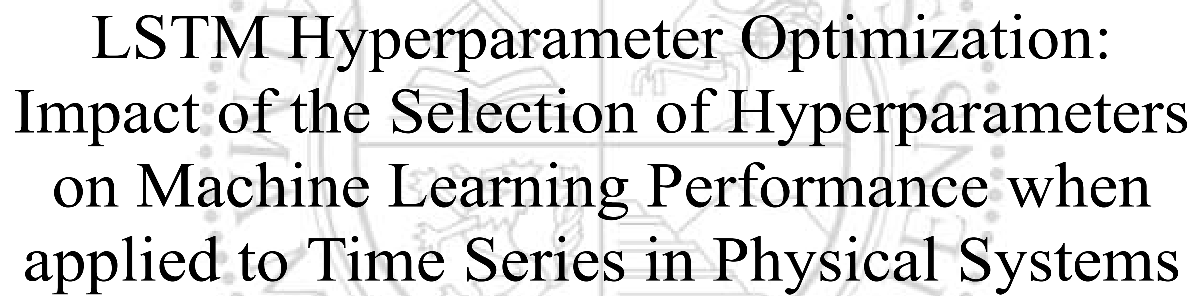
Chair of Automation

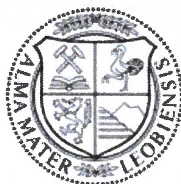
Master's Thesis



LSTM Hyperparameter Optimization: Impact of the Selection of Hyperparameters on Machine Learning Performance when applied to Time Series in Physical Systems

Anika Teresa Terbuch, BSc

May 2021



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 25.05.2021

Anika Teresa Terbuch

Unterschrift Verfasser/in
Anika Teresa Terbuch

Danksagung - Dedication

I’d like to thank all people that supported me in the process of writing this thesis.

I would like to express my gratitude to my supervisor Professor Paul O’Leary for the opportunity to write my master thesis, for all the helpful discussions and for the freedom to implement my ideas. Many thanks to everybody of the Chair of Automation.

I could not thank my boyfriend, my family and friends enough for supporting me during my whole studies. Prisrčna hvala! Danke!

Kurzfassung

Diese Masterarbeit untersucht den Einsatz von genetischen Algorithmen zur Optimierung von Hyperparametern des maschinellen Lernens, insbesondere für den Anwendungsfall von Echtzeitdaten, welche in industriellen Prozessen anfallen. Der Einsatz und die Eignung der Kombination eines Algorithmus des maschinellen Lernens und einer Metaheuristik, welche unter dem Überbegriff der genetischen Algorithmen zusammengefasst wird, wird untersucht. Als Modell des maschinellen Lernens wird ein sogenannter Variational-Autoencoder, welcher über Schichten des langen Kurzzeitgedächtnisses verfügt, verwendet. Der Rückgabewert dieses Modelles ist der Rekonstruktionsfehler. Da dieser nicht normalverteilt ist, wird für die Anomaliedetektion eine spezielle Art des Boxplots für schiefe, nicht normalverteilte Daten verwendet. Eine neue Variation des genetischen Algorithmus mit maximal einer Evaluation pro Individuum und Teilmenge wird zur Laufzeitreduktion vorgestellt. Damit lässt sich darüber hinaus auch das benötigte Expertenwissen, welches bei manuellen Ansätzen der Hyperparameteroptimierung benötigt wird, vermindern. Weiters wird die Kombination zweier Kreuzungsfunktionen eingeführt, um eine bessere Untersuchung guter Regionen des Suchraumes zu gewährleisten. Für die Ausreißererkennung werden Methoden des unüberwachten Lernens eingesetzt. Für die Hyperparameteroptimierung und das Trainieren des Netzwerkes werden nur fehlerfreie Daten verwendet. Nach diesem Schritt wird das damit trainierte Netzwerk auf den gesamten Datensatz angewandt. Diese Vorgehensweise verbessert die Performance der Anwendung auf Daten mit einem unausgewogenen Verhältnis zwischen gewöhnlichen Daten und Ausreißern. Der entwickelte Ansatz wurde erfolgreich auf Daten, welche in einem industriellen Prozess erhoben wurden, angewandt.

