Chair of Automation

Master's Thesis

Optimization of Autoencoders for Anomaly Detection in Multivariate Real-Time Measurement Data Acquired From Production Machinery

Gernot Steiner, BSc

September 2022

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 11.09.2022

Gernot Steiner

Unterschrift Verfasser/in
Gernot Steiner

Dedication

First of all, I want to thank Prof. Paul O'Leary and Dipl.-Ing. Anika Terbuch from the Chair of Automation for their tremendous support during the elaboration of this thesis and their advice beyond the scope of this work, be it in scientific or personal matters. In a lot of friendly discussions, they introduced me to the exciting field of machine learning and were always willing to help me during the challenging times of the COVID-19 crisis.

Special thanks go to my fellow students and friends in Leoben, with whom I had a great time and who always kept me motivated.

Last but not least, I would like to thank my family and friends, especially my parents, who have always encouraged me to follow my passion and supported me where they could.

Kurzfassung

Die vorliegende Arbeit widmet sich der Optimierung von Autoencodern, welche zur Klassifizierung von multivariaten Echtzeit-Messdaten aus Produktionsprozessen eingesetzt werden sollen. Ziel ist die Erkennung von Zeitreihen mit Anomalien, die auf mögliche Prozessfehler hindeuten könnten. Die Güte der Klassifizierung wird von unterschiedlichen Einflussfaktoren bestimmt, von denen die wichtigsten in separaten Testreihen analysiert wurden. Mittels eines existierenden Frameworks wurden Autoencoder mit uni- oder bidirektionalen Schichten des Typs Long Short-Term Memory (LSTM) erzeugt, um Langzeit-Abhängigkeiten in den Daten zu berücksichtigen. Die Abweichung des vom Autoencoder rekonstruierten Signals vom entsprechenden Eingangssignal wurde als Maß für den Grad der Abnormalität einer Zeitreihe verwendet. Mittels eines Fehlerschwellwertes wurden die Proben als normal oder abnormal klassifiziert. Bei der Schwellwertbestimmung wurde die Schiefe der Verteilung der Rekonstruktionsfehler verschiedener Zeitreihen berücksichtigt. In einer Testreihe wurden Autoencoder mit unterschiedlichen Architekturen hinsichtlich ihrer Eignung zur Anomalieerkennung verglichen. In weiteren Experimenten wurden Techniken zur Initialisierung der Gewichtsparameter von neuronalen Netzwerken behandelt. Darüber hinaus wurden unterschiedliche Methoden zur Optimierung der maßgebenden Hyperparameter von Autoencodern untersucht. Unter Berücksichtigung der erlangten Erkenntnisse aus den zuvor genannten Versuchsreihen wurden separate Autoencoder für die zwei Hauptphasen des überwachten Prozesses optimiert und getestet. Diese wurden mit einem statistischen Tool zur Anomalieerkennung basierend auf Leistungskennzahlen kombiniert, um hybride Lernmodelle zu erzeugen. Der verwendete Datensatz wurde von Sensoren an Baumaschinen aufgezeichnet, welche zur Baugrundverbesserung mittels Rütteldruckverdichtung eingesetzt werden. Ziel dieser Arbeit war die Unterstützung laufender Forschung am *Lehrstuhl für Automation*, welche sich mit Methoden des maschinellen Lernens in der kombinierten Anwendung mit klassischen Methoden zur Maschinendatenanalyse befasste.

Abstract

This thesis investigates the optimization of autoencoders for the classification of multivariate real-time measurement data emanating from production machinery. The objective is to detect anomalous time-series samples that may indicate process failures. The classification performance depends on a variety of factors, the most important of which were analyzed in separate series of experiments. The autoencoders were set up using an existing framework that allows the inclusion of uni- or bidirectional Long Short-Term Memory (LSTM) layers in order to track long-term dependencies in the data. The error between the original signal and the corresponding reconstruction obtained by the autoencoder was used as a measure of the degree a sample is believed to be anomalous. Via an error threshold, the data samples were classified as either anomalous or non-anomalous. Since the distribution over the reconstruction errors of different samples was right-skewed, a skewness-adjusted threshold setting was performed. In a series of tests, autoencoders with different architectures were compared with regard to their suitability for anomaly detection. Further experiments involved the use of several techniques for initializing the weight parameters. In addition, various methods for optimizing the most impactful hyperparameters were evaluated. Considering the results of the experiments mentioned above, separate autoencoders for the two main phases of the monitored process were optimized and tested. These models were combined with a statistical outlier detection tool based on key performance indicators in order to form hybrid learning models. The data set analyzed in the experiments was gathered from instrumented machinery used in a ground improvement process for building foundations. The aim of this thesis was to support ongoing research at the *Chair of Automation* involving the use of machine learning techniques in combination with classical methods for machine data analysis.

List of Acronyms

Adam	Adaptive moment estimation
AE	Undercomplete autoencoder
ANN	Artificial neural network
AR model	Autoregressive model
ARIMA model	Autoregressive integrated moving-average model
ARMA model	Autoregressive moving-average model
AUC	Area under the ROC Curve
biLSTM	Bi-directional long short-term memory
BPTT	Backpropagation through time
CNN	Convolutional neural network
ELBO	Evidence lower bound
GA	Genetic algorithm
HPO	Hyperparameter optimization
IQR	Interquartile range
KL divergence	Kullback-Leibler divergence
k-NN	k-nearest neighbors
KPI	Key performance indicator
LSTM	Long short-term memory
MA model	Moving-average model
MAE	Mean absolute error
MAPE	Mean absolute percentage error
MB	Mini-batch
MC	Medcouple
MCC	Matthew's correlation coefficient
MSE	Mean squared error
MVTS	Multivariate time-series
PCA	Principal component analysis
ReLU	Rectified linear unit
RMSE	Root mean squared error
RNN	Recurrent neural network
ROC	Receiver operating characteristic
seq2seq	Sequence-to-sequence
SGD	Stochastic gradient descent
SMAPE	Symmetric mean absolute percentage error
SSE	Sum-of-squares error
SVM	Support vector machine
VAE	Variational autoencoder

Contents

1 Introduction **1**

2 Basics of Machine Learning **3**

2.1 Machine Learning Tasks 3

2.2 Gaining Experience 4

2.2.1 Supervised Learning 4

2.2.2 Unsupervised Learning 6

2.3 Performance Metrics 7

2.3.1 Metrics for Regression Tasks 7

2.3.2 Metrics for Classification Tasks 8

3 Artificial Neural Networks (ANN) **12**

3.1 Model of a Neuron 12

3.2 Network Architectures 13

3.2.1 Feedforward Neural Networks 14

3.2.2 Recurrent Neural Networks 16

3.3 Training 21

3.3.1 Gradient Descent 22

3.3.2 Backpropagation 23

3.3.3 Weight Initialization 26

3.4 Hyperparameter Optimization 28

3.4.1 Grid Search 29

3.4.2 Random Search 29

3.4.3 Bayesian Optimization 30

3.4.4 Genetic Algorithm 31

4 Basics of Probability and Information Theory **33**

4.1 Probability Theory 33

4.1.1 Probability Distributions 33

4.1.2 Marginal Probability 36

4.1.3 Conditional Probability and Chain Rule . . . 37
4.1.4 Bayes' Theorem . . . 37
4.2 Information Theory . . . 38

5 Autoencoders 40
5.1 Types of Autoencoders . . . 40
5.1.1 Undercomplete Autoencoders . . . 40
5.1.2 Sparse Autoencoders . . . 41
5.1.3 Denoising Autoencoders . . . 43
5.1.4 Variational Autoencoders . . . 43
5.2 Encoder-Decoder Sequence-To-Sequence Architectures . . . 47

6 Anomaly Detection in Time-Series Data 48
6.1 Time-Series . . . 48
6.2 Anomalies . . . 48
6.3 Methods for Anomaly Detection in Time-Series . . . 49

7 Hybrid Learning Tool for Anomaly Detection 53
7.1 Structure of the Hybrid Learning Tool . . . 53
7.2 Analyzed Process . . . 54
7.3 Machine Learning Model . . . 57
7.3.1 Autoencoder Framework . . . 57
7.3.2 Reconstruction Error and Threshold Setting . . . 60
7.4 Statistical Model . . . 64

8 Test Results 65
8.1 Evaluation of Different Architectures . . . 65
8.1.1 Architectures With One Hidden Layer . . . 66
8.1.2 Architectures With Two Hidden Layers . . . 69
8.2 Evaluation of Hyperparameter Optimization Methods . . . 73
8.3 Evaluation of Weight Initializing Methods . . . 77
8.4 Optimizing Models for Phase-Wise Anomaly Detection . . . 81
8.4.1 Parallel Hybrid Model . . . 82
8.4.2 Parallel Serial Hybrid Model . . . 83

9 Conclusion and Future Work 87

List of Figures 89

List of Tables 92

Bibliography 93

Chapter 1
Introduction

Modern sensor technology laid the foundation for monitoring and analyzing processes in real-time, inspiring developments in the fields of quality control [1], predictive maintenance [2], optimization of process efficiency [3] or reduction of energy consumption and sustainability issues [4]. Time-series data can be collected by instrumenting a production facility or machinery with sensors that measure a physical value in certain predefined time intervals [5]. This allows gathering useful information about the underlying process, including the presence of extreme values or unusual behavior. The corresponding procedure of detecting abnormal patterns in the data is referred to as *anomaly detection* [6]. It can help to identify products that may not meet the requirements, e.g. in terms of quality, or detect a failure of the machinery [4]. Manual anomaly detection is error-prone and cost-intensive and may not be feasible in an acceptable amount of time, depending on the amount of data to be analyzed [6], [7]. This inspired research on methods that perform this task automatically [6]. In the case of time-series data, these methods have to take the context in which a data point appears into account, e.g., long- or short-term trends [8].
A multivariate time-series includes data of multiple variables stored in different channels. The data points of different channels typically originate from the same process and therefore are correlated in some form. An anomaly detection tool for multivariate time-series data should be able to account for this correlation [9].
In this thesis, an approach for anomaly detection using a type of neural network called *autoencoder* [10] is presented. An autoencoder learns how to compute a hidden representation of the input data, on which it can be well reconstructed [11]. If this hidden encoding has a lower dimension than the input signal, the autoencoder is forced to capture the most salient features of the data [11]. By adding LSTM [12] and biLSTM [13] layers, the model is able to track long-term dependencies in the time-series. The performance of autoencoders in anomaly detection tasks highly depends on their architecture and the training hyperparameters [14], [15]. In this thesis, several series of experiments to optimize the machine learning models are presented. Section 8.1 shows the results of experiments aimed at finding the most suitable autoencoder architecture. Different hyperparameter optimization methods were tested and compared with each other, as shown in Section 8.2. In a test series presented in Section 8.3, the impact of weight initialization was investigated, and different

common methods were applied.

The data set of multivariate time-series samples used for the presented experiments was collected during a vibro ground improvement process for building foundations. The purpose of this process is the stabilization of cohesionless soils by creating subsurface columns of compacted gravel or sand [16]. Since columns with deficiencies are a potential safety hazard, the process is monitored, and abnormal behavior has to be detected [17]. Considering the results of the test series performed in Section 8.1-8.3, separate autoencoders for the two main phases of the analyzed process were optimized. These models were combined with a statistical tool for anomaly detection based on *key performance indicators (KPI)* [18] to form a hybrid learning model. Two different approaches, a parallel hybrid model and a parallel serial hybrid model, were tested.

This thesis was intended to support ongoing research [17] at the *Chair of Automation* investigating how machine learning techniques and classical methods for machine data analysis can be combined to benefit from the strengths of both.

Chapter 2
Basics of Machine Learning

Machine Learning has gained significant popularity in recent decades and nowadays is one of the core areas of information technology. The fields of application range from speech recognition, web page ranking in search engines, and autonomous driving to classification tasks in automated industrial processes [19], [20]. A possible definition of what machine learning is about can be found in a work of Tom Mitchell from 1997:

> A computer program is said to *learn* from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. [21, p.2]

Take, for example, the task T of classifying images of handwritten digits: A possible measure P for the performance of the program is simply the ratio of images that were classified correctly. Experience E can be gained by feeding the program with sets of images and corresponding known class labels. With an increasing amount of labeled images that are available to the program, it tends to improve its ability to correctly classify a random unlabeled image of a handwritten digit, i.e., the program *learns* [21].
Over the past decades, a variety of machine learning methods have been developed and become established in many areas of application in industry and personal life [20], [21]. This chapter is intended to give a brief overview of the most important methods and techniques one should know when dealing with this matter.

2.1 Machine Learning Tasks

For the decision of whether to prefer machine learning to directly coding a program, two main aspects of the problem at hand must be considered: the complexity and the required adaptability. Tasks that are performed by humans or animals routinely, e.g., driving or interpreting images, are often too complex to directly write a program that covers all aspects of the problem. Other tasks require the program to adapt to varying input data of the same underlying problem, such as e.g. recognition of handwritings of different persons. These are typical fields where it is beneficial to

let a program learn how to fulfill a task by itself [20].
The following listing shows some typical areas in which machine learning has become established:

- **Classification:** Produce a function $f : \mathbb{R}^n \to \{c_1, ..., c_k\}$ that assigns the input $\boldsymbol{x} \in \mathbb{R}^n$ to one class of a given set of k possible classes $\mathcal{C} = \{c_1, ..., c_k\}$ or returns a probability distribution over all of them. Possible task: image recognition [22].
- **Regression:** Determine a function $f : \mathbb{R}^n \to \mathbb{R}$ that predicts a numerical value based on provided input data $\boldsymbol{x} \in \mathbb{R}^n$. Possible task: predicting the future prices of securities [22].
- **Machine translation:** Convert a given sequence of words of some language into a grammatically correct sequence of words in another language [22].
- **Anomaly detection:** Identify patterns in provided input data that indicate abnormal behavior [22]. Possible tasks: outlier detection in records of credit card usage [22] or industrial data [17].
- **Synthesis and sampling:** Generate data that is similar to the input samples. Possible task: automated texture generation for landscapes in video games [22].
- **Imputation:** Predict the values of missing entries in a given data set [22].
- **Denoising:** Remove the noise of corrupted input data [22].

2.2 Gaining Experience

According to the way they gain experience E in training, machine learning methods can be roughly divided into two groups: *supervised* and *unsupervised* learning methods. Some algorithms can not be categorized into either of these groups, e.g., *reinforcement learning* algorithms [22].
Reinforcement learning involves an agent that learns to perform an optimal sequence of actions to reach a specific goal. The agent receives feedback for his actions from a trainer in the form of *rewards*, which indicate the desirability of the currently achieved change in the environment [22], [21]. Since this type of learning is not addressed in this thesis, it will not be discussed in detail.

2.2.1 Supervised Learning

Algorithms that learn in a supervised training process receive a set $\mathcal{Y} = \{\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_n\}$ of n target vectors, values, or labels, that are associated with the feature vectors of the input data set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n\}$. Thus, the user (*supervisor*) provides the desired output $\boldsymbol{y}_i$ for each training sample $\boldsymbol{x}_i$, where $i = \{1, \ldots, n\}$. With this data, the algorithm produces a function that maps an unseen data vector $\boldsymbol{x_i}$ to a predicted class, vector, or value $\hat{\boldsymbol{y}}_i$, aiming for the prediction to be equal to or near the target $\boldsymbol{y}_i$ [22].

Some common supervised machine learning methods are:

- **Support Vector Machines:** SVMs are used to compute a linear decision boundary in high dimensional feature spaces while maximizing the minimum distance between the data points of the separated classes and the boundary hyperplane, the so-called *margin*. The *support vectors* are the data points closest to the decision boundary, which therefore define the hyperplane, i.e., *support* it [19], [20].
- **k-Nearest Neighbors:** k-NN is a technique for assigning a label to a given input based on the labels of the k nearest (e.g. in terms of Euclidean distance) training samples in the feature space. It can be used for classification and regression [20].
- **Naive Bayes:** The Naive Bayes classifier receives an input vector of attribute values $\boldsymbol{x} = [x_1, x_2, ..., x_m]^{\mathrm{T}}$, and assigns the most probable label $\hat{y}$ that is part of a set $\mathcal{Y} = \{y_1, y_2, ..., y_k\}$ of k labels. Thus, the conditional probability $P(y_i|x_1, x_2, ..., x_m)$ of a label $y_i \in \mathcal{Y}$ given the attribute values in $\boldsymbol{x}$ has to be maximized. Using *Bayes' Theorem*, the following classifier can be obtained [20], [21]:

$$\begin{aligned}\hat{y} = \underset{y_i \in \mathcal{Y}}{\arg\max}\, P(y_i|x_1, x_2, ..., x_m) &= \underset{y_i \in \mathcal{Y}}{\arg\max}\, \frac{P(x_1, x_2, ..., x_m|y_i)P(y_i)}{P(x_1, x_2, ..., x_m)} \\ &= \underset{y_i \in \mathcal{Y}}{\arg\max}\, P(x_1, x_2, ..., x_m|y_i)P(y_i). \end{aligned} \tag{2.1}$$

  The evidence $P(x_1, x_2, ..., x_m)$ is a constant scaling factor and therefore can be left out. Providing a set of training data that covers all possible combinations of attribute values and labels to estimate the corresponding conditional probabilities $P(x_1, x_2, ..., x_m|y_i)$ is only feasible in the rarest cases. Therefore, Naive Bayes assumes the attribute values in $\boldsymbol{x}$ to be conditionally independent given a label y_i. Using the *chain rule* (see Section 4.1.3), the classifier in Equation (2.1) can be rewritten, as shown in Equation (2.2). The simplification mentioned above yields the Naive Bayes classifier defined in Equation (2.3) [21]:

$$\hat{y} = \underset{y_i \in \mathcal{Y}}{\arg\max}\, P(y_i)P(x_1|y_i) \prod_{j=2}^{m} P(x_j|y_i, x_1, ..., x_{j-1}) \tag{2.2}$$

$$= \underset{y_i \in \mathcal{Y}}{\arg\max}\, P(y_i) \prod_{j=1}^{m} P(x_j|y_i). \tag{2.3}$$

  For more information on conditional probabilities and Bayes' Theorem, refer to Chapter 4.
- **Neural Networks:** These machine learning models are inspired by neuroscience and consist of multiple units (*neurons*) that form a network via connections among each other. This enables the algorithm, analogous to the brain, to solve highly complex problems [22]. A detailed description of neural networks is given in Chapter 3.

- **Linear Regression:** The program predicts an output value $\hat{y}$ based on an input vector $\boldsymbol{x} = [x_1, x_2, ..., x_m]^{\mathrm{T}}$, where $\hat{y} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ is a linear function of the input. The vector of learnable parameters is denoted as $\boldsymbol{w}$ [22].

2.2.2 Unsupervised Learning

Unsupervised learning models are trained on an unlabeled data set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n\}$, i.e., they do not receive any targets vectors or labels $\boldsymbol{y}_i$ associated with the samples $\boldsymbol{x}_i$. Rather than predicting a target $\hat{\boldsymbol{y}}_i$, the algorithm aims to gather useful information about the structure of the given data [20], [22]. This information can be used, for example, to divide a set of data points into clusters according to their similarity [23]. Some common unsupervised machine learning techniques are:

- **k-Means Clustering:** This algorithm decomposes a given input data set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n\}$ into a set of k disjoint clusters $\mathcal{C} = \{c_1, c_2, ..., c_k\}$, with $1 \leq k \leq n$. The objective is to minimize the *sum-of-squares error (SSE)* between the objects and the mean vectors, or *centroids*, $\mathcal{M} = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k\}$ of the clusters in $\mathcal{C}$ they are assigned to [23]:

$$SSE(\mathcal{C}) = \sum_{j=1}^{k} \sum_{\boldsymbol{x}_i \in c_j} ||\boldsymbol{x}_i - \boldsymbol{\mu}_j||^2 \text{ with } \boldsymbol{\mu}_j = \frac{\sum_{\boldsymbol{x}_i \in c_j} \boldsymbol{x}_i}{|c_j|}. \tag{2.4}$$

  Equation 2.4 shows how for each cluster $c_j \in \mathcal{C}$, the distances between the centroid $\boldsymbol{\mu}_j$ of this cluster, and the elements $\boldsymbol{x}_i \in \mathcal{X}$ it consists of, are summed up. Note that $||\cdot||$ is the notation for the norm, which is typically the Euclidean norm. The number of elements in each cluster c_j is denoted as $|c_j|$. The performance of this method is highly dependent on the initial choice of cluster centroids [23].
- **Fuzzy Clustering:** This method is an extension of the standard k-means clustering algorithm. The objects are not assigned to one cluster exclusively but belong to different clusters to a certain degree expressed by probabilistic weights. Such a weight is denoted as $V_{ij} \in [0, 1]$ and indicates an increasing degree of membership of data vector $\boldsymbol{x}_i$ in cluster c_j the higher its value is. Given a set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n\}$ of n input vectors and a set $\mathcal{C} = \{c_1, c_2, ..., c_k\}$ of k clusters, the weights are organized in an $n \times k$ weight matrix V. The objective of fuzzy clustering is to minimize the sum of weighted squared errors between the data points in $\mathcal{X}$ and the centroids $\mathcal{M} = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k\}$, which can be formulated as [23]:

$$F(\mathcal{C}, \mathsf{V}) = \sum_{i=1}^{n} \sum_{j=1}^{k} V_{ij}^{p} ||\boldsymbol{x}_i - \boldsymbol{\mu}_j||^2 \text{ with } \boldsymbol{\mu}_j = \frac{\sum_{i=1}^{n} V_{ij}^{p} \boldsymbol{x}_i}{\sum_{i=1}^{n} V_{ij}^{p}}, \tag{2.5}$$

  where the exponent p defines the degree of fuzziness of the object memberships [23].

- **Autoencoders:** An autoencoder is a type of neural network that maps a given input $\boldsymbol{x}$ to a hidden representation $\boldsymbol{z}$ and computes a reconstruction $\hat{\boldsymbol{x}}$ of the original input from $\boldsymbol{z}$, such that the error between $\hat{\boldsymbol{x}}$ and $\boldsymbol{x}$ is minimized [24]. A detailed description of autoencoders is given in Chapter 5.

2.3 Performance Metrics

Machine learning models require a measure to evaluate their performance in the training process in order to adjust the learnable parameters. These measures are selected depending on the task to be accomplished [21]. Some of the most common metrics for regression and classification tasks are presented in the following.

2.3.1 Metrics for Regression Tasks

Regression tasks require performance metrics that quantify the error between the outputs $\hat{y}_i$ computed by the machine learning model and the corresponding targets y_i. If a set of n outputs $\hat{\mathcal{Y}} = \{\hat{y}_1, \hat{y}_2, ..., \hat{y}_n\}$ is obtained, a set of n targets $\mathcal{Y} = \{y_1, y_2, ..., y_n\}$ has to be provided [22].
Metrics for regression tasks can be divided into *scale-dependent* metrics and *percentage* metrics. Scale-dependent metrics provide values that are on the same scale and have the same units as $\hat{y}_i$ and y_i [25]. The most common metrics of this type are presented in the following:

- **Mean Squared Error (MSE)** [26]:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2. \tag{2.6}$$

- **Root Mean Squared Error (RMSE)** [26]:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}. \tag{2.7}$$

- **Mean Absolute Error (MAE)** [26]:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|. \tag{2.8}$$

Note that even though the mean squared error is not expressed in the same units as the data, it still is assigned to the group of scale-dependent metrics [25].
In contrast to scale-dependent metrics, percentage metrics are unit-free. Thus, they allow to com-

pare the errors of data sets with different units and are easier to interpret, especially if the user is not familiar with the units of the data samples [26]. Examples of this type of metrics are:

- **Mean Absolute Percentage Error (MAPE)** [25]:

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{\hat{y}_i - y_i}{y_i} \right|. \tag{2.9}$$

- **Symmetric Mean Absolute Percentage Error (SMAPE)** [25]:

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{\hat{y}_i - y_i}{(|\hat{y}_i| + |y_i|)/2} \right|. \tag{2.10}$$

2.3.2 Metrics for Classification Tasks

Classification involves the categorization of given input samples into classes according to the probability of membership [22]. In the following, some commonly applied metrics for binary classification tasks are presented since this type of classification is addressed in this work. For an overview of performance metrics for multi-class classifiers, refer to [27].
In binary classification, the objective is to allocate the n samples in data set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n\}$ to one of two given classes c_1 and c_2, which in the following will be referred to as class *positive* and class *negative*, respectively. By comparing the classifier's predictions with the corresponding known true classes of a data set, the following categorization can be made [28]:

- *True positive*: A data set of class *positive* is correctly assigned to the class *positive*.
- *True negative*: A data set of class *negative* is correctly assigned to the class *negative*.
- *False positive*: A data set of class *negative* is erroneously assigned to the class *positive*.
- *False negative*: A data set of class *positive* is erroneously assigned to the class *negative*.

In the context of this work, the class *negative* represents the class of non-anomalous data samples, and the class *positive* is the class of data samples that contain anomalies. The variables TP, TN, FP and FN denote the number of true positive, true negative, false positive, and false negative classifications, respectively. Combining these values in the form of a table yields a commonly used tool for interpreting the classifier's performance, the *confusion matrix* [28] (see Figure 2.1). Some typically used performance metrics for binary classification tasks are listed in the following:

- **Accuracy**: This is the most common metric for quantifying a classifier's performance. It represents the correctly classified portion of all given data samples [28]:

$$acc = \frac{TP + TN}{TP + TN + FP + FN}. \tag{2.11}$$

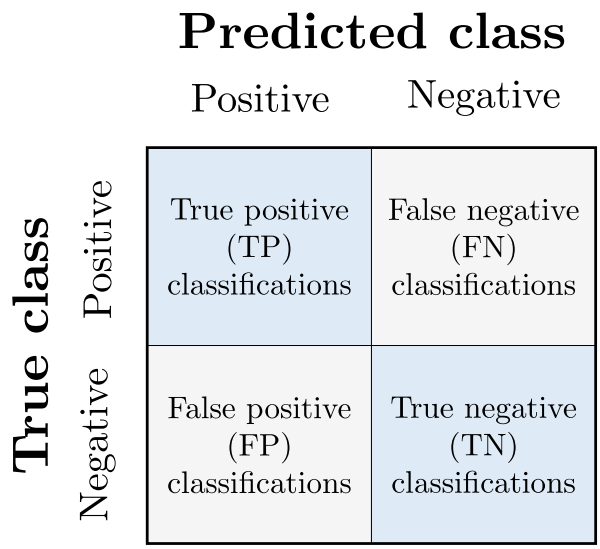


Fig. 2.1: Confusion matrix for binary classification tasks. Adapted from [28].

- **Precision**: The precision is a measure for the portion of all samples labeled as *positive* by the classifier that also belong to the class *positive* [28]:

$$precision = \frac{TP}{TP+FP}. \tag{2.12}$$

- **Recall**: The recall, also referred to as *sensitivity* or *true positive rate t*, represents the portion of data samples that belong to the class *positive*, which also got labeled as *positive* by the classifier [28], [29]:

$$recall = \frac{TP}{TP+FN}. \tag{2.13}$$

- **Specificity**: Analogous to the recall, the specificity denotes the correctly classified portion of all samples of the class *negative* [28]:

$$specificity = \frac{TN}{TN+FP}. \tag{2.14}$$

- **False positive rate**: The false positive rate represents the ratio of *negative* data samples that were erroneously assigned to the class *positive* [29]:

$$f = \frac{FP}{TN+FP} = 1 - specificity. \tag{2.15}$$

- **F_β-measure**: The F_β-measure combines the recall (see Equation 2.13) and precision (see Equation 2.12) to form the following metric [28]:

$$F_\beta = \frac{(\beta^2+1) \cdot precision \cdot recall}{\beta^2 \cdot precision + recall}, \tag{2.16}$$

where β is a weighting factor to favor precision ($\beta > 1$) or recall ($\beta < 1$). The weighting of both metrics is balanced for $\beta = 1$, which yields the commonly used F_1-measure [28].

- **Area under the ROC curve (AUC)**: The *receiver operating characteristic (ROC) curve* visualizes the trade-off between the sensitivity, or true positive rate, t (see Equation 2.13) and the false positive rate f (see Equation 2.15) [29]. Exemplary ROC curves are shown in Figure 2.2. Consider the task of labeling each sample of a data set as either anomalous

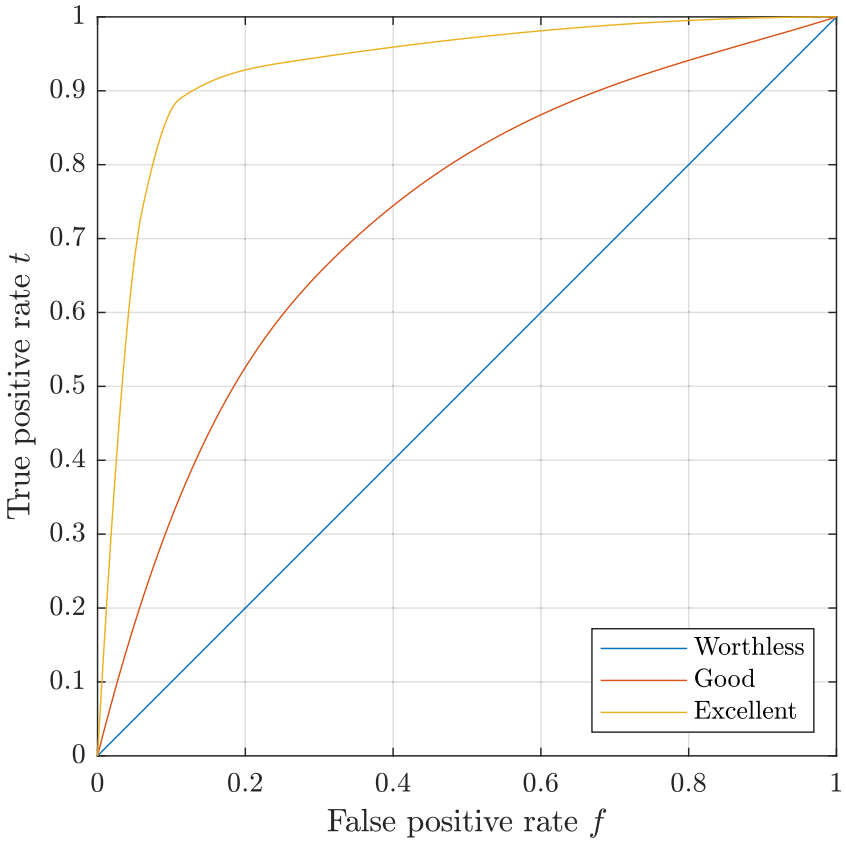


Fig. 2.2: Exemplary receiver operating characteristic (ROC) curves that depict the relationship between the true positive rate and the false positive rate. The aim is to produce curves that get close to the left upper corner [29]. Adapted from [30].

(positive class) or non-anomalous (negative class). The objective would be to assign as many anomalous data sets as possible to the class positive, i.e., achieve a high true positive rate. On the other hand, it is aspired that the number of non-anomalous data sets erroneously flagged as outliers is low, which is equal to a low false positive rate. The ROC can be a helpful tool to decide where to place the threshold in order to achieve a good compromise. Figure 2.2 indicates that the ROC curve should be close to the upper left corner, i.e., the area below the curve should be maximized. This would enable a high true positive rate and an acceptable false positive rate [29].

Thus, the area under the ROC curve (AUC) can also be used as a metric to determine a

classifier's performance [31]:

$$AUC = \int_0^1 t \, \mathrm{d}f. \tag{2.17}$$

- **Matthew's Correlation Coefficient (MCC)**: The MMC was proven to be a more reliable performance metric for binary classification tasks than the popularly used accuracy and F_1-measure. This is because it only delivers a high score, indicating a good prediction, if the classifier performs well in all four categories of the confusion matrix (TP, TN, FP, FN) [32]. Suppose a classifier is tested with a set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n\}$ of n data samples $\boldsymbol{x}_i$. Based on the corresponding predicted classes $\hat{y}_i$ and true classes y_i, two vectors $\boldsymbol{a} = [a_1, a_2, ..., a_n]^{\mathrm{T}}$ and $\boldsymbol{b} = [b_1, b_2, ..., b_n]^{\mathrm{T}}$ are defined as follows [33]:

$$a_i = \begin{cases} 1 & \text{if } \boldsymbol{x}_i \text{ belongs to class } y_i = \text{positive} \\ 0 & \text{if } \boldsymbol{x}_i \text{ belongs to class } y_i = \text{negative} \, , \end{cases} \tag{2.18}$$

$$b_i = \begin{cases} 1 & \text{if } \boldsymbol{x}_i \text{ has been assigned to class } \hat{y}_i = \text{positive} \\ 0 & \text{if } \boldsymbol{x}_i \text{ has been assigned to class } \hat{y}_i = \text{negative} \, . \end{cases} \tag{2.19}$$

The MCC represents *Pearson's correlation coefficient* of these two given binary vectors [33]. It can be obtained as follows [32]:

$$MMC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP) \times (TP+FN) \times (TN+FP) \times (TN+FN)}}. \tag{2.20}$$

A higher correlation between the target vector $\boldsymbol{a}$ and the prediction vector $\boldsymbol{b}$ indicates a better classification performance [33]. The MCC delivers values between -1 (perfect misclassification) and $+1$ (perfect classification) [32].

Chapter 3
Artificial Neural Networks (ANN)

As described in Section 2.2.1, a neural network is a machine learning model that is inspired by the structure of the brain [20]. It contains a variety of simple processing units, referred to as neurons, that form a net structure through massive interconnections similar to the synopses in the brain of humans or animals. By the use of learnable weighting factors, the importance of each connection can be modified and optimized in the training process. This type of structure enables the network to efficiently store gathered information and solve highly complex tasks [34]. This chapter gives a brief introduction to the basics of artificial neural networks.

3.1 Model of a Neuron

Neurons process a set of weighted inputs by passing them through a nonlinear function, the so-called *activation function*. This designation stems from the fact that the generated output of the activation function is also referred to as *activation* of the respective neuron [20], [34]. Figure 3.1 shows a schematic diagram of a neuron k that receives an m-dimensional input vector $\boldsymbol{x} = [x_1, x_2, ..., x_m]^{\mathrm{T}}$, whose elements are weighted via weighting factors w_{kj}, with $j = \{1, ..., m\}$ [34].

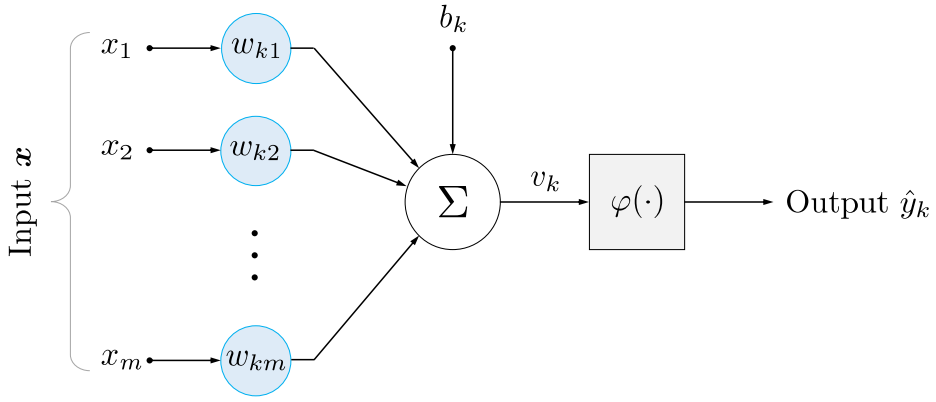


Fig. 3.1: Schematic diagram of a neuron that sums up the weighted inputs $w_{kj}x_j$ and adds a bias b_k, then squashes this sum v_k through an activation function φ to compute $\hat{y}_k$ [34]. Adapted from [34].

The weighted inputs are summed up, and a bias b_k is added. The result of this summation v_k is handed over to an activation function φ to compute the output $\hat{y}_k$. These operations can be formulated as follows [34]:

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k, \tag{3.1}$$

$$\hat{y}_k = \varphi(v_k). \tag{3.2}$$

In modern neural networks, a variety of different activation functions are applied, which are selected depending on the task to be accomplished. Some of the most common ones are presented in the following listing and Figure 3.2:

- **Sigmoid:** The differentiable S-shaped sigmoid function σ outputs values in the range [0,1] and is characterized by a smooth transition between linear and nonlinear behavior [22], [34]. When referring to the sigmoid function in the context of neural networks, usually a special case, the *logistic* sigmoid function, is meant [21], [34]. Given the sum of weighted inputs v_k and the slope parameter a, it is defined as follows [34]:

$$\sigma(v_k) = \frac{1}{1 + \mathrm{e}^{-a v_k}}. \tag{3.3}$$

- **Hyperbolic tangent:** In cases where a function with a good balance between linearity and nonlinearity similar to the sigmoid function, but with an output range of [-1,1], is required, the hyperbolic tangent is a popular option. It is defined by [34]:

$$\tanh(v_k) = \frac{\mathrm{e}^{v_k} - \mathrm{e}^{-v_k}}{\mathrm{e}^{v_k} + \mathrm{e}^{-v_k}}. \tag{3.4}$$

- **Rectified linear unit (ReLU):** The ReLU function outputs all input values v_k with the property $v_k > 0$ and returns a zero for inputs $v_k \leq 0$. It can be formulated as follows [22]:

$$\varphi(v_k) = \max(0, v_k). \tag{3.5}$$

  The ReLU function exhibits a constant first derivative of $\varphi'(v_k) = 1$ for all values that activate the neuron and a second derivative of $\varphi''(v_k) = 0$ everywhere it can be defined. In modifications like the *leaky ReLU*, nonzero values for the slope in the domain with negative input values are defined or even optimized in the training process [22].

3.2 Network Architectures

A neural network consists of multiple processing units that are organized in the form of layers. Over the past decades, various different network architectures emerged that differ, e.g., in the way how neurons communicate with each other or how the learnable parameters are optimized

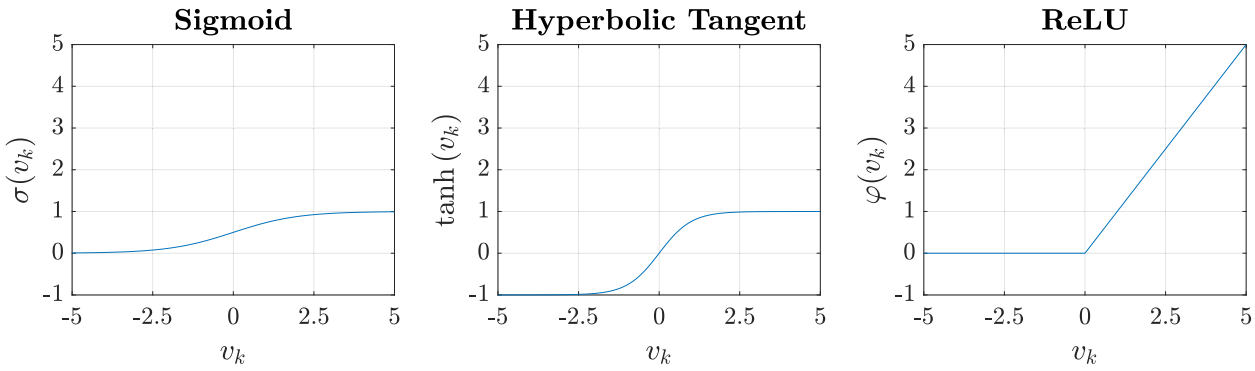


Fig. 3.2: Common activation function types used in neural networks. Adapted from [35].

in the training process. *Feedforward neural networks*, *recurrent neural networks (RNN)*, and *convolutional neural networks (CNN)* constitute the most significant part of architectures in use today [20], [34]. The two first-mentioned architectures are of interest in the context of this thesis and therefore will be presented in more detail in the following sections.

