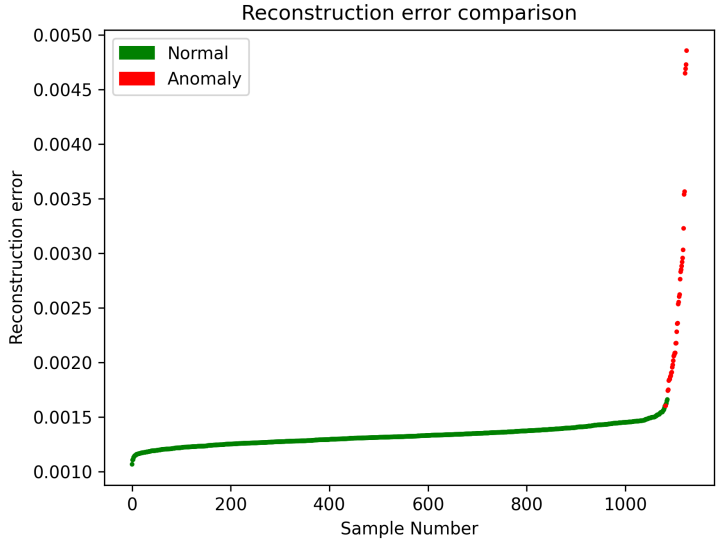

Figure 5.5: Comparison of reconstruction errors obtained from the convolutional autoencoder for normal and anomaly hydropower spectrograms.

positive rate on the x-axis against the true positive rate on the y-axis for varying threshold values [4]. Figure 5.6 shows the effect of the various threshold candidates on the positive rates in the example of the dense autoencoder with MIMII test data. A perfect classifier would generate a threshold candidate which correctly classifies all samples, which would be visible in the ROC plot as a true positive rate of one combined with a false positive rate of zero.

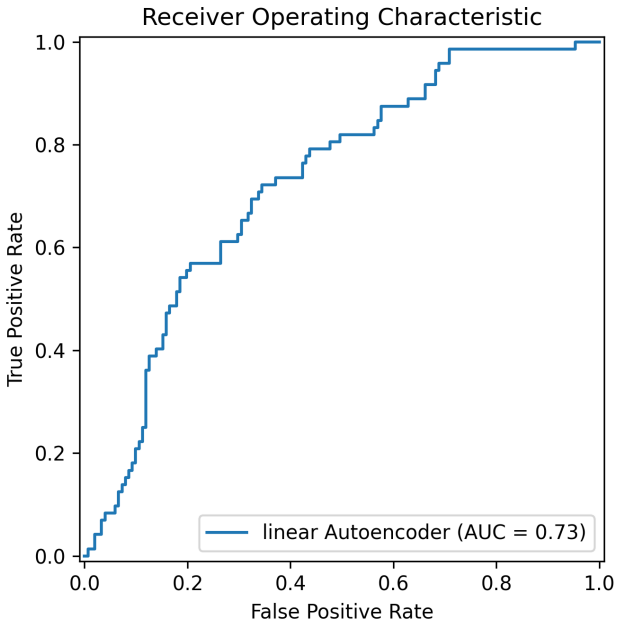

Figure 5.6: ROC curve for the dense autoencoder that reconstructed MIMII test samples

5.3 Self-Organizing Map

The input data represented as various features is not easy to visualize in a meaningful way due to its multidimensional nature. The comparison between feature vectors that are normal and ones containing an anomaly is therefore omitted here.

The SOM is trained with the normal training data of the hydropower dataset with the hyperparameters specified in Section 4.4. During an epoch, every training sample is fitted to the map exactly once in a randomized order. Randomization is introduced to prevent the model from learning patterns that exist only based on the input order. In the end, the map updates itself the number of training samples times the number of epochs: $training\_samples * epochs = 5'063 * 5 = 25'315$.

The SOM does not have the objective of minimizing the quantization error during training. That is why the training progress cannot be checked by looking at the quantization error for the training and validation data. To gain insight into what is happening during training, a hit map is created, showing how many times each neuron was the best matching unit for some training sample. In figure 5.7 it is observable that only few neurons make up the vast majority of the best matching units. This could be a sign, that the map is too big and the number of neurons could be reduced. In the following section, the hyperparameters are varied to observe their effect on the classification strength.