3.2.1 Feedforward Neural Networks

A feedforward network contains, at minimum, an input layer that receives and forwards the input $\boldsymbol{x} \in \mathbb{R}^m$, and an output layer consisting of neurons that outputs the processed data in the required dimension. This simplest form of a feedforward network is referred to as *single-layer feedforward network*. In many applications, adding additional layers between the input and output layer may be beneficial. Since the outputs of these layers are not directly accessible, these layers are called *hidden layers* [34].

A feedforward network is characterized by a directed acyclic graph. The term *acyclic* refers to the fact that feedforward networks do not exhibit connections that feed back outputs into the model, as it is practiced in recurrent networks [20]. Typically, each neuron of a hidden layer or the output layer only receives data from the previous network layer. If each neuron is connected to all neurons in the adjacent layers, the network is called *fully connected*. In contrast, networks with missing connections are referred to as *partially connected* networks [34].

A possible architecture of a fully connected feedforward network is shown in Figure 3.3. As described in Section 3.1, a neuron's inputs are weighted via the use of weighting factors. For a network with multiple layers, the concept presented in Section 3.1 needs to be extended. Each of the l layers in a network, apart from the input layer, is identified by an index i, with $i = \{1, ..., l\}$. a_k^i represents the activation of the k-th neuron in layer i, where $k = \{1, ..., n_i\}$ for a layer with n_i hidden units. The activations of a layer i are combined to form an activation vector $\boldsymbol{a}^i$ [36]:

$$\boldsymbol{a}^i = \left[a_1^i, a_2^i, \ldots, a_{n_i}^i\right]^{\mathrm{T}}. \tag{3.6}$$

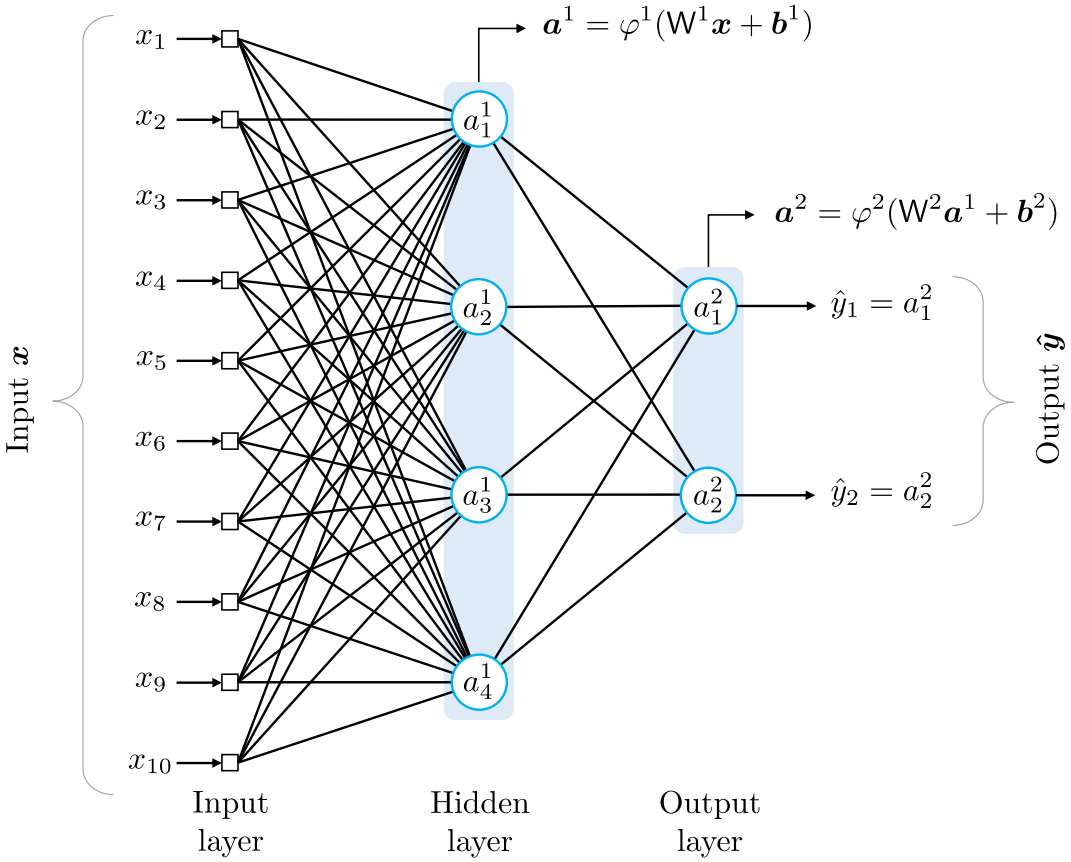


Fig. 3.3: Exemplary graph of a fully connected feedforward network with one hidden layer. The network processes an input vector $\boldsymbol{x}$ of dimension $m = 10$ and outputs a vector $\boldsymbol{y}$ of dimension $n_l = 2$. The blue circles represent the neurons with the corresponding activations a_k^i written inside. Adapted from [34].

Furthermore, $w_{k,j}^i$ represents the weighting factor of the k-th neuron in layer i, which scales the input a_j^{i-1} received from the j-th neuron in the previous layer $i-1$. These weighting factors are organized in a *weight matrix* W^i, which is defined for each layer i separately as follows [36]:

$$\mathsf{W}^i = \begin{bmatrix} w_{1,1}^i & \dots & w_{1,n_{i-1}}^i \\ \vdots & \ddots & \vdots \\ w_{n_i,1}^i & \dots & w_{n_i,n_{i-1}}^i \end{bmatrix}, \tag{3.7}$$

where n_{i-1} represents the number of neurons in layer $i-1$. Note that n_0 is equal to the dimension of the input vector $\boldsymbol{x}$,

$$\boldsymbol{x} = \begin{bmatrix} x_1, x_2, \dots, x_m \end{bmatrix}^{\mathrm{T}}, \tag{3.8}$$

i.e., $n_0 = m$. The vector containing the biases b_k^i of all neurons k in a layer i is defined by [36]:

$$\boldsymbol{b}^i = \begin{bmatrix} b_1^i, b_2^i, \dots, b_{n_i}^i \end{bmatrix}^{\mathrm{T}}. \tag{3.9}$$

By adding this bias vector to the weighted sum of inputs, the vector $\boldsymbol{v}^i$ is obtained [36]:

$$\boldsymbol{v}^i = \mathsf{W}^i \boldsymbol{a}^{i-1} + \boldsymbol{b}^i. \tag{3.10}$$

The entries v_k^i of $\boldsymbol{v}^i$ are squashed through an activation function $\boldsymbol{\varphi}^i$ to determine the activation of the corresponding neurons k. This operation can be formulated as follows [36]:

$$\boldsymbol{a}^i = \boldsymbol{\varphi}^i(\boldsymbol{v}^i). \tag{3.11}$$

For computing the activation vector $\boldsymbol{a}^1$ of the first hidden layer, it has to be considered that $\boldsymbol{a}^0 = \boldsymbol{x}$, where $\boldsymbol{x}$ represents the input vector. The activation vector of the last layer represents the output of the network $\hat{\boldsymbol{y}}$ [36]:

$$\hat{\boldsymbol{y}} = \left[\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{n_l}\right]^{\mathrm{T}} = \left[a_1^l, a_2^l, \ldots, a_{n_l}^l\right]^{\mathrm{T}}. \tag{3.12}$$

3.2.2 Recurrent Neural Networks

A recurrent neural network processes sequential or time-series data and is able to preserve information from previous computations. It receives a sequence $\mathcal{X} = \{\boldsymbol{x}(1), \ldots, \boldsymbol{x}(\tau)\}$ of input vectors $\boldsymbol{x}(t)$, each associated with a time instant or indexing variable $t = \{1, ..., \tau\}$ [22],

$$\boldsymbol{x}(t) = \left[x_1(t), x_2(t), \ldots, x_m(t)\right]^{\mathrm{T}}, \tag{3.13}$$

and produces a corresponding sequence $\hat{\mathcal{Y}} = \{\hat{\boldsymbol{y}}(1), \ldots, \hat{\boldsymbol{y}}(\tau)\}$ of output vectors $\hat{\boldsymbol{y}}(t)$ [22],

$$\hat{\boldsymbol{y}}(t) = \left[\hat{y}_1(t), \hat{y}_2(t), \ldots, \hat{y}_{n_l}(t)\right]^{\mathrm{T}}. \tag{3.14}$$

The current output is influenced by past elements of the time-series or sequence by feeding back information via directed cycles referred to as *recurrent connections*. The majority of recurrent neural networks are capable of processing time-series or sequences with varying lengths [21], [22].
If a recurrent network contains multiple hidden layers that exhibit feedback connections, it is referred to as *deep recurrent neural network* [37]. Each neuron k of a recurrent network layer i is associated with a time-dependent *hidden state* $h_k^i(t)$, which represents the memory of the hidden unit. The hidden states of all neurons in a layer i are concatenated to form a vector $\boldsymbol{h}^i(t)$ [22], [37],

$$\boldsymbol{h}^i(t) = \left[h_1^i(t), h_2^i(t), \ldots, h_{n_i}^i(t)\right]^{\mathrm{T}}, \tag{3.15}$$

which is updated at each time instant t. This hidden state vector is a function of the current input $\boldsymbol{x}^i(t)$ of layer i and the hidden state vector of the previous time step $\boldsymbol{h}^i(t-1)$ [22], [37]:

$$\boldsymbol{h}^i(t) = \boldsymbol{\varphi}_h^i\left(\mathsf{W}_h^i \boldsymbol{x}^i(t) + \mathsf{R}_h^i \boldsymbol{h}^i(t-1) + \boldsymbol{b}_h^i\right). \tag{3.16}$$

In addition to a bias vector $\boldsymbol{b}_h^i$ and a weight matrix W_h^i for the layer input $\boldsymbol{x}^i(t)$, a recurrent weight matrix R_h^i for the hidden state $\boldsymbol{h}^i(t-1)$ is introduced. The subscript h indicates that these matrices or vectors contain the parameters for the computation of the hidden state vector. An activation function φ_h^i, e.g., a hyperbolic tangent, is applied to the weighted sum. The output vector $\hat{\boldsymbol{y}}^i(t)$ at time instant t is a function of the hidden state vector $\boldsymbol{h}^i(t)$ and can be determined by [22], [37]:

$$\hat{\boldsymbol{y}}^i(t) = \varphi_y^i\left(\mathsf{R}_y^i \boldsymbol{h}^i(t) + \boldsymbol{b}_y^i\right), \tag{3.17}$$

where R_y^i represents the recurrent weight matrix, $\boldsymbol{b}_y^i$ the bias vector, and φ_y^i the activation function for the computation of the layer output vector $\hat{\boldsymbol{y}}^i(t)$. Figure 3.4 shows a graphical representation of the previously presented concept. On the left side, a network containing one recurrent layer i can be seen, represented by a dashed blue square. It receives a sequence of input vectors $\boldsymbol{x}^i$ and computes a corresponding sequence of output vectors $\hat{\boldsymbol{y}}^i$. This figure also visualizes how the hidden state is fed back to the same layer to preserve information from previous time steps. An alternative representation of this recurrent network is shown on the right side of Figure 3.4. Here, a recurrent network layer is represented by separate layers for each time instant $t = \{1, ..., \tau\}$. Each of these layers applies the same weight matrices, bias vectors, and activation functions for the computation, i.e., the parameters are shared across the layers [22], [37]. In Figure 3.5, a deep RNN with two re-

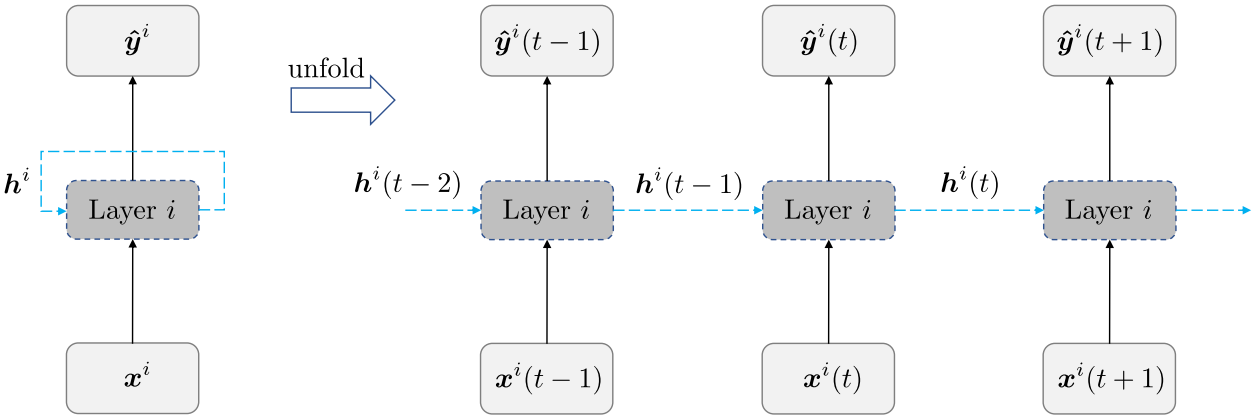


Fig. 3.4: Schematic diagram of an RNN with one recurrent network layer. The left side of the figure visualizes the cycle that feeds information of previous time steps back to the recurrent layer. The right side shows the same network unfolded across time, i.e., represented by separate layers for each time instant t [22]. Adapted from [38].

current network layers is shown. Equation (3.16) and (3.17) also hold for multilayer networks, where it has to be considered that the input vector of layer i at time instant t is equal to the output vector of the previous layer, i.e., $\boldsymbol{x}^i(t) = \hat{\boldsymbol{y}}^{i-1}(t)$. Note that the first recurrent layer receives the input data vector $\boldsymbol{x}(t)$, i.e., $\boldsymbol{x}^1(t) = \boldsymbol{x}(t)$ [37].

For training of recurrent neural networks, a variant of the *backpropagation* algorithm, the so-called *backpropagation through time (BPTT)* algorithm, can be used (see Section 3.3.2) [22]. Major weak-

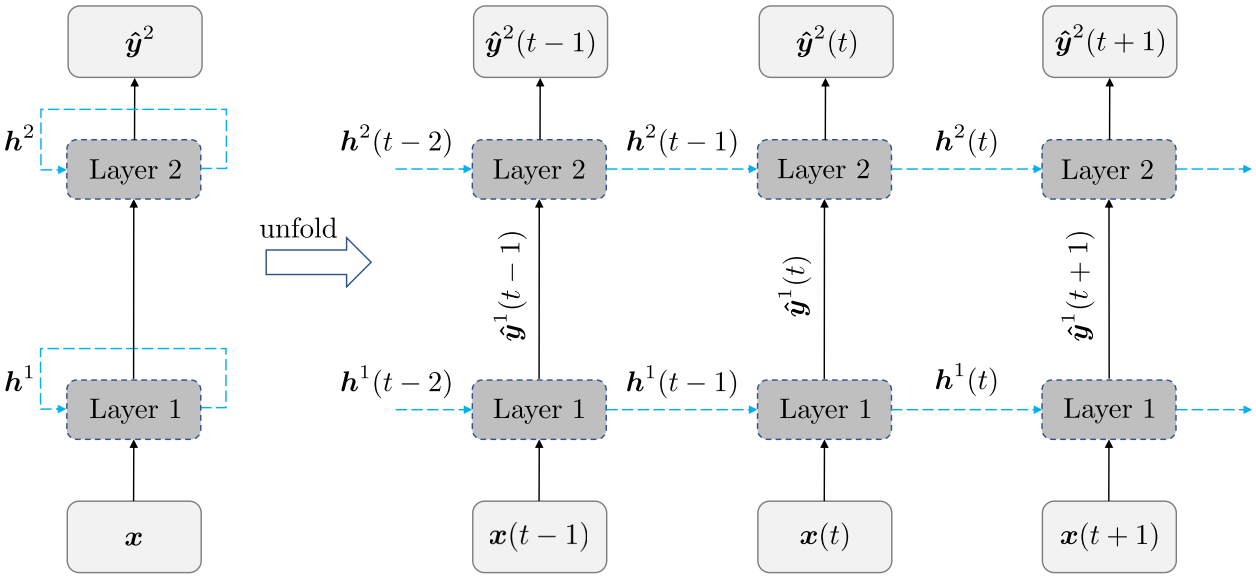


Fig. 3.5: Schematic diagram of an RNN with two recurrent layers. On the left side of the figure, the cycles that feed information of previous time steps back to the network layers are shown. The right side visualizes the same network unfolded across time, i.e., both layers are represented by separate layers for each time instant t [22]. Adapted from [37].

nesses of RNNs when using BPTT are vanishing or exploding gradients, i.e., gradients that shrink or grow with each time step. In the case of vanishing gradients, the training takes an unacceptable amount of time or even does not work at all, whereas exploding gradients lead to an oscillation of the weights. These phenomena make simple RNNs unable to capture long-term dependencies in the input data [12].
A significant improvement regarding the problem of vanishing gradients can be achieved by using special types of RNN architectures, which will be explained in the following [39].

3.2.2.1 Long Short-Term Memory (LSTM)

LSTMs were first introduced by Sepp Hochreiter and Juergen Schmidhuber [12] in 1997 as a new type of recurrent network architecture that is able to keep track of long-term dependencies in time-series and sequential data. The authors addressed the problem of vanishing gradients state-of-the-art recurrent networks suffered from and introduced a so-called *cell state*. The cell state acts as a memory that stores information about past time steps and is updated via *gate* units that decide which information is important to keep. In 1999, Gers et al. [40] presented an improved version of the LSTM by adding a *forget gate* to the existing *input* and *output gate*, which allows the removal of information from the cell state that is out of date.
The LSTM architectures used nowadays usually consist of a cell and four gate units: a forget gate, an input gate, a cell candidate gate, and an output gate (see Figure 3.6). An LSTM network may consist of multiple network layers. Thus, the notation for recurrent networks presented in Sec-

tion 3.2.2 is applied and extended. For each gate, an input weight matrix W^i, a recurrent weight matrix R^i, a bias vector $\boldsymbol{b}^i$, and an activation function φ^i are defined. The subscript indicates the corresponding gate these parameters belong to, with: e (input gate), f (forget gate), g (cell candidate gate), and o (output gate). As in Section 3.2.2, the superscript i represents the layer index [41].

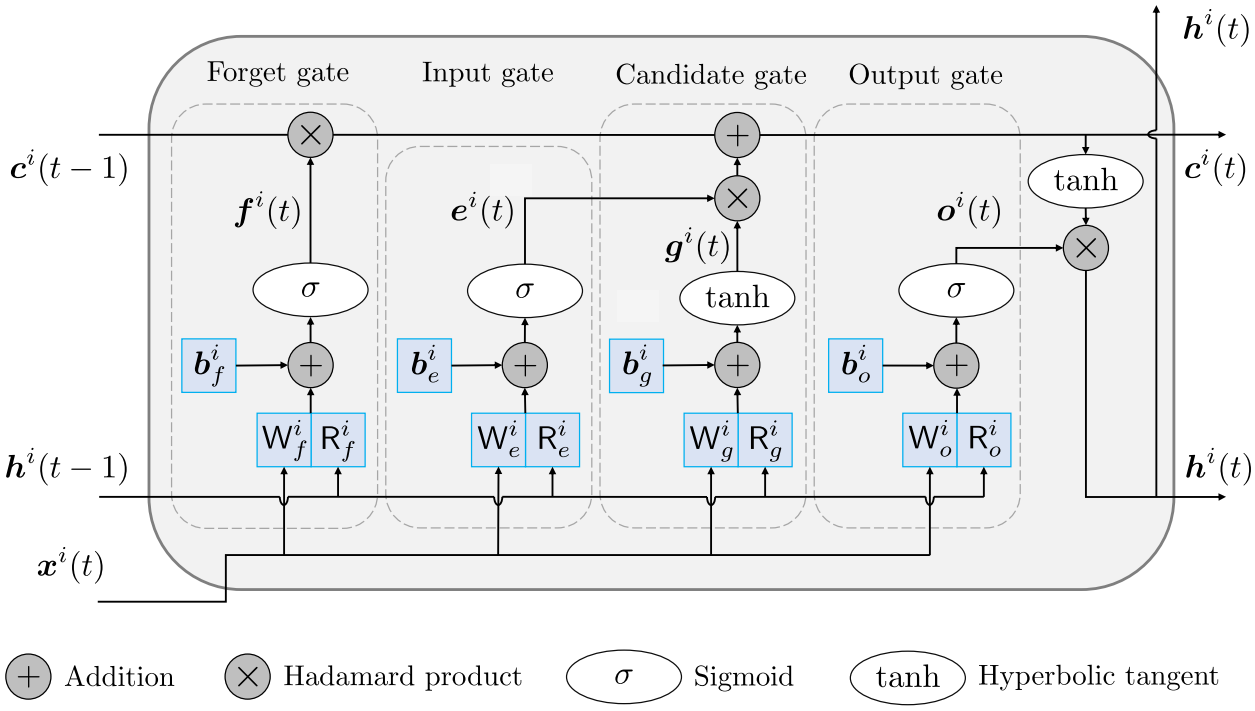


Fig. 3.6: Schematic diagram of an LSTM architecture. The vector of cell states $\boldsymbol{c}^i(t)$ acts as the long-term memory of an LSTM layer with index i and is modified at each time step via gate units. The output of an LSTM layer i at time instant t is the hidden state $\boldsymbol{h}^i(t)$ [22]. Adapted from [42].

Forget gate:

The forget gate defines which information is dispensable at the current time instance and therefore has to be discarded from the cell. This prevents the cell state from unbounded growing, which would lead to the loss of the memorizing capability of the LSTM [40]. Applying the sigmoid function on the weighted inputs yields the forget vector [41]:

$$\boldsymbol{f}^i(t) = \boldsymbol{\sigma}\left(\mathsf{W}_f^i \boldsymbol{x}^i(t) + \mathsf{R}_f^i \boldsymbol{h}^i(t-1) + \boldsymbol{b}_f^i\right). \tag{3.18}$$

The sigmoid function forces the entries in $\boldsymbol{f}^i(t)$ to be between zero and one [34], [41].

Input gate and cell candidate gate:

These two gates define which information is relevant to memorize and control the information flow to the cell. Analogous to the forget gate, a vector $\boldsymbol{e}^i(t)$ is computed by squeezing a weighted sum of the input $\boldsymbol{x}^i(t)$ and the hidden state $\boldsymbol{h}^i(t-1)$ through a sigmoid layer [41]:

$$\boldsymbol{e}^i(t) = \sigma\left(\mathsf{W}_e^i \boldsymbol{x}^i(t) + \mathsf{R}_e^i \boldsymbol{h}^i(t-1) + \boldsymbol{b}_e^i\right). \tag{3.19}$$

Additionally, a vector $\boldsymbol{g}^i(t)$ containing possible candidates for updating the cell state is determined as follows [41]:

$$\boldsymbol{g}^i(t) = \varphi_g^i\left(\mathsf{W}_g^i \boldsymbol{x}^i(t) + \mathsf{R}_g^i \boldsymbol{h}^i(t-1) + \boldsymbol{b}_g^i\right). \tag{3.20}$$

In modern LSTM architectures, the activation function φ_g^i of the candidate gate is a hyperbolic tangent [41], as shown in Figure 3.6, rather than a sigmoid function, as proposed in the original work of Hochreiter and Schmidhuber [12].

The vectors $\boldsymbol{f}^i(t)$, $\boldsymbol{e}^i(t)$, and $\boldsymbol{g}^i(t)$ are used to update the cell state. Element-wise multiplication (*Hadamard product* with operator $\circ$) of the cell state $\boldsymbol{c}^i(t-1)$ at the previous time instant $t-1$ with the forget vector $\boldsymbol{f}^i(t)$ removes outdated memory from the cell. New information is then added from the vector of cell candidates $\boldsymbol{g}^i(t)$, where a multiplication of $\boldsymbol{g}^i(t)$ with the vector $\boldsymbol{e}^i(t)$ of the input gate ensures only relevant information flows to the cell [41]:

$$\boldsymbol{c}^i(t) = \boldsymbol{f}^i(t) \circ \boldsymbol{c}^i(t-1) + \boldsymbol{e}^i(t) \circ \boldsymbol{g}^i(t). \tag{3.21}$$

Due to the use of a sigmoid function at the input gate, the values in $\boldsymbol{e}^i(t)$ lie in the range between zero and one, where a higher value denotes a higher importance of the corresponding value in the cell candidate vector $\boldsymbol{g}^i(t)$ [34], [41].

Output gate:

The output gate decides which information of the memory cell is used as output at the current time step [41]. Again, a sigmoid function is applied to a weighted sum of the input $\boldsymbol{x}^i(t)$ and the hidden state $\boldsymbol{h}^i(t-1)$ at the previous time step in order to create a vector $\boldsymbol{o}^i(t)$ of values between zero and one [34], [41]. This vector regulates the information flow from the cell and is determined by [41]:

$$\boldsymbol{o}^i(t) = \sigma\left(\mathsf{W}_o^i \boldsymbol{x}^i(t) + \mathsf{R}_o^i \boldsymbol{h}^i(t-1) + \boldsymbol{b}_o^i\right). \tag{3.22}$$

The hidden state vector $\boldsymbol{h}^i(t)$, which is also the output vector $\hat{\boldsymbol{y}}^i(t)$ of the LSTM layer i at time instant t, is computed as follows [22], [41]:

$$\boldsymbol{h}^i(t) = \boldsymbol{o}^i(t) \circ \tanh\left(\boldsymbol{c}^i(t)\right). \tag{3.23}$$

In most cases, the cell state vector $\boldsymbol{c}^i(t)$ is squashed through a hyperbolic tangent function prior to the element-wise multiplication with the output vector $\boldsymbol{o}^i(t)$, as shown in Figure 3.6 and Equation (3.23). Alternatively, a ReLU function could be used. The computed hidden state $\boldsymbol{h}^i(t)$ and cell state $\boldsymbol{c}^i(t)$ get passed on to the computation at the next time instant $t+1$ [22], [41].

3.2.2.2 Bi-directional Long Short-Term Memory (biLSTM)

An approach for a further improvement of recurrent neural networks was made by Mike Schuster and Kuldip K. Paliwal [43] in 1997, who introduced *bidirectional* recurrent neural networks. These networks consist of two hidden layers connected to the same input and output nodes. One of them, the *forward* layer, receives the time-series in the forward (positive) direction $t = \{0, ..., \tau\}$. The other layer, referred to as *backward layer*, is fed with the data presented in backward (negative) direction $t = \{\tau, ..., 0\}$. While this offers the potential of improving the prediction process by consideration of future context, this architecture cannot be used in online settings, as data of future time steps would have to be available [41]. For training, the BPTT algorithm can be used [43]. In 2005, Alex Graves and Juergen Schmidhuber [13] applied the concept of bidirectional RNNs to Long-Short Term memory architectures nowadays called *biLSTM*.
As shown in Figure 3.7, hidden state vectors $\boldsymbol{h}_F^i(t)$ and $\boldsymbol{h}_B^i(t)$ are computed separately in the forward and the backward layer of a biLSTM layer with index i, respectively. These vectors are determined using the equations for the standard LSTM architecture, as presented in Section 3.2.2.1 in Equation (3.18) - (3.23). Note that separate weight matrices and bias vectors are used for the forward and the backward layer. The hidden state vectors $\boldsymbol{h}_F^i(t)$ and $\boldsymbol{h}_B^i(t)$ are combined to compute the output vector $\hat{\boldsymbol{y}}^i(t)$, using a function ψ^i [44]:

$$\hat{\boldsymbol{y}}^i(t) = \psi^i(\boldsymbol{h}_F^i(t), \boldsymbol{h}_B^i(t)). \tag{3.24}$$

This function can be a concatenating function, an averaging function, a multiplication function, or a summation function [44].

3.3 Training

As described in Chapter 2, the learnable parameters of machine learning models are modified in the training process in order to improve performance, measured by a given metric [21]. This section provides a brief introduction to the methods that are commonly used to optimize the parameters of neural networks.

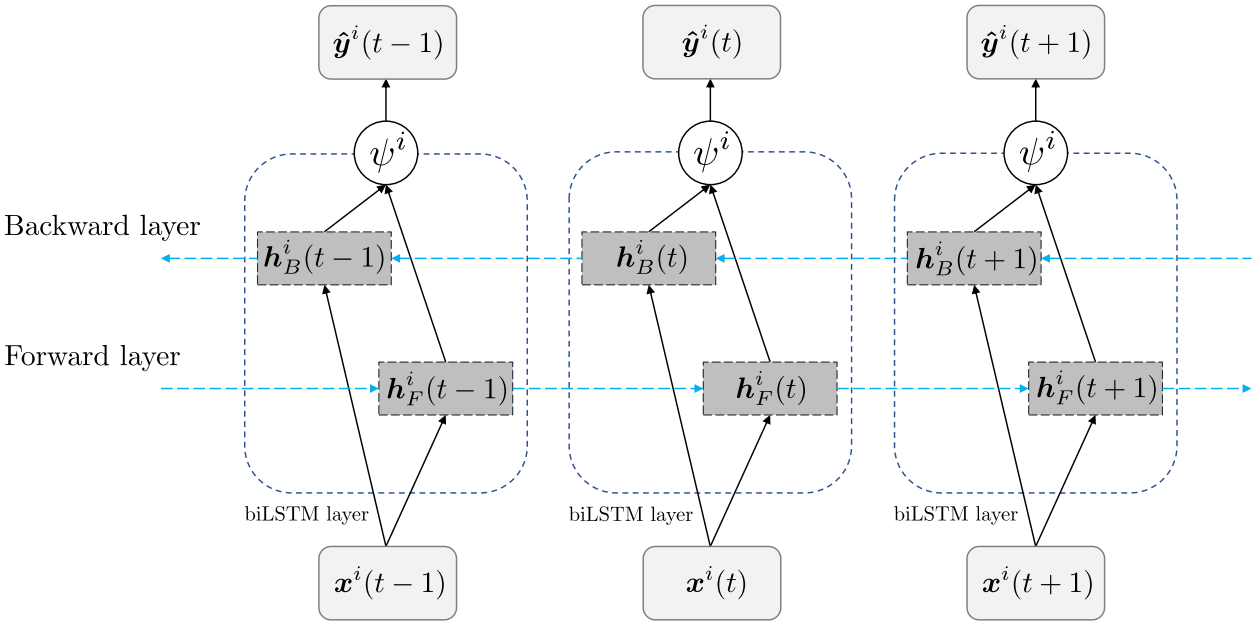


Fig. 3.7: Schematic diagram of a biLSTM architecture unfolded across time. At each time instant, the hidden state vectors $\boldsymbol{h}_F^i$ and $\boldsymbol{h}_B^i$ are computed in the *forward* layer and the *backward* layer, respectively. Applying a function ψ^i on these vectors yields the output $\hat{\boldsymbol{y}}^i$ of layer i at the corresponding time step [44]. Adapted from [44].

3.3.1 Gradient Descent

Gradient descent is an optimization procedure that iteratively updates the learnable parameters by taking a step in the direction of the negative gradient of some differentiable loss function $\mathcal{L} : \mathbb{R}^{n_l} \to \mathbb{R}$. The loss function maps the difference of the n_l-dimensional vector containing the obtained outputs of the network $\hat{\boldsymbol{y}} = [\hat{y}_1, \hat{y}_2, ..., \hat{y}_{n_l}]^{\mathrm{T}}$ and the vector with the corresponding targets $\boldsymbol{y} = [y_1, y_2, ..., y_{n_l}]^{\mathrm{T}}$ to a numerical value. The aim of training a network is to find the global minimum of this loss function, which is selected based on the task at hand [20], [21]. A commonly used metric is the mean squared error [21], [26]:

$$\mathrm{MSE} = \frac{1}{n_l} \sum_{i=1}^{n_l} (\hat{y}_i - y_i)^2. \tag{3.25}$$

See Section 2.3 for some further examples of performance metrics. To update the parameters, a step along the negative direction of the gradient $\nabla\mathcal{L}$ of the cost function $\mathcal{L}$ w.r.t. the corresponding parameter is taken [36]:

$$w_{k,j}^i(t+1) = w_{k,j}^i(t) - \eta \frac{\partial \mathcal{L}}{\partial w_{k,j}^i(t)}, \tag{3.26}$$

$$b_k^i(t+1) = b_k^i(t) - \eta \frac{\partial \mathcal{L}}{\partial b_k^i(t)}, \tag{3.27}$$

where the factor η represents the *learning rate*. For a given gradient, the learning rate defines the step size for updating the parameters. If it is set too low, the updates are too small. A learning rate that is too high, on the other hand, can lead to an oscillation of the loss around an optimum and therefore to the algorithm becoming unstable. Both scenarios can cause an unsuccessful or unacceptably long training process. Thus, the learning rate is often set to a higher initial value to approach the optimum quickly. After a defined number of epochs, it is decreased in order to prevent oscillation and allow convergence to the optimal value [36].
For the training of neural networks, a finite set of observations is used. The *batch gradient descent* algorithm computes and averages the gradient $\nabla\mathcal{L}$ of the cost function w.r.t. to all data samples. For large data sets, this method can be computationally very expensive, as it requires the allocation of significant computer resources to store the data [45].
Therefore, a variant of the standard gradient descent method, the so-called *stochastic gradient descent (SGD)* algorithm, was introduced. The SGD algorithm determines an approximation $\tilde{\nabla}\mathcal{L}$ by just computing the gradient of the loss function w.r.t. one random sample of the given data set, whereby $\mathbb{E}(\tilde{\nabla}\mathcal{L}) = \nabla\mathcal{L}$ [20]. Another method, referred to as *mini-batch gradient descent*, involves the computation of the gradients w.r.t. only a subset of the training data set, the *mini-batch (MB)* [45]. A variety of variants to further improve the optimization performance were developed. One of the most popular methods is the *Adam (adaptive moment estimation)* optimizer [46]. This algorithm determines individual learning rates for the optimizable parameters using estimates of the first and second moments of the gradient [46].
Choosing an appropriate optimization method is crucial for the performance in the training process and can help to reduce the duration of training significantly [47].

3.3.2 Backpropagation

3.3.2.1 Backpropagation in Feedforward Networks

In the prior section, it was explained how gradient descent is applied to update the network parameters using partial derivatives of the loss function. The computation of these gradients is carried out by the so-called *backpropagation* algorithm. As shown in Equation (3.10), the vector containing the sum of weighted inputs to the activation functions of the nodes $k = \{1,...,n_i\}$ in layer i is denoted by $\boldsymbol{v}^i$. Each entry v_k^i of this vector is the sum of a bias b_k^i and a linear combination of the activations a_j^{i-1} in the previous layer, where the coefficients are the weighting parameters $w_{k,j}^i$ [36]:

$$v_k^i = w_{k,1}^i a_1^{i-1} + w_{k,2}^i a_2^{i-1} + \cdots + w_{k,n_{i-1}}^i a_{n_{i-1}}^{i-1} + b_k^i. \tag{3.28}$$

The *sensitivity* of the loss function $\mathcal{L}$ w.r.t. to the input of the activation function v_k^i in layer i and neuron k is denoted as δ_k^i. It describes how much a change in the sum of weighted inputs v_k^i (and

thus a change in the weight matrix W^i or the bias vector $\boldsymbol{b}^i$) affects the value of the loss function $\mathcal{L}$, and is computed by [36], [48]:

$$\delta^i_k = \frac{\partial \mathcal{L}}{\partial v^i_k}. \tag{3.29}$$

The vector of activations $\boldsymbol{a}^i$ of layer i is computed by applying an activation function $\boldsymbol{\varphi}^i$ on $\boldsymbol{v}^i$ [36], as shown in Equation (3.11). Thus, the sensitivity δ^l_k for neuron k in the output layer l using the chain rule is computed by [36], [48]:

$$\delta^l_k = \frac{\partial \mathcal{L}}{\partial a^l_k}\frac{\partial a^l_k}{\partial v^l_k} = \frac{\partial \mathcal{L}}{\partial a^l_k}\frac{\partial \varphi^l(v^l_k)}{\partial v^l_k}. \tag{3.30}$$

In matrix-based notation, the vector $\boldsymbol{\delta}^l$ containing the sensitivity values δ^l_k for all nodes of the last layer l can be determined by [36], [48]:

$$\boldsymbol{\delta}^l = \dot{\boldsymbol{\Phi}}^l(\boldsymbol{v}^l)\nabla_{\boldsymbol{a}^l}\mathcal{L}, \tag{3.31}$$

where $\dot{\boldsymbol{\Phi}}^l(\boldsymbol{v}^l)$ indicates how fast the activation function $\boldsymbol{\varphi}^l$ is changing at the input values v^l_k of the neurons in the last layer. For any layer i, this matrix is defined as follows [36], [48]:

$$\dot{\boldsymbol{\Phi}}^i(\boldsymbol{v}^i) = \begin{bmatrix} \frac{\partial \varphi^i(v^i_1)}{\partial v^i_1} & 0 & \dots & 0 \\ 0 & \frac{\partial \varphi^i(v^i_2)}{\partial v^i_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial \varphi^i(v^i_{n_i})}{\partial v^i_{n_i}} \end{bmatrix}. \tag{3.32}$$

The gradient $\nabla_{\boldsymbol{a}^l}\mathcal{L}$ of the loss function $\mathcal{L}$ w.r.t. the vector of activations $\boldsymbol{a}^l$ in the last layer is defined by [36], [48]:

$$\nabla_{\boldsymbol{a}^l}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial a^l_1}, \frac{\partial \mathcal{L}}{\partial a^l_2}, \dots, \frac{\partial \mathcal{L}}{\partial a^l_{n_l}}\right]^{\mathrm{T}}. \tag{3.33}$$

The sensitivity vector $\boldsymbol{\delta}^i$ of the cost function w.r.t. a hidden layer i depends on the sensitivity vector $\boldsymbol{\delta}^{i+1}$ of the next layer in the network. To derive this relationship, consider how the input to node k in layer $i+1$ is affected by changes in the input to a neuron j in layer i [36]:

$$\begin{aligned} \frac{\partial v^{i+1}_k}{\partial v^i_j} &= \frac{\partial\left(\sum_{p=1}^{n_i} w^{i+1}_{k,p}a^i_p + b^{i+1}_k\right)}{\partial v^i_j} = w^{i+1}_{k,j}\frac{\partial a^i_j}{\partial v^i_j} \\ &= w^{i+1}_{k,j}\frac{\partial \varphi^i(v^i_j)}{\partial v^i_j}, \end{aligned} \tag{3.34}$$

where a_p^i represents the activation of a neuron p in layer i and $w_{k,p}^{i+1}$ the corresponding weighting factor of neuron k in layer $i+1$. From Equation (3.34), it can be derived that the *Jacobian matrix* $\frac{\partial \boldsymbol{v}^{i+1}}{\partial \boldsymbol{v}^i}$ is defined by [36]:

$$\frac{\partial \boldsymbol{v}^{i+1}}{\partial \boldsymbol{v}^i} = \mathsf{W}^{i+1} \dot{\boldsymbol{\Phi}}^i(\boldsymbol{v}^i). \tag{3.35}$$

Now the computation of the sensitivity vector $\boldsymbol{\delta}^i$ of layer i from the sensitivity vector $\boldsymbol{\delta}^{i+1}$ of layer $i+1$ using the chain rule can be formulated as [36]:

$$\begin{aligned} \boldsymbol{\delta}^i = \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}^i} &= \left(\frac{\partial \boldsymbol{v}^{i+1}}{\partial \boldsymbol{v}^i} \right)^{\mathrm{T}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}^{i+1}} = \dot{\boldsymbol{\Phi}}^i(\boldsymbol{v}^i)(\mathsf{W}^{i+1})^{\mathrm{T}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}^{i+1}} \\ &= \dot{\boldsymbol{\Phi}}^i(\boldsymbol{v}^i)(\mathsf{W}^{i+1})^{\mathrm{T}} \boldsymbol{\delta}^{i+1}. \end{aligned} \tag{3.36}$$

This operation can be thought of as a backpropagation of the sensitivity $\boldsymbol{\delta}^{i+1}$ of layer $i+1$ to the i-th layer. Consequently, the sensitivities are backpropagated from the last layer of the network to the first one, hence the name *backpropagation* algorithm. The computation of the sensitivity vectors for all layers in the network, as shown in Equation (3.31) and (3.36), allows determining the partial derivatives of the loss function w.r.t. the weights and biases simply by [36], [48]:

$$\frac{\partial \mathcal{L}}{\partial b_k^i} = \underbrace{\frac{\partial \mathcal{L}}{\partial a_k^i} \frac{\partial a_k^i}{\partial v_k^i}}_{\delta_k^i} \underbrace{\frac{\partial v_k^i}{\partial b_k^i}}_{\text{see (3.28)}} = \delta_k^i, \tag{3.37}$$

$$\frac{\partial \mathcal{L}}{\partial w_{k,j}^i} = \underbrace{\frac{\partial \mathcal{L}}{\partial a_k^i} \frac{\partial a_k^i}{\partial v_k^i}}_{\delta_k^i} \underbrace{\frac{\partial v_k^i}{\partial w_{k,j}^i}}_{\text{see (3.28)}} = \delta_k^i a_j^{i-1}, \tag{3.38}$$

where j represents the index of a neuron in layer $i-1$. For a more detailed derivation of the presented equations, the reader is referred to [48] and [36].