The trained SOM is ready for inference. The model is now used to compute the quantization errors for the normal and anomaly test samples. Figure 5.8 visualizes the quantization errors obtained for the normal samples in green and the samples containing an anomaly in red. In contrast to the autoencoder models, the errors for the normal and anomalous cases do not overlap. There is a considerable gap between the lowest error for an anomaly

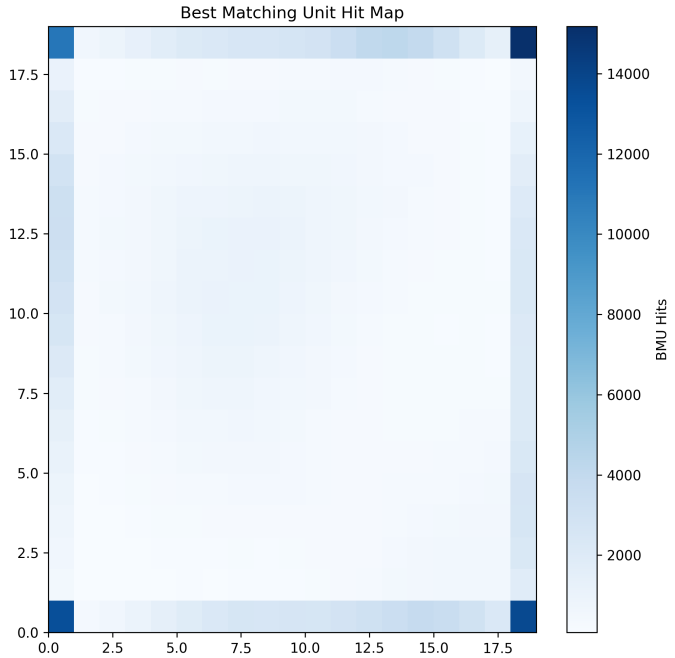

Figure 5.7: Hit Map that indicates how many times each neuron has been the best matching unit for a training sample using the hydropower dataset.

and the highest error for a normal sample. Any quantization error between these two samples can be used as a decision threshold and will lead the model to correctly classify all test samples.

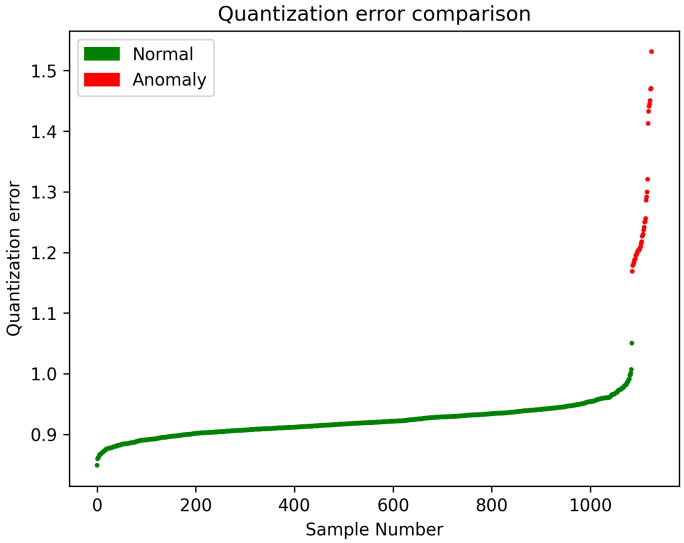

Figure 5.8: Comparison of quantization errors obtained at inference time from the Self-Organizing Map for hydropower data

5.4 Model Comparison

To assess the models' ability to distinguish between normal samples and anomalies, a quality score is computed using the area under the curve (AUC) of the receiver operating characteristic (ROC) plot. A perfect binary classifier achieves an AUC of 1, while a random classifier scores 0.5. Table 5.2 summarizes the AUC scores of the models evaluated for both data sets.

Anomalies are detected more effectively in the Hydropower dataset than in MIMII. One reason for this is the difference in sample duration. The Hydropower data set is labeled with fine granularity, whereas MIMII samples are labeled only once per ten-second segment, making it harder to distinguish anomalies. In addition, the MIMII dataset is more diverse and contains a wider range of anomalies, increasing the complexity of the classification task.

The best results on both datasets were achieved with the SOM, which perfectly classified all Hydropower samples and reached an AUC of 0.89 on MIMII. This is a surprising outcome, as the SOM is less commonly used in related work and its application to sound data has not been observed yet, in contrast to the widely adopted autoencoder.