3.3.2.2 Backpropagation in Recurrent Networks

Training of recurrent neural networks requires a variant of the standard backpropagation algorithm to compute the gradients, the backpropagation through time algorithm [49]. Suppose the RNN is unfolded across time, as shown in Figure 3.4. At each time instant t, the network computes an output vector $\hat{\boldsymbol{y}}(t)$. With the corresponding target vector $\boldsymbol{y}(t)$, a loss $\ell^{(t)}$ at time instant t is determined. The overall loss function $\mathcal{L}$ is obtained by adding up the losses $\ell(t)$ at each time instant t [50]:

$$\mathcal{L} = \sum_{t=1}^{\tau} \ell(t), \tag{3.39}$$

with $t = \{1, \ldots, \tau\}$. A complete derivation of the algorithm is presented in the original paper of Paul J. Werbos [49]. In the following, it is discussed how the problem of vanishing or exploding gradients arises when training an RNN with BPTT. For simplicity, a network with one hidden layer is assumed. Therefore, the superscript *i* denoting the layer is omitted in the notation of the following equations. Updating the recurrent weight matrix R_h (see Section 3.2.2) involves computing the gradient of the loss function $\ell(t)$ at each time instant t w.r.t. this matrix [50]:

$$\frac{\partial \ell(t)}{\partial \mathsf{R}_h} = \frac{\partial \ell(t)}{\partial \hat{\boldsymbol{y}}(t)} \frac{\partial \hat{\boldsymbol{y}}(t)}{\partial \boldsymbol{h}(t)} \frac{\partial \boldsymbol{h}(t)}{\partial \mathsf{R}_h}. \tag{3.40}$$

The hidden state vector $\boldsymbol{h}(t)$ is a function of the hidden state vector $\boldsymbol{h}(t-1)$ at the previous time instant $t-1$, as shown in Equation (3.16). The vector $\boldsymbol{h}(t-1)$, in turn, is a function of the recurrent weight matrix R_h and therefore cannot be treated as a constant when computing $\frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{R}_h}$ in Equation (3.40). As each hidden state vector depends on the corresponding vector at the previous time instant, $\boldsymbol{h}(t)$ is a function of all hidden state vectors $\boldsymbol{h}(p)$ up to time instant $t-1$, where $p = \{1, ..., t-1\}$. For each $t > 1$, applying the chain rule yields [50], [51]:

$$\frac{\boldsymbol{h}(t)}{\partial \mathsf{R}_h} = \sum_{p=1}^{t-1} \frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{h}(p)} \frac{\partial \boldsymbol{h}(p)}{\partial \mathsf{R}_h}, \tag{3.41}$$

whereby [50], [51],

$$\frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{h}(p)} = \prod_{d=p+1}^{t} \frac{\partial \boldsymbol{h}(d)}{\partial \boldsymbol{h}(d-1)} = \prod_{d=p+1}^{t} (\mathsf{R}_h)^{\mathsf{T}} \operatorname{diag} \boldsymbol{\varphi}_h' [\boldsymbol{h}(d-1)]. \tag{3.42}$$

The recurrent weight matrix R_h appears in the product in Equation (3.42) $t-p$ times. When dealing with long-term dependencies, i.e., $p << t$, this can cause the gradients to explode or vanish depending on the value of the largest eigenvalue λ_1 of R_h [51]. A detailed explanation of under which circumstances the gradients explode or shrink is given in [51].
A solution for the exploding gradient problem is the *truncated backpropagation through time* algorithm, which clips the gradients if they grow above a certain threshold. An improvement regarding the vanishing gradient problem can be achieved with LSTM (see Section 3.2.2.1) [50].

3.3.3 Weight Initialization

In Section 3.3, it was illustrated how the weights of a network are updated based on the gradient of the loss function. The initial values for these parameters are set according to some method specified by the user. While they were set to zero or one in the early days of neural networks, some more elaborate weight initializing methods were developed over the course of the years [52]. These initializers usually sample the initial weight values from a defined distribution, whose pa-

rameters may depend on the network architecture, e.g., the input dimension n_{i-1} and the output dimension n_i of a layer i in a fully connected network [53], [54]. In the following listing, some of the most popular weight initializers for neural networks are shown. Unless specified otherwise, the presented methods are defined according to their implementation in MATLAB [55]:

- **Narrow-normal initializer:** This initializer samples the initial parameter values from a normal distribution $\mathcal{N}$ with zero mean and a fixed standard deviation, which is set to $\sigma = 0.01$ in MATLAB [55],

$$w_{k,j}^{i} \sim \mathcal{N}(0, \sigma^2). \tag{3.43}$$

  In *Keras* [56], a deep learning library written in PYTHON, a similar method called *random normal* is implemented.
- **Random uniform:** This initializing method included in Keras [56] draws the values from a uniform distribution $\mathcal{U}$ with limits a and b, e.g., $a = -0.05$ and $b = 0.05$,

$$w_{k,j}^{i} \sim \mathcal{U}(a, b). \tag{3.44}$$

  Currently, no comparable method is available in MATLAB [55].
- **Glorot initializer:** The *Glorot* (also called *Xavier*) initializer [53] draws the parameter values from a uniform distribution with limits that depend on the input and the output dimension of the layer,

$$w_{k,j}^{i} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{i-1}+n_i}}, \sqrt{\frac{6}{n_{i-1}+n_i}}\right). \tag{3.45}$$

  In *Keras* [56], this method is referred to as *Glorot uniform*. Based on the theoretical considerations formulated in [53], a so-called *Glorot normal* initializer is defined in *Keras*. This initializer samples the initial values from a truncated normal distribution that is centered around zero, with a standard deviation that depends on the input and output dimension [56],

$$w_{k,j}^{i} \sim \mathcal{N}\left(0, \frac{2}{n_{i-1}+n_i}\right). \tag{3.46}$$

- **He initializer:** The *He* initializer [54] samples the parameter values from a truncated normal distribution with a mean of zero and a standard deviation that depends on the input dimension,

$$w_{k,j}^{i} \sim \mathcal{N}\left(0, \frac{2}{n_{i-1}}\right). \tag{3.47}$$

  Analogous to the Glorot initializer, Keras [56] distinguishes between the *He normal* initializer, as defined in Equation (3.47), and the *He uniform* initializer. This derived variant draws the initial parameter values from a uniform distribution,

$$w_{k,j}^{i} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{i-1}}}, \sqrt{\frac{6}{n_{i-1}}}\right). \tag{3.48}$$

- **Orthogonal:** This technique presented in [57] initializes the weights with an orthogonal matrix Q computed by a QR-decomposition of a random matrix $\mathsf{Z} = \mathsf{QR}$. The matrix Z is sampled from a unit normal distribution [55].

3.4 Hyperparameter Optimization

Machine learning enables the user to solve highly complex tasks without the need to have full insight into the actual mechanisms involved in the computation process. However, it requires some knowledge and experience to choose a suitable machine learning model and its parameters that are not optimized in the learning process. These parameters can either configure a model and define its architecture or specify the optimizer that updates the weights and biases. An appropriate setting of these so-called *hyperparameters* is crucial for the performance of the machine learning model. Traditionally, suitable hyperparameter values were determined manually in a trial-and-error process. This method is ineffective for complex models with long computation times and hyperparameters that exhibit nonlinear interdependencies. This inspired the application of optimization techniques that tune the hyperparameters in an automated fashion. These methods aim to minimize (or maximize) a mostly non-convex and non-differentiable objective function $f(\boldsymbol{x})$ [58], [59]:

$$\boldsymbol{x}^\star = \underset{\boldsymbol{x}\in\mathcal{X}}{\arg\min}\, f(\boldsymbol{x}), \tag{3.49}$$

whereby $\boldsymbol{x}^\star$ denotes the hyperparameter configuration that delivers the best value of the objective function, and $\mathcal{X}$ represents the search space of the hyperparameters $\boldsymbol{x}$. Usually, constraints are imposed on a hyperparameter domain, and it must be defined if the domain is discrete, continuous, binary, or categorical [58].

A hyperparameter configuration can be evaluated via k-fold cross-validation. In this case, a given set of training samples is divided into k subsets. A machine learning model is set up using the hyperparameters to be tested. In a cyclic manner, this model is then trained k times using $k-1$ subsets, and tested on the remaining one. This is to mitigate or prevent overfitting of the model, i.e., it should also perform well on unseen data rather than just on the training samples [34], [59].

This chapter will focus on the optimization of hyperparameters of neural networks, as described in [14] and [15]. Possible hyperparameters are:

- **Number of epochs (discrete):** The (maximum) number of epochs in the training process, i.e., how many times each sample in the training data set passes the network [34].
- **Learning rate (continuous):** The factor defining the step size for the weight updates when using gradient descent (see Section 3.3.1) or its variants [20], [21].
- **Mini-batch size (discrete):** The number of data samples that pass the network before an update of the learnable parameters is performed [19].

- **Number of layers (discrete):** The number of layers of a certain type in the network [34].
- **Number of neurons (discrete):** The number of hidden units in a specific layer [34].
- **Activation function (categorical):** The activation function of the neurons in a specific layer [34].

Simple methods like *random search* or *grid search* attempt to exploit the whole search space without considering previous results. Other techniques, such as *Bayesian optimization* or *genetic algorithms*, focus the further search on regions in the search space that already were found to deliver suitable hyperparameters. Therefore, these methods usually enable a faster convergence to an optimum of the objective function. However, this increases the risk of getting stuck in a local optimum. Thus, selecting an appropriate hyperparameter optimization method and its parameters, as well as suitable constraints on the search space of the hyperparameters, are crucial tasks [58]. In the following, the most common hyperparameter optimization (HPO) methods are explained.

3.4.1 Grid Search

Grid search is a relatively simple brute-force optimization method used to exploit predefined regions in the search space. For each hyperparameter, the user defines a finite set of reasonable values or categories. Grid search evaluates the Cartesian product of these sets, i.e., all possible combinations of the distinct hyperparameter values or categories are tested. This method does not explore promising regions in the search space automatically. Thus, further runs with user-defined hyperparameter sets based on the results of previous runs may have to be performed until a good hyperparameter configuration is found. Grid search is only suitable for low-dimensional search spaces, as the number of evaluations to be performed increases exponentially with the number of hyperparameter sets [58].

3.4.2 Random Search

Instead of selecting the hyperparameters from given user-defined sets, the random search method draws them from predefined distributions. The user has to set the number of evaluations to perform and the parameters of the distributions the hyperparameters are sampled from. Random search is able to exploit a bigger search space than grid search for the same number of evaluations. Suppose one hyperparameter has negligible influence on the objective. Grid search tests a particular configuration of the remaining hyperparameters with each value or category from the set of the non-influential hyperparameter, with marginal changes in the objective. In random search, on the other hand, the parameters are drawn at random in each new test run [58], [60]. However, this method also does not narrow the search space based on the results of previous runs. Some more

advanced methods, which consider values of the objective function in already explored regions of the search space to determine future evaluation points, are presented in the following [58].

3.4.3 Bayesian Optimization

Bayesian optimization is a machine-learning-based optimization technique that attempts to find the global minimum or maximum of a continuous objective function $f(\boldsymbol{x})$ that is computationally expensive to evaluate [61]. Therefore, it was found to be a suitable tool for the optimization of hyperparameters, as described in [62]. The function input $\boldsymbol{x} \in \mathbb{R}^m$ usually has a dimension of $m \leq 20$ and can be corrupted by noise. The main components of the Bayesian optimization method are a Bayesian statistical model called *surrogate*, which approximates the objective function, and an *acquisition function*. The acquisition function defines a point in the search space where the objective function should be evaluated next. In the first step, values of the objective function are computed at a defined number of points in the search space according to some experimental design. The surrogate model then determines a Bayesian posterior probability distribution on the objective function $f(\boldsymbol{x})$. That is, estimates for the values of $f(\boldsymbol{x})$ at unobserved points in the search space with corresponding Bayesian credible intervals are computed. This posterior distribution gets updated after each new evaluation of the objective function. If the surrogate model is a Gaussian process, the prior distribution is assumed to be multivariate normal. The mean vector $\boldsymbol{\mu}_0(\boldsymbol{x}_{1:k})$ and the covariance matrix $\Sigma_0(\boldsymbol{x}_{1:k}, \boldsymbol{x}_{1:k})$ that describe the distribution are computed via a mean function and a kernel function, respectively. The mean function is evaluated at each observed point $\boldsymbol{x}_i$ of the sequence $\boldsymbol{x}_{1:k} = \boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_k$, while the kernel function is evaluated at each pair $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ of this sequence, whereby $i = \{1, ..., k\}$ and $j = \{1, ..., k\}$. The observations of the objective function are assumed to be drawn at random from the following distribution [61]:

$$f(\boldsymbol{x}_{1:k}) \sim \mathcal{N}(\boldsymbol{\mu}_0(\boldsymbol{x}_{1:k}), \Sigma_0(\boldsymbol{x}_{1:k}, \boldsymbol{x}_{1:k})). \tag{3.50}$$

The objective function is evaluated in the next run at the point in the search space that delivers an optimum of the acquisition function, which depends on the posterior distribution currently provided by the surrogate model [61]. Figure 3.8 shows two observations of the objective function and the obtained posterior distribution for an exemplary Bayesian optimization process.
The attempt is to find a good balance between exploration and exploitation. Exploration describes the process of evaluating the objective function in mostly unobserved regions of the search space and therefore prevents the optimizer from getting stuck in a local optimum. Exploitation involves the observation of points that are near the current optimum estimated by the surrogate model. Bayesian optimization takes the track record of previous observations into consideration when selecting a new hyperparameter configuration to test the network with. Hence, it usually approaches the optimum of the objective function a lot faster than the previously presented methods [58], [63].

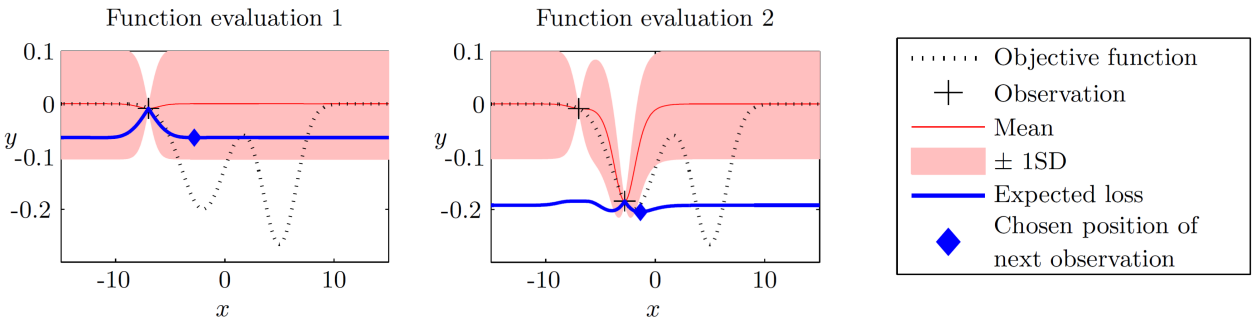


Fig. 3.8: Exemplary plots related to Bayesian optimization with a one-dimensional function input using a Gaussian process. The location where the objective function is evaluated first is chosen at random, as shown in the left plot. The point that delivers an optimum (in this case, a minimum) of the acquisition function (blue) is where the objective is observed next. After each observation, the posterior distribution is updated [61], [64]. Adapted from [64].

Bayesian optimization in the context of this work (see Section 8.2) was performed with the MATLAB function *bayesopt* [55] using the default settings. This function starts by testing four configurations of hyperparameters that are randomly drawn from uniform distributions defined by the corresponding domain boundaries. It then fits a Gaussian process regression model to the observations and determines a Bayesian posterior distribution. The acquisition function being used is referred to as *expected improvement*. It determines the point in the search space where the improvement in the observed function value, compared to the current best observation, is maximal [61]. Additionally, this function includes a procedure that prevents the algorithm from getting stuck at a local minimum of the objective function. That is, the kernel function of the acquisition function is modified if a particular criterion is met, which can be adapted by the user [55].

3.4.4 Genetic Algorithm

A genetic algorithm (GA) is a special type of metaheuristic capable of solving optimization problems with non-convex and non-continuous objective functions [58]. Same as most other metaheuristics, it mimics a biological process. More specifically, genetic algorithms are inspired by the theory of evolution and the concept of *survival of the fittest* [65], [66].
In the beginning, an initial population of possible candidate solutions, called *individuals*, is created [66]. Each individual consists of one or more chromosomes that, in turn, are characterized by a set of genes. These genes can be represented, e.g., by binary digits [67]. Evolutionary theory states that individuals, which have higher survival capabilities and better adapt to their environment than others, are more likely to pass their genes to the next generation. Therefore, the genetic algorithm requires a metric to evaluate the capabilities of each individual and identify promising candidates, the so-called *fitness function*. The fitness function is chosen based on the problem at

hand. For inheritance, a portion of the population is selected based on the fitness of the individuals, whereby candidates with a better fitness value have a higher probability of getting chosen. This operation, referred to as *selection*, leads to further optimization of well-performing individuals in the following generations, while candidates with low fitness gradually disappear [68].
The creation of new candidate solutions is performed by genetic operations called *crossover* and *mutation*. Via crossover, the individuals of the following generation inherit their characteristics from random individuals of the selected pool of promising candidates, representing their *parents*. Traditional crossover methods involve the exchange of a random portion of genes between two associated parents. The creation of child solutions by exchanging the bit sequences of the two parental individuals after a randomly selected element is referred to as *single-point crossover*. In *two-point crossover*, the portions of the parent's chromosomes between two random points in the sequence are exchanged to form the offspring. Mutation is performed by randomly altering one or more genes of an individual. In the case of a binary representation of the genes, this can be done by flipping bits, i.e., changing a 1 to a 0 and vice versa. The bits in the parent chromosome to be flipped are selected according to the bit sequence of a randomly generated mutation chromosome of the same length. If a bit in the mutation chromosome is a 1, it triggers a flipping of the bit at the corresponding position of the parent chromosome. Another mutation method exchanges two randomly selected bits in a chromosome. The crossover and mutation operations allow the algorithm to approach a possible optimum of the fitness function. Yet, they ensure a sufficient degree of diversity is maintained in the population in order to explore further regions of the search space [65], [66]. The algorithm is executed until the maximum number of generations is reached, or another termination criterion is met [66].
The genetic algorithm used for the test series presented in this thesis was written at the Chair of Automation [15]. The fitness function is based on the error between the original signal and the corresponding reconstruction obtained by an autoencoder (see Chapter 5). More precisely, it is determined by summing up the absolute errors between the reconstructed and the original signal at each time step in all channels of all samples of an MVTS data set. The three individuals with the best fitness are passed to the next generation without being modified via crossover operations. The mating pool consists of individuals with better fitness than the median fitness in the current population. Unfit candidates are added by a specified small probability. Crossover is performed on two randomly chosen parental individuals from the mating pool via one of two possible operations selected at random. In the first variant, each chromosome of the child is inherited from a randomly chosen parent. Alternatively, the child's chromosomes are determined by computing the mean of the corresponding parental chromosomes. The value is then rounded to the next integer. Note that the learning rate is defined in units of 10^{-5}. Eventually, the individuals are mutated by a predefined probability. That is, the value of the chromosome to be mutated is replaced by a random value from the corresponding hyperparameter domain.

Chapter 4
Basics of Probability and Information Theory

Probability and information theory are fundamental tools for machine learning and computer science in general. Although a machine learning program is often considered a black box that does not require the user to have a deeper knowledge of the processes and algorithms involved, a basic understanding of the theory is beneficial for interpreting results and optimizing the model [22]. The purpose of this chapter is to provide a brief overview of some essential basics of probability and information theory, which help to understand the theoretical background associated with the machine learning methods addressed in this work.

4.1 Probability Theory

Probability theory provides a mathematical framework for expressing the uncertainty or plausibility of statements or events [22], [69], where an event represents the result of a random experiment [69]. In artificial intelligence, it is an important tool for deriving the mathematical foundations of various machine learning algorithms or evaluating and analyzing their behavior [22]. The focus of this section will be on the fundamentals of probability theory that are applied in neural networks, particularly in autoencoders.

4.1.1 Probability Distributions

In statistics, the result or assumed outcome of a random experiment is described by a *random variable* X. This random variable is associated with a probability distribution that indicates how likely X is to take on each of its possible values. Generally, two types of probability distributions are distinguished: discrete and continuous distributions. If X is sampled from a discrete distribution, it only can take on certain distinct values $x_i \in \mathbb{R}$, with $i \in \{1, ..., m\}$ [69].
In this case, the distribution is described by a so-called *probability mass function*. This function maps the value or state of the random variable to the probability of it being the outcome of a ran-

dom experiment, i.e., the probability $P(X = x_i)$ of X taking on the value or state x_i. A probability mass function has to fulfill certain criteria [22]:

- The domain of the probability mass function has to be the set of all possible values or states $\{x_1, ..., x_m\}$ the random variable X can take on [22].
- $\forall x_i \in X : 0 \leq P(x_i) \leq 1$, i.e., the probability of the random variable X taking on the value or state x_i has to lie in the interval [0,1], where $P(x_i) = 0$ represents an impossible outcome and $P(x_i) = 1$ a certain outcome [22].
- $\sum_{x_i \in X} P(x_i) = 1$. The probabilities of all possible events must add up to one. Therefore, a mass function is referred to as being *normalized* [22].

In the following, some of the most common discrete distributions are presented:

Discrete Uniform Distribution

A random variable X that can take on m different states x_i with $i = \{1, ..., m\}$, which are equally likely, is said to have a discrete uniform distribution. The corresponding probability mass function can be defined by [22]:

$$P(X = x_i) = \frac{1}{m}. \tag{4.1}$$

Bernoulli Distribution

The Bernoulli distribution is a distribution over a random variable X that can take on two possible states $x_i \in \{0, 1\}$, where the probability of $X = 1$ is denoted as $P(X = 1) = \phi$. Consequently, the state $X = 0$ has a probability of $P(X = 0) = 1 - \phi$. The probability mass function can be formulated as follows [22]:

$$P(X = x_i) = \phi^{x_i}(1 - \phi)^{1 - x_i}. \tag{4.2}$$

Binomial Distribution

The binomial distribution describes the probability of k successes in n independent random experiments. The probability of success is denoted as ϕ, and the probability of failure, therefore, is equal to $1 - \phi$. The corresponding probability mass function is defined by [69]:

$$P(k; n, \phi) = \binom{n}{k} \phi^k (1 - \phi)^{n-k} \text{ with } k \in \{0, 1, ..., n\}. \tag{4.3}$$

The term $\phi^k(1 - \phi)^{n-k}$ can be viewed as the probability of the first k experiments being successes and the following $n - k$ experiments being failures. However, the sequential order in which the successes and failures occur is not restricted. That is, altogether $\binom{n}{k}$ combinations of outcomes of the probability $\phi^k(1 - \phi)^{n-k}$ are possible, given the number of successes is equal to k [69].

If X is a continuous random variable, it can take on infinitely many values $x \in \mathbb{R}$, typically in the range $-\infty \leq x \leq \infty$ [69]. The probability distribution associated with X is described by a *probability density function* $p(x)$. Analogous to the mass function, it has to obey some rules [22]:

- The domain of the probability density function has to be the set of all possible values or states the random variable X can take on [22].
- $\forall x \in X : 0 \leq p(x)$, i.e., the probability of the random variable X taking on the value or state x has to be non-negative [22].
- $\int p(x)\mathrm{d}x = 1$, i.e., the probability density function is normalized [22].

Some continuous distributions that are relevant for the field of machine learning are shown in the following:

Continuous Uniform Distribution

The continuous uniform distribution is fully described by two parameters, a and b, representing the boundary values for the interval in which the density function is $p(x) > 0$. The term *uniform* refers to the fact that the density function is constant in the interval [a,b]. The continuous uniform distribution is defined by [69]:

$$p(x;a,b) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b \\ 0 & \text{otherwise .} \end{cases} \tag{4.4}$$

Gaussian Distribution

The Gaussian or *normal* distribution is the most commonly used distribution to describe real numbers. It is characterized by a symmetric graph, often referred to as *bell curve*, which is centered around the mean μ. In science, normal distributions are the most popular default choice for independent real numbers if there is a lack of prior knowledge about their real distribution [22].
One reason for that is the *central limit theorem*, which states that the sum of n independent variables with arbitrary distributions moves asymptotically towards a normal distribution for $n \to \infty$, given some very general conditions. This theorem is particularly important for cases where the random variable X takes on values from a measurement, that is influenced by various causes in the instrument and the atmosphere. X can then be assumed to originate from a variety of independent random variables with different distributions. The parameters that fully describe the normal distribution, the mean μ and the variance σ^2, can be easily approximated from the measurement data. This is another reason for the frequent use of the Gaussian distribution [69].
In the one-dimensional case, the density function of the normal distribution is defined by [22]:

$$p(x;\mu,\sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{4.5}$$

The generalized version of the Gaussian distribution for higher dimensional data $\boldsymbol{x} \in \mathbb{R}^n$ is referred to as *multivariate normal distribution*. With a mean vector $\boldsymbol{\mu}$ and a covariance matrix Σ, it is defined as follows [22]:

$$p(\boldsymbol{x};\boldsymbol{\mu},\Sigma) = \sqrt{\frac{1}{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^{\mathrm{T}}\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right). \tag{4.6}$$

Exponential and Laplace Distributions

In certain tasks in machine learning, it is required to use a density function that exhibits a sharp point at the origin [22]. The exponential distribution is a possible choice for this type of problems [69]:

$$p(x;\gamma) = \begin{cases} \frac{1}{\gamma}\exp(-\frac{x}{\gamma}) & \text{for } x \geq 0 \\ 0 & \text{otherwise .} \end{cases} \tag{4.7}$$

The Laplace distribution is closely related to the exponential distribution. It is described by the parameter μ that defines the location of the peak, and a parameter γ [22]:

$$p(x;\mu,\gamma) = \frac{1}{2\gamma}\exp\left(-\frac{|x-\mu|}{\gamma}\right). \tag{4.8}$$

4.1.2 Marginal Probability

A *joint probability distribution* $P(X^{(1)},X^{(2)},...,X^{(n)})$ is a probability distribution that is defined over a set $\mathcal{D} = \{X^{(1)},X^{(2)},...,X^{(n)}\}$ of n random variables. Sometimes it is required to compute the probability distribution over a subset of these variables, which is referred to as *marginal probability distribution*. The k_i possible values a random variable $X^{(i)} \in \mathcal{D}$ can take on are denoted as x_{i,m_i}, where $m_i = \{1,\ldots,k_i\}$. In order to obtain the marginal distribution over one discrete variable $X^{(i)}$, a summation must be performed over all possible values of each random variable in $\mathcal{D}$ except $X^{(i)}$ [22], [70]:

$$\forall x_{i,m_i} \in X^{(i)} : P(X^{(i)} = x_{i,m_i}) = \sum_{m_1=1}^{k_1} \cdots \sum_{m_{i-1}=1}^{k_{i-1}} \sum_{m_{i+1}=1}^{k_{i+1}} \cdots \sum_{m_n=1}^{k_n} P(x_{1,m_1},\ldots,x_{i,m_i},\ldots,x_{n,m_n}). \tag{4.9}$$

For a set $\mathcal{C} = \{X^{(1)},X^{(2)},...,X^{(n)}\}$ of continuous random variables with the corresponding joint probability distribution $p(x_1,x_2,...,x_n)$, the marginal distribution $p(x_i)$ over a variable $X^{(i)} \in \mathcal{C}$ can be determined as follows [22], [70]:

$$p(x_i) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} p(x_1,...,x_i,...,x_n)\, \mathrm{d}x_1 \ldots \mathrm{d}x_{i-1}\mathrm{d}x_{i+1} \ldots \mathrm{d}x_n. \tag{4.10}$$

4.1.3 Conditional Probability and Chain Rule

The conditional probability $P(A|B)$ expresses the probability of an event A, given another event B has happened. Take, e.g., the conditional probability $P(Y = y_i|X = x_i)$ of the random variable Y taking on the value y_i, given the random variable X is equal to some value x_i. The corresponding formula to compute this conditional probability is [22]:

$$P(Y = y_i|X = x_i) = \frac{P(Y = y_i, X = x_i)}{P(X = x_i)}. \tag{4.11}$$

The background of Equation (4.11) will be further discussed in Section 4.1.4. An important rule that follows directly from this equation is the so-called *chain rule*. For that, Equation (4.11) is rearranged and extended in order to be used for n random variables $\mathcal{X} = \{X^{(1)}, ..., X^{(n)}\}$. The joint probability distribution $P(X^{(1)}, ..., X^{(n)})$ over these random variables can then be expressed as a product of conditional probability distributions, e.g., as shown in the following equation [22]:

$$P(X^{(1)}, ..., X^{(n)}) = P(X^{(1)}) \prod_{i=2}^{n} P(X^{(i)}|X^{(1)}, ..., X^{(i-1)}). \tag{4.12}$$

4.1.4 Bayes' Theorem

Bayes' Theorem is a famous equation of statistics that describes the probability of an event A given an event B [22]:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{4.13}$$

This theorem can be used for testing hypotheses. Suppose a *null hypothesis* H_0 and an *alternative hypothesis* H_1 are given [69]. The probability of the null hypothesis, given some observed data $\mathcal{D}$, can then be determined by [71]:

$$P(H_0|\mathcal{D}) = \frac{P(H_0)P(\mathcal{D}|H_0)}{P(\mathcal{D})}, \tag{4.14}$$

with following probabilities [71]:

- $P(H_0|\mathcal{D})$: Conditional probability, also called the *posterior* probability. It indicates how confidently the null hypothesis H_0 can be stated to be true, given some data $\mathcal{D}$ that was observed [71].
- $P(H_0)$: *Prior* probability. It is an estimate of the probability of the null hypothesis H_0 prior to observing data $\mathcal{D}$ [71].
- $P(\mathcal{D}|H_0)$: *Likelihood*, i.e., the probability of observing data $\mathcal{D}$ assuming the null hypothesis H_0 is true [71].

- $P(\mathcal{D})$: *Evidence* of the data, also referred to as *marginal likelihood*. It is the probability of observing data $\mathcal{D}$, irrespective of the hypothesis H_0 being true or not. The evidence can be obtained as follows [71]:

$$P(\mathcal{D}) = P(H_0)P(\mathcal{D}|H_0) + P(H_1)P(\mathcal{D}|H_1). \tag{4.15}$$

The posterior $P(H_1|\mathcal{D})$ of the alternative hypothesis H_1 given data $\mathcal{D}$ can be determined analogously [71].
A related task that can be dealt with using Bayes' Theorem is the estimation of model parameters. By replacing the hypothesis in Equation (4.14) with a parameter vector θ, the corresponding posterior probability $P(\theta|\mathcal{D})$ can be obtained as follows [72]:

$$P(\theta|\mathcal{D}) = \frac{P(\theta)P(\mathcal{D}|\theta)}{P(\mathcal{D})}. \tag{4.16}$$

The aim is to find the parameter vector θ of the parameter space Θ with the highest posterior probability $P(\theta|\mathcal{D})$. $P(\theta)$ represents the prior probability of the parameter vector θ, irrespective of the observed data $\mathcal{D}$, and $P(\mathcal{D}|\theta)$ the probability of the data $\mathcal{D}$ being generated by the parameters θ. Since $P(\mathcal{D})$ only acts as a normalizing constant, it is sufficient to consider the following relation [72]:

$$P(\theta|\mathcal{D}) \propto P(\theta)P(\mathcal{D}|\theta). \tag{4.17}$$

4.2 Information Theory

Information theory is a subarea of applied mathematics that deals with the quantification of the information content in signals. In its early years, the primary purpose was to provide design guidelines for the encoding schemes of messages. The encoding and the length of a message were intended to depend on the probability of the corresponding event it informed about, i.e., the amount of information that was gained by receiving that message. The basic intuition behind this is that a message about an event that is likely to happen provides less information than a message about the occurrence of a more improbable event. Suppose a discrete random variable X can take on the values x_i of a set $\mathcal{M} = \{x_1, x_2, ..., x_n\}$ by a certain probability $P(x_i)$. For the quantification of the information content, a metric called *self-information* $I(x_i)$ of an event x_i is defined [22]:

$$I(x_i) = -\log P(x_i). \tag{4.18}$$

If the natural logarithm is used, $I(x_i)$ is given in units of *nats*. One *nat* represents the information gain when being informed about an event of probability $P(x_i) = \mathrm{e}^{-1}$. In computer science and machine learning, this concept plays an important role, even though the interpretation of information in the context of messages or signals does not always apply [22]. The average information $H(X)$,

associated with the set $\mathcal{M}$ of possible values x_i the random variable X can take on, is referred to as *Shannon entropy*. It is defined by [73]:

$$H(X) = \sum_{i=1}^{n} P(x_i) I(x_i) = -\sum_{i=1}^{n} P(x_i) \log P(x_i). \tag{4.19}$$

In classical information theory, the Shannon entropy represents the lower bound for the average number of bits that are needed to encode messages informing about possible events. In this case, the logarithm with base 2 has to be used in Equation (4.19). For continuous random variables X with a probability density function $p(x)$, the Shannon entropy has the following form [22]:

$$H(X) = \mathbb{E}_{X\sim p}[I(x)] = -\mathbb{E}_{X\sim p}[\log p(x)], \tag{4.20}$$

whereby $H(X)$ is also referred to as *differential entropy*. $H(X)$ expresses the expected information gain when sampling x from the distribution $p(x)$ [22]. Another important metric of information theory is the *Kullback-Leibler (KL) divergence* $KL[p(x)||q(x)]$ [74], which quantifies the similarity of two probability distributions $p(x)$ and $q(x)$ [22], [75]. The KL divergence is non-negative, whereby $KL[p(x)||q(x)] = 0$ if $p(x) = q(x)$. The value of the KL divergence increases as the two distributions diverge. The Kullback-Leibler divergence does not fulfill all criteria that are required to be a distance measure, as it does not satisfy the triangular inequality and is not symmetric, i.e., $KL[p(x)||q(x)] \neq KL[q(x)||p(x)]$ [76]. It can be computed as follows [22]:

$$KL[p(x)||q(x)] = \mathbb{E}_{X\sim p}\left[\log \frac{p(x)}{q(x)}\right] = \int p(x) \log \frac{p(x)}{q(x)} \mathrm{d}x. \tag{4.21}$$

With the definition of a closely related quantity referred to as *cross-entropy* [22],

$$H[p(x), q(x)] = -\mathbb{E}_{X\sim p} \log q(x), \tag{4.22}$$

the Kullback-Leibler divergence can be written as follows [22]:

$$KL[p(x)||q(x)] = H[p(x), q(x)] - H[p(x)]. \tag{4.23}$$

The KL divergence can also be formulated with probability mass functions $P(x_i)$ and $Q(x_i)$ associated with distinct values x_i of a set $\mathcal{D}$ [76]:

$$KL(P||Q) = \sum_{x_i \in \mathcal{D}} P(x_i) \log \frac{P(x_i)}{Q(x_i)}. \tag{4.24}$$

Chapter 5
Autoencoders

Autoencoders were first described by Rumelhart et al. [10] in 1986, although the term *autoencoder* was not introduced in this work. The authors used a simple neural network to compute a $\log_2 N$-bit pattern from an N-bit input pattern, on which the original input should be well reconstructible.
In a more generalized manner, the term autoencoder today refers to a neural network with a specific type-dependent architecture consisting of two main parts: the encoder and the decoder. The encoder $E : \boldsymbol{x} \rightarrow \boldsymbol{z}$ learns a mapping of the input $\boldsymbol{x}$ to a hidden representation $\boldsymbol{z}$. The decoder $D : \boldsymbol{z} \rightarrow \hat{\boldsymbol{x}}$ computes a reconstruction $\hat{\boldsymbol{x}}$ of the input from this hidden encoding. Autoencoders are trained in an unsupervised manner attempting to minimize a loss function $\mathcal{L}(\boldsymbol{x}, \hat{\boldsymbol{x}})$ that is based on the error between the original and reconstructed input, $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$, respectively. In order to prevent autoencoders from learning just to copy the input, certain restrictions depending on the specific variant are made. This forces the autoencoders to extract the relevant information from the input data, which is why they are often used for nonlinear dimensionality reduction or feature extraction [22]. A variant called *variational autoencoder* [77] can also be used as a generative model, as described in Section 5.1.4. In the following, some common autoencoder variants will be presented.