The convolutional autoencoder was expected to outperform the dense autoencoder, given its ability to capture spatial dependencies in spectrograms. However, the obtained results do not support this expectation. Although the convolutional autoencoder slightly outperformed the dense model for the Hydropower dataset, the opposite was observed for MIMII. While this model comparison already gives insights, there is the possibility of a bias introduced through the choices of hyperparameters. This bias could also be the explanation for the convolutional autoencoder failing to meet the expectation. In order to reduce this potential distortion, hyperparameter tuning is performed in the following subsection.

Table 5.2: Comparison of obtained ROC-AUC for each model

	Hydropower	MIMII
Dense Autoencoder	0.9947	0.7264
Convolutional Autoencoder	0.9999	0.6915
Self-Organizing Map	1.0	0.8990

5.4.1 Hyperparameter Tuning

The selection of hyperparameters, both architectural and training related, was informed by previous studies in related work and recommendations in the documentation of the packages used. To further refine these choices, hyperparameter tuning is performed. This process provides insight into how different parameter combinations influence the models'

Table 5.3: Hyperparameter Overview

Parameter	Hydropower Base Model	MIMII Base Model	Parameter range
Dense Autoencoder			
First layer output size	64	64	[64-512]
Bottleneck size	8	8	[4-128]
Batch size	64	64	[8-128]
Epochs	10	75	[5-100]
Convolutional Autoencoder			
First layer filter number	16	16	[4-128]
Bottleneck size	64	64	[4-128]
Batch size	64	64	[8-128]
Epochs	25	75	[5-100]
Self-Organizing Map			
Side lengths	19	12	[8-20]
Sigma	24	15	[1-20]
Learning rate	0.5	0.5	[0.1-1.5]
Epochs	5	5	[2-10]

anomaly detection performance. As a next step, the goal is to compare the models in their best performing configurations.

The models trained in the previous sections are hereafter referred to as the base models. Table 5.3 summarizes the parameters used for the base models along with the parameter ranges explored during the tuning.

For the dense autoencoder, the output size of the first layer cannot take arbitrary values within the specified range, as it must be repeatedly divisible by two. To satisfy this constraint, the parameter is varied within the specified range, but is limited to powers of two. To further reduce the search space, both the bottleneck size and batch size are also restricted to powers of two. In addition, the number of epochs varies in increments of five. Despite these constraints, the resulting search space still consists of 2'400 possible parameter combinations.

For the convolutional autoencoder, all parameters except the number of epochs are restricted to powers of two. As with the dense autoencoder, the number of epochs is varied with a step size of five. This parameter configuration results in a total of 3'600 distinct combinations.

In the case of the SOM, the learning rate can take values within the specified range with a step size of 0.1, while all other parameters can assume any integer value within their

respective ranges. The search space consists of 35'100 combinations to be explored.

Exhaustively computing the entire search space is impractical due to its large size and the substantial computational resources required, particularly for training convolutional autoencoder configurations with a high number of filters and epochs. Instead, a random search is expected to yield sufficiently good results. Each model is trained with randomly selected hyperparameters over 100 iterations for each dataset.

To rank the models parameter combinations, the ROC-AUC score described in the last section is used again as a quality measure. This time it is computed on the validation samples instead of the test samples. As a second step, the model that yields the best validation score is picked and evaluated using the test set. This separation is conducted in order to prevent data leakage from the test set into the model selection, which can lead to overly optimistic performance estimations.

Table 5.4: Comparison of tuned models with base models

	AUC Base Model	AUC Tuned Model	Score Increase
Hydropower			
Dense Autoencoder	0.9947	0.9999	0.0052
Convolutional Autoencoder	0.9999	1.0	0.0001
Self-Organizing Map	1.0	1.0	0
MIMII			
Dense Autoencoder	0.7264	0.8181	0.0917
Convolutional Autoencoder	0.6915	0.8045	0.1130
Self-Organizing Map	0.8915	0.9671	0.0756

Table 5.4 presents the AUC scores in the test sets for the best model in each category, together with the absolute score improvement compared to the corresponding base model. The results indicate that parameter tuning has a significant impact when there is room for improvement, namely in the classification of MIMII samples. Even small increases in AUC can substantially enhance the real-world applicability of a model, highlighting the importance of tuning.