5.1 Types of Autoencoders

5.1.1 Undercomplete Autoencoders

A simple way to prevent the autoencoder from just learning to copy the input is forcing a dimensionality reduction. That is, the encoder maps the input $\boldsymbol{x} \in \mathbb{R}^m$ to a hidden representation $\boldsymbol{z} \in \mathbb{R}^p$, where $p < m$ (see Figure 5.1). Thus, an *undercomplete autoencoder (AE)* learns to compute a representation of the input data with a lower dimension while preserving the relevant information [22]. The simplest form of an undercomplete autoencoder contains only an input layer, a hidden layer that outputs a compressed version of the input, and an output layer. The hidden encoding is then defined by [24]:

$$\boldsymbol{z} = \boldsymbol{\varphi}^1(\mathsf{W}^1\boldsymbol{x} + \boldsymbol{b}^1). \tag{5.1}$$

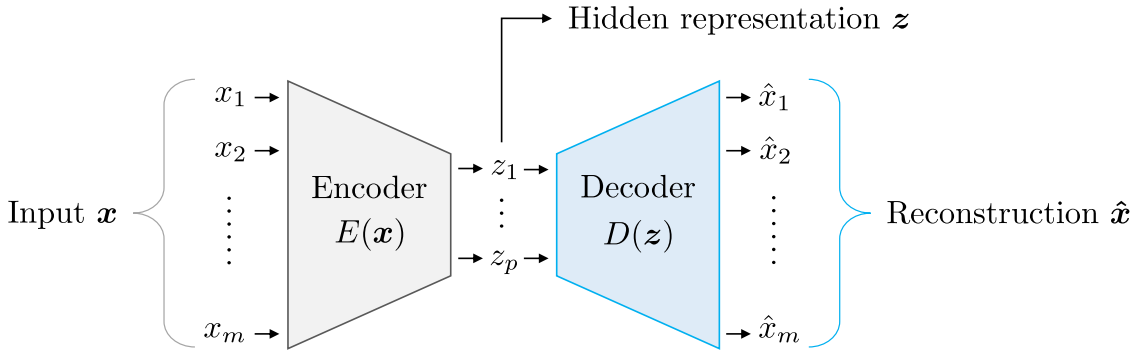


Fig. 5.1: Schematic diagram of an undercomplete autoencoder that compresses an input vector $\boldsymbol{x}$ to a hidden representation $\boldsymbol{z}$ and computes a reconstruction $\hat{\boldsymbol{x}}$ of the signal from this hidden encoding [22]. Adapted from [78].

Via the decoder, a reconstruction $\hat{\boldsymbol{x}}$ of the original signal from the hidden representation $\boldsymbol{z}$ is computed as follows [24]:

$$\hat{\boldsymbol{x}} = \varphi^2(\mathrm{W}^2\boldsymbol{z} + \boldsymbol{b}^2). \tag{5.2}$$

W^1, $\boldsymbol{b}^1$, W^2, and $\boldsymbol{b}^2$ represent the weight matrices and bias vectors of the encoder and the decoder, respectively. The activation functions of layer 1 and 2 are respectively denoted as φ^1 and φ^2 [24]. When using the sum-of-squares error as the loss function, this primitive form of an autoencoder projects the input data to the p-dimensional subspace that is spanned by the first p principal components [79], [80]. This also holds if a nonlinear activation function is used in the neurons of the hidden layer [80], [81]. The principal components are a set of basis vectors defined sequentially so that each new basis vector is orthogonal to the existing ones while minimizing the squared distance to the data points [81]. The corresponding method to determine these vectors is referred to as *principal component analysis (PCA)* [82]. Even though the simple autoencoder projects the data to the same subspace as a PCA, the weight vectors that span this subspace, i.e., the row vectors of the weight matrix, are generally neither orthogonal nor normalized [81]. However, adding hidden layers with nonlinear activation functions to the encoder and the decoder enables the network to perform a nonlinear dimensionality reduction. This allows autoencoders to compute a more powerful compression of the input data than a standard linear PCA [83].

5.1.2 Sparse Autoencoders

Another way of preventing the autoencoder from learning the identity function, even if the dimension of the hidden representation is higher than the dimension of the input vector, is to impose a *sparsity constraint* on the neurons in the hidden layers [84]. The first approaches to applying sparsity penalties in neural networks were presented in [85] and [86]. In sparse autoencoders, the hidden neurons are forced to be inactive (low activation) most of the time, and only a small frac-

tion is activated in each run [84]. This allows the network to find interesting structures in the input data [84] and revive neurons with low average activation [86]. In the following, the activation functions of the neurons in the hidden layers are assumed to be sigmoid functions, but other function types could also be used. As described in Section 3.2.1, $a_k^i(\boldsymbol{x}^{(m)}) \in [0,1]$ denotes the activation of node k in the hidden network layer i, if the network is fed with the input vector $\boldsymbol{x}^{(m)}$. Furthermore, $i = \{1,\dots,h\}$ is defined for a network with h hidden layers. The average activation $\hat{\rho}_k^i$ of node k in layer i over the whole training set $\mathcal{X} = \{\boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(p)}\}$ is defined by [84]:

$$\hat{\rho}_k^i = \frac{1}{p}\sum_{m=1}^{p}[a_k^i(\boldsymbol{x}^{(m)})]. \tag{5.3}$$

The aim is to force the value of the average activation $\hat{\rho}_k^i$ of each neuron k to be as near as possible to a given target value ρ, the *sparsity parameter*. This constraint value ρ is usually chosen to be close to zero in order to keep the neurons inactivated most of the time. A Bernoulli random variable X with a mass function P (see Section 4.1.1) is introduced. $P(X=1) = \rho$ and $P(X=0) = 1-\rho$ respectively denote the predefined target probabilities of a neuron getting activated or not. For each neuron k in a hidden layer i, a Bernoulli random variable Y_k^i is defined. The mass function Q_k^i with the associated probabilities $Q_k^i(Y_k^i = 1) = \hat{\rho}_k^i$ and $Q_k^i(Y_k^i = 0) = 1-\hat{\rho}_k^i$ expresses the actual probabilities of neuron k in layer i being activated or not, respectively [84]. For the task of forcing the probabilities $\hat{\rho}_k^i$ to be as near as possible to the sparsity parameter ρ, the cross entropies [86] or KL divergences [84] of the probability distributions defined by P and Q_k^i are added to the cost function. Using the KL divergence to enforce sparsity, the cost function has the following form [84]:

$$\mathcal{L}(\boldsymbol{x},\hat{\boldsymbol{x}}) + \beta\sum_{i=1}^{h}\sum_{k=1}^{n_i} KL(P||Q_k^i), \tag{5.4}$$

where n_i is the number of neurons in the hidden layer i and β a weighting factor. The KL divergence for Bernoulli distributions is formulated as follows [84]:

$$KL(P||Q_k^i) = \rho\log\frac{\rho}{\hat{\rho}_k^i} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_k^i}. \tag{5.5}$$

It is non-negative and $KL(P||Q_k^i) = 0$ if $\hat{\rho}_k^i = \rho$. It increases the more $\hat{\rho}_k^i$ is diverging from ρ [76]. Thus, minimizing the loss function and hence the Kullback-Leibler divergences enforces the average activations $\hat{\rho}_k^i$ to converge to ρ [84]. A more detailed explanation of the KL divergence is given in Section 4.2.

5.1.3 Denoising Autoencoders

A *denoising autoencoder* is trained to reconstruct the original input data $\boldsymbol{x}$ from a corrupted version $\tilde{\boldsymbol{x}}$ of it. Thus, the network is prevented from learning just to copy the input and is trained to find the underlying relevant information in noisy data [87]. Most commonly, corruption is carried out by one of the following methods [24]:

- *Binary noise:* Set randomly chosen elements of the input data to zero.
- *Gaussian noise:* Add a number of random Gaussian values to the input data.

The encoder receives the corrupted vector $\tilde{\boldsymbol{x}}$ and computes a hidden encoding $\boldsymbol{z} = E(\tilde{\boldsymbol{x}})$. From this hidden representation, the decoder aims to reconstruct the original signal $\boldsymbol{x}$ rather than the distorted version $\tilde{\boldsymbol{x}}$. Thus, the autoencoder is trained to minimize a cost function $\mathcal{L}(\boldsymbol{x}, \hat{\boldsymbol{x}})$. This function measures the error between the original data $\boldsymbol{x}$ and the reconstruction $\hat{\boldsymbol{x}}$ obtained from the hidden representation of the corrupted input $\tilde{\boldsymbol{x}}$ [22]:

$$\mathcal{L}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \mathcal{L}\left[\boldsymbol{x}, D(E(\tilde{\boldsymbol{x}}))\right]. \tag{5.6}$$

5.1.4 Variational Autoencoders

Although it is classified as a type of autoencoder, the mathematical basis of a *variational autoencoder (VAE)* is fundamentally different from other autoencoder types, like the ones presented before [88]. Instead of just mapping the input vector to data points in the latent space, the VAE maps the input to parameters of a latent distribution. Via a regularization term in the cost function, this distribution is forced to be as close as possible to a predefined target distribution [77]. The aim is to enforce a continuous and complete latent space [89]. This enables the variational autoencoder to generate new data that is similar to the data of the training set. A variational autoencoder consists of two main parts: the inference model $q_{\Phi}(\boldsymbol{z}|\boldsymbol{x})$, which is the *stochastic encoder*, and the generative model $p_{\theta}(\boldsymbol{x}|\boldsymbol{z})$, the *stochastic decoder* [77]. Figure 5.2 shows a schematic diagram of a VAE. In the following, the mathematical foundations of the variational autoencoder are presented.

5.1.4.1 Variational Inference

Any vector of observed variables $\boldsymbol{x}$ that is fed into the network is a random sample drawn from an unknown true probability distribution $p^*(\boldsymbol{x})$. In order to approximate this distribution, a model $p_{\theta}(\boldsymbol{x})$ with parameters $\boldsymbol{\theta}$ is used [77]:

$$\boldsymbol{x} \sim p_{\theta}(\boldsymbol{x}). \tag{5.7}$$

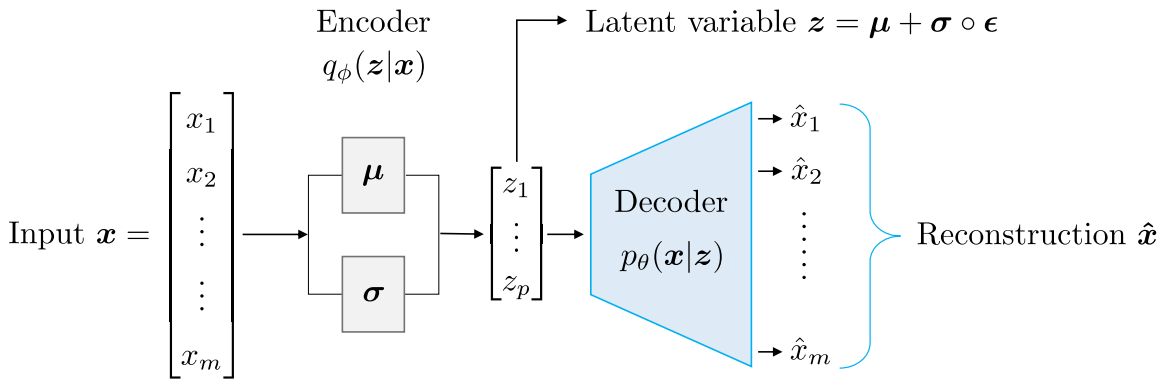


Fig. 5.2: Schematic diagram of a variational autoencoder. The encoder maps an input signal $\boldsymbol{x}$ to a vector of means $\boldsymbol{\mu}$ and a vector of standard deviations $\boldsymbol{\sigma}$ of a latent distribution. The decoder computes a reconstruction $\hat{\boldsymbol{x}}$ from the latent variable $\boldsymbol{z}$, which is sampled from this latent distribution [77]. Adapted from [78].

The aim of learning is to determine the parameters θ such that the model $p_\theta(\boldsymbol{x})$ is a good approximation for the real distribution $p^*(\boldsymbol{x})$ for any sampled variable $\boldsymbol{x}$ [77]:

$$p_\theta(\boldsymbol{x}) \approx p^*(\boldsymbol{x}). \tag{5.8}$$

Variational autoencoders are a type of *Deep Latent Variable Models (DLVM)* [77]. These models explain the received input data $\boldsymbol{x}$ using a hidden representation $\boldsymbol{z}$ of it [90]. The unobserved vector $\boldsymbol{z}$ is referred to as *latent variable*, which is not part of the data but part of the model. The conditional distribution $p^*(\boldsymbol{z}|\boldsymbol{x})$ represents the distribution over the latent variable $\boldsymbol{z}$, conditioned on the observations $\boldsymbol{x}$. This distribution again can be approximated by a model $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ with parameters θ. Using Bayes' Theorem, $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ can be defined by [77]:

$$p_\theta(\boldsymbol{z}|\boldsymbol{x}) = \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{x})} = \frac{p_\theta(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})}{p_\theta(\boldsymbol{x})}. \tag{5.9}$$

The marginal probability of data under the model, i.e., the evidence, can be computed by [77]:

$$p_\theta(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})d\boldsymbol{z}. \tag{5.10}$$

This integral can be high-dimensional and computationally too expensive to evaluate [90]. Since the probabilities $p_\theta(\boldsymbol{x})$ and $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ are related via Bayes' Theorem in Equation (5.9), $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ is also intractable. In order to solve this problem, an inference model $q_\Phi(\boldsymbol{z}|\boldsymbol{x})$ with the *variational parameters* Φ is introduced. If a neural network is used, these parameters are its weights and biases. The aim is now to optimize the parameters Φ such that $q_\Phi(\boldsymbol{z}|\boldsymbol{x})$ is a good approximation for the posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ [77]:

$$q_\Phi(\boldsymbol{z}|\boldsymbol{x}) \approx p_\theta(\boldsymbol{z}|\boldsymbol{x}). \tag{5.11}$$

Hence, instead of computing integrals to determine the evidence, an optimization problem is solved [91]. As the joint probability distribution $p_\theta(\boldsymbol{x},\boldsymbol{z})$ is tractable, the evidence $p_\theta(\boldsymbol{x})$ is now also computable via Bayes' Theorem using the approximation for the posterior, see Equation (5.9). This concept of approximating an intractable posterior is referred to as *variational inference* [77].

5.1.4.2 ELBO Loss

The loss function of the VAE is referred to as *Evidence Lower Bound (ELBO)*, sometimes called *Variational Lower Bound*. It contains a regularization term in addition to the reconstruction error, the Kullback-Leibler divergence $KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})]$. The KL divergence quantifies how accurately the inference model $q_\Phi(\boldsymbol{z}|\boldsymbol{x})$ approximates the posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ [77]. As explained in Section 4.2, $KL = 0$ if the compared distributions are equivalent, and it increases with the distributions diverging from each other [76]. Since $q_\Phi(\boldsymbol{z}|\boldsymbol{x})$ should be a good approximation of the posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$, the objective is to minimize the KL divergence, which is defined by [77]:

$$KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})] = \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right]. \tag{5.12}$$

Rewriting Equation (5.12) yields [77], [91]:

$$\begin{aligned}
\mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] &= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})p_\theta(\boldsymbol{x})}{p_\theta(\boldsymbol{x},\boldsymbol{z})}\right] \\
&= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{x},\boldsymbol{z})}\right] + \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x})\right] \\
&= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{x},\boldsymbol{z})}\right] + \log p_\theta(\boldsymbol{x}).
\end{aligned} \tag{5.13}$$

By rearranging this equation, the log-likelihood of data $\log p_\theta(\boldsymbol{x})$ is given as the sum of the ELBO loss $\mathcal{L}_{\theta,\Phi}(\boldsymbol{x})$ and the KL divergence $KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})]$ [77]:

$$\begin{aligned}
\log p_\theta(\boldsymbol{x}) &= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\right] + \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathcal{L}_{\theta,\Phi}(\boldsymbol{x}) + KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})].
\end{aligned} \tag{5.14}$$

The goal is to maximize the marginal likelihood $p_\theta(\boldsymbol{x})$ in order to improve the generative model [77]. Equation (5.14) indicates that this can be achieved by maximizing the ELBO loss, which in turn minimizes the KL divergence $KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})]$ and therefore also enforces a better approximation $q_\Phi(\boldsymbol{z}|\boldsymbol{x})$ of the posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$ [91]. As the KL divergence is non-negative, i.e., $KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})] \geq 0$, it can be stated that the ELBO loss is the lower bound for the log-likelihood of the data [77], see Equation (5.15). That is where the name *Evidence Lower Bound* stems from [91]:

$$\log p_\theta(\boldsymbol{x}) \geq \mathcal{L}_{\theta,\Phi}(\boldsymbol{x}). \tag{5.15}$$

To get a deeper insight into the ELBO loss, Equation (5.14) is rewritten as follows [91]:

$$\begin{aligned}
\mathcal{L}_{\theta,\Phi}(\boldsymbol{x}) &= \log p_\theta(\boldsymbol{x}) - KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})] \\
&= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] - \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] - \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})\right] - \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z})}\right] \\
&= \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})\right] - KL[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z})]. \qquad (5.16)
\end{aligned}$$

The first term on the right-hand side (RHS) represents the expected log-likelihood of the data. The second term on the RHS forces $q_\Phi(\boldsymbol{z}|\boldsymbol{x})$ to be as close as possible to a prior $p(\boldsymbol{z})$ [91]. The prior distribution $p(\boldsymbol{z})$ can be chosen freely. Usually, a simple distribution is used for this task, e.g., a multivariate Gaussian distribution with a diagonal covariance structure [88].
The two terms of the loss function have contrasting effects. Minimizing the reconstruction error for different samples tends to increase the distance between the corresponding representations in the latent space to make their embeddings distinguishable. The regularization term of the ELBO loss, on the other hand, aims to create a continuous latent space to facilitate the generation process. This may lead to an overlapping of the latent variables and hence a noisy encoding. Thus, an important subject of current research is finding the right balance between these terms via the use of a weighting factor λ [92], [93], [94]:

$$\mathcal{L}_{\theta,\Phi}(\boldsymbol{x}) = \mathbb{E}_{q_\Phi(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})\right] - \lambda \mathrm{KL}[q_\Phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z})]. \qquad (5.17)$$

5.1.4.3 Computation

Variational autoencoders can be trained via mini-batch stochastic gradient descent or extensions of it, e.g., the *Adam* or *Adamax* optimization algorithm. Gradients of the individual-datapoint ELBO w.r.t. the parameters θ of the generative model $\nabla_\theta \mathcal{L}_{\theta,\Phi}(\boldsymbol{x})$ are easy to obtain. The computation of gradients w.r.t. the variational parameters $\nabla_\Phi \mathcal{L}_{\theta,\Phi}(\boldsymbol{x})$, however, is not feasible without the use of a so-called *reparameterization trick*. In this case, $\boldsymbol{z}$ is expressed as a differentiable transformation of a random variable $\boldsymbol{\epsilon}$ [77],

$$\boldsymbol{z} = \boldsymbol{g}(\boldsymbol{\epsilon}, \Phi, \boldsymbol{x}), \qquad (5.18)$$

where $\boldsymbol{x}$ denotes the input variable and Φ the parameters of the inference model. The dimension of $\boldsymbol{\epsilon}$ is equal to the latent dimension p, and the associated distribution is independent of $\boldsymbol{x}$ and Φ [77]. A widely used approach is a Gaussian encoder [88]:

$$q_\Phi(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}, \Sigma), \qquad (5.19)$$

where $\boldsymbol{\mu} \in \mathbb{R}^p$ and $\Sigma \in \mathbb{R}^{p \times p}$ respectively represent the vector of means μ_i and the covariance matrix. Typically, only the diagonal elements of the covariance matrix, i.e., the variances, are computed. The non-diagonal entries are set to zero. Hence, the covariance matrix has the following form [77]:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_p^2 \end{bmatrix}. \tag{5.20}$$

The parameters in $\boldsymbol{\mu}$ and Σ are obtained using a neural network that represents the encoder E [77]:

$$[\boldsymbol{\mu}, \log(\operatorname{diag} \Sigma)] = E(\boldsymbol{x}). \tag{5.21}$$

The process of sampling $\boldsymbol{z}$ using the reparameterization trick can be formulated as follows [77]:

$$\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \circ \boldsymbol{\epsilon} \text{ with } \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathrm{I}). \tag{5.22}$$

The vector $\boldsymbol{\sigma} \in \mathbb{R}^p$ of standard deviations σ_i is defined by [77]:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_1, \sigma_2, \dots, \sigma_p \end{bmatrix}^{\mathrm{T}}. \tag{5.23}$$

Reparameterization allows applying the backpropagation algorithm (see Section 3.3.2) to train the VAE [77]. For a detailed description of the training process of VAEs, refer to [77].

5.2 Encoder-Decoder Sequence-To-Sequence Architectures

Sequence-to-sequence (seq2seq) models, as introduced in [95] and [96] in 2014, generate an output sequence Y from an input sequence X without the restriction of both having the same length.
A possible task that requires this type of capability is machine translation. A sequence of words in one language may have a different length than the corresponding sequence in the language it should be translated into. Sequence-to-sequence architectures typically consist of two recurrent neural networks representing the encoder (reader) and the decoder (writer). The encoder receives an input sequence $X = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(m)}\}$ and outputs a typically fixed-size *context vector* C. The decoder receives the vector C and generates an output sequence $Y = \{\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, ..., \boldsymbol{y}^{(n)}\}$. The learnable parameters of the encoder and the decoder are jointly optimized in an unsupervised training process to maximize the mean of $\log P(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, ..., \boldsymbol{y}^{(n)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(m)})$ for all $\boldsymbol{x}$-$\boldsymbol{y}$ pairs in the training data set [22].
The architectures of the encoder and the decoder can be defined independently [22]. The authors of [96] suggest the use of LSTM layers to handle long-term dependencies in the input sequences.

Chapter 6
Anomaly Detection in Time-Series Data

6.1 Time-Series

A time-series represents a discrete sequence of data points organized in chronological order. If it only contains the time-dependent values of one single variable, the time-series is referred to as being *univariate*. A univariate time-series is denoted as a set $\mathcal{X} = \{x(1), \ldots, x(\tau)\}$ of distinct values $x(t)$, each associated with a time instant $t = \{1, \ldots, \tau\}$. A *multivariate* time-series, as used in the context of this thesis, is obtained by sampling two or more variables simultaneously. Since the collected data points related to these variables typically originate from the same process, they often exhibit some kind of correlation. In order to clarify that the multivariate time-series does not consist of independent univariate time-series, this type is sometimes referred to as *continuous* multivariate time-series. It is defined by a set $\mathcal{X} = \{\boldsymbol{x}(1), \ldots, \boldsymbol{x}(\tau)\}$ of time-dependent vectors $\boldsymbol{x}(t)$ containing the data points $x_i(t)$ of all m variables sampled at the respective time instant t [9]:

$$\boldsymbol{x}(t) = \Big[x_1(t), \ldots, x_m(t)\Big]^{\mathrm{T}}, \tag{6.1}$$

where $i = \{1, \ldots, m\}$.

6.2 Anomalies

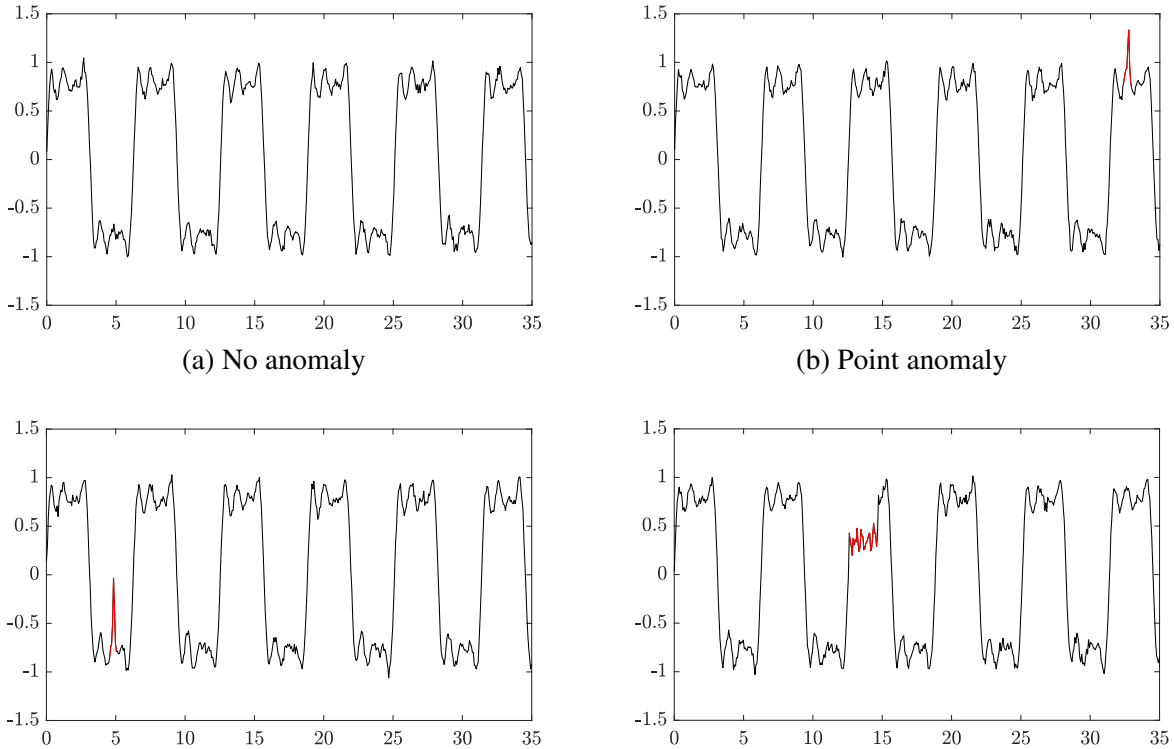
A typical engineering task involving univariate or multivariate time-series is the collection of sensor data from a process, e.g., for monitoring purposes [5]. A possible objective of process monitoring is the detection of *outliers* or *anomalies* [6], which can be formally defined as follows:

> An outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism. [97, p.1]

Thus, an outlier noticeably differs from other collected data points and hence may indicate an unwanted behavior of the analyzed process [6]. According to their characteristics, anomalies can

be divided into three different types (see Figure 6.1) [8]:

- **Point Anomalies:** Data points that significantly deviate from the rest of the data [8].
- **Contextual Anomalies:** Data points that are not anomalous w.r.t. the data set as a whole, but in a certain context. In time-series analysis, contextual anomalies represent data points that considerably differ from adjacent points in the time sequence [8].
- **Collective Anomalies:** Data points that are anomalous due to being part of a collective with suspicious characteristics compared to the rest of the data. The respective data points might not be outliers individually but as a part of an anomalous collection [8].



(a) No anomaly (b) Point anomaly

(c) Contextual anomaly (d) Collective anomaly

Fig. 6.1: Time-series plots of an exemplary periodical process without anomalous behavior (a) and with different types of anomalies plotted in red color (b)-(d). An anomalous data point may represent a global anomaly (b), deviate significantly from adjacent points in the sequence (c), or belong to a collective with suspicious characteristics [8]. Adapted from [98].

6.3 Methods for Anomaly Detection in Time-Series

Since manual anomaly detection, especially in large data sets, is slow and error-prone, various automatic anomaly detection methods have emerged. Time-series data is usually temporally continuous, i.e., abrupt changes in the trend of a signal are not expected and may indicate an unusual event

in the underlying process. This characteristic has to be considered when selecting the anomaly detection method to be used [6].

A simple regression-based model for detecting point anomalies and contextual anomalies in time-series data is an *autoregressive model (AR)*. An $AR(p)$ model for univariate time-series assumes the value $x(t)$ to be linearly dependent on the p previous observations [6]:

$$x(t) = \underbrace{\sum_{j=1}^{p} a_j x(t-j) + c}_{\text{expected value}} + \varepsilon(t), \tag{6.2}$$

where $a_1, ..., a_p$ and c denote the learnable regression parameters. The term $\varepsilon(t)$ represents the error between the expected value and the real value of $x(t)$ and thus can be used to quantify the degree of abnormality. A data point $x(t)$ is considered an outlier if the corresponding error $\varepsilon(t)$ is above a predefined threshold. The model parameters are determined via least-squares regression. A vector $\boldsymbol{y}$ contains the real values the model aims to predict [6]:

$$\boldsymbol{y} = \Big[x(p+1), x(p+2), \ldots, x(\tau)\Big]^{\mathrm{T}}. \tag{6.3}$$

The first p values in the time-series can not be predicted, since there are not enough previous data points available. With the definition of a matrix D and a parameter vector $\boldsymbol{a}$ [6],

$$\mathsf{D} = \begin{bmatrix} x(1) & x(2) & \ldots & x(p) & 1 \\ x(2) & x(3) & \ldots & x(p+1) & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x(\tau-p) & x(\tau-p+1) & \ldots & x(\tau-1) & 1 \end{bmatrix}, \tag{6.4}$$

$$\boldsymbol{a} = \Big[a_p, a_{p-1}, \ldots, a_1, c\Big]^{\mathrm{T}}, \tag{6.5}$$

the over-determined system of equations can be defined by [6]:

$$\boldsymbol{y} \approx \mathsf{D}\boldsymbol{a}. \tag{6.6}$$

The solution for the model parameters in a least-squares sense can be obtained as follows [6]:

$$\boldsymbol{a} = (\mathsf{D}^{\mathrm{T}}\mathsf{D})^{-1}\mathsf{D}^{\mathrm{T}}\boldsymbol{y} = \mathsf{D}^{+}\boldsymbol{y}, \tag{6.7}$$

where D^{+} denotes the *Moore-Penrose pseudo-inverse* [99]. To increase robustness, the *AR* model can be combined with a *moving-average model (MA)*. The $MA(q)$ model assumes the value $x(t)$ to be linearly dependent on the deviations $\varepsilon(t-q), ..., \varepsilon(t-1)$ at the previous q time steps [6]:

$$x(t) = \sum_{j=1}^{q} b_j \varepsilon(t-j) + \mu + \varepsilon(t), \tag{6.8}$$

where μ represents the mean of the time-series and $b_1, ..., b_q$ denote the model parameters to be optimized. Combining an $AR(p)$ model with an $MA(q)$ model yields an *autoregressive moving-average* model $ARMA(p,q)$, defined by [6]:

$$x(t) = \sum_{j=1}^{p} a_j x(t-j) + \sum_{j=1}^{q} b_j \varepsilon(t-j) + c + \varepsilon(t). \tag{6.9}$$

An extension of the *ARMA* model is the so-called *autoregressive integrated moving-average model (ARIMA)*. It can be used on time-series data that exhibits a persistent trend. By differencing the time-series prior to applying the *ARMA* model, the *ARIMA* model eliminates the trend and enforces a stationary (time-invariant) mean [6]. First-order differencing of the data is performed as follows [100]:

$$u(t) = x(t) - x(t-1), \tag{6.10}$$

where $u(t)$ denotes the value at time instant t of the differenced time-series to which the *ARMA* model is applied. Further differencing operations can be carried out in an analogous manner [100]:

$$w(t) = u(t) - u(t-1), \tag{6.11}$$

where $w(t)$ represents the component of the time-series at time instant t after second-order differencing. The $ARIMA(p,d,q)$ model has a parameter d denoting the differencing order, in addition to the parameters p and q of the $ARMA(p,q)$ model [6], [100].

The autoregression methods presented above can be extended to be applicable to multivariate time-series data. Theoretically, each channel could be analyzed separately and treated as a univariate time-series. However, it is recommended to use a more elaborate approach to also account for the correlation between the different channels. The $AR(p)$ model for multivariate time-series assumes the value in channel i to be linearly dependent on the p previous observations in each of the m channels [6]:

$$x_i(t) = \left[\sum_{k=1}^{m} \sum_{j=1}^{p} a_{ijk} x_k(t-j) \right] + c_i + \varepsilon_i(t), \tag{6.12}$$

where a_{ijk} expresses the predictive power of a data point in channel k at time instant $t-j$ on the value in channel i at time instant t. The parameter c_i is a constant, and $\varepsilon_i(t)$ denotes the error between the predicted value and the real value of channel i at time step t. The *ARMA* model and the *ARIMA* model can be extended analogously in order to be used on multivariate time-series data [6]. Applications of these models for anomaly detection can be found in [101], [102] or [103].

Recurrent neural networks with LSTM layers (see Section 3.2.2.1) are frequently used for anomaly detection and time-series prediction due to their ability to keep track of long-term trends in the data [12]. In [104], an approach for detecting outliers based on the prediction error is presented. An LSTM network is trained on non-anomalous data and tested with unseen data sets. It aims to predict the time-series data at future time instances in a prediction window of length l, i.e.,

$\hat{\boldsymbol{x}}(t+1),...,\hat{\boldsymbol{x}}(t+l)$. Hence, data vectors $\boldsymbol{x}(t)$ at time instances $t > l$ are predicted l times at the time instances $t-l,...,t-1$. An error vector $\boldsymbol{\epsilon}_i(t)$ containing the errors $\varepsilon_{i,j}(t)$ between the real value $x_i(t)$ of channel i at time instant t and the corresponding prediction made at time instant $t-j$ is defined by [104]:

$$\boldsymbol{\epsilon}_i(t) = \left[\varepsilon_{i,l},\, \varepsilon_{i,l-1},\, \ldots,\, \varepsilon_{i,1}\right]^{\mathrm{T}}. \tag{6.13}$$

Based on the error vectors $\boldsymbol{\epsilon}_i(t)$ of each channel $i = \{1,...,m\}$, the time-series data at time instant t is either labeled as normal or anomalous [104]. A simplified version of the method presented in [104] is described in [105]. An LSTM network forecasts the values of each channel i only at the next time step $t+1$. With the obtained prediction vector $\hat{\boldsymbol{x}}(t)$ and the corresponding vector $\boldsymbol{x}(t)$ of true values, a criterion for a data vector to be an anomaly can be formulated as follows [105]:

$$||\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t)|| > \Pi, \tag{6.14}$$

where Π denotes the threshold value. Similar prediction-based anomaly detection approaches were made with *convolutional neural networks (CNN)*, e.g., in [106].

Another group of anomaly detection methods based on neural networks involves different types of autoencoders. As explained in Section 5, autoencoders learn to compute a hidden feature representation of the input data, on which the original signal can be well reconstructed. In contrast to the aforementioned prediction-based techniques, outlier detection is based on the error between the original and the reconstructed input signal [11]. The autoencoder is usually trained on non-anomalous data, and the network learns how to reconstruct data sets similar to the training samples [107], [108]. If the network receives a test sample that contains anomalies, it fails to reasonably reconstruct the signal from the hidden encoding, which results in a high reconstruction error. Given an input vector $\boldsymbol{x}(t)$ at time instant t and the corresponding reconstructed vector $\hat{\boldsymbol{x}}(t)$, the reconstruction error $\varepsilon(t)$ can, e.g., be determined by [11]:

$$\varepsilon(t) = ||\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t)||^2. \tag{6.15}$$

Various autoencoder types with different architectures have been studied for their anomaly detection capabilities, such as undercomplete autoencoders with LSTM layers [107], [108], variational autoencoders with LSTM layers [109] or convolutional variational autoencoders [110].

This section only contains an excerpt of the anomaly detection tools for time-series in use. Various other approaches have been made in research. These include clustering techniques like the *DBSCAN (density-based spatial clustering of applications with noise)* [111] algorithm, applied in [112], a modified *k-means clustering* algorithm, as shown in [113], or *one-class support vector machines* [114], used in [115]. The authors of [116] or [117] propose the use of the *isolation forest* [118] algorithm for outlier detection. Other research papers, such as [119], address the application of *local outlier factor (LOF)* [120] algorithms for this task.

Chapter 7
Hybrid Learning Tool for Anomaly Detection

In this section, a hybrid learning tool developed for the detection of anomalies in multivariate time-series data is explained in detail. It consists of a machine learning model, described in Section 7.3, and a statistical model, based on key performance indicators, presented in Section 7.4. Via the statistical model, systematic changes in the data across a set of samples can be analyzed. The autoencoder enables to take nonlinear interdependencies between the different channels of the MVTS into account [17]. These capabilities of the hybrid learning model should make it a reliable tool for detecting anomalies that are difficult to spot using analytical methods alone or that were previously not even known to indicate abnormal behavior. Two approaches for the combination of the machine learning model and the statistical model, and therefore the structure of the hybrid learning tool, are described in Section 7.1.
The hybrid learning tool generally is applicable to all sorts of multivariate time-series data. However, it has been further adapted to analyze data collected from a ground improvement process for building foundations. Therefore, this process and the recorded sensor data sets are explained in Section 7.2.

7.1 Structure of the Hybrid Learning Tool

An important issue that has already been addressed in previous work, e.g., in [121] or [122], is how to combine the machine learning model and the statistical model to achieve the best possible performance in anomaly detection. In this thesis, two hybrid model structures are presented. The first one, referred to as *parallel hybrid*, treats the statistical model and the machine learning model as independent, i.e., they are fed with the same data, and the results are compared [122]. Usually, an autoencoder should be trained only on non-anomalous data in order to learn how to best possibly compress and reconstruct it [11]. In the absence of pre-labeled samples, it can not be guaranteed that the training set does not contain anomalous samples. However, they are assumed to be significantly less prevalent in a data set than non-anomalous data samples. Hence, they should not have a significant impact on the learning process. Furthermore, anomalous samples are expected to be

less similar to each other than non-anomalous data samples. This should force the optimizer to modify the learnable parameters in favor of the non-anomalous data samples in the training process in order to minimize the overall loss. The second option, a *parallel serial hybrid* model, uses the statistical model to pre-label the data samples. The machine learning model then only receives samples that were considered non-anomalous by the statistical model for training. Therefore, the portion of samples in the training set that are in fact non-anomalous should be higher than in parallel hybrid models. This may improve the anomaly detection performance of the machine learning model. However, the models are not fully independent anymore [121], [122]. An application of these hybrid learning models is presented in Section 8.4.

7.2 Analyzed Process

The data used for the experiments presented in this thesis is related to a vibro ground improvement process for stabilizing cohesionless granular soils, which do not fulfill the requirements for the construction of building foundations. This process involves the creation of subsurface columns of compacted gravel or sand via a cylindrical depth vibrator suspended from a crane or mounted on a custom-built rig. These columns are arranged in a predefined pattern on the building site that depends on the local soil conditions. The benefits of ground improvement are a reduction in foundation settlement and an increase in the bearing capacity and stiffness of the ground. It also makes a shallow footing construction possible. The process can be divided into different phases, as shown in Figure 7.1 [16], [123]. In this work, the first phase is referred to as *initial phase*. Here, a wheel loader fills stones into a skip that transports them to a chamber, from where they are unloaded into the inside of the cylindrical vibrator and flow to its tip. The subsequent phase is called *penetration phase*. A horizontal oscillation of the cylindrical vibrator is induced by a rotating eccentric weight. The vibrator is then lowered into the ground and displaces the soil until the required depth is reached. In the *compaction phase*, the vibrator is pulled up slightly, which causes stones at the tip of the vibrator to flow into the originated cavity. The vibrator is then pulled down again in order to compact the filled-in stones and press them into the surrounding soil. These steps of unloading and compacting the stones are performed alternatingly until the ground level is reached, eventually completing a subsurface column [16], [123].

Since columns with deficiencies are a potential safety hazard and can be difficult and costly to rectify, the machinery is equipped with sensors to monitor the process permanently. For each column, a time-series with $n_S = 16$ channels and a sampling interval of $t_S = 1\,\mathrm{s}$ is created [17].

Figure 7.2 shows a plot with data of nine channels segmented into the three different phases of the process. The signals in the first and the second channel describe the current depth at which the drilling rig is located and the corresponding gradient w.r.t. time, respectively. A measurement of the force applied when drilling or compacting gravel yields the data in the third channel. The

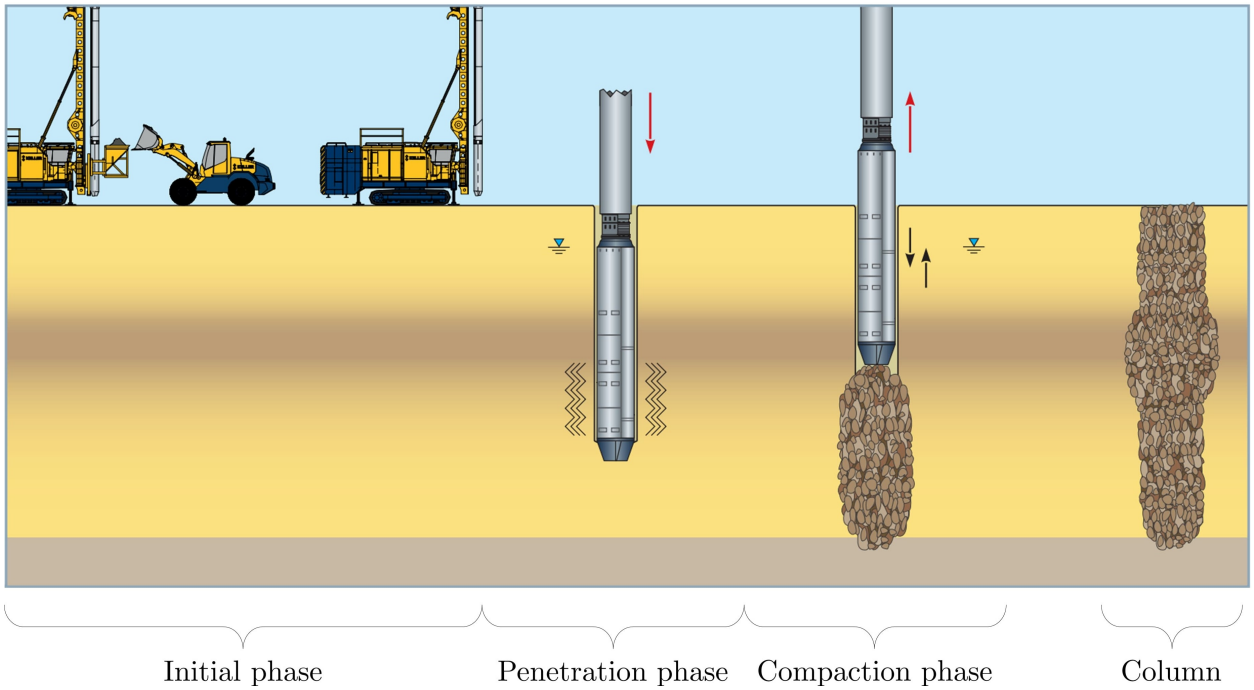


Fig. 7.1: Phases of the vibro ground improvement process. Adapted from [16].

fourth, the fifth, and the sixth channel are related to the power supply and motor of the vibrator. The seventh channel provides information about the weight of the loaded gravel. The last two channels indicate how much the vibrator is tilted w.r.t. two directions, X and Y. As can be seen in Figure 7.3a, this time-series data may contain segments with dead time, i.e., phases where the feed rate remains zero for a noticeable amount of time. This is the case, for example, when the gravel is filled into the skip in the initial phase or refilled during the compaction phase. These long segments with dead time in a time-series could impede the reconstruction of the signal so that the corresponding sample would be considered anomalous. Longer pauses could indeed indicate an anomaly, e.g., a shutdown of the vibrator due to overheating. They could also be caused by the operator taking a break or an exceptionally long gravel refilling process. These long idle times might indicate problems in the site logistics but typically do not have an impact on the quality of the created column. However, detecting this type of anomalous behavior can help to improve process efficiency. In the context of this work, the identification of faulty columns is of higher interest. Therefore, dead time segments are removed from the signal before feeding it into the autoencoder (see Figure 7.3b), whereby dead time is characterized by the gradient of the depth channel falling below a defined threshold [17].

Since a building site can contain up to thousands of columns, manual identification of anomalous samples is error-prone and cost-intensive. This inspired the development of a hybrid learning tool that is able to detect abnormal behavior automatically, reducing the risk of overseeing anomalous data samples [17].

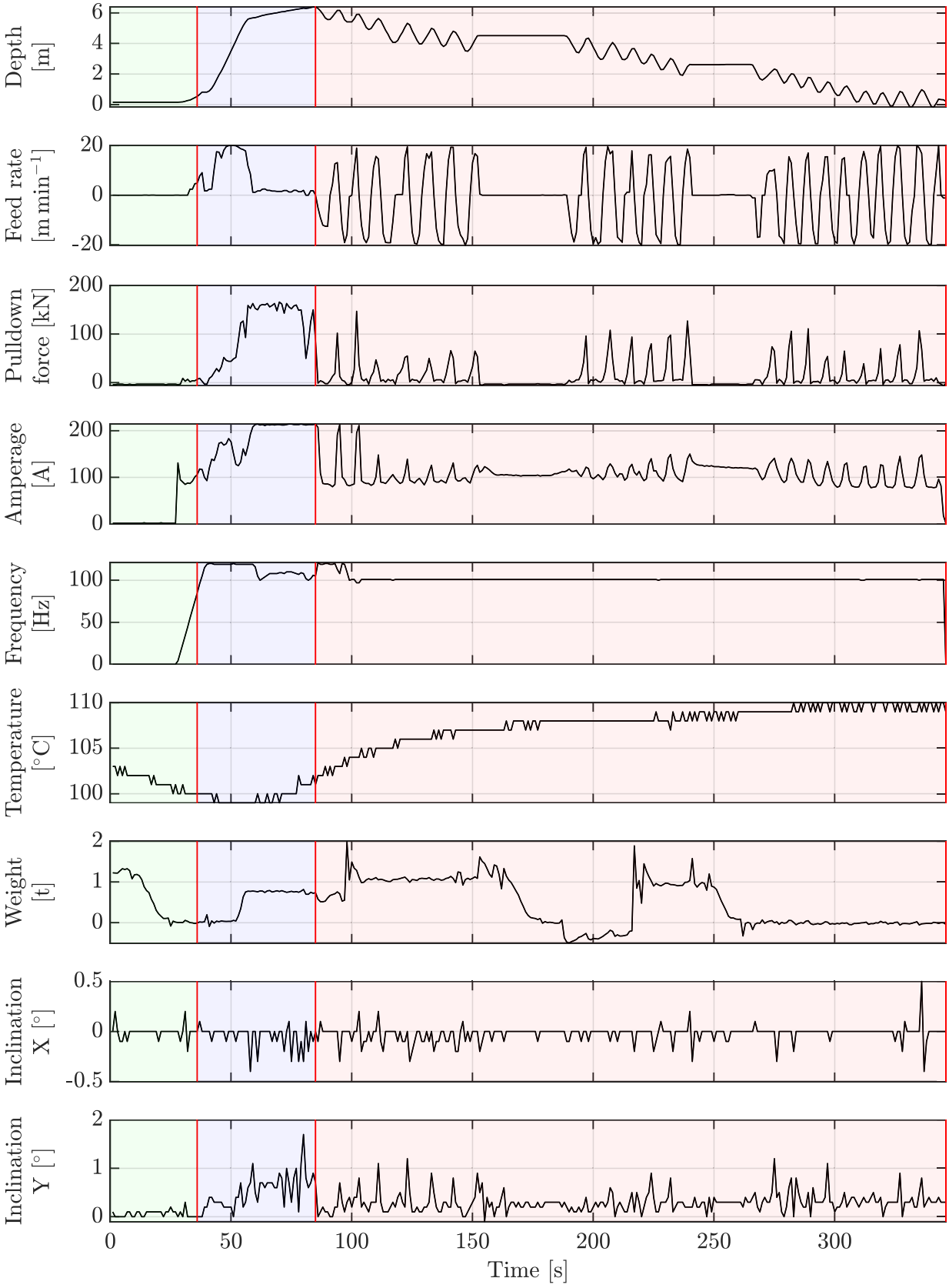
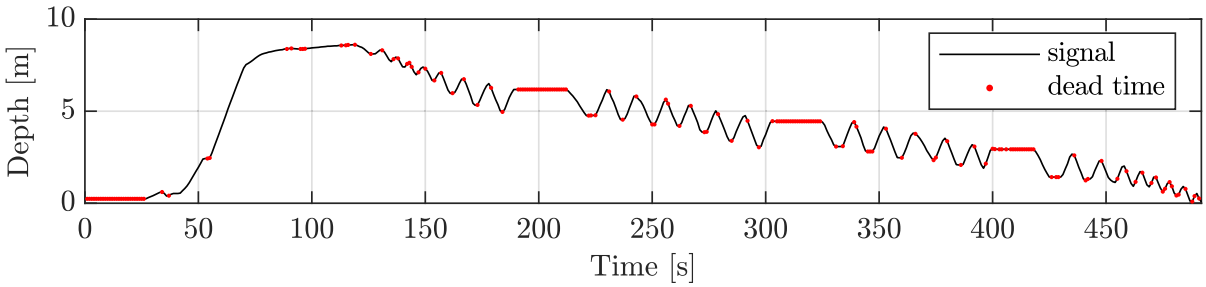
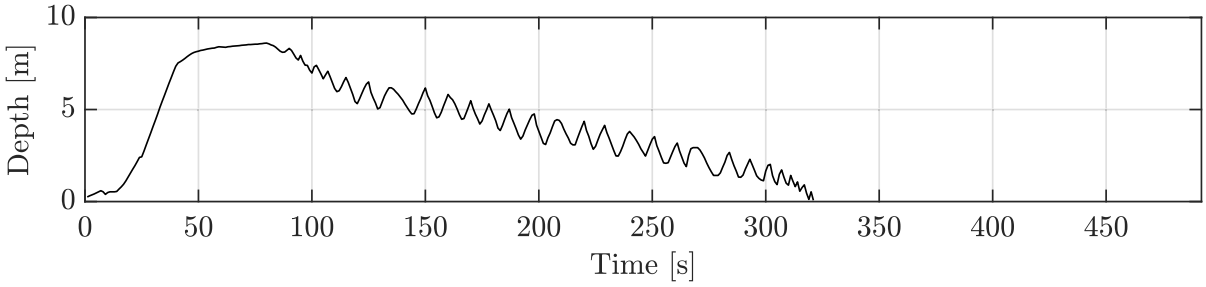


Fig. 7.2: Data of $m = 9$ channels collected from a ground improvement process. The process can be segmented into an initial phase (green), a penetration phase (blue), and a compaction phase (red).



(a) Signal with dead time segments



(b) Signal without dead time segments

Fig. 7.3: Data of the depth channel collected from a vibro ground improvement process. The signal in (a) contains dead time segments (red), where the gradient falls below a defined threshold. These segments are removed to produce the signal shown in (b).

7.3 Machine Learning Model

The machine learning models used in the context of this thesis are autoencoders, see Chapter 5. This work focuses on the optimization and performance evaluation of these models in different series of experiments presented in Chapter 8. Autoencoders of different types with varying architectures were set up via a framework [124] built at the Chair of Automation. This framework is explained in Section 7.3.1. Anomaly detection via autoencoders is based on the error between the input signal and the output signal of the autoencoder, which is reconstructed from a hidden representation of the original data [11], as described in Section 6.3. The determination of a threshold on the reconstruction error in order to detect anomalous samples is explained in Section 7.3.2.

7.3.1 Autoencoder Framework

The networks presented in this thesis were set up using a framework for object-oriented implementation of deep autoencoders in MATLAB, using the provided functions and tools of the *Deep Learning Toolbox*™. The code has been made public and can be accessed on the *Central File Exchange* of MATLAB [124]. The framework was designed to process multivariate time-series of varying lengths.

7.3.1.1 Network Architecture

The current version of the framework (accessed on 15.05.2022) allows choosing between two basic autoencoder types: undercomplete autoencoder (see Section 5.1.1) and variational autoencoder (see Section 5.1.4). An *overcomplete autoencoder* [125], whose latent dimension is at least as high as the input dimension, could also be created. However, this variant is not of interest in the context of this thesis. An encoder network and a decoder network are set up separately. Both at least contain an input layer for feeding sequential data into the network and a fully connected layer defining the dimension of the network output. Optionally, the decoder can additionally include an output layer that applies a nonlinear function on the output values of the previous fully connected layer, forcing them to lie within a certain interval. The currently available function types for this operation are sigmoid, which has been used for the work presented in this chapter, and hyperbolic tangent (see Section 3.1). Parts of the results presented in this thesis were obtained with older versions of the framework, where this layer was not included. This is pointed out in the relevant sections. The user can specify the number and types of the hidden layers as well as the corresponding numbers of neurons. The selectable layer types are LSTM (see Section 3.2.2.1), biLSTM (see Section 3.2.2.2) and fully connected (see Section 3.2.1) layers.

7.3.1.2 Data Preprocessing

The data is rescaled in each channel separately before being fed into the network. This is of high importance as the measurement data of different channels may be associated with different units and scales. Rescaling prevents data with large ranges from dominating over data with lower ranges when optimizing the network parameters, and increases the training speed. The data in each channel is rescaled to lie in the interval $[0,1]$ using *min-max normalization*. Consider a data set $\mathcal{M} = \{X_1, \ldots, X_n\}$ of n multivariate time-series samples X_j, with $j = \{1, ..., n\}$. Each sample $X_j = \{\boldsymbol{x}_j(1), \ldots, \boldsymbol{x}_j(\tau_j)\}$ consists of τ_j data vectors $\boldsymbol{x}_j(t)$, which contain the data of m variables $x_{j,i}(t)$ collected at time instant $t = \{1, \ldots, \tau_j\}$ [126]:

$$\boldsymbol{x}_j(t) = \left[x_{j,1}(t),\, x_{j,2}(t),\, \ldots,\, x_{j,m}(t)\right]^{\mathrm{T}}, \tag{7.1}$$

where $i = \{1, \ldots, m\}$. Furthermore, $x_{j,i}^{\max}$ and $x_{j,i}^{\min}$ respectively denote the maximum and minimum values of channel i over all time instances t of a sample X_j [126]:

$$x_{j,i}^{\max} = \max\left(\{x_{j,i}(1), \ldots, x_{j,i}(\tau_j)\}\right), \tag{7.2}$$

$$x_{j,i}^{\min} = \min\left(\{x_{j,i}(1), \ldots, x_{j,i}(\tau_j)\}\right). \tag{7.3}$$

The rescaled value $x_{j,i}^{\text{resc}}(t)$ of sample j in channel i at time instant t is then computed by [126]:

$$x_{j,i}^{\text{resc}}(t) = \frac{x_{j,i}(t) - x_{j,i}^{\min}}{x_{j,i}^{\max} - x_{j,i}^{\min}}. \tag{7.4}$$

From now on, the superscript *resc* will be omitted, and all variables $x_{j,i}(t)$ will be assumed to be rescaled. Due to the rescaling process, information about the absolute channel values gets lost since each channel of each sample is mapped independently to the defined interval. The use of constant scaling parameters $x_i^{\max}$ and $x_i^{\min}$ to apply to the data of all samples is not recommended when analyzing the vibro ground improvement process. This is because particular attributes, like an appropriate maximum depth, depend on the local soil conditions, which can vary substantially within a site and inter-site. The task of detecting anomalies in unscaled data is covered by the use of key performance indicators.

For training, the set of data samples, i.e., files containing time-series data, is divided into mini-batches of a user-defined size b. Before splitting, the samples are sorted according to their length, i.e., the number of time steps at which data was collected during the measurement. Thus, each mini-batch consists of data samples of similar lengths. This ensures only little information from the data gets lost since the time-series files are downsampled to the length of the shortest in the respective mini-batch.

7.3.1.3 Training

The networks are trained using the Adam optimizer [46] (see Section 3.3.1) for a user-defined number of epochs. Unless specified otherwise, the weight parameters are initialized using the He initializer [54] (see Section 3.3.3). Which loss function is used in the learning process depends on the selected type of autoencoder. For an undercomplete autoencoder, the loss of a sample X_j only consists of the sum-of-squares error of all channels i and time steps t:

$$\mathcal{L}_{AE}(X_j) = \sum_{i=1}^{m} \sum_{t=1}^{\tau_j} \left[\hat{x}_{j,i}(t) - x_{j,i}(t)\right]^2, \tag{7.5}$$

where $\hat{x}_{j,i}(t)$ denotes the reconstructed value of channel i in sample X_j at time instant t. The loss function of the variational autoencoder additionally contains a regularization term [77]:

$$\mathcal{L}_{VAE}(X_j) = -ELBO = \sum_{i=1}^{m} \sum_{t=1}^{\tau_j} \left[\hat{x}_{j,i}(t) - x_{j,i}(t)\right]^2 + \lambda KL\left[\mathcal{N}(\boldsymbol{\mu}, \Sigma) || \mathcal{N}(\mathbf{0}, \mathrm{I})\right], \tag{7.6}$$

whereby the covariance matrix Σ has a diagonal structure. The parameter λ represents a weighting factor that enables to balance the two terms of the loss function according to the requirements [94]. See Section 5.1.4.2 for a detailed description of the ELBO loss, the loss function of the VAE.

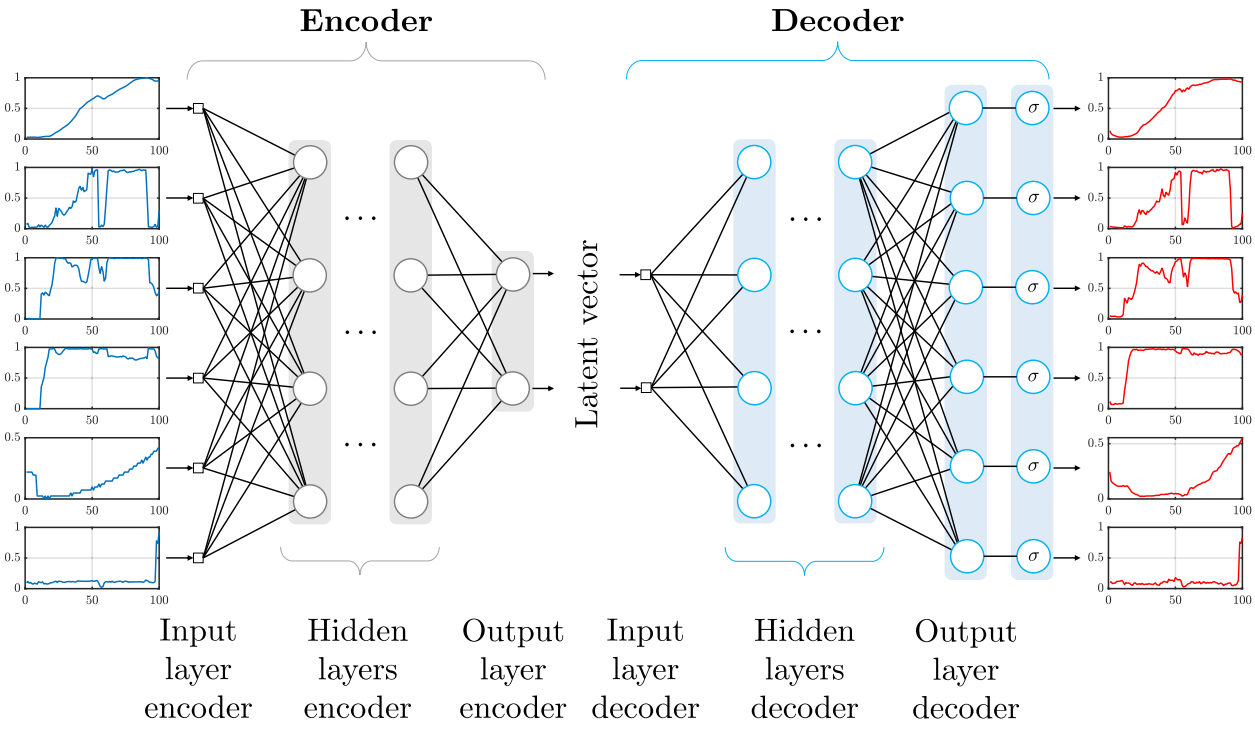


Fig. 7.4: Exemplary schematic diagram of an autoencoder based on the framework of the Chair of Automation [124]. The number of neurons in each layer, as well as the number and types of the hidden layers, can be set by the user and thus may differ from this graphic.

7.3.2 Reconstruction Error and Threshold Setting

The machine-learning-based anomaly detection model labels a data sample X_j as either normal or anomalous based on the associated reconstruction error per time step. The data set $\mathcal{M} = \{X_1, \ldots, X_n\}$ of multivariate time-series samples is split into a *training set* $\mathcal{A}$ with k samples and a *test set* $\mathcal{B}$ containing $n-k$ samples. In case a parallel hybrid approach is used, the members of the training set are selected at random from $\mathcal{M}$ [122]. In parallel serial hybrid models, the training set consists of k randomly selected samples that were pre-labeled as non-anomalous by the statistical model [122]. The learnable parameters of the autoencoder are optimized using the training set $\mathcal{A}$ by minimizing the loss function of the undercomplete or variational autoencoder shown in Equation (7.5) and (7.6), respectively. After training, the autoencoder is fed with the data samples of the test set $\mathcal{B}$ and outputs a reconstructed signal $\hat{X}_j = \{\hat{\boldsymbol{x}}_j(1), \ldots, \hat{\boldsymbol{x}}_j(\tau_j)\}$ of each sample X_j, with:

$$\hat{\boldsymbol{x}}_j(t) = \left[\hat{x}_{j,1}(t), \hat{x}_{j,2}(t), \ldots, \hat{x}_{j,m}(t)\right]^{\mathrm{T}}. \tag{7.7}$$

The normalized sum-of-squares reconstruction error E_j of sample X_j is obtained by summing up the squared errors of all channel values at each time step, and normalizing it by the duration τ_j:

$$E_j = \frac{1}{\tau_j} \sum_{i=1}^{m} \sum_{t=1}^{\tau_j} \left[x_{j,i}(t) - \hat{x}_{j,i}(t)\right]^2. \tag{7.8}$$

A data sample X_j is flagged as an outlier if the reconstruction error E_j exceeds a certain threshold Π [15], i.e.:

$$E_j > \Pi. \tag{7.9}$$

This threshold Π is determined via a *skewness-adjusted boxplot* [127]. This is a modified version of the standard boxplot that accounts for the skewness of the distribution associated with the data it is applied to. More specifically, the lengths of the whiskers are adjusted based on a measure for the skewness that is more robust against outliers than conventional measures, the *medcouple (MC)* [128]. Suppose the reconstruction errors E_j of each sample X_j in a test data set $\mathcal{B}$ are members of an error set $\mathcal{E}$. The median error or second quartile of this set is denoted as Q_2. Considering all pairs of errors below or equal Q_2, $E_l \leq Q_2$, and errors above or equal Q_2, $E_u \geq Q_2$, the medcouple is determined by [128]:

$$MC = \underset{E_l \leq Q_2 \leq E_u}{\text{med}} h(E_l, E_u). \tag{7.10}$$

The kernel function $h(E_l, E_u)$ for all $E_l \neq E_u$ is defined as follows [128], [129]:

$$h(E_l, E_u) = \frac{(E_u - Q_2) - (Q_2 - E_l)}{E_u - E_l}. \tag{7.11}$$

The special case of $E_l = Q_2 = E_u$ is treated by [128]:

$$h(E_l, E_u) = \begin{cases} +1 & \text{if } l > u \\ 0 & \text{if } l = u \\ -1 & \text{if } l < u\,. \end{cases} \tag{7.12}$$

The medcouple represents the median kernel function value of all pairs of errors E_l and E_u. It lies in the interval $[-1, 1]$, where $MC > 0$ indicates a right-skewed distribution, $MC < 0$ a left-skewed distribution and $MC = 0$ a symmetric distribution [129]. The lengths of the whiskers define the location of the boundaries that separate normal data points from outliers. In standard boxplots [130], these lengths are fixed to $1.5\,IQR$, where IQR represents the interquartile range [128],

$$IQR = Q_3 - Q_1, \tag{7.13}$$

and Q_1 and Q_3 denote the first and third quartile, respectively. The corresponding interval that contains the data points assumed to be normal is defined by [128]:

$$[Q_1 - 1.5\,IQR, \quad Q_3 + 1.5\,IQR]\,. \tag{7.14}$$

For asymmetric distributions, too many data points lie outside this interval, i.e., they are flagged as outliers. The analogous interval of the skewness-adjusted boxplot for right-skewed or symmetric distributions, i.e., $MC \geq 0$, is defined as shown in Equation (7.15) [129],

$$\left[Q_1 - 1.5\mathrm{e}^{-4MC}\,IQR, \quad Q_3 + 1.5\mathrm{e}^{3MC}\,IQR\right]. \tag{7.15}$$

For left-skewed distributions ($MC < 0$) it is defined by [129]:

$$\left[Q_1 - 1.5\mathrm{e}^{-3MC}\,IQR, \quad Q_3 + 1.5\mathrm{e}^{4MC}\,IQR\right]. \tag{7.16}$$

The distribution over the reconstruction errors E_j of all data samples in the test was found to be right-skewed (see Figure 7.5). Hence, the skewness-adjusted boxplot was supposed to be more suitable for setting the threshold than the standard version. This method for outlier detection has already been applied in previous work at the Chair of Automation [15]. Note that only samples with reconstruction errors above the upper interval boundary are considered as outliers since a reconstruction error below the lower interval limit does not indicate anomalous behavior. An ex-

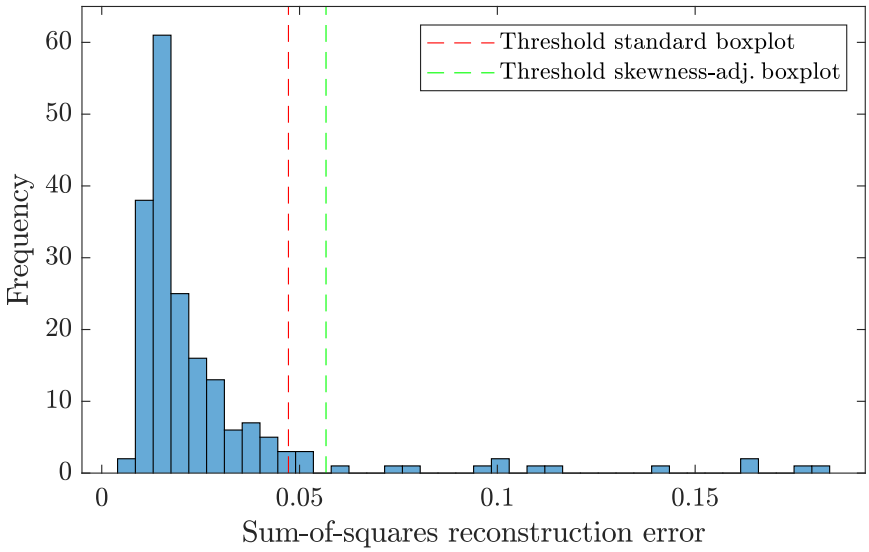


Fig. 7.5: Right-skewed distribution over the reconstruction errors of all samples of a test data set related to the vibro ground improvement process. The thresholds were determined using a standard boxplot [130] (red) and a skewness-adjusted boxplot [128] (green).

emplary plot of the sum-of-squares reconstruction error E_j of all samples in the test data set, and the threshold Π separating non-anomalous samples from outliers, is shown in Figure 7.6.
A core goal of this thesis is to determine which autoencoder architectures are the most suitable for anomaly detection, see Section 8.1. Usually, this would be done by comparing the obtained classifications with the known true labels of a test data set [22]. In the absence of labeled data sets, some intuitive measures to assess the suitability of a model for anomaly detection were defined [15]:

- **Median reconstruction error of non-anomalous samples** $\tilde{E}_{normal}$: This measure describes the capability of a model to reconstruct non-anomalous signals. Since these signals should be well reconstructed, a low value of $\tilde{E}_{normal}$ is aspired.

- **Median reconstruction error of anomalous samples** $\tilde{E}_{outlier}$: The reconstruction error of an anomalous data sample is aimed to be significantly higher than the error of a normal sample since the reconstruction error is a measure of the degree of abnormality.
- **Distance of medians** $d_{med} = \tilde{E}_{outlier} - \tilde{E}_{normal}$: The distance of the median errors of anomalous and non-anomalous data samples should be high in order to enhance discrimination of these two classes.
- **Standard deviation of reconstruction error of non-anomalous samples** σ_{normal}: A low value of this measure is aspired, as it allows to distinguish between non-anomalous data and outliers more accurately.

See Figure 7.6 for a graphical representation of these measures. While they provide an indication of the anomaly detection capabilities of a tested network, these measures are not fully reliable. This is because it can not be ensured that the data labels obtained by the model are correct.

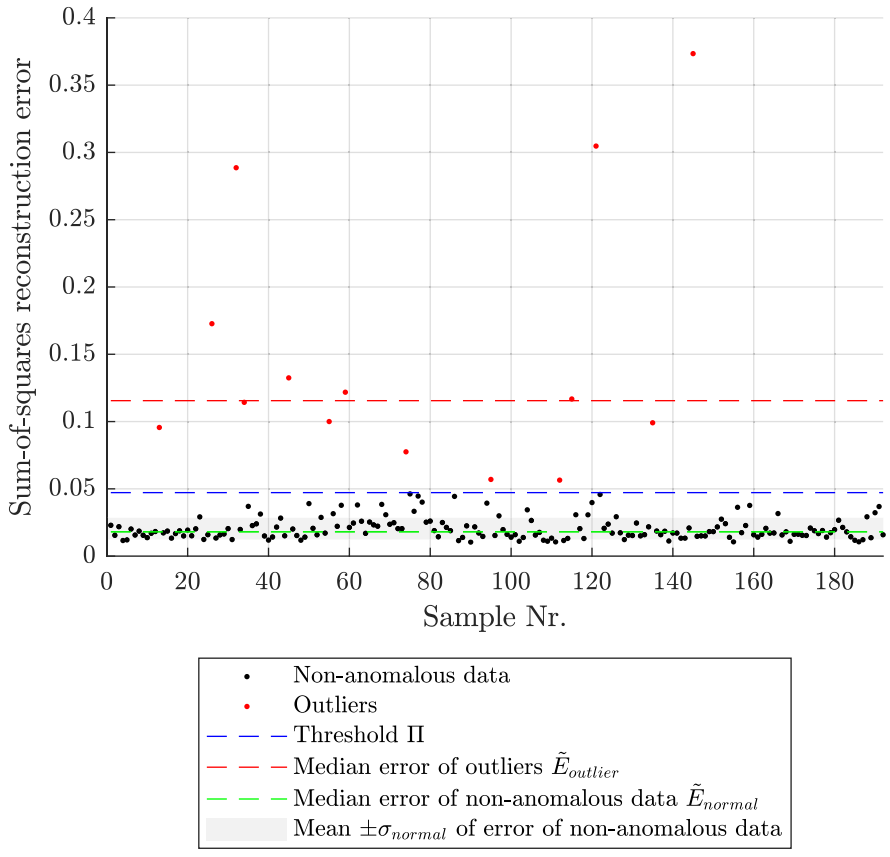


Fig. 7.6: Exemplary plot of the reconstruction errors associated with the data samples of a test set. It shows the threshold for separating non-anomalous data from outliers and some statistical measures.

7.4 Statistical Model

Statistical outlier detection was performed via the use of key performance indicators at the Chair of Automation, as presented in [18]. Statistical measures were defined for the process as a whole, particular phases of the process, or specific data channels. Figure 7.7 shows a heat map with different KPIs evaluated for a set of test samples collected at the same site.

A data sample X_j is considered to be anomalous in terms of the quantity measured by a particular KPI if the value of the respective KPI significantly deviates from the values obtained for the other samples in test set $\mathcal{B}$ [17]. The boundaries for separating non-anomalous data points from outliers are defined according to the criterion used to determine outliers in standard boxplots [128], [130]. That is, a data point is flagged as an outlier if it is below $Q_1 - 1.5\,IQR$ or above $Q_3 + 1.5\,IQR$ [17]. Hence, the interval of data points assumed to be non-anomalous is defined as shown in Equation 7.14.

The number of KPIs of a data sample that lie outside this interval can be used to quantify the degree to which the sample is believed to be anomalous by the statistical model. In this work, this measure is referred to as *outlierness* [17]. For ease of interpretation, the outlierness is normalized, i.e., the outlierness of each sample is divided by the highest obtained outlierness in the test set. Hence, the values of the outlierness lie in the interval [0,1]. As with the machine learning model, a threshold could be defined to distinguish anomalous and non-anomalous samples. This, however, is not in the scope of this work.

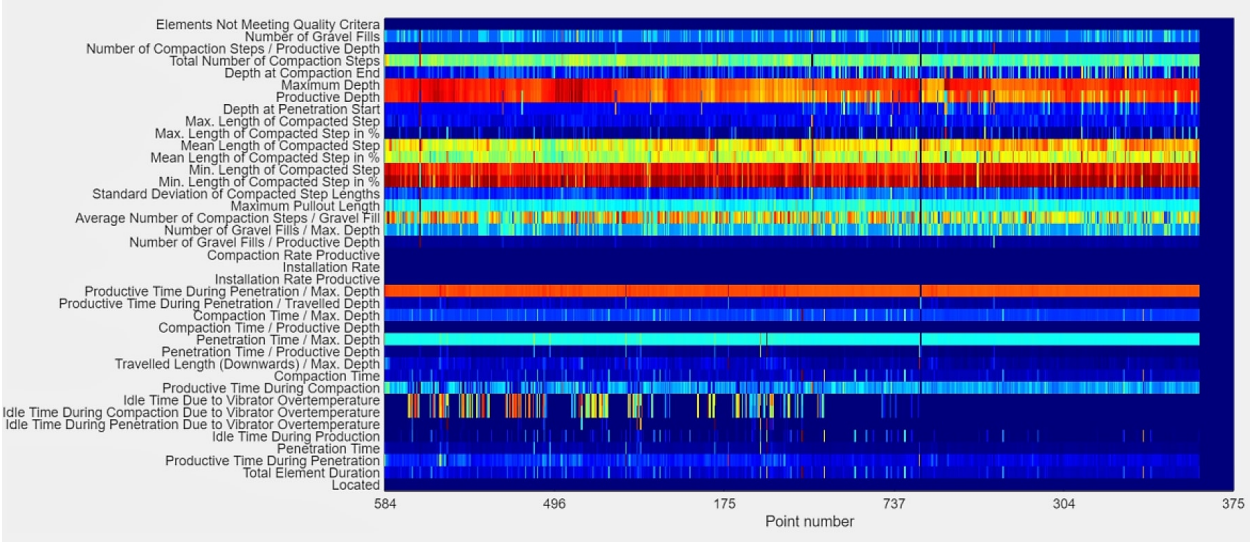


Fig. 7.7: Heat map of different KPIs evaluated for a set of samples (points), which are sorted chronologically. It allows detecting systematic patterns within a site and relationships between different KPIs. All samples were collected at the same construction site [17], [18]. Adapted from [18].

Chapter 8
Test Results

This chapter contains the results of practical work that was performed in the context of this thesis. The aim was to optimize a machine learning model that should be used for anomaly detection in time-series data in combination with a statistical tool based on key performance indicators. In separate series of experiments, different network architectures, weight initializing techniques, and hyperparameter optimization methods were investigated. The test data was acquired from a ground improvement process described in detail in Section 7.2. For the tests related to this thesis, only data from one construction site was used. Since the conditions vary across different sites, a trained model should only be applied to data from the same site as the training samples. Results of tests with data from other sites can be found, e.g., in [15]. The scope of the test series related to this thesis was limited due to the time-intensive optimization and training processes. Still, the results should give a good indication of how to set up and optimize an autoencoder to achieve a satisfying performance in anomaly detection. In the absence of labeled data sets, the idea was to analyze the data samples that were considered anomalous by the autoencoder but not by the statistical model. This should help to gain a deeper understanding of the process and optimize the hybrid learning tool. A detailed description of the applied hybrid learning tool is given in Chapter 7.
Setting the model parameters appropriately is crucial for the performance of the autoencoder and requires knowledge and experience. Also, the optimal choice of the parameters is dependent on the data the network receives, i.e., the parameters have to be adapted to the problem at hand [58]. For the test series related to this thesis, some of these parameters were optimized via a genetic algorithm written at the Chair of Automation [15]; see Section 3.4.4 for a rough description. In one series of tests, alternative HPO methods have been applied, as shown in Section 8.2.

8.1 Evaluation of Different Architectures

The aim of the following experiments was to determine a model parameter setting that enables the best performance in anomaly detection when using data from the vibro ground improvement process. For this purpose, the performances of undercomplete autoencoders and variational autoen-

coders with LSTM or biLSTM layers were compared. One of the objectives was to evaluate the benefit of using bidirectional LSTM layers instead of simple LSTM layers and if it would justify the more time-consuming training process.
Related work addressed the performance of simple neural networks with LSTM or biLSTM layers in time-series prediction, as presented in [131], [132], or [133], for example. In all these papers, networks with biLSTM layers were found to outperform their counterparts with univariate LSTM layers. While in [131], the prediction RMSE could be reduced by 57.43% when using biLSTM instead of LSTM layers, the reduction of the average RMSE was only approx. 5% at most in the experiments presented in [133]. In [132], the networks were tested with eleven different time-series data sets. The reduction in RMSE when using biLSTM-networks instead of LSTM-networks was between 12.93% and 77.60%, depending on the data set being analyzed. These results indicate that the achievable improvement in tracking temporal dependencies in the data when using biLSTM layers over LSTM layers highly depends on the data the network receives.
The experiments shown in this section were also intended to provide an indication of which autoencoder type is better suited for anomaly detection in the given MVTS data sets. As the selection of the training and network hyperparameters is a crucial task for the performance of a neural network, a hyperparameter optimization was performed using a genetic algorithm written at the Chair of Automation [15]. This should allow a reliable comparison of different network architectures independent of external influences, like the user's experience or preferences. Since the genetic algorithm involves stochastic operations, the results of different runs with the same parameter settings may vary. Due to limited resources, only one HPO run was performed with each architecture. As input data, $m = 6$ channels of the collected time-series data were chosen according to their importance for the analyzed ground improvement process: depth, pulldown force, vibrator amperage, vibrator frequency, vibrator temperature, and weight (see Section 7.2). The training set consisted of 80 MVTS data samples with varying time-series lengths, which were pre-labeled as non-anomalous by the statistical model. The trained networks were tested with 192 unseen data samples. The dimension of the latent space was chosen to be $p = 3$ based on previous work [15], where a compression rate of 50% was found to be well-suited. In the first series of experiments, autoencoders with a single hidden layer in the encoder and the decoder were tested. In the second series, the possible benefit of adding an additional layer to the networks was investigated. It has to be mentioned that in the experiments presented in this section, the decoder did not contain a separate sigmoid layer, as shown in Figure 7.4. Also, the initial weight parameters were determined using the Xavier initializer. This is because an older version of the framework was used.