For the Hydropower dataset, the tuned convolutional autoencoder and the Self-Organizing Map (SOM) achieve perfect binary classification on the test samples. The dense autoencoder performs nearly as well, but struggles with one anomaly, reconstructing it too accurately. As a result, its reconstructed anomalies overlap with normal samples, reducing classification effectiveness.

In contrast, the AUC scores for the MIMII dataset show greater variation. Once again, the SOM achieves the highest performance, with an AUC exceeding 0.96, demonstrating

its potential to detect anomalies even in more challenging scenarios. The expectation that the convolutional autoencoder would outperform the dense autoencoder was initially attributed to a suboptimal configuration. However, since this observation persists even after hyperparameter tuning, the finding is considered to be robust.

5.4.2 Runtime Analysis

For an anomaly detection model to be useful as an early warning system for machine defects, it needs to be able to perform its detection in real-time. This subsection performs a runtime analysis of the implemented models. The goal is to assess how the models compare regarding their runtime and whether anomaly detection with the proposed preprocessing steps and models is feasible in real-time.

Training the models is resource-intensive and time-consuming. However, since the training is conducted only once, it is not relevant for the analysis of real-time feasibility. Only data pre-processing and inference steps are performed repeatedly for each time unit. The runtime analysis is therefore performed only on these steps.

The runtime of a piece of code depends on multiple factors, including the device where the code was executed, as well as other tasks that the machine was performing in parallel. This analysis is performed on a MacBook Pro with an Apple M4 Pro chip, 14 cores, and 24 GB of Random-Access Memory (RAM). To obtain a somewhat more reliable time measurement, the execution is repeated several times. Then the minimal execution time is used, as it is expected to be the most accurate, since it corresponds to the execution with the least number of other tasks running in the background.

For both datasets, 150 samples are repeatedly preprocessed in 20 rounds. Table 5.5 gives an overview of the runtimes obtained for the pre-processing of the spectrograms and features. The samples of the MIMII dataset take an order of magnitude longer to be preprocessed, which is due to their longer duration per sample. The extraction and pre-processing of features take more time than for the spectrograms. In general, the preprocessing times are neglectable compared to the combined duration of the samples which is around 110 seconds for the 150 Hydropower samples and 1'500 seconds for the same number of MIMII samples.

Table 5.5: Execution time of the preprocessing steps

	Spectrograms	Features
Hydropower	0.1038s	0.3718s
MIMII	2.0346s	4.1575s

To measure the inference time, the same procedure is used again. For 20 repetitions, for each category, the best model obtained after hyperparameter tuning is used to classify 150 samples. The minimal runtime is considered to be the best estimate regarding the executing time. Table 5.6 displays the obtained runtimes. The dense autoencoder is a

magnitude faster at classification when compared to its convolutional counterpart, which is explainable by the increased computational effort required by the convolutional layers. The inference time of the SOM is comparable to that of the dense autoencoder. All the obtained execution times are low, imposing no performance bottleneck for the applicability to real-time anomaly detection.

Table 5.6: Runtimes for the classification of 150 samples for each model.

	Dense Autoencoder	Convolutional Autoencoder	Self-Organizing Map
Hydropower	0.0400s	0.4526s	0.0348s
MIMII	0.1178s	4.0070s	0.23329s

5.5 Discussion

The goal of this Bachelor Thesis is the exploration of sound-based anomaly detection techniques and built on that, implementing approaches in the context of a hydropower plant. With the dense autoencoder, the convolutional autoencoder and the Self-Organizing Map, three models were designed and implemented. An important part is the preprocessing, whose importance needs to be highlighted. There are various decisions to be taken that can influence how well a model is able to detect anomalies, such as the frame length or the amount of mel bands, determining the frequency resolution in the case of a mel spectrogram.

The anomaly detection accuracy improved significantly following hyperparameter tuning. Although the relative ranking of the models remained unchanged, the absolute detection performance increased noticeably. This underscores the importance of fine-tuning in achieving reliable and effective anomaly detection results.

The evaluation of the SOM showed a high accuracy in detecting anomalies, compared to the autoencoders. One possible explanation to this is that the features used as input to the SOM possess more useful information compared to the spectrograms, which were the input for the other models. The limited number of training samples is expected to give the SOM a relative advantage due to its competitive learning strategy. It is therefore not clear, whether the results would be the same for larger datasets. Another possibility is that the SOM as a model is specifically well suited for sound-based anomaly detection tasks. Generally, it is recommended to take the SOM into account when building a sound-based anomaly detection system.

The convolutional autoencoder has significantly higher computational costs during training. This additional effort was not rewarded in terms of classification results. One possible explanation is the relatively small size of the training dataset, which may have been insufficient to fully exploit the capacity of the more complex convolutional architecture. In the case of the Hydropower dataset with over 5'000 training samples, the dense model

was outperformed by the convolutional autoencoder. For the MIMII data set with only 700 samples for training, the convolutional model demonstrated poorer anomaly detection results.

Another contributing factor could be the use of strided convolutions to reduce the dimensionality of the spectrograms. This approach may have led to an overly aggressive loss of detail, potentially discarding subtle features essential for distinguishing anomalies.

For models to classify sound in practice, a suitable decision threshold needs to be defined. Simply balancing precision and recall as in the F1 score is a possibility. Better would be to come up with quantitative estimates of what the consequences of a missed anomaly are compared to the impacts of a false alarm. Based on that, a threshold can be computed that minimizes potential costs for the test data. If the test data represent the real sound during operation well and the cost estimates are somewhat accurate, an accurate anomaly detection system has a great potential to ensure a secure and cost-effective operation of a hydroelectric power plant.

Chapter 6

Final Considerations

This chapter summarizes the parts of this thesis and describes what other authors can focus their research in future work.

6.1 Summary

This thesis investigates the potential of unsupervised anomaly detection techniques for analysing sound signals in hydropower plants. Therefore the baseline approach from [21] leveraging a dense autoencoder is implemented and compared with other approaches.

Section 2.2 reviews relevant research on unsupervised anomaly detection methods. Based on this review, two additional models were selected: the convolutional autoencoder, and the Self-Organizing Map (SOM).

Sound data was recorded in the turbine building of the PSHPP Rodundwerk II in Vorarlberg, Austria. Since the models operate in an unsupervised manner, they are trained exclusively on normal operating sounds. However, to evaluate their anomaly detection capabilities, anomalous sound samples were also required. To obtain these, different anomalies were induced and recorded.

Chapter 3 introduces the preprocessing pipelines for each model. The autoencoders use normalized mel spectrograms as input, while the SOM operates on a combination of extracted sound features. The dense autoencoder is designed based on the work of Huber & Krummenacher [21]. The designs of the convolutional autoencoder and the SOM are introduced as well.

Chapter 4 describes the implementation of the preprocessing pipelines and models with Python code. The training and evaluation of the models are then presented in Chapter 5. To facilitate an unbiased comparison, a random search was performed to optimize hyperparameters. Among the evaluated models, the SOM achieved the best detection results, whereas the convolutional autoencoder fell short of expectations. Despite differences in classification performance, all models exhibited low execution times for preprocessing and inference, making them suitable for real-time classification.

6.2 Conclusions

This thesis set out to explore the potential of state of the art anomaly detection models for identifying acoustic anomalies in hydropower environments. The evaluated models, especially the Self-Organizing Map, show a reasonable degree of accuracy, demonstrating their potential in early detection of machine faults, potentially preventing failures from occurring.

Another goal was to explore the effects of hyperparameter tuning. Given the infeasibility of exhaustive search in the high-dimensional parameter spaces, a random search was performed. The results confirm that even basic tuning significantly improves detection accuracy, highlighting the importance of this step in developing effective anomaly detection systems.

Overall, this thesis contributes to the ongoing exploration of unsupervised learning techniques in industrial settings and provides a foundation for the anomaly detection system that will be deployed in the PSHPP Rodundwerk II.

6.3 Future Work

One hypothesis emerging from the evaluation is that the SOM's strong performance, along with the convolutional autoencoder's underperformance, may be attributed to the limited number of available samples. It would be valuable to investigate, how more complex deep learning models benefit from larger training datasets and whether their performance surpasses that of simpler approaches under such conditions.

Hyperparameter tuning resulted in a significant improvement in anomaly detection performance for the implemented models. For future research, it is recommended to perform thorough hyperparameter optimization prior to model evaluation, in order to ensure that comparisons are made between well-configured versions of each model.