8.1.1 Architectures With One Hidden Layer

In this section, the results of a series of experiments with autoencoders containing one hidden layer in the encoder and the decoder are presented. These hidden layers were of the same type, i.e., both

were either LSTM or biLSTM layers. First, suitable hyperparameters were determined via a genetic algorithm for each architecture separately. Table 8.1 contains the hyperparameters that were optimized in this process and the corresponding predefined bounds. As can be seen, the interval

Table 8.1: Hyperparameters to optimize and corresponding bounded domains for architectures with one hidden layer

Hyperparameter	**Domain**	**Lower Bound**	**Upper Bound**
Nr. of epochs	discrete	1	100
Nr. of neurons in encoder	discrete	1	100
Nr. of neurons in decoder	discrete	1	100
Learning rate	continuous	3×10^{-3}	1×10^{-1}
Mini-batch size	discrete	2	53

boundaries for the values a hyperparameter could take have been set generously. The idea was to allow the optimizer to explore a large search space at the beginning of the HPO process in order to identify promising regions on which to focus the further search.
The performance of each network associated with a set of hyperparameters was tested via k-fold cross-validation, where $k = 3$. When using this method, the training set is divided into k subsets or folds. The network is then trained k times in a cyclic manner using $k-1$ subsets in each run and tested on the remaining one [34]. A mini-batch can not consist of more elements than the training set, which contains the samples of 2 out of 3 folds. These are at minimum 53 samples, which therefore is set as the upper limit for the mini-batch size in Table 8.1. In addition to the hyperparameter domains, some parameters of the genetic algorithm had to be defined, as shown in Table 8.2. The

Table 8.2: Parameters of the genetic algorithm for architectures with one hidden layer

Parameter	**Selected value**
Population size	20
Maximum number of generations	7
Probability of mutation	0.05 %
Probability of unfit individual in mating pool	0.05 %

population size represents the number of individuals, i.e., hyperparameter configurations, of each generation [65]. The genetic algorithm proposed in [15] is executed until at least one out of three termination criteria is met. In order to prevent unnecessary computations, execution is stopped if all hyperparameter configurations in a generation are equal. Also, the process is terminated if the average fitness in the current generation is worse than in all of the four previous generations. If none of the aforementioned events occurs, the computation is stopped after the specified maximum number of generations is reached. The probability of mutation expresses the probability of an individual's genes being altered according to a defined mutation method [65]. The probability

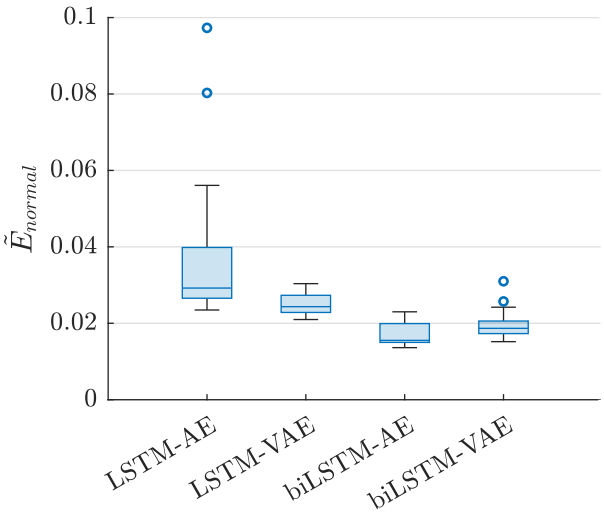
of an unfit individual being added to the mating pool describes how likely it is that an individual, which would not be considered for mating based on its fitness value, is selected to act as a parent. The use of unfit individuals for mating is not included in conventional genetic algorithms but the algorithm applied in this work [15]. For more information on genetic operations, see Section 3.4.4. The hyperparameter configurations with the best fitness values obtained by the genetic algorithm are shown in Table 8.3. As can be seen, the optimum number of epochs was found to be near the

Table 8.3: HPO results of autoencoder architectures with one hidden layer in encoder and decoder

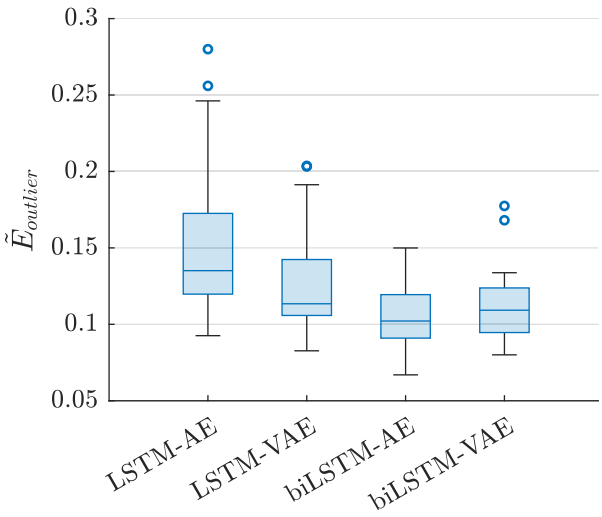
Autoencoder Type	**Number Epochs**	**Neurons Encoder**	**Neurons Decoder**	**Learning Rate**	**MB Size**
LSTM-AE	93	59	19	3.691×10^{-2}	12
LSTM-VAE	77	55	30	5.133×10^{-2}	4
biLSTM-AE	96	70	32	1.877×10^{-2}	3
biLSTM-VAE	96	57	47	2.571×10^{-2}	6

upper predefined limit of possible values. This indicates that a better performance might be expected if the training process would span over more epochs than obtained in the HPO. The upper limit was set to ensure an acceptable training time. It appears beneficial to have a significantly higher number of neurons in the encoder than in the decoder. Also, a small mini-batch size seems to improve the network performance.

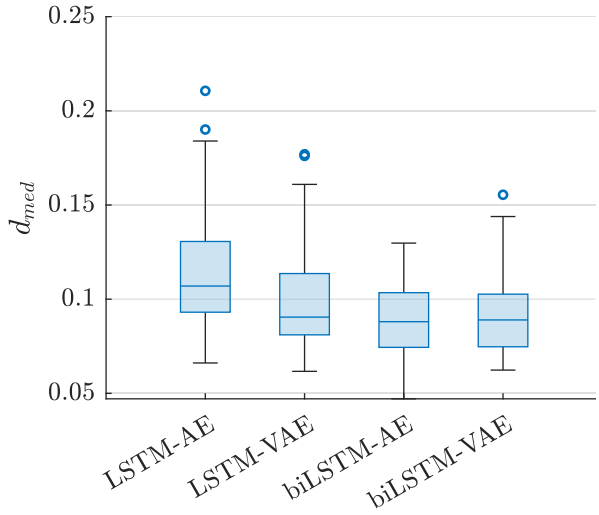
After optimizing the hyperparameters, networks were set up using the framework described in Section 7.3.1. In order to compensate for the stochastic nature of weight initialization, each autoencoder was trained and tested $n_{runs} = 25$ times. The four statistical measures described in Section 7.3.2 were computed separately for each architecture and test run. The corresponding results are shown in Figure 8.1. As can be seen in Figure 8.1a, the autoencoders with hidden biLSTM layers outperformed the ones with LSTM layers in terms of the reconstruction error of data labeled as non-anomalous. Yet, they also allowed a better reconstruction of signals considered as outliers, see Figure 8.1b. However, a significantly lower standard deviation of the reconstruction errors of data labeled as normal could be achieved when using a biLSTM-AE or a biLSTM-VAE, compared to the LSTM variants, as shown in Figure 8.1d. This suggests that they allow easier separation of outliers from non-anomalous data. Thus, despite the lower reconstruction error of data samples flagged as outliers, networks with hidden biLSTM layers were found to potentially perform better in anomaly detection tasks than their LSTM counterparts. Also, the distribution over the statistical measures in Figure 8.1 seems nearer to a Gaussian distribution when using variational autoencoders rather than undercomplete autoencoders. This could be advantageous but needs to be verified by further experiments.



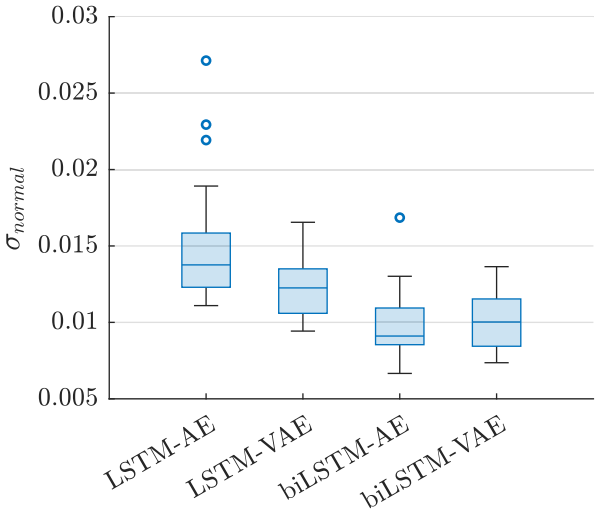
(a) Median error of samples labeled as non-anomalous by the machine learning model



(b) Median error of samples labeled as anomalous by the machine learning model



(c) Distance of median errors of non-anomalous samples (a) and anomalous samples (b) as labeled by the machine learning model



(d) Standard deviation of errors of samples labeled as non-anomalous by the machine learning model

Fig. 8.1: Statistical measures of sum-of-squares sample errors obtained with different autoencoder architectures containing one hidden layer in encoder and decoder for $n_{runs} = 25$ test runs.

8.1.2 Architectures With Two Hidden Layers

In a second series of experiments, performance tests with autoencoders containing two hidden layers in the encoder and decoder were executed. These layers were either LSTM layers or biLSTM layers. The bounded domains for the optimizable hyperparameters were defined as shown in Table 8.4. Based on the results obtained in the HPO process in Section 8.1.1, the bounds for the number of epochs were modified such that higher values were possible. The upper limits for the number of neurons in the inner hidden layers were set to lower values than the corresponding limits

of the outer hidden layers. This was done to encourage an hourglass structure of the autoencoder, which was expected to improve the performance. Again, the hyperparameters were optimized with

Table 8.4: Hyperparameters to optimize and corresponding bounded domains for architectures with two hidden layers

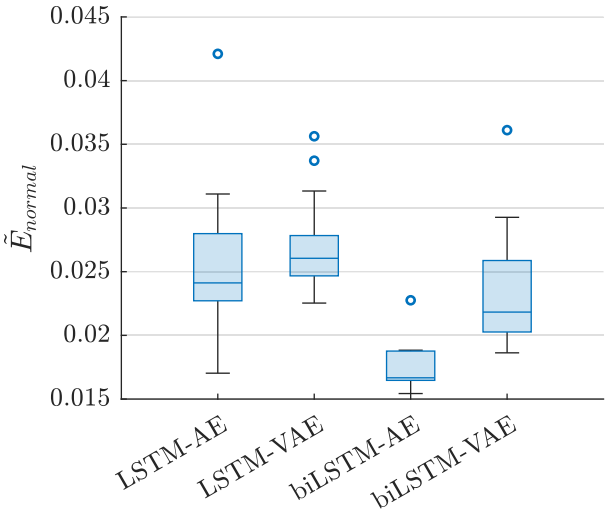
Hyperparameter	Domain	Lower Bound	Upper Bound
Nr. of epochs	discrete	50	200
Nr. of neurons in 1st layer of encoder	discrete	10	80
Nr. of neurons in 2nd layer of encoder	discrete	10	60
Nr. of neurons in 1st layer of decoder	discrete	10	60
Nr. of neurons in 2nd layer of decoder	discrete	10	80
Learning rate	continuous	3×10^{-3}	1×10^{-1}
Mini-batch size	discrete	2	53

a genetic algorithm via 3-fold cross-validation, as described in Section 8.1.1. For consistency, the parameters of the genetic algorithm were set to the same values as presented in Table 8.2. The hyperparameter configurations for each tested architecture that achieved the best fitness values are shown in Table 8.5. Autoencoders were set up, trained, and tested $n_{runs} = 25$ times with the

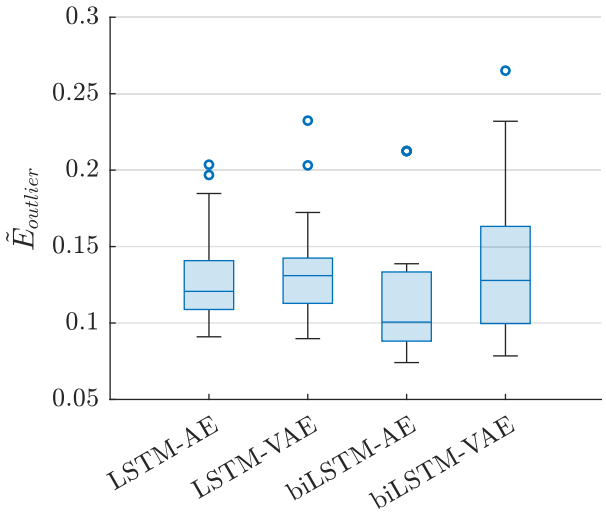
Table 8.5: HPO results of autoencoder architectures with two hidden layers in encoder and decoder

Autoencoder	Nr. of	Neurons Encoder		Neurons Decoder		Learning	MB
Type	Epochs	Layer 1	Layer 2	Layer 1	Layer 2	Rate	Size
LSTM-AE	148	40	47	45	63	1.248×10^{-2}	6
LSTM-VAE	153	68	31	46	43	1.310×10^{-2}	10
biLSTM-AE	169	37	52	48	37	1.608×10^{-2}	5
biLSTM-VAE	155	47	37	46	35	1.748×10^{-2}	12

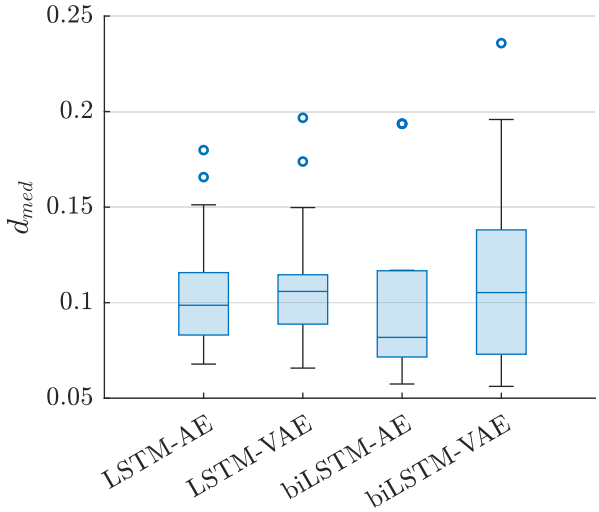
optimized hyperparameters. Again, four statistical measures were computed separately for each architecture and test run. The results are visualized in the form of boxplots in Figure 8.2. As can be seen in Table 8.5, the aspired hourglass structure did not turn out to enable the best performance. It is quite noticeable that the optimized number of neurons in the layer that receives the data from the latent space as input, i.e., the first layer of the decoder, is similar for all architectures. The same applies to the learning rate and the number of epochs. Figure 8.2a indicates that autoencoders with hidden biLSTM layers, especially the biLSTM-AE, could better reconstruct data samples labeled as non-anomalous than LSTM autoencoders. However, they did not necessarily allow easier separation of non-anomalous data from outliers. This can be derived from Figure 8.2c and 8.2d. The distance of medians d_{med} and the standard deviation σ_{normal} of the reconstruction errors of normal data varied widely across different test runs for networks with biLSTM layers. That is, they exhibited poor reproducibility, and multiple training runs might be necessary to ensure the network performs well. Therefore, it is not possible to draw a definitive conclusion as to which of these



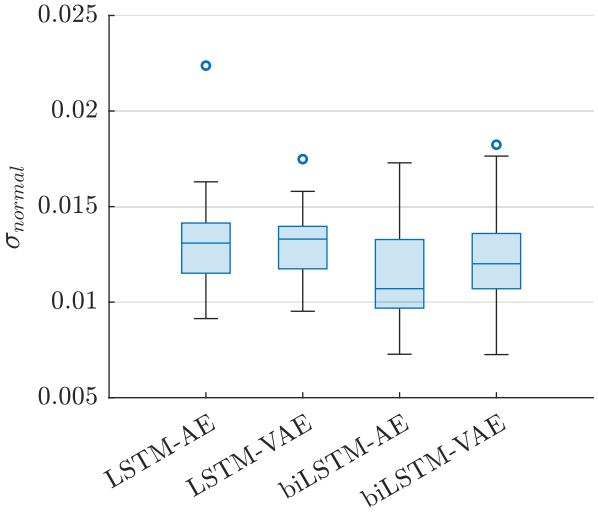
(a) Median error of samples labeled as non-anomalous by the machine learning model



(b) Median error of samples labeled as anomalous by the machine learning model



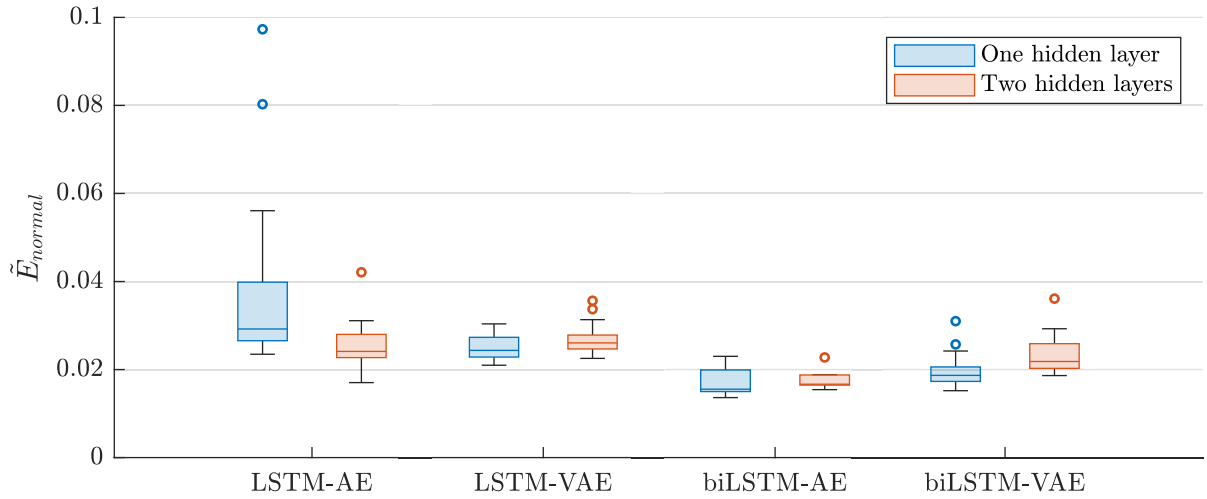
(c) Distance of median errors of non-anomalous samples (a) and anomalous samples (b) as labeled by the machine learning model



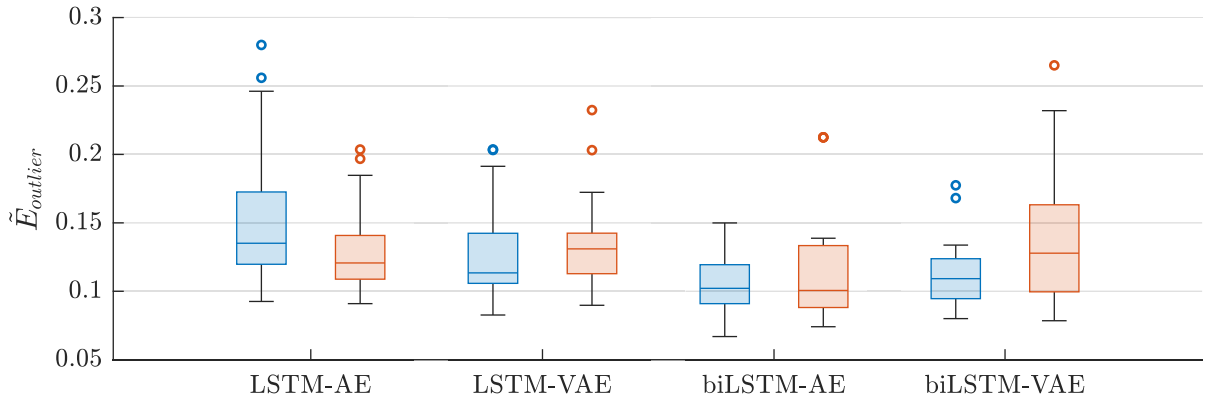
(d) Standard deviation of errors of samples labeled as non-anomalous by the machine learning model

Fig. 8.2: Statistical measures of sum-of-squares sample errors obtained with different autoencoder architectures containing two hidden layers in encoder and decoder for $n_{runs} = 25$ test runs.

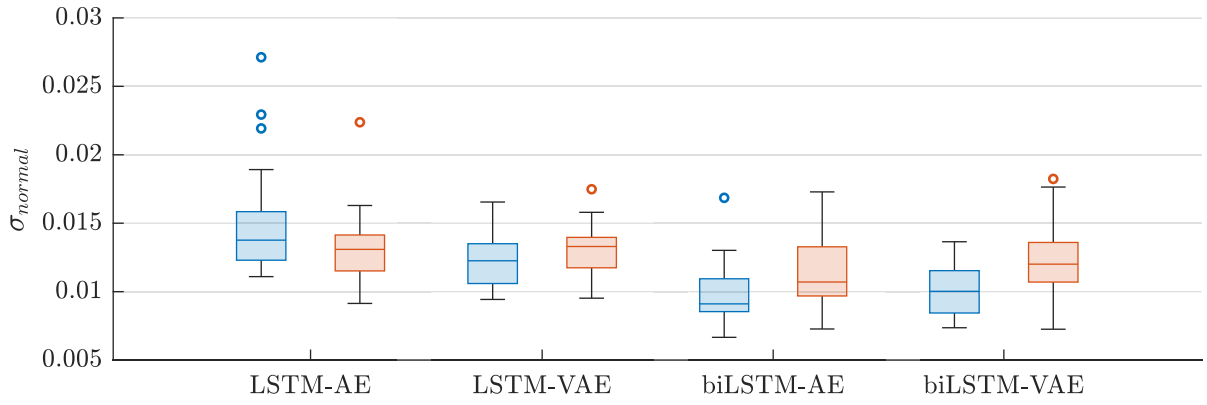
network architectures is the most suitable for anomaly detection. In Figure 8.3 the results of this section are compared to the ones presented in Section 8.1.1. Except for the LSTM-AE, the performance of the autoencoders could not be improved by adding an additional hidden layer to the networks. Encoders and decoders with two hidden layers do not seem to enable a lower reconstruction error of data labeled as non-anomalous than their counterparts with one hidden layer (see Figure 8.3a). In terms of σ_{normal}, they even got outperformed. Due to their lower complexity and training effort at a comparable performance, architectures with one hidden layer were found to be preferable to those with two hidden layers.



(a) Median error of samples labeled as non-anomalous by the machine learning model



(b) Median error of samples labeled as anomalous by the machine learning model

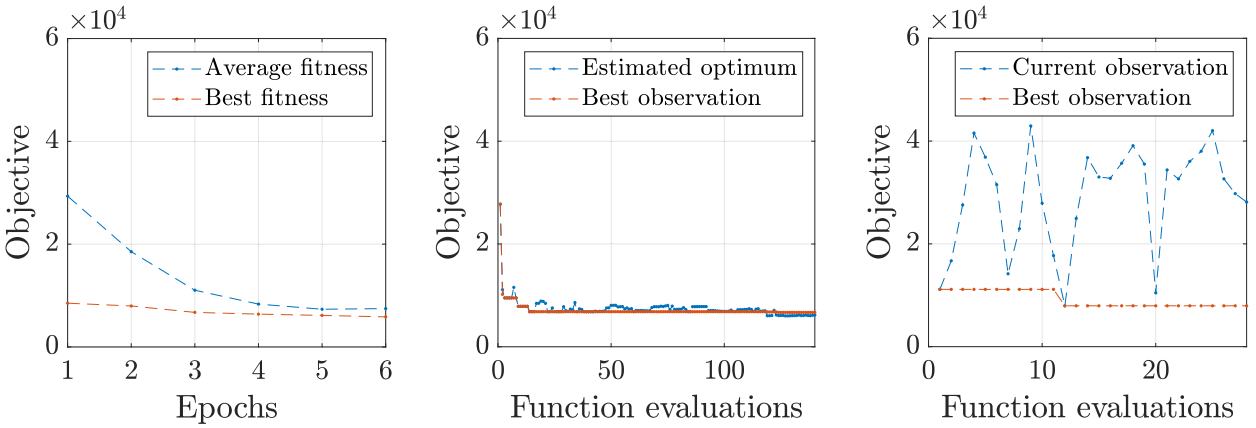


(c) Standard deviation of errors of samples labeled as non-anomalous by the machine learning model

Fig. 8.3: Statistical measures of sum-of-squares sample errors obtained with different autoencoder architectures for $n = 25$ test runs. The boxplots in blue are associated with autoencoders, whose encoder and decoder contain one hidden layer. The boxplots in orange correspond to autoencoders, whose encoder and decoder have two hidden layers.

8.2 Evaluation of Hyperparameter Optimization Methods

Choosing a network model's architectural and training hyperparameters is a crucial task and may require a lot of experience and trials [58]. Several methods exist that search for a suitable hyperparameter configuration automatically (see Section 3.4). One of these techniques, a genetic algorithm, has already been used for the tests shown in Section 8.1. In the following, the results of a series of experiments, which also included the use of other hyperparameter optimization methods, are presented. The aim was to test if Bayesian optimization (see Section 3.4.3) or random search (see Section 3.4.2) can outperform the genetic algorithm. Each method was run $n_{HPO} = 3$ times with an AE and a VAE, whose encoder and decoder contained one hidden biLSTM layer. As in previous experiments described in Section 8.1, 80 training samples pre-labeled as non-anomalous by the statistical model and 192 test samples of the same site were used. The latent dimension was set to $p = 3$, and data of the following $m = 6$ channels was fed into the networks: depth, pulldown force, vibrator amperage, vibrator frequency, vibrator temperature, and weight. The bounds on the hyperparameters were defined as shown in Table 8.1. Each hyperparameter configuration was evaluated in a 3-fold cross-validation process. The sum of the reconstruction errors of all validation samples was used as the objective function to be minimized in the optimization process [15]. In each HPO run, a total of 140 networks were tested in this manner. Figure 8.6 shows exemplary plots of the optimization process for a biLSTM-biLSTM AE performed with the formerly mentioned methods. It can be seen that the evaluations of the objective function tend to yield lower values as the optimization process progresses when using the GA or Bayesian optimization. With random search, no systematic improvement in the evaluated function values can be observed.



(a) Genetic algorithm (b) Bayesian optimization (c) Random search

Fig. 8.6: Plots of the hyperparameter optimization progress with different methods. The plot related to the genetic algorithm (a) visualizes the average fitness in each generation of 20 individuals and the fitness of the best individual. The plot associated with Bayesian optimization (b) shows the estimated values of the objective function at the points with the greatest expected improvement as well as the current best observations of the objective. In (c), the current observations and the lowest observed values so far obtained with random search are depicted.

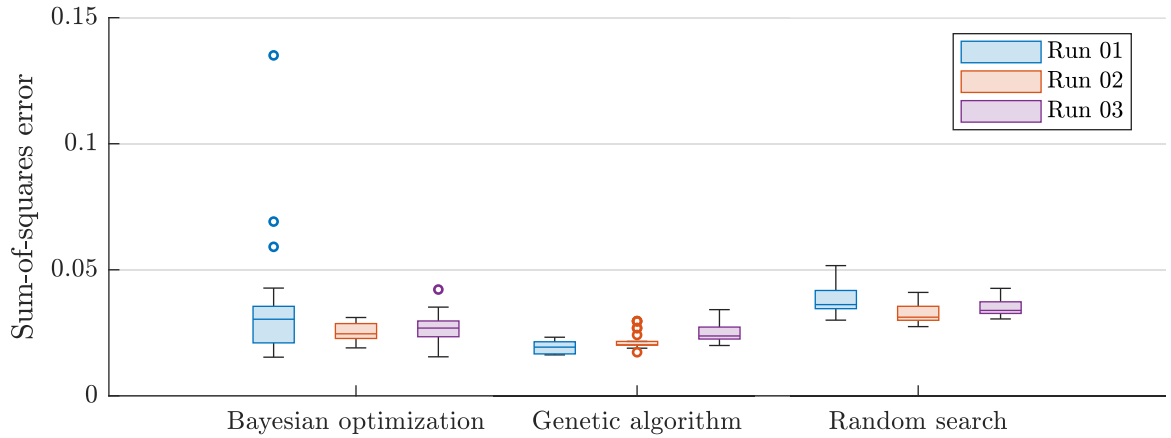
It has to be noted that the same subsets were used for each hyperparameter evaluation in Bayesian optimization and random search. This allows a more reliable comparison of different hyperparameter configurations. However, the HPO results may vary when using different splittings of the data. The genetic algorithm used in this work [15] redefines the data splitting after each generation in order to achieve a better generalization. However, even with the same subsets, the results of multiple runs may differ due to the stochastic nature of weight initialization, which leads to noisy observations of the objective function. To account for this phenomenon, the MATLAB function *bayesopt* [55] estimates a noise level during optimization. In genetic algorithms, the effects of weight initialization are also partially compensated without the need for further modifications. This is because characteristics are inherited from previous generations; thus, similar hyperparameter configurations are tested throughout the performed iterations [15]. In order to take the problem of weight initialization also into account in random search, a complete cross-validation process was performed five times with each hyperparameter setting. The median validation error of these five runs was then used as an evaluation basis for the goodness of a hyperparameter configuration. In Table 8.6, the hyperparameters optimized with the aforementioned methods are shown. Note

Table 8.6: Hyperparameter configurations of autoencoders optimized with different HPO methods: Bayesian optimization (BO), genetic algorithm (GA) and random search (RS)

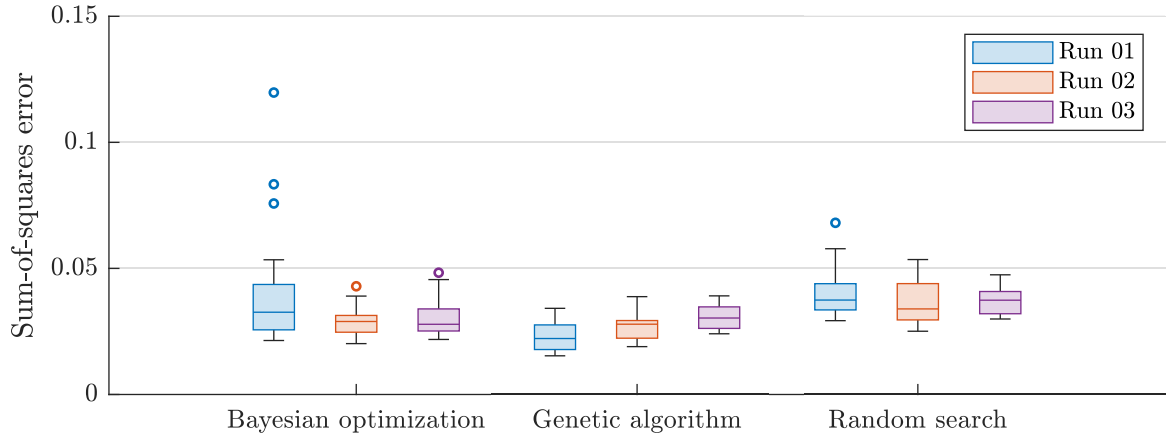
Autoencoder Type	**HPO Method**	**Run Nr.**	**Number Epochs**	**Neurons Encoder**	**Neurons Decoder**	**Learning Rate**	**MB Size**
biLSTM-AE	BO	1	82	20	52	5.867×10^{-2}	2
biLSTM-AE	BO	2	97	83	100	2.027×10^{-2}	6
biLSTM-AE	BO	3	88	47	51	3.403×10^{-2}	5
biLSTM-AE	GA	1	96	70	32	1.877×10^{-2}	3
biLSTM-AE	GA	2	83	45	29	3.672×10^{-2}	2
biLSTM-AE	GA	3	93	39	51	3.334×10^{-2}	6
biLSTM-AE	RS	1	53	63	58	1.271×10^{-2}	7
biLSTM-AE	RS	2	96	30	32	2.982×10^{-2}	12
biLSTM-AE	RS	3	93	81	62	0.432×10^{-2}	9
biLSTM-VAE	BO	1	61	59	55	3.116×10^{-2}	6
biLSTM-VAE	BO	2	100	15	30	5.329×10^{-2}	5
biLSTM-VAE	BO	3	99	64	62	3.009×10^{-2}	6
biLSTM-VAE	GA	1	96	57	47	2.571×10^{-2}	6
biLSTM-VAE	GA	2	86	12	27	5.187×10^{-2}	12
biLSTM-VAE	GA	3	75	43	63	2.695×10^{-2}	3
biLSTM-VAE	RS	1	90	69	12	1.587×10^{-2}	4
biLSTM-VAE	RS	2	87	75	54	2.491×10^{-2}	16
biLSTM-VAE	RS	3	98	36	100	1.540×10^{-2}	22

that, equal to Section 8.1, an older version of the framework was used, which did not include a

separate sigmoid or hyperbolic tangent layer in the decoder. Furthermore, the initial weights were determined using the Xavier initializer. With each hyperparameter setting shown in Table 8.6, networks were trained $n_{runs} = 25$ times to compensate for the stochastic nature of weight initialization. The training and validation errors per sample at the end of training were computed and are visualized in the form of boxplots in Figure 8.7 for biLSTM-AEs and Figure 8.8 for biLSTM-VAEs. The



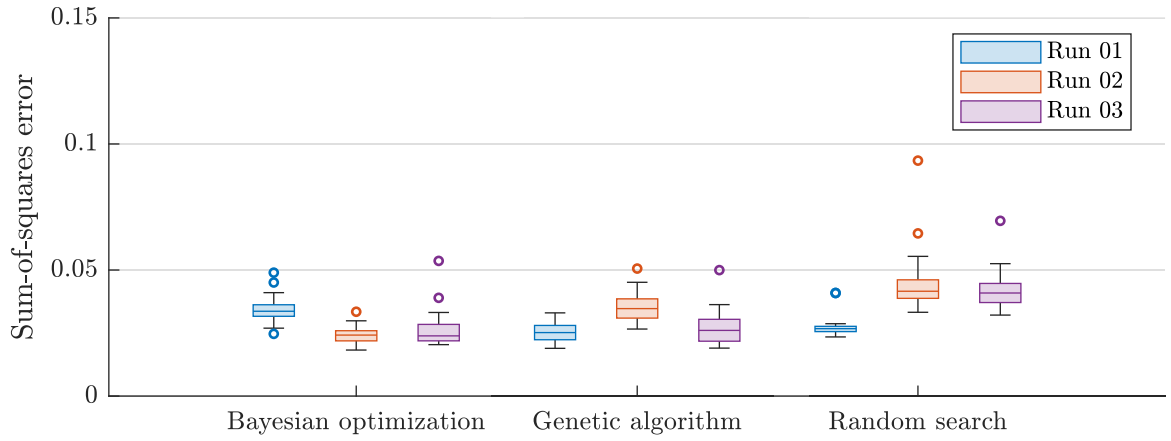
(a) Training error per sample



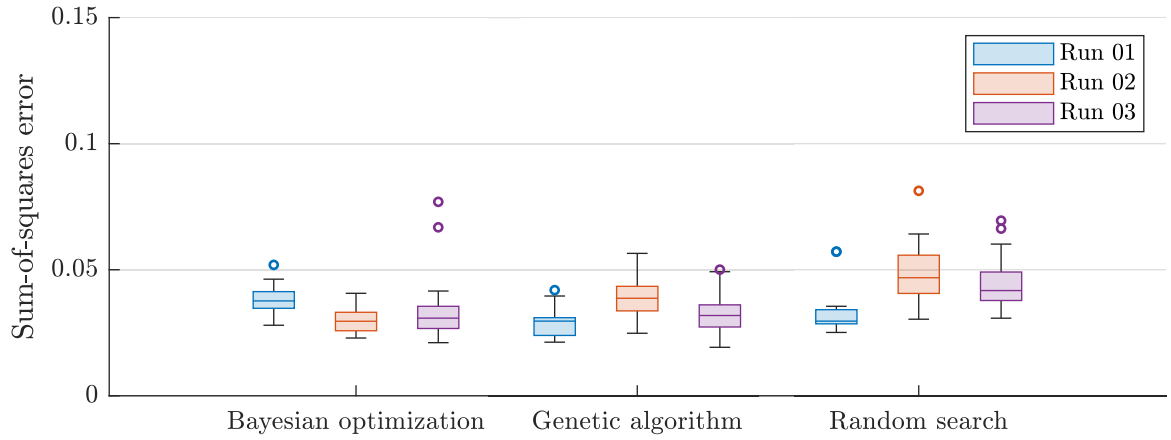
(b) Validation error per sample

Fig. 8.7: Sum-of-squares errors per sample of training samples (a) and validation samples (b) after the last epoch with biLSTM-AEs. The hyperparameters were optimized $n_{HPO} = 3$ times with each of the tested HPO methods, and $n_{runs} = 25$ test runs were performed with each hyperparameter configuration.

results for undercomplete autoencoders with hidden biLSTM layers shown in Figure 8.7 indicate that the genetic algorithm outperformed the other two HPO methods. It obtained hyperparameter configurations that led to low training and validation errors and a relatively small interquartile range over the performed test runs compared to the other two variants. The hyperparameters determined by Bayesian optimization resulted in lower median training and validation errors than the ones found with random search. However, the boxplots associated with the first Bayesian op-



(a) Training error per sample



(b) Validation error per sample

Fig. 8.8: Sum-of-squares errors per sample of training samples (a) and validation samples (b) after the last epoch with biLSTM-VAEs. The hyperparameters were optimized $n_{HPO} = 3$ times with each of the tested HPO methods, and $n_{runs} = 25$ test runs were performed with each hyperparameter configuration.

timization run include some outliers that significantly deviate from the median. That is, there is a high risk for performance fluctuations.