Additionally, research into the costs associated with missed anomalies and false alarms in hydropower plants would be highly beneficial. Establishing a cost matrix based on these findings could enable the optimization of decision thresholds for classification models, improving their practical applicability.

Bibliography

[1] Edmund Fosu Agyemang. "Anomaly detection using unsupervised machine learning algorithms: A simulation study". In: *Scientific African* 26 (2024), e02386.

[2] *Aktueller Marktpreis gemäß § 41 Abs. 1 Ökostromgesetz 2012*. URL: https://www.e-control.at/industrie/oeko-energie/oekostrommarkt/marktpreise-gem-paragraph-20#:~:text=Dieser%20Marktpreis%20betr%C3%A4gt%20demzufolge%2097,Quartal%202024)..

[3] Alexander von Birgelen et al. "Self-Organizing Maps for Anomaly Localization and Predictive Maintenance in Cyber-Physical Production Systems". In: *Procedia CIRP* 72 (2018), pp. 480–485.

[4] Andrew P. Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern Recognition* 30.7 (1997), pp. 1145–1159.

[5] Barış Bayram, Taha Berkay Duman, and Gökhan Ince. "Real time detection of acoustic anomalies in industrial processes using sequential autoencoders". In: *Expert Systems* 38.1 (2021).

[6] Alessandro Betti et al. "Condition monitoring and predictive maintenance methodologies for hydropower plants equipment". In: *Renewable Energy* 171 (2021), pp. 246–253.

[7] Breeze Paul. "Hydropower". In: *Hydropower*. 2018, pp. i–iii.

[8] Sanjay Chakraborty et al. "Detection and Classification of Novel Attacks and Anomaly in IoT Network using Rule based Deep Learning Model". In: *SN Computer Science* 5.8 (2024), p. 1056.

[9] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Unsupervised and Semi-supervised learning*. Adaptive computation and machine learning. 2006.

[10] Chuxu Zhang et al. *A Deep Neural Network for Unsupervised Anomaly Detection and Diagnosis in Multivariate Time Series Data*. 2018.

[11] Giuseppe Ciaburro, V. Kishore Ayyadevara, and Alexis Perrier. *Hands-On Machine Learning on Google Cloud Platform: Implementing smart and efficient analytics using Cloud ML Engine*. 2018.

[12] *Cluster Data with a Self-Organizing Map*. 2024. URL: https://ch.mathworks.com/help/deeplearning/gs/cluster-data-with-a-self-organizing-map.html.

[13] Anthony Deschênes et al. *Planing It by Ear: Convolutional Neural Networks for Acoustic Anomaly Detection in Industrial Wood Planers*. URL: http://arxiv.org/pdf/2501.04819v1.

[14] Emanuele Di Fiore et al. "An anomalous sound detection methodology for predictive maintenance". In: *Expert Systems with Applications* 209 (2022), p. 118324.

[15] Monika Dörfler, Roswitha Bammer, and Thomas Grill. "Inside the spectrogram: Convolutional Neural Networks in audio processing". In: *2017 International Conference on Sampling Theory and Applications (SampTA)*. 2017, pp. 152–155.

[16] Taha Berkay Duman, Barış Bayram, and Gökhan İnce. "Acoustic Anomaly Detection Using Convolutional Autoencoders in Industrial Processes". In: pp. 432–442.

[17] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd. 2019.

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. 2016.

[19] Bingjun Guo et al. *A Comparative Evaluation of SOM-based Anomaly Detection Methods for Multivariate Data*. 2019.

[20] Ricardo Hormann and Eric Fischer. "Detecting Anomalies by using Self-Organizing Maps in Industrial Environments". In: 2019, pp. 336–344.

[21] Valentin Huber and Stefan Krummenacher. *Acoustic-based Anomaly Detection for Remote Pumped Storage Hydroelectric Power Plants*. 2025.

[22] Muhammad Ayaz Hussain et al. "Comparison of Anomaly Detection between Statistical Method and Undercomplete Autoencoder". In: pp. 32–38.

[23] International Hydropower Association. "2024 World Hydropower Outlook". In: *International Hydropower Association* 2024 ().

[24] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015).

[25] K. Choi et al. "A Comparison of Audio Signal Preprocessing Methods for Deep Neural Networks on Music Tagging". In: *2018 26th European Signal Processing Conference (EUSIPCO)*. 2018, pp. 1870–1874.

[26] Jonathan Elliott Katz et al. "The Basic Physics of Waves, Soundwaves, and Shockwaves for Erectile Dysfunction". In: *Sexual medicine reviews* 8.1 (2020), pp. 100–105.

[27] Ken'iti Kido. *Digital Fourier Analysis: Fundamentals*. 2014.

[28] T. Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[29] Ning Li et al. *Anomaly Detection via Self-organizing Map*. 2021.

[30] *Librosa: Python library for audio and music analysis*. 2025. URL: `https://github.com/librosa/librosa`.

[31] Wenting Ma et al. "Anomaly Detection of Mountain Photovoltaic Power Plant Based on Spectral Clustering". In: *IEEE Journal of Photovoltaics* 13.4 (2023), pp. 621–631.

[32] A. Mardin. "Neural Network Learning and Expert Systems". In: *AI Commun* 7.3–4 (1994), pp. 238–240.

[33] *Matplotlib: plotting with Python*. 2024. URL: `https://github.com/matplotlib/matplotlib`.

[34] Mattia Fanan et al. "Anomaly Detection for Hydroelectric Power Plants: a Machine Learning-based Approach". In: *2024 4th International Conference on Energy Engineering and Power Systems (EEPS)* (2023).

[35] Kishan G. Mehrotra, Chilukuri K. Mohan, and HuaMing Huang. *Anomaly Detection Principles and Algorithms*. 2017.

[36] U. Michelucci. *Advanced Applied Deep Learning: Convolutional Neural Networks and Object Detection*. For professionals by professionals. 2019.

[37] Amirhossein Moshrefi et al. "Industrial Fault Detection Employing Meta Ensemble Model Based on Contact Sensor Ultrasonic Signal". In: *Sensors (Basel, Switzerland)* 24.7 (2024).

[38] *Numpy: The fundamental package for scientific computing with Python*. 2024. URL: `https://github.com/numpy/numpy`.

[39] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[40] Cheng-Yu Peng et al. "Sound Detection Monitoring Tool in CNC Milling Sounds by K-Means Clustering Algorithm". In: *Sensors (Basel, Switzerland)* 21.13 (2021).

[41] John G. Proakis and Dimitris G. Manolakis. *Digital signal processing (3rd ed.): principles, algorithms, and applications*. 1996.

[42] Harsh Purohit et al. "MIMII Dataset: Sound Dataset for Malfunctioning Industrial Machine Investigation and Inspection". In: *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2019 Workshop (DCASE2019)*. 2019, pp. 209–213.

[43] Lawrence R. Rabiner and Ronald W. Schafer. *An Introduction to Digital Speech Processing*. 2007.

[44] Springer Series in Information Sciences, ed. *Self-Organizing Maps*. 2001.

[45] Elias M. Stein and Rami Shakarchi. *Fourier analysis: An introduction*. Vol. 1. Princeton lectures in analysis. 2003.

[46] Jérôme Sueur. *Sound Analysis and Synthesis with R*. 1st ed. 2018. Use R! 2018.

[47] J. Tabak. *Geometry: The Language of Space and Form*. Facts on File math library. 2014.

[48] *Tensorflow Keras: The high-level API for TensorFlow*. 2024. URL: `https://www.tensorflow.org/guide/keras`.

[49] R. Venkatesan and B. Li. *Convolutional Neural Networks in Visual Computing: A Concise Guide*. Data-enabled engineering. 2018.

[50] Nishchal K. Verma et al. "Intelligent condition based monitoring of rotating machines using sparse auto-encoders". In: *2013 IEEE Conference on Prognostics and Health Management (PHM)*. 2013, pp. 1–7.

[51] Giuseppe Vettigli. *MiniSOM: minimalistic implementation of the Self Organizing Maps*. 2025. URL: `https://github.com/JustGlowing/minisom/`.

[52] Marc Wegmann et al. "A review of systematic selection of clustering algorithms and their evaluation". In: (2021).

[53] Zhiyan Feng et al. "Power Plant Production Equipment Sound Recognition Method Combined with Attention Mechanism". In: *2022 IEEE 4th International Conference on Power, Intelligent Computing and Systems (ICPICS)* (2022).

List of Figures

List of Tables

Listings