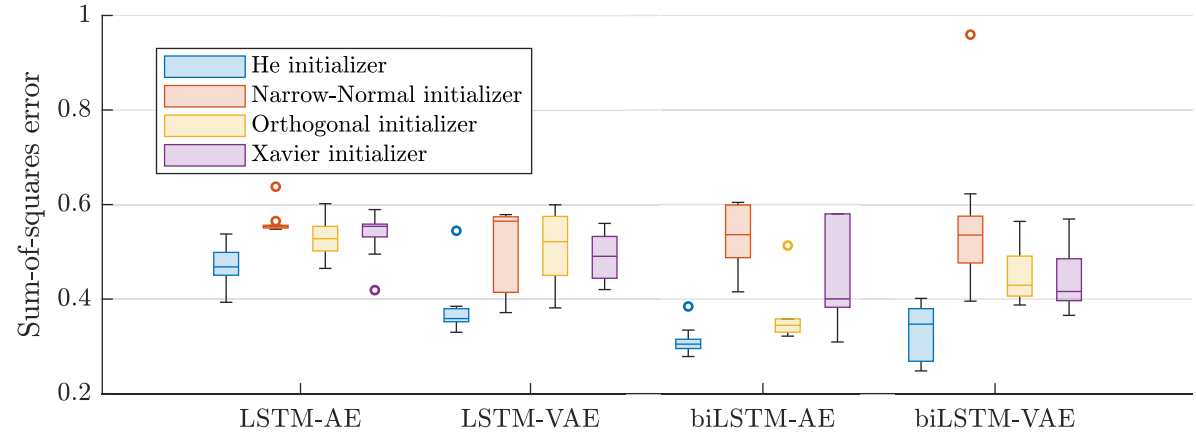
From the results of biLSTM-VAEs shown in Figure 8.8, no definitive conclusion can be drawn as to which HPO method performed best. When taking all $n_{HPO} = 3$ runs into consideration, the genetic algorithm and Bayesian optimization outperformed the random search method. However, it can be seen that in the first run of random search, a well-suited hyperparameter configuration was obtained. It led to lower training and validation errors than some of the hyperparameter settings provided by the genetic algorithm or Bayesian optimization. Thus, it is also possible to find suitable hyperparameters with random search, although a higher number of optimization runs than with the other two methods is necessary on average [58].

8.3 Evaluation of Weight Initializing Methods

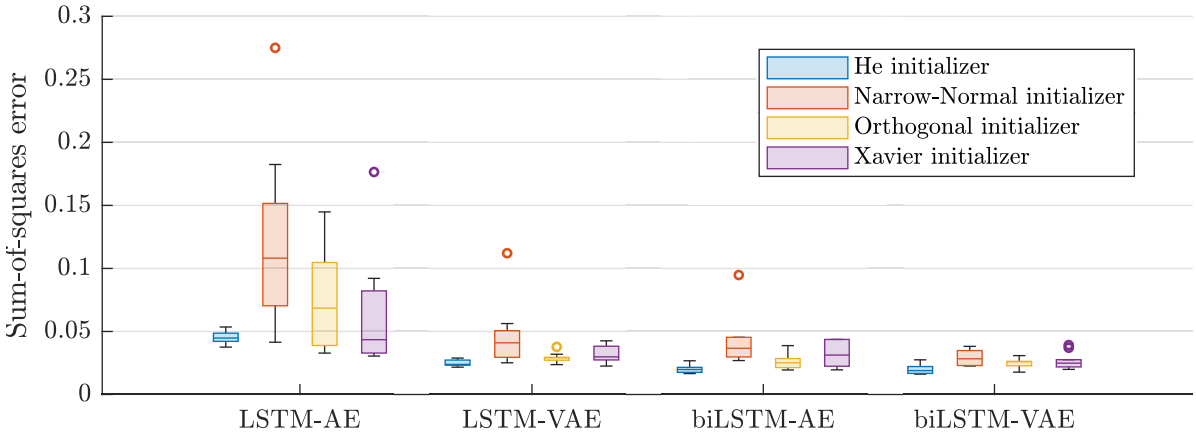
The initial values of the weight parameters can have a significant impact on the training performance and duration [52]. This inspired the development of various weight initializing methods [52], as described in more detail in Section 3.3.3. In order to determine the most suitable initializer for the networks used in the context of this work, some methods implemented in MATLAB were tested on autoencoders with different architectures. These were the narrow-normal initializer [55], the He initializer [54], the Xavier initializer [53], and a method MATLAB refers to as orthogonal initializer [57]. The networks were set up and trained using the optimized hyperparameters presented in Section 8.1.1. An up-to-date version of the framework, as described in Section 7.3.1, was used. In order to account for the stochastic nature of weight initialization, each network was trained $n_{runs} = 10$ times. A set of 80 random data samples considered non-anomalous by the statistical model was divided into a training set of 72 samples and a validation set of 8 samples. Again, the input signal of dimension $m = 6$ was mapped to a latent space with dimension $p = 3$, using the following data channels: depth, pulldown force, vibrator amperage, vibrator frequency, vibrator temperature, and weight. The training errors per sample in the learning process for each initializing method and network after $n_{ep} = 1$ and $n_{ep} = 45$ epochs, as well as after the training was finished, are shown in Figure 8.9. The corresponding validation errors per sample can be seen in Figure 8.10.

Across all autoencoder types and architectures, the He initializer delivered the most suitable initial weight parameters. In terms of training and validation error per sample after $n_{ep} = 1$ epoch, it significantly outperformed all other initializing methods. After $n_{ep} = 45$ epochs and after the last epoch, the median training and validation errors per sample of the LSTM-AE were as low or slightly lower when using the Xavier initializer instead of the He initializer. However, the variation in the results across different runs was noticeably higher, and the network seemed more prone to poor learning performances with slow convergence. The technique that initializes the weight parameters with an orthogonal matrix also achieved good results after the last epoch across all architectures. Networks whose parameters were initialized with the narrow-normal method performed worst, especially if they contained LSTM layers.

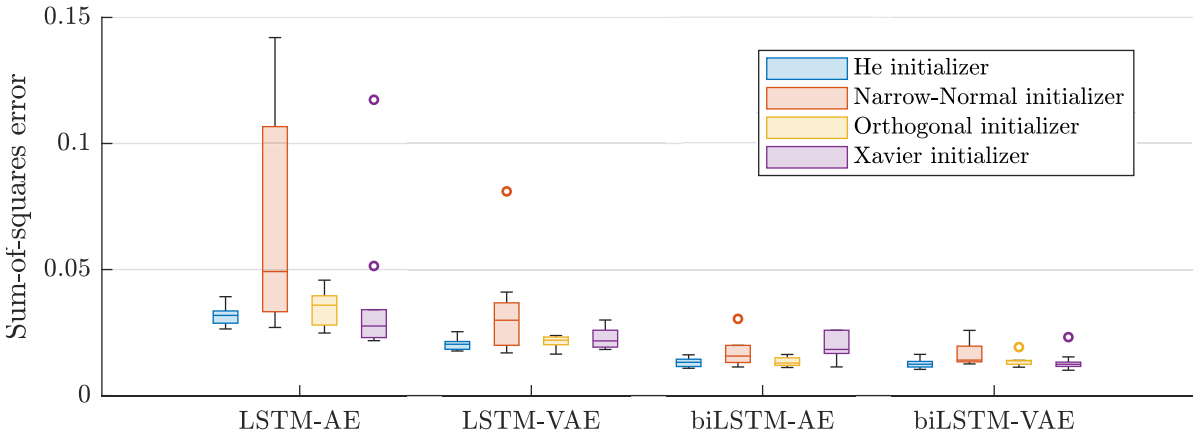
Figure 8.11 shows plots of the sum-of-squares validation error per sample at each epoch for the tested initializers and autoencoder architectures. In each case, the run with the fifth lowest validation error after training out of $n_{runs} = 10$ runs is shown. These plots confirm the formerly made deductions that the He initializer usually enables the fastest convergence in the training process, while networks typically perform worst when using the narrow-normal initializer. It can also be seen that initialization with the method of Xavier can lead to instabilities in the training process. These peaks may slow down the convergence during learning and result in poor network parameters if they appear at the end of the training process.



(a) Training error per sample after $n_{ep} = 1$ epoch

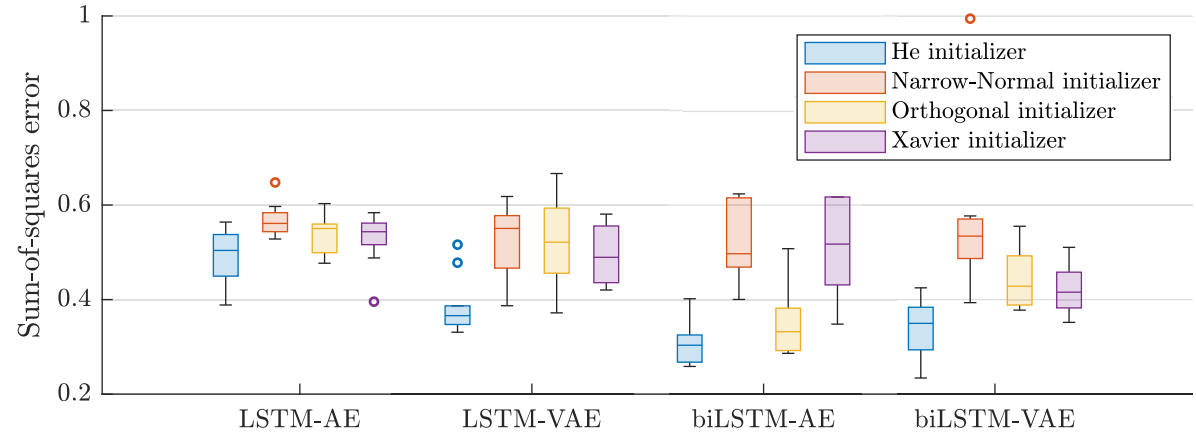


(b) Training error per sample after $n_{ep} = 45$ epochs

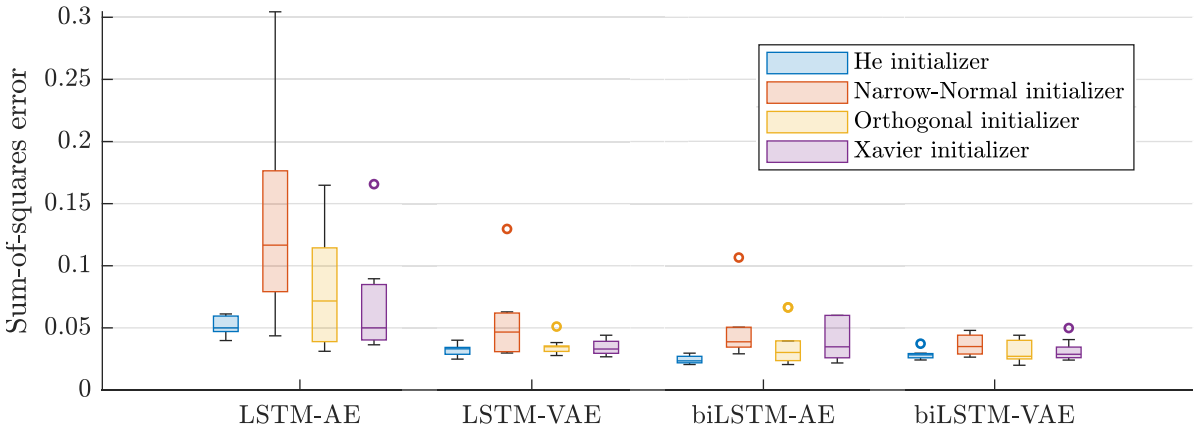


(c) Training error per sample after the last epoch

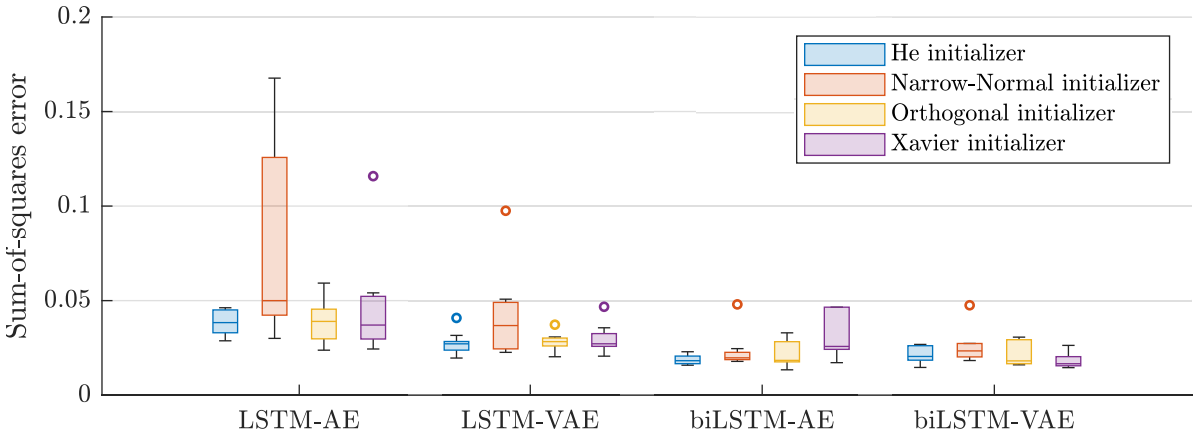
Fig. 8.9: Boxplots of the sum-of-squares training error per sample of different autoencoders after a particular number of epochs using four different weight initializing methods. With each combination of initializing methods and autoencoder architectures, $n_{runs} = 10$ runs were performed.



(a) Validation error per sample after $n_{ep} = 1$ epoch

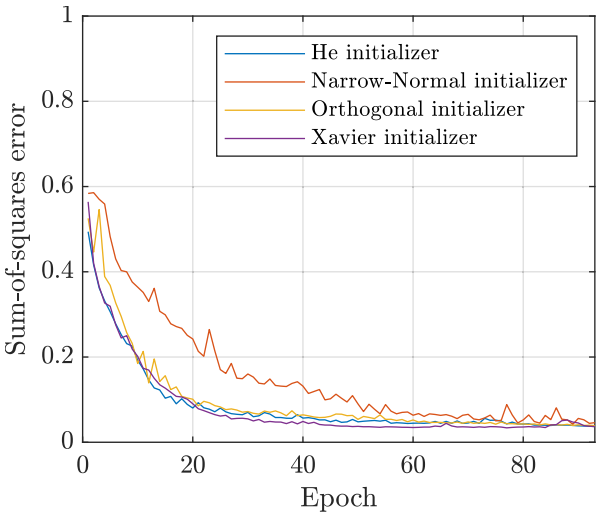


(b) Validation error per sample after $n_{ep} = 45$ epochs

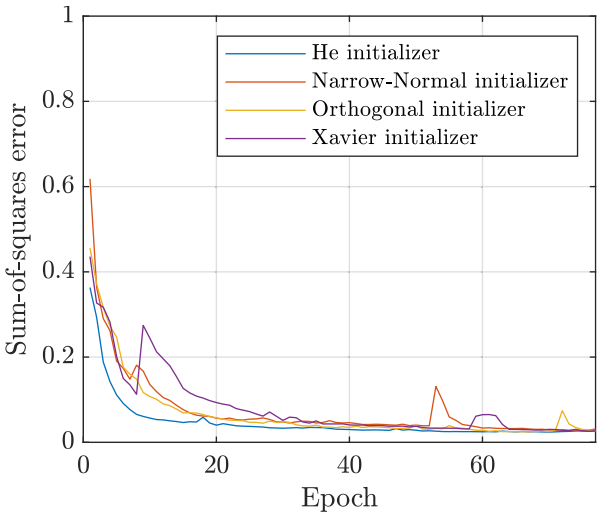


(c) Validation error per sample after the last epoch

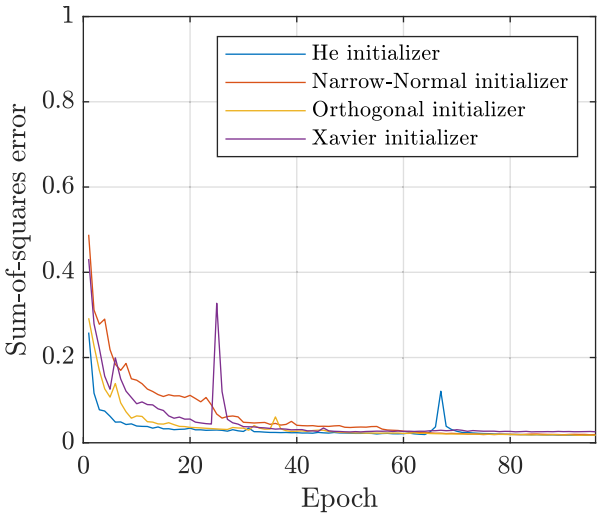
Fig. 8.10: Boxplots of the sum-of-squares validation error per sample of different autoencoders after a particular number of epochs using four different weight initializing methods. With each combination of initializing methods and autoencoder architectures, $n_{runs} = 10$ runs were performed.



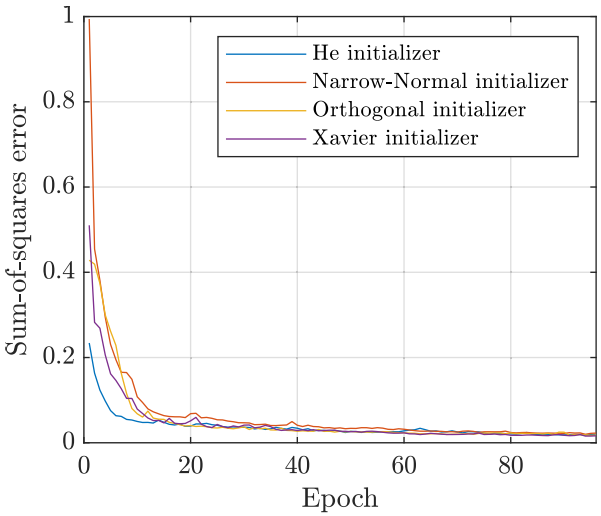
(a) Validation error per sample of LSTM-AE



(b) Validation error per sample of LSTM-VAE



(c) Validation error per sample of biLSTM-AE



(d) Validation error per sample of biLSTM-VAE

Fig. 8.11: Plots of the sum-of-squares validation error per sample with different autoencoders using four initializing methods. Out of $n_{runs} = 10$ runs, the one with the 5th lowest validation error per sample at the end of training is shown.

In summary, regardless of the type or architecture of the autoencoder, the He initializer seems to provide the most suitable initial weights that result in low errors, especially in the first epochs of the training process. The Xavier initializer might enable a lower median error for LSTM-AE networks after training. However, the variation in the results of different runs is noticeably higher compared to networks initialized with the method of He.

8.4 Optimizing Models for Phase-Wise Anomaly Detection

In consideration of the results presented in Section 8.1 - 8.3, separate autoencoders were set up for the process phases that are of interest. These are the penetration and the compaction phase, as shown in Figure 7.1 and 7.2. The autoencoder architectures that achieved the best performance in the experiments shown in Section 8.1 were undercomplete and variational autoencoders with one hidden biLSTM layer. As the variational autoencoder had already been used in previous work at the Chair of Automation [15], it was also chosen for the tests presented in this section. Furthermore, the weights were initialized with the He initializer, which was found to allow the fastest convergence in training, as shown in Section 8.3. The hyperparameters were optimized with a genetic algorithm. This method, together with Bayesian optimization, provided the most suitable hyperparameter configurations (see Section 8.2) and had already been successfully applied to the same data in other work [15]. Based on the results of hyperparameter optimization performed in the previous test series, the hyperparameter domains were adapted, as shown in Table 8.7. The limits for the number of epochs were raised, and the upper limit for the mini-batch size was lowered. Additionally, the lower limits for the number of neurons were increased. The parameters of the genetic algorithm

Table 8.7: Hyperparameters to optimize and corresponding bounded domains for phase-wise anomaly detection models

Hyperparameter	**Domain**	**Lower Bound**	**Upper Bound**
Nr. of epochs	discrete	50	200
Nr. of neurons in encoder	discrete	10	100
Nr. of neurons in decoder	discrete	10	100
Learning rate	continuous	3×10^{-3}	1×10^{-1}
Mini-batch size	discrete	2	20

were set as shown in Table 8.2, except for the maximum number of generations. This value was set to $n_{gen} = 10$ in order to allow an extended search for optimal hyperparameters. The results of the HPO performed with 80 data samples considered non-anomalous by the statistical model are shown in Table 8.8. Equal to the test series in Section 8.1 - 8.3, the latent dimension was set

Table 8.8: HPO results of autoencoders for phase-wise anomaly detection

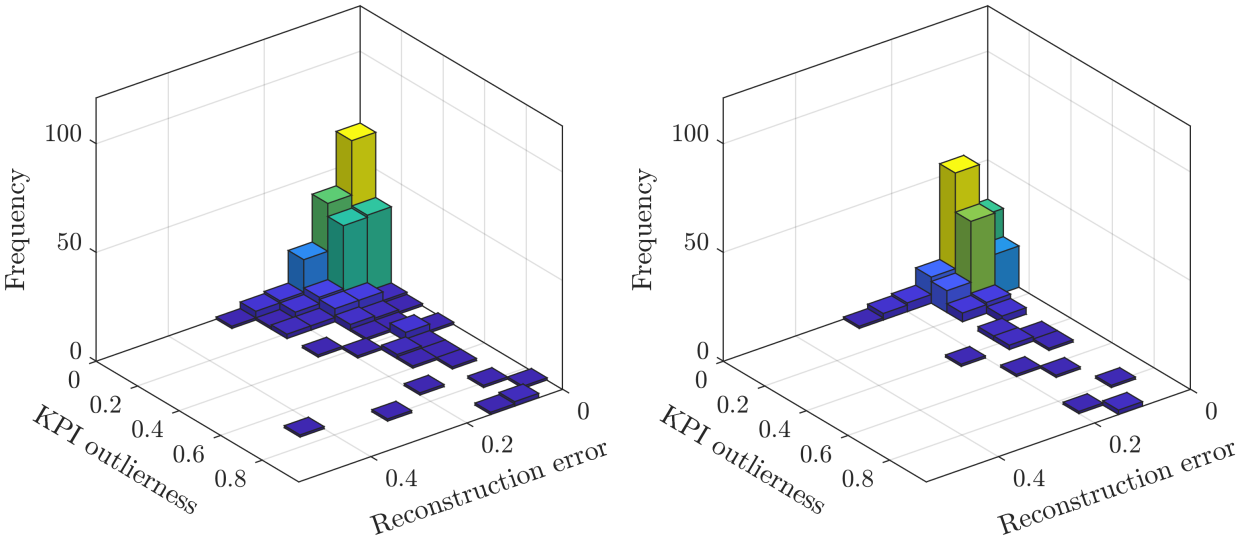
Autoencoder Type	**Process Phase**	**Number Epochs**	**Neurons Encoder**	**Neurons Decoder**	**Learning Rate**	**MB Size**
biLSTM-VAE	Penetration	151	69	55	2.340×10^{-2}	13
biLSTM-VAE	Compaction	192	52	53	1.727×10^{-2}	7

to $p = 3$, and data of the following $m = 6$ channels was fed into the networks: depth, pulldown force, vibrator amperage, vibrator frequency, vibrator temperature, and weight. The data set that

was analyzed consisted of 272 samples, each collected during the creation of a subsurface column at the same site. The latest version of the framework (see Section 7.3.1) was used to set up and train the networks. Two hybrid learning models, as described in Section 7.1, were applied to the data samples. For the statistical model, 42 key performance indicators for the separate phases or the process as a whole were defined at the Chair of Automation [18]. The criterion for a sample being an outlier w.r.t. the quantity measured by a particular KPI is described in Section 7.4.

8.4.1 Parallel Hybrid Model

In a parallel hybrid model, the autoencoder receives a random portion of the data set for training, which is not pre-labeled by a statistical model [122]. Via a procedure similar to 4-fold cross-validation, each of the 272 samples in the data set was used for training and testing. First, the data set was randomly divided into four subsets. In a cyclic manner, training was then carried out with one subset and testing with the remaining subsets. This differs from classical k-fold cross-validation, where only one subset would be used for testing and the remaining ones for training in each iteration [34]. One reason for this type of approach is to have a size of the training set similar to the one used for hyperparameter optimization. Furthermore, each sample is tested three times, and the error is averaged, which should lead to more reliable results. The averaged errors in the penetration and the compaction phase were summed up for each sample. After the first completed run, i.e., after each sample had been used for training once and for testing three times, the samples that had been labeled as outliers were eliminated from the data set, and a second run was performed. The results of statistical outlier detection performed on the same data set at the Chair of Automation [18] were provided for this work. Figure 8.12 shows bivariate histograms of the outlierness obtained with KPIs and the reconstruction error after the first and the second run for all test samples. As can be seen, the majority of data samples was considered non-anomalous by both methods, i.e., the reconstruction error and outlierness were low. The fact that these samples were analyzed with two independent models strengthens the assumption that these are indeed non-anomalous. A lot of samples, however, exhibited a high statistical outlierness but a rather low reconstruction error. A possible explanation is that the statistical model was applied to data that still contained dead time segments. Thus, it also considered an unusual process duration or long pauses as anomalous behavior. Anomalies of this type might be related to efficiency problems but usually do not indicate faulty columns. The opposite case of samples with a high reconstruction error but low statistical outlierness is of higher interest. The anomalies in these samples can e.g. be contextual deviations, i.e., data points with atypical values compared to adjacent points in the time-series. These data points may not be anomalous w.r.t. the time-series as a whole, but in a certain context. The machine learning model does not provide information on why it fails to reconstruct the signal of particular data samples. Hence, the help of an expert with knowledge about the analyzed process is necessary to determine the cause and if it is related to a failure in the process.



(a) Histogram of first run

(b) Histogram of second run

Fig. 8.12: Bivariate histograms of the reconstruction error and the outlierness obtained with KPIs [18]. The training was performed with unlabeled data via 4-fold cross-validation, and the average test error was computed for each sample. For the second run, the anomalous samples found in the first run were eliminated from the data set.

8.4.2 Parallel Serial Hybrid Model

In a parallel serial hybrid learning model, the autoencoder receives a training set consisting of samples that were pre-labeled as non-anomalous by the statistical model [122]. The networks for both process phases were trained on a random subset of 80 data samples without anomalous KPI values, and tests were performed with the whole data set. The reconstruction errors of both phases are plotted for all test samples in Figure 8.13. Additionally, the right-skewed error distributions and the thresholds for separating non-anomalous data from outliers are shown. It can be seen that the majority of outliers was only considered anomalous w.r.t. to one of the process phases. Therefore, detecting anomalies in only one phase could result in many abnormal samples going undetected.
Again, the summed reconstruction error of both phases is compared to the outlierness obtained by the statistical model, as shown in Figure 8.14a. It can be seen that a substantially higher number of data samples had a reconstruction error and an outlierness in the interval with the lowest values than it was the case with the parallel hybrid model (see Figure 8.12). This is not surprising, as the training samples had also been left in the test set. Furthermore, some data samples exhibited a high statistical outlierness but a low reconstruction error. As explained in Section 8.4.1, a reason might be the elimination of dead time segments before the data was fed into the autoencoder, while the KPIs were computed with data that still contained them. Samples with a low statistical outlierness but a high reconstruction error have to be inspected by an expert. The knowledge gained in this process might be helpful to optimize the hybrid learning tool further. The histogram in Figure 8.14b compares the reconstruction errors computed by an autoencoder in a parallel serial

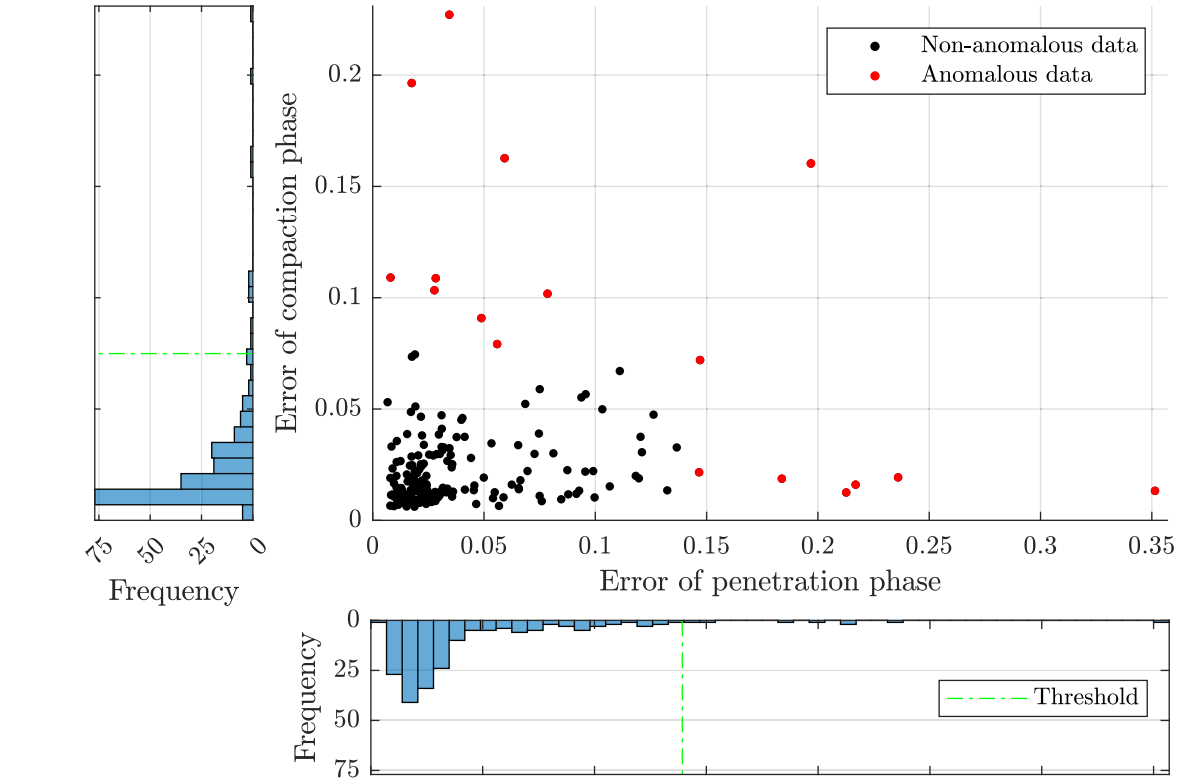
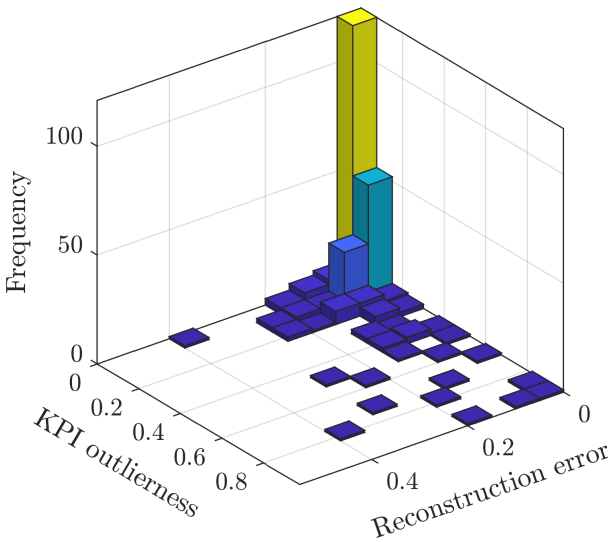
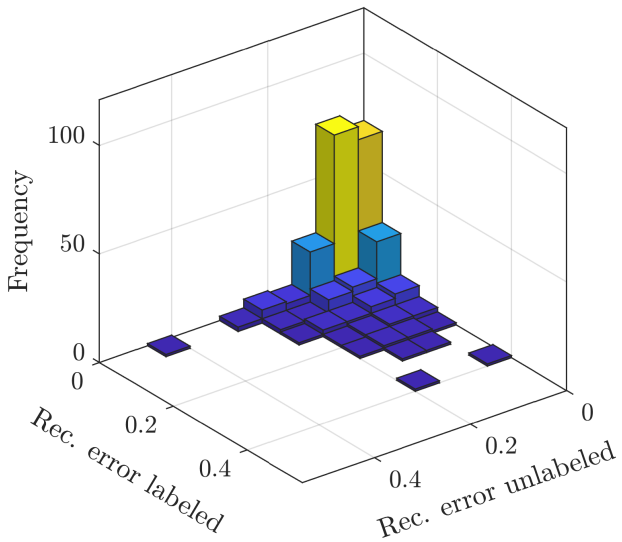


Fig. 8.13: Phase-wise error plot of the test data samples with the corresponding histograms of the error distributions and threshold values for separating normal from anomalous data samples.



(a) Outlierness and reconstruction error with parallel serial hybrid model



(b) Reconstruction error with parallel hybrid model or parallel serial hybrid model

Fig. 8.14: Subfigure (a) shows a bivariate histogram of the summed reconstruction error when training with a labeled data set as well as the outlierness determined via KPIs [18] for all test samples. Subfigure (b) is a bivariate histogram of the reconstruction errors obtained with autoencoders that were trained with labeled (parallel serial hybrid) or unlabeled (parallel hybrid) data sets.

hybrid model with the errors obtained by an autoencoder in a parallel hybrid model. It can be seen that although the majority of samples has a low reconstruction error with both approaches, the errors differ significantly between the two hybrid learning approaches for some other time-series files. This suggests that the pre-labeling of data sets with statistical methods has a noticeable influence on the machine learning model.

Which hybrid model structure works best remains to be investigated. At the current stage of research, the aim is to investigate why the autoencoder fails at reconstructing the signal of particular samples and thus considers them anomalous. Figure 8.15, for example, shows the signal of a sample collected during compaction, which was labeled as non-anomalous by the machine learning model. The reconstruction error is low, and the data does not show any abnormalities from a layman's point of view. The signal plot in Figure 8.16, however, exhibits a suspicious pattern in the depth channel after approx. 235 s until the end of the process. At first glance, it seems like the compaction was not carried out properly in the end phase, which might result in loose gravel in the near-surface zone of the column. However, by taking a look at the raw signal, it gets clear

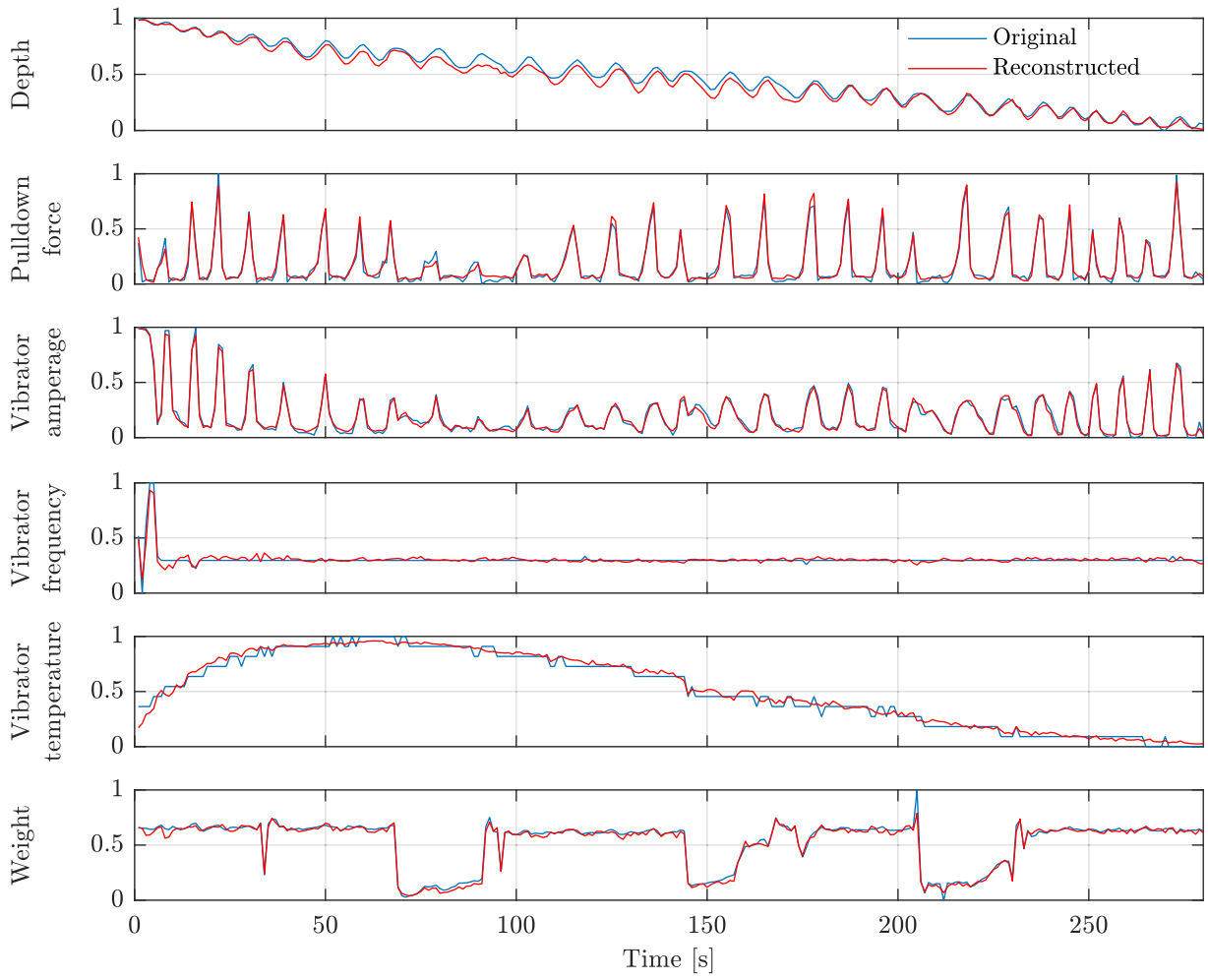


Fig. 8.15: Original and reconstructed data of the compaction phase of an exemplary MVTS test sample. The low reconstruction error indicates that no process failure occurred during compaction.

that the process was actually continued after the last time instant shown in the plot, and just some error in preprocessing of the input data occurred. Still, it shows that the autoencoder is able to detect unusual patterns in the data that may be difficult to spot with statistical methods. Another interesting anomaly can be found in the temperature channel after approx. 5 s. The discontinuity was introduced by removing dead time segments from the data in preprocessing. The temperature continued to rise even though the process was in idle mode. In the raw data plots, it can be seen that the vibrator was still running during a part of the pause. Presumably, this does not affect the quality of the column but results in unnecessary energy consumption and lower process efficiency. A discontinuity could also occur if the temperature decreases during the standstill, which might result in the machine learning model considering the corresponding sample anomalous too. Thus, the final interpretation of the results is up to experts with knowledge about the process.

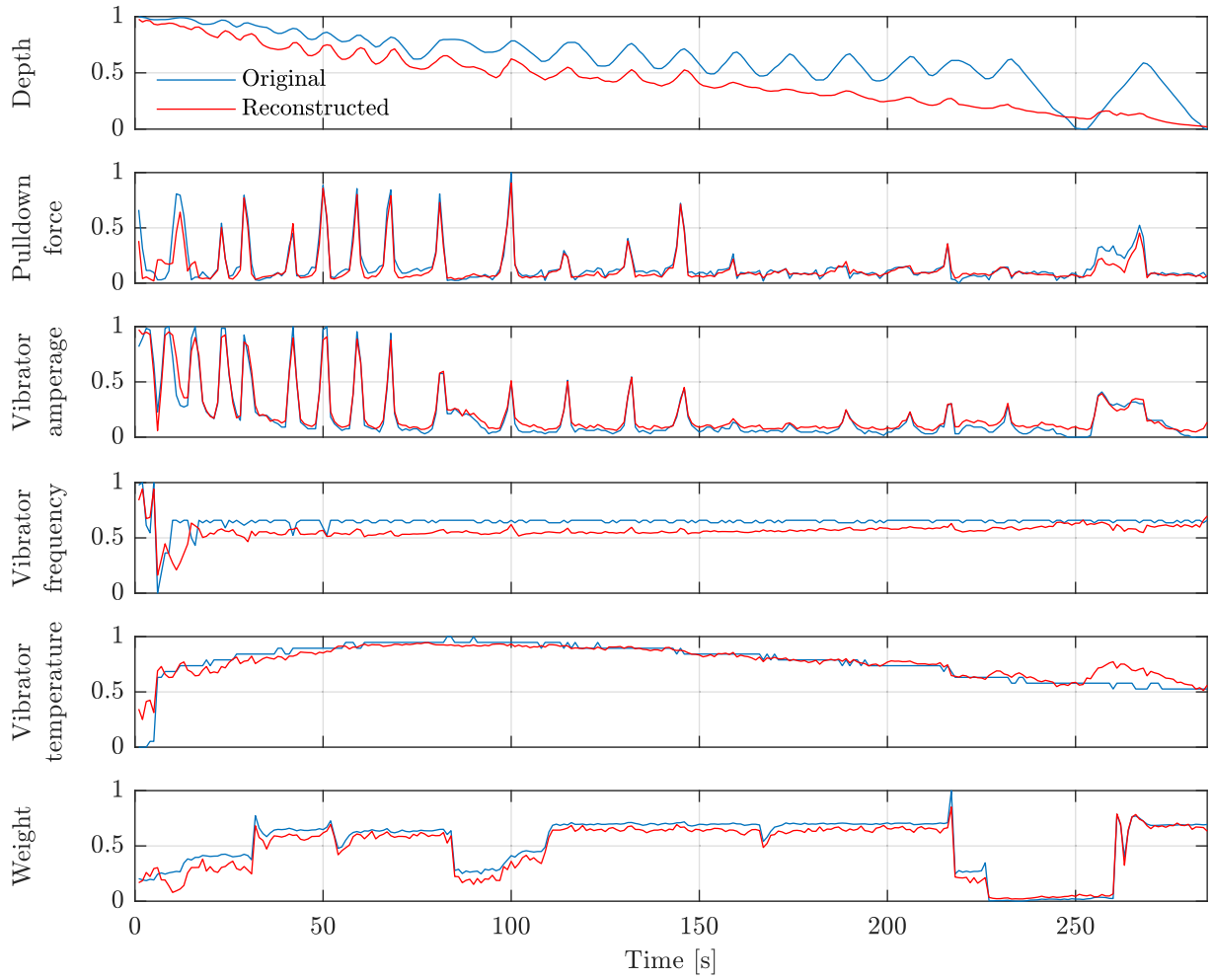


Fig. 8.16: Original and reconstructed data of the compaction phase of an exemplary MVTS test sample. The high reconstruction error indicates anomalous behavior during compaction.

Chapter 9
Conclusion and Future Work

The results of the test series presented in Section 8.1 - 8.3 provide valuable information for the optimization of autoencoders used for anomaly detection in MVTS data. It was found that architectures with one hidden biLSTM layer in the encoder and the decoder seem to enable a better reconstruction of non-anomalous data samples than their counterparts with hidden unidirectional LSTM layers. Furthermore, they allow to easier distinguish them from samples that exhibit anomalies. The use of additional hidden layers did not result in an improvement of the model.
The optimization of hyperparameters was found to enhance the reconstruction performance of an autoencoder considerably. Methods that focus on promising regions in the search space based on the results of previous runs [58], i.e., Bayesian optimization and genetic algorithms, outperformed the less elaborate random search technique.
Weight initialization was found to have a significant impact on convergence in the learning process. The best results were achieved with the He initializer throughout all tested autoencoder types and architectures. For particular networks, the Xavier and the orthogonal initializer seemed to be good alternatives.
The optimized machine learning model detected unusual patterns in data samples that were not considered anomalous by the statistical model. It remains to be investigated what caused the autoencoder to fail at reconstructing these samples and if it is related to a process failure. The machine learning model is not intended to replace the statistical tool for outlier detection but to extend its capabilities. To some degree, it takes the correlation between different data channels into account and detects anomalies that are difficult to spot with other techniques [17]. This should make the hybrid learning model a more reliable tool for anomaly detection than the statistical model alone.
Future work to further improve the existing machine learning model might include a modification of the rescaling process. As mentioned in Section 7.3.1.2, fixed scaling parameters are not recommended due to the varying soil conditions within the site and inter-site, which affect the analyzed process. Combining the data samples with the GPS coordinates makes it possible to derive scaling parameters from the process data collected at nearby columns. For example, the maximum reached depth at different GPS coordinates, as shown in Figure 9.1, can be used to infer a model on which the definition of scaling parameters can be based [134]. For setting the scaling parameters of other

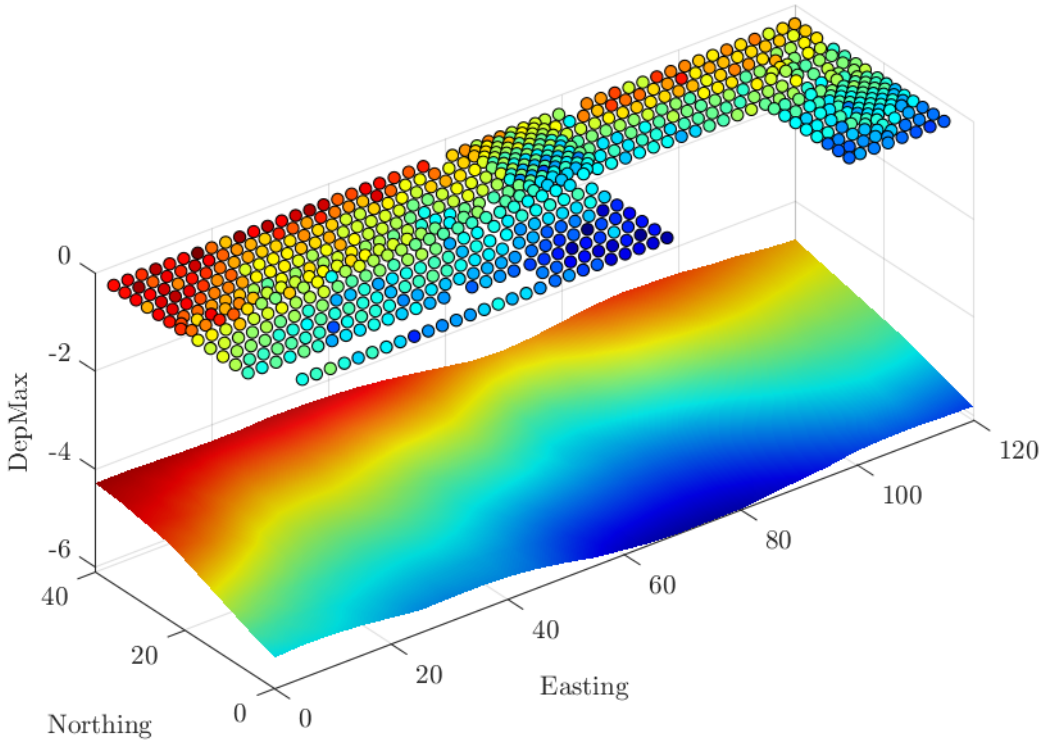


Fig. 9.1: Maximum reached depth during vibro ground improvement at different GPS coordinates of a site and inferred model. Reprinted from [134].

variables, such as the vibrator temperature, the limits of the machinery can be used.

A further approach for improving the model is the derivation of additional variables from the measured data. This could, for example, be the work performed during penetration or compaction, which is currently a topic of research at the Chair of Automation.

A possibility to improve a VAE-based model is to determine an appropriate weighting factor for the KL divergence in the loss function. This allows balancing the error-based term and the regularization term according to the requirements [92], [93], [94].

List of Figures

2.1 Confusion matrix for binary classification tasks. Adapted from [28]. 9
2.2 Exemplary receiver operating characteristic (ROC) curves. Adapted from [30]. . . . 10

3.1 Schematic diagram of a neuron. Adapted from [34]. 12
3.2 Common activation function types used in neural networks. Adapted from [35]. . . 14
3.3 Exemplary graph of a fully connected feedforward network with one hidden layer. Adapted from [34]. 15
3.4 Schematic diagram of an RNN with one recurrent layer. Adapted from [38]. 17
3.5 Schematic diagram of an RNN with two recurrent layers. Adapted from [37]. . . . 18
3.6 Schematic diagram of an LSTM architecture. Adapted from [42]. 19
3.7 Schematic diagram of a biLSTM architecture unfolded across time. Adapted from [44]. 22
3.8 Exemplary plots related to Bayesian optimization using a Gaussian process. Adapted from [64]. 31

5.1 Schematic diagram of an undercomplete autoencoder. Adapted from [78]. 41
5.2 Schematic diagram of a variational autoencoder. Adapted from [78]. 44

6.1 Time-series plots of a process without anomalous behavior and with different types of anomalies. 49

7.1 Phases of the vibro ground improvement process. Adapted from [16]. 55
7.2 Data of $m = 9$ channels collected from a ground improvement process, segmented into the main phases. 56
7.3 Comparison of depth channel data with and without dead time. 57
7.4 Exemplary schematic diagram of an autoencoder based on the framework of the Chair of Automation [124]. The number of neurons in each layer, as well as the number and types of the hidden layers, can be set by the user and thus may differ from this graphic. 60

7.5 Right-skewed distribution over the reconstruction errors of test samples with thresholds for outlier detection. 62
7.6 Exemplary plot of reconstruction errors of test data samples and associated statistical measures. 63
7.7 Heat map of different KPIs evaluated for a set of samples (points) collected at the same site. Adapted from [18] 64

8.1 Statistical measures of sum-of-squares sample errors obtained with different autoencoder architectures containing one hidden layer in encoder and decoder for $n_{runs} = 25$ test runs. 69
8.2 Statistical measures of sum-of-squares sample errors obtained with different autoencoder architectures containing two hidden layers in encoder and decoder for $n_{runs} = 25$ test runs. 71
8.3 Statistical measures of sum-of-squares sample errors obtained with autoencoders with different architectures and number of hidden layers for $n = 25$ test runs. 72
8.6 Plots of the hyperparameter optimization progress with different methods. 73
8.7 Sum-of-squares training and validation error per sample for biLSTM-AEs, whose hyperparameters were optimized with three different HPO methods in $n_{HPO} = 3$ runs each. For each hyperparameter configuration $n_{runs} = 25$ test runs were performed. 75
8.8 Sum-of-squares training and validation error per sample for biLSTM-VAEs, whose hyperparameters were optimized with three different HPO methods in $n_{HPO} = 3$ runs each. For each hyperparameter configuration $n_{runs} = 25$ test runs were performed. 76
8.9 Boxplots of training error per sample of different autoencoders after a particular number of epochs using four different weight initializing methods. 78
8.10 Boxplots of validation error per sample of different autoencoders after a particular number of epochs using four different weight initializing methods. 79
8.11 Plots of the validation error per sample of different autoencoders using four different initializing methods. 80
8.12 Bivariate histograms of the reconstruction error and the outlierness obtained with KPIs when training without a labeled data set. 83
8.13 Phase-wise error plot of the test data samples with the corresponding histograms of the error distributions and threshold values for separating normal from anomalous data samples. 84
8.14 Bivariate histogram of the reconstruction error and the outlierness obtained using KPIs when training with a labeled data set, and bivariate histogram comparing the errors of networks trained with labeled and unlabeled data sets. 84

8.15 Original and well reconstructed data of the compaction phase of an exemplary MVTS test sample. 85
8.16 Original and bad reconstructed data of the compaction phase of an exemplary MVTS test sample. 86

9.1 Maximum reached depth during vibro ground improvement at different GPS coordinates of a site and inferred model. Reprinted from [134]. 88

List of Tables

8.1 Hyperparameters to optimize and corresponding bounded domains for architectures with one hidden layer . . . 67
8.2 Parameters of the genetic algorithm for architectures with one hidden layer . . . 67
8.3 HPO results of autoencoder architectures with one hidden layer in encoder and decoder . . . 68
8.4 Hyperparameters to optimize and corresponding bounded domains for architectures with two hidden layers . . . 70
8.5 HPO results of autoencoder architectures with two hidden layers in encoder and decoder . . . 70
8.6 Hyperparameter configurations of autoencoders optimized with different HPO methods: Bayesian optimization (BO), genetic algorithm (GA) and random search (RS) . . . 74
8.7 Hyperparameters to optimize and corresponding bounded domains for phase-wise anomaly detection models . . . 81
8.8 HPO results of autoencoders for phase-wise anomaly detection . . . 81

References

[1] R. Hamzeh, L. Thomas, J. Polzer, X. W. Xu, and H. Heinzel, "A Sensor Based Monitoring System for Real-Time Quality Control: Semi-Automatic Arc Welding Case Study," *Procedia Manufacturing*, vol. 51, pp. 201–206, 2020.

[2] P. Kamat and R. Sugandhi, "Anomaly Detection for Predictive Maintenance in Industry 4.0 - A survey," *E3S Web of Conferences*, vol. 170, 2020. DOI: 10.1051/e3sconf/202017002007.

[3] G. Chryssolouris, N. Papakostas, and D. Mavrikios, "A perspective on manufacturing strategy: Produce more with less," *CIRP Journal of Manufacturing Science and Technology*, vol. 1, no. 1, pp. 45–52, 2008.

[4] P. Stavropoulos, D. Chantzis, C. Doukas, A. Papacharalampopoulos, and G. Chryssolouris, "Monitoring and Control of Manufacturing Processes: A Review," *Procedia CIRP*, vol. 8, pp. 421–425, 2013.

[5] S. Jeschke, C. Brecher, H. Song, and D. B. Rawat, *Industrial Internet of Things: Cybermanufacturing Systems* (Springer Series in Wireless Technology). Cham, CH: Springer International Publishing, 2017.

[6] C. C. Aggarwal, *Outlier Analysis* (Springer eBook Collection Computer Science), 2nd ed. Basel, CH: Springer International Publishing, 2017.

[7] N. Vandeput, *Data science for supply chain forecasting*, 2nd ed. Berlin, GE: De Gruyter, 2021.

[8] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, 2009. DOI: 10.1145/1541880.1541882.

[9] M. Hoarau, *Time Series Analysis on AWS: Learn how to build forecasting models and detect anomalies in your time series data*. Birmingham, UK: Packt Publishing, 2022.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[11] G. Pang, C. Shen, L. Cao, and A. Hengel, "Deep Learning for Anomaly Detection: A Review," *ACM Computing Surveys*, vol. 54, pp. 1–38, 2021.

[12] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural computation*, vol. 9, pp. 1735–1780, 1997.

[13] A. Graves and J. Schmidhuber, "Framewise Phoneme Classification With Bidirectional LSTM and Other Neural Network Architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005.

[14] J. Kalliola, J. Kapočiūtė-Dzikienė, and R. Damaševičius, "Neural Network Hyperparameter Optimization For Prediction Of Real Estate Prices In Helsinki," *PeerJ. Computer science*, vol. 7, e444, 2021.

[15] A. Terbuch, "LSTM Hyperparameter Optimization: Impact of the Selection of Hyperparameters on Machine Learning Performance when applied to Time Series in Physical Systems," M.S. Thesis, Chair of Automation, Montanuniversitaet Leoben, Leoben, AT, 2021.

[16] Keller UK Ltd, *Vibro: Where ground improvement begins*. [Online]. Available: https://www.keller.co.uk/sites/keller-uk/files/2019-03/vibro-techniques-brochure-keller-uk.pdf (visited on 04/23/2022).

[17] A. Terbuch, P. O'Leary, and P. Auer, "Hybrid Machine Learning for Anomaly Detection in Industrial Time-Series Measurement Data," in *2022 IEEE International Instrumentation and Measurement Technology Conference (I2MTC 2022)*, Ottawa, CA, 2022.

[18] N. Khalili-Motlagh-Kasmaei, D. Ninevski, P. O'Leary, C. J. Rothschedl, V. Winter, and A. Zöhrer, "A Digital Twin for Deep Vibro Ground Improvement," in *International Conference on Deep Foundations and Ground Improvement: Smart Construction for the Future (DFI-EFFC 2022)*, Berlin, GE, 2022.

[19] A. Smola and S. Vishwanathan, *Introduction to Machine Learning*. Cambridge, UK: Cambridge University Press, 2008.

[20] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, UK: Cambridge University Press, 2014.

[21] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill Science/Engineering/Math, 1997.

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, [Online]. Available: http://www.deeplearningbook.org. (visited on 05/02/2022).

[23] D. Greene, P. Cunningham, and R. Mayer, "Unsupervised Learning and Clustering," in *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*, ser. SpringerLink Bücher, M. Cord and P. Cunningham, Eds., Berlin, GE: Springer, 2008, pp. 51–90.

[24] G. Dong, G. Liao, H. Liu, and G. Kuang, "A Review of the Autoencoder and Its Variants: A Comparative Perspective from Target Recognition in Synthetic-Aperture Radar Images," *IEEE Geoscience and Remote Sensing Magazine*, vol. 6, pp. 44–68, 2018.

[25] R. Hyndman, "Another Look at Forecast Accuracy Metrics for Intermittent Demand," *Foresight: The International Journal of Applied Forecasting*, vol. 4, pp. 43–46, 2006.

[26] A. Jierula, S. Wang, T.-M. OH, and P. Wang, "Study on Accuracy Metrics for Evaluating the Predictions of Damage Locations in Deep Piles Using Artificial Neural Networks with Acoustic Emission Data," *Applied Sciences*, vol. 11, no. 5, 2021. DOI: 10.3390/app11052314.

[27] C. Ferri, J. Hernández-Orallo, and R. Modroiu, "An experimental comparison of performance measures for classification," *Pattern Recognition Letters*, vol. 30, no. 1, pp. 27–38, 2009.

[28] M. Sokolova, N. Japkowicz, and S. Szpakowicz, "Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation," in *AI 2006: Advances in Artificial Intelligence*, A. Sattar and B.-h. Kang, Eds., Berlin, GE: Springer, 2006, pp. 1015–1021.

[29] S. Yang and G. Berdine, "The receiver operating characteristic (ROC) curve," *The Southwest Respiratory and Critical Care Chronicles*, vol. 5, pp. 34–36, 2017.

[30] V. A. Ferraris, "Commentary: Should we rely on receiver operating characteristic curves? From submarines to medical tests, the answer is a definite maybe!" *The Journal of thoracic and cardiovascular surgery*, vol. 157, no. 6, pp. 2354–2355, 2019.

[31] Y. Liu, Y. Zhou, S. Wen, and C. Tang, "A Strategy on Selecting Performance Metrics for Classifier Evaluation," *International Journal of Mobile Computing and Multimedia Communications*, vol. 6, pp. 20–35, 2014.

[32] D. Chicco and G. Jurman, "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, pp. 1–13, 2020.

[33] R. Delgado and X.-A. Tibau Alberdi, "Why Cohen's Kappa should be avoided as performance measure in classification," *PLoS ONE*, vol. 14, pp. 1–26, 2019.

[34] S. S. Haykin, *Neural networks and learning machines*, 3. ed. New York, NY, USA: Pearson Education, 2009.

[35] J. Feng, X. He, Q. Teng, C. Ren, H. Chen, and Y. Li, "Reconstruction of porous media from extremely limited information using conditional generative adversarial networks," *Physical Review E*, vol. 100, no. 3, 2019. DOI: 10.1103/PhysRevE.100.033308.

[36] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. de Jesus, *Neural network design*, 2nd ed. Stillwater, OK, USA: Martin Hagan, 2014. [Online]. Available: https://hagan.okstate.edu/NNDesign.pdf (visited on 06/03/2022).

[37] A. Murad and J.-Y. Pyun, "Deep Recurrent Neural Networks for Human Activity Recognition," *Sensors*, vol. 17, no. 11, 2017. DOI: 10.3390/s17112556.

[38] P. Venugopal and V. T., "State-of-Health Estimation of Li-ion Batteries in Electric Vehicle Using IndRNN under Variable Load Condition," *Energies*, vol. 12, 2019. DOI: 10.3390/en12224338.

[39] P. Le and W. Zuidema, "Quantifying the Vanishing Gradient and Long Distance Dependency Problem in Recursive Neural Networks and Recursive LSTMs," in *Proceedings of the 1st Workshop on Representation Learning for NLP*, Berlin, GE, 2016, pp. 87–93.

[40] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with LSTM," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, Edinburgh, UK, 1999, pp. 850–855.

[41] Z. C. Lipton, J. Berkowitz, and C. Elkan, *A Critical Review of Recurrent Neural Networks for Sequence Learning*, 2015. [Online]. Available: arXiv:1506.00019v4 (visited on 05/02/2022).

[42] K. Zarzycki and M. Ławryńczuk, "LSTM and GRU Neural Networks as Models of Dynamical Processes Used in Predictive Control: A Comparison of Models Developed for Two Chemical Reactors," *Sensors*, vol. 21, no. 16, 2021. DOI: 10.3390/s21165625.

[43] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[44] Z. Cui, R. Ke, Z. Pu, and Y. Wang, "Stacked bidirectional and unidirectional LSTM recurrent neural network for forecasting network-wide traffic state with missing values," *Transportation Research Part C: Emerging Technologies*, vol. 118, 2020. DOI: 10.1016/j.trc.2020.102674.

[45] L. Bottou, "On-line learning and stochastic approximations," in *In On-line Learning in Neural Networks*, D. Saad, Ed., Cambridge, UK: Cambridge University Press, 1998, pp. 9–42.

[46] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *3rd International Conference on Learning Representations (ICLR 2015)*, Y. Bengio and Y. LeCun, Eds., San Diego, CA, USA, 2015.

[47] R. Zaheer and H. Shaziya, "A Study of the Optimization Algorithms in Deep Learning," in *2019 Third International Conference on Inventive Systems and Control (ICISC)*, Coimbatore, IN, 2019, pp. 536–539. DOI: 10.1109/ICISC44355.2019.9036442.

[48] M. A. Nielsen, *Neural Networks and Deep Learning: How the backpropagation algorithm works*, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap2.html (visited on 05/02/2022).

[49] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[50] S. Raschka and V. Mirjalili, *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow* (Expert insight), 2nd ed. Birmingham, UK: Packt Publishing, 2017.

[51] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, GA, USA, 2013, pp. 1310–1318.

[52] H. Li, M. Krček, and G. Perin, "A Comparison of Weight Initializers in Deep Learning-Based Side-Channel Analysis," in *Applied Cryptography and Network Security Workshops ACNS 2020*, J. Zhou, M. Conti, C. M. Ahmed, *et al.*, Eds., Cham, CH: Springer International Publishing, 2020, pp. 126–143.

[53] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, vol. 9, Sardinia, IT, 2010, pp. 249–256.

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, CL, 2015, pp. 1026–1034.

[55] The MathWorks, Inc., *MATLAB Documentation version 9.11.0.1769968 (R2021b)*, 2021.

[56] *Keras documentation: Layer weight initializers*. [Online]. Available: `https://keras.io/api/layers/initializers/` (visited on 04/04/2022).

[57] A. M. Saxe, J. L. McClelland, and S. Ganguli, *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*, 2013. [Online]. Available: `arXiv:1312.6120v3` (visited on 05/02/2022).

[58] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[59] E. Elgeldawi, A. Sayed, A. R. Galal, and A. M. Zaki, "Hyperparameter Tuning for Machine Learning Algorithms Used for Arabic Sentiment Analysis," *Informatics*, vol. 8, no. 4, 2021. DOI: `10.3390/informatics8040079`.

[60] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012.

[61] P. I. Frazier, *A Tutorial on Bayesian Optimization*, 2018. [Online]. Available: `arXiv:1807.02811v1` (visited on 05/02/2022).

[62] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12, Red Hook, NY, USA: Curran Associates Inc, 2012, pp. 2951–2959.

[63] E. Hazan, A. Klivans, and Y. Yuan, "Hyperparameter optimization: a spectral approach," in *International Conference on Learning Representations (ICLR)*, Vancouver, CA, 2018. [Online]. Available: `https://openreview.net/forum?id=H1zriGeCZ` (visited on 06/02/2022).

[64] R. Garnett, M. A. Osborne, and S. J. Roberts, "Bayesian optimization for sensor set selection," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Stockholm, SE, 2010, pp. 209–219. DOI: `10.1145/1791212.1791238`.

[65] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*. Berlin, GE: Springer, 2008.

[66] C. Reeves, "Genetic Algorithms," in *Handbook of Metaheuristics*, ser. International Series in Operations Research & Management Science, vol. 146, 2010, pp. 109–139.

[67] S. Lessmann, R. Stahlbock, and S. Crone, "Optimizing Hyperparameters of Support Vector Machines by Genetic Algorithms," in *Proceedings of the 2005 International Conference on Artificial Intelligence (ICAI)*, H. R. Arabnia and R. Joshua, Eds., Las Vegas, NV, USA, 2005, pp. 74–82.

[68] F. Itano, M. A. de Abreu Sousa, and E. Del-Moral-Hernandez, "Extending MLP ANN hyper-parameters Optimization by using Genetic Algorithm," in *2018 International Joint Conference on Neural Networks (IJCNN)*, Rio de Janeiro, BR, 2018, pp. 1–8.

[69] K.-R. Koch, *Introduction to Bayesian Statistics*, 2nd ed. Berlin, GE: Springer, 2007.

[70] A. Gut, *Probability: A Graduate Course* (Springer Texts in Statistics (STS)). New York, NY, USA: Springer, 2005.

[71] B. J. Brewer, *Introduction to Bayesian Statistics: STATS 331*. [Online]. Available: `https://www.stat.auckland.ac.nz/%C2%A0brewer/stats331.pdf` (visited on 05/02/2022).

[72] S. V. Vaseghi, *Advanced Digital Signal Processing and Noise Reduction*, 2nd ed. Chichester, UK: John Wiley & Sons, Inc, 2000.

[73] J. Stone, *Information Theory: A Tutorial Introduction*. Sebtel Press, 2015.

[74] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[75] A. Bulinski and D. Dimitrov, "Statistical Estimation of the Kullback–Leibler Divergence," *Mathematics*, vol. 9, no. 5, 2021. DOI: `10.3390/math9050544`.

[76] T. M. Cover and J. A. Thomas, *Elements of information theory*, 2nd ed. Hoboken, NJ, USA: Wiley-Interscience, 2006.

[77] D. P. Kingma and M. Welling, "An Introduction to Variational Autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[78] L. Weng, *From Autoencoder to Beta-VAE*, 2018. [Online]. Available: `https://lilianweng.github.io/posts/2018-08-12-vae/` (visited on 04/15/2022).

[79] P. Baldi and K. Hornik, "Neural networks and principal component analysis: Learning from examples without local minima," *Neural Networks*, vol. 2, no. 1, pp. 53–58, 1989.

[80] H. Bourlard and Y. Kamp, "Auto-Association by Multilayer Perceptrons and Singular Value Decomposition," *Biological cybernetics*, vol. 59, pp. 291–294, 1988.

[81] C. M. Bishop, *Pattern Recognition and Machine Learning* (Information Science and Statistics), 1st ed. New York, NY, USA: Springer, 2016.

[82] K. Pearson, "LIII. On lines and planes of closest fit to systems of points in space," *Philosophical Magazine Series 1*, vol. 2, pp. 559–572, 1901.

[83] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science (New York, N.Y.)*, vol. 313, no. 5786, pp. 504–507, 2006.

[84] A. Ng, *CS294A Lecture notes*, 2011. [Online]. Available: `https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf` (visited on 05/02/2022).

[85] H. Lee, C. Ekanadham, and A. Ng, "Sparse deep belief net model for visual area V2," in *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., ser. NIPS'07, vol. 20, Vancouver, CA, 2007, pp. 873–880.

[86] V. Nair and G. E. Hinton, "3D Object Recognition with Deep Belief Nets," in *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems*, vol. 22, Vancouver, CA, 2009, pp. 1339–1347.

[87] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *ICML '08: Proceedings of the 25th International Conference on Machine Learning*, New York, NY, USA: Association for Computing Machinery, 2008, pp. 1096–1103.

[88] D. P. Kingma and M. Welling, *Auto-Encoding Variational Bayes*, 2014. [Online]. Available: arXiv:1312.6114v10 (visited on 02/06/2022).

[89] P. Mehta, S. Kumar, R. Kumar, and C. S. Babu, "Demystifying Tax Evasion Using Variational Graph Autoencoders," in *Electronic Government and the Information Systems Perspective*, A. Kö, E. Francesconi, G. Kotsis, A. M. Tjoa, and I. Khalil, Eds., Cham, CH: Springer International Publishing, 2021, pp. 155–166.

[90] P.-A. Mattei and J. Frellsen, "Leveraging the Exact Likelihood of Deep Latent Variable Models," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, Red Hook, NY, USA: Curran Associates Inc, 2018, pp. 3859–3870.

[91] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational Inference: A Review for Statisticians," *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[92] A. Asperti and M. Trentin, "Balancing Reconstruction Error and Kullback-Leibler Divergence in Variational Autoencoders," *IEEE Access*, vol. 8, pp. 199 440–199 448, 2020. DOI: 10.1109/ACCESS.2020.3034828.

[93] A. Alemi, B. Poole, I. Fischer, J. Dillon, R. A. Saurous, and K. Murphy, "Fixing a Broken ELBO," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, Stockholm, SE, 2018, pp. 159–168.

[94] S. Lin, S. J. Roberts, N. Trigoni, and R. Clark, *Balancing Reconstruction Quality and Regularisation in Evidence Lower Bound for Variational Autoencoders*. [Online]. Available: arXiv:1909.03765v1 (visited on 05/03/2022).

[95] K. Cho, B. van Merriënboer, C. Gulcehre, *et al.*, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds., Doha, Qatar, 2014, pp. 1724–1734.

[96] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Cambridge, MA, USA: MIT Press, 2014, pp. 3104–3112.

[97] D. Hawkins, *Identification of Outliers* (Monographs on Statistics and Applied Probability Ser). Dordrecht, NL: Springer, 1980.

[98] B. Lindemann, B. Maschler, N. Sahlab, and M. Weyrich, "A Survey on Anomaly Detection for Technical Systems using LSTM Networks," *Computers in Industry*, vol. 131, 2021. DOI: 10.1016/j.compind.2021.103498.

[99] G. Strang, *Introduction to linear algebra*, 5th ed. Wellesley, MA, USA: Wellesley-Cambridge Press, 2016.

[100] J. D. Salas, J. W. Delleur, V. Yevjevich, and W. L. Lane, *Applied Modelling of Hydrologic Time Series*. Littleton, CO, US: Water Resources Publications, 1980.

[101] V. Kozitsin, I. Katser, and D. Lakontsev, "Online Forecasting and Anomaly Detection Based on the ARIMA Model," *Applied Sciences*, vol. 11, no. 7, 2021. DOI: 10.3390/app11073194.

[102] Q. Yu, L. Jibin, and L. Jiang, "An Improved ARIMA-Based Traffic Anomaly Detection Algorithm for Wireless Sensor Networks," *International Journal of Distributed Sensor Networks*, vol. 2016, pp. 1–9, 2016.

[103] E. H. M. Pena, M. V. O. de Assis, and M. L. Proença, "Anomaly Detection Using Forecasting Methods ARIMA and HWDS," in *2013 32nd International Conference of the Chilean Computer Science Society (SCCC)*, Temuco, CL, 2013, pp. 63–66.

[104] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long Short Term Memory Networks for Anomaly Detection in Time Series," in *23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2015)*, Bruges, BE, 2015, pp. 89–94.

[105] K. P. Tran, H. Du Nguyen, and S. Thomassey, "Anomaly detection using Long Short Term Memory Networks and its applications in Supply Chain Management," *IFAC-PapersOnLine*, vol. 52, no. 13, pp. 2408–2412, 2019.

[106] M. Munir, S. Siddiqui, A. Dengel, and S. Ahmed, "DeepAnT: A Deep Learning Approach for Unsupervised Anomaly Detection in Time Series," *IEEE Access*, vol. 7, pp. 1991–2005, 2019. DOI: 10.1109/ACCESS.2018.2886457.

[107] M. Said Elsayed, N.-A. Le-Khac, S. Dev, and A. D. Jurcut, "Network Anomaly Detection Using LSTM Based Autoencoder," in *Proceedings of the 16th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, ser. Q2SWinet '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 37–45.

[108] T. Sabata and M. Holena, "Active Learning for LSTM-autoencoder-based Anomaly Detection in Electrocardiogram Readings," in *Proceedings of the Workshop on Interactive Adap-*

tive Learning co-located with European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2020), ser. CEUR Workshop Proceedings, vol. 2660, CEUR-WS.org, 2020, pp. 72–77.

[109] D. Park, Y. Hoshi, and C. C. Kemp, "A Multimodal Anomaly Detector for Robot-Assisted Feeding Using an LSTM-Based Variational Autoencoder," *IEEE Robotics and Automation Letters*, vol. 3, pp. 1544–1551, 2018.

[110] D. Kim, H. Yang, M. Chung, and S. Cho, "Squeezed Convolutional Variational AutoEncoder for unsupervised anomaly detection in edge device industrial Internet of Things," *2018 International Conference on Information and Computer Technologies (ICICT)*, pp. 67–71, 2018.

[111] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *KDD'96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, OR, USA, 1996, pp. 226–231.

[112] M. Celik, F. Dadaser-Celik, and A. Dokuz, "Anomaly Detection in Temperature Data Using DBSCAN Algorithm," in *2011 International Symposium on Innovations in Intelligent Systems and Applications (NISTA 2011)*, Istanbul, TR, 2011, pp. 91–95. DOI: 10.1109/INISTA.2011.5946052.

[113] U. Rebbapragada, P. Protopapas, C. E. Brodley, and C. Alcock, "Finding anomalous periodic time series," *Machine Learning*, vol. 74, no. 3, pp. 281–313, 2009.

[114] B. Schölkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, "Support Vector Method for Novelty Detection," in *Advances in Neural Information Processing Systems*, vol. 12, MIT Press, 2000, pp. 582–588.

[115] J. Ma and S. Perkins, "Time-series novelty detection using one-class support vector machines," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 3, Portland, OR, USA, 2003, pp. 1741–1745.

[116] R. N. Calheiros, K. Ramamohanarao, R. Buyya, C. Leckie, and S. Versteeg, "On the effectiveness of isolation-based anomaly detection in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 18, e4169, 2017.

[117] M. U. Togbe, M. Barry, A. Boly, *et al.*, "Anomaly Detection for Data Streams Based on Isolation Forest Using Scikit-Multiflow," in *Computational Science and Its Applications – ICCSA 2020*, ser. Lectures in Computer Science, O. Gervasi, B. Murgante, S. Misra, *et al.*, Eds., vol. 12252, Cham, CH: Springer International Publishing, 2020, pp. 15–30.

[118] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *2008 Eighth IEEE International Conference on Data Mining*, Pisa, IT, 2008, pp. 413–422.

[119] O. Alghushairy, R. Alsini, T. Soule, and X. Ma, "A Review of Local Outlier Factor Algorithms for Outlier Detection in Big Data Streams," *Big Data and Cognitive Computing*, vol. 5, no. 1, 2020. DOI: 10.3390/bdcc5010001.

[120] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 93–104.

[121] E. J. Hagendorfer, "Knowledge Incorporation for Machine Learning in Condition Monitoring: A Survey," in *2021 International Symposium on Electrical, Electronics and Information Engineering*, ser. ISEEIE 2021, New York, NY, USA: Association for Computing Machinery, 2021, pp. 230–240.

[122] M. Haider, "Machine Learning and KPI Analysis applied to Time-Series Data in Physical Systems: Comparison and Combination," M.S. Thesis, Chair of Automation, Montanuniversitaet Leoben, Leoben, AT, 2021.

[123] Keller UK Ltd, *Vibro compaction*. [Online]. Available: https://www.keller.co.uk/expertise/techniques/vibro-compaction (visited on 04/23/2022).

[124] A. Terbuch, *Generic Deep Autoencoder for Time-Series: Version 1.0.1*, MATLAB Central File Exchange, 2022. [Online]. Available: https://ch.mathworks.com/matlabcentral/fileexchange/111110-generic-deep-autoencoder-for-time-series?s_tid=ta_fx_results (visited on 06/02/2022).

[125] D. Charte, F. Charte, S. García, M. J. Del Jesus, and F. Herrera, "A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines," *Information Fusion*, vol. 44, pp. 78–96, 2018. DOI: 10.1016/j.inffus.2017.12.007.

[126] J. Han and M. Kamber, *Data Mining: Concepts and Techniques* (The Morgan Kaufmann Series in Data Management Systems Ser), 2nd ed. Amsterdam, NL: Elsevier, 2006.

[127] E. Vandervieren and M. Hubert, "An adjusted boxplot for skewed distributions," in *COMPSTAT 2004 - Proceedings in Computational Statistics: 16th symposium*, J. Antoch, Ed., Heidelberg, GE: Physica, 2004, pp. 1933–1940.

[128] G. Brys, M. Hubert, and A. Struyf, "A Comparison of Some New Measures of Skewness," in *Developments in Robust Statistics: International Conference on Robust Statistics 2001*, R. Dutter, P. Filzmoser, U. Gather, and P. Rousseeuw, Eds., Berlin, GE: Springer, 2003, pp. 98–113.

[129] M. Hubert and E. Vandervieren, "An Adjusted Boxplot for Skewed Distributions," *Computational Statistics & Data Analysis*, vol. 52, pp. 5186–5201, 2008.

[130] J. W. Tukey, *Exploratory Data Analysis*. Reading, MA, USA: Addison-Wesley Publishing, 1977.

[131] M. Yang and J. Wang, "Adaptability of Financial Time Series Prediction Based on BiLSTM," *Procedia Computer Science*, vol. 199, pp. 18–25, 2022. DOI: 10.1016/j.procs.2022.01.003.

[132] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "The Performance of LSTM and BiLSTM in Forecasting Time Series," in *2019 IEEE International Conference on Big Data*, Los Angeles, CA, USA, 2019, pp. 3285–3292.

[133] F. Masri, D. Saepudin, and D. Adytia, "Forecasting of Sea Level Time Series using Deep Learning RNN, LSTM, and BiLSTM: Case Study in Jakarta Bay, Indonesia," *e-Proceeding of Engineering*, vol. 7, no. 2, pp. 8544–8551, 2020.

[134] A. Terbuch, A. Zöhrer, V. Winter, P. O'Leary, N. Khalili-MotlaghKasmaei, and G. Steiner, "Quality monitoring in vibro ground improvement – A hybrid machine learning approach," *Geomechanics and Tunnelling*, 2022. DOI: 10.1002/geot.202200028.