Abstract

This thesis investigates the use of genetic algorithms to optimize the hyperparameters of machine learning; the focus is on the application to real-time series data gathered during industrial processes. The combination of machine learning and the meta-heuristic genetic algorithm is reviewed to determine their suitability for hyperparameter optimization for anomaly detection. Because the machine learning model consists of a variational autoencoder with long short-term memory layers, the output of the model is the reconstruction error. Further, a skewness-adjusted boxplot for non-normal distributed data is applied for outlier detection on the reconstruction error. A new approach of the genetic algorithm with maximal one evaluation of each individual per generation and fold was introduced. The genetic algorithm is developed to overcome the long runtime and expert knowledge that is needed for the popular approach of manual hyperparameter optimization. Further, a combination of two crossover functions is introduced for a better exploration of good regions of the search space. The outlier detection is done in an unsupervised manner. For the hyperparameter optimization as well as the training only non-anomalous data was used; then the trained network is applied to all the data. This improves performance for highly biased training data. The approach was successfully applied to datasets gained during an industrial process.

Acronyms

BPTT	Backpropagation Through Time
GA	Genetic Algorithm
HPO	Hyperparameter Optimization
LSTM	Long Short-Term Memory
LSTM-VAE	LSTM-Based Variational Autoencoder
IQR	Interquantile Range
MAD	Mean Absolute Deviation
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MC	Medcouple
ML	Machine Learning
MLP	Multi Layer Perceptron
MSE	Mean Square Error
ReLU	Rectified Linear Unit
RMSE	Root Mean Square Error
RMSPE	Root Mean Square Percentage Error
RNN	Recurrent Neural Network
SD	Standard Deviation
VAE	Variational Autoencoder

Contents

List of figures . . . viii

List of tables . . . x

List of listings . . . xi

1 Introduction . . . 1
- 1.1 Goals . . . 2
- 1.2 Organization . . . 2

2 Machine Learning Basics . . . 3
- 2.1 What Machine Learning is . . . 3
- 2.2 Classification vs. Regression . . . 5
  - 2.2.1 Classification . . . 5
  - 2.2.2 Regression . . . 5
- 2.3 Error Measurement . . . 6
  - 2.3.1 Scale Dependant Errors . . . 6
  - 2.3.2 Percentage Errors . . . 7
- 2.4 Artificial Neural Networks . . . 7
  - 2.4.1 Perceptron . . . 8
  - 2.4.2 Multilayer Perceptron . . . 10
  - 2.4.3 Recurrent Neural Network . . . 11
- 2.5 Architectures . . . 12
  - 2.5.1 Sequence-Vector Model . . . 12
  - 2.5.2 Vector-Sequence Model . . . 13
  - 2.5.3 Sequence-Sequence Model . . . 14

3 Time Series . . . 18
- 3.1 Definition . . . 18
- 3.2 Components . . . 18

3.3 Characteristics of Time Series in ML . . . 18
3.3.1 Backpropagation Through Time . . . 19
3.3.2 Time Series Forecasting . . . 19
3.3.3 Outlier Detection . . . 20

4 Long-Short Term Memory . . . 22
4.1 Basic Architecture Description . . . 23
4.2 Combination of LSTM and VAE . . . 23

5 Genetic Algorithm . . . 25
5.1 Description . . . 25
5.2 Representation . . . 26
5.3 Initialization . . . 26
5.4 Genetic Operators . . . 27
5.4.1 Selection . . . 28
5.4.2 Crossover . . . 30
5.4.3 Mutation . . . 31
5.4.4 Replacement . . . 31
5.5 Hyperparameters of the Genetic Algorithm . . . 31
5.6 Why GA works . . . 32
5.6.1 Convergence of Genes . . . 33

6 LSTM Hyperparameter-Selection using Heuristics based on GA . . . 34
6.1 Hyperparameter Optimization . . . 34
6.1.1 Importance/ Impact of LSTM hyperparameters on the Performance . . . 35
6.2 Classical Techniques for Choosing Hyperparameters . . . 36
6.2.1 Manual Search . . . 36
6.2.2 Automatic Search . . . 37
6.3 Hyperparameter Optimization Using Genetic Algorithms . . . 38

7 Statistical Approaches for Outlier Detection . . . 40
7.1 Outlier Detection for Normal Distributed Data . . . 41
7.1.1 Standard Deviation (SD) . . . 41
7.1.2 Median Absolute Deviation (MAD) . . . 41
7.1.3 Interquantile Range (IQR) . . . 42
7.1.4 Z-Score . . . 43
7.1.5 Two-Stage Threshold . . . 44
7.2 Outlier Detection for Non-Normal Distributed Data . . . 44
7.2.1 Boxplot . . . 44

8 Approach for HPO Using GA 47
8.1 Used Time Series Data 47
8.1.1 Preprocessing 47
8.2 Optimized ML-Model 48
8.3 Applied Genetic Algorithm 50
8.3.1 Termination Condition 51
8.3.2 Fitness Function 52
8.3.3 Chromosomes 53
8.3.4 Genetic Operators 54
8.3.5 Algorithm Applied to Real Data 56

9 Summary and Conclusion 74

Bibliography

List of Figures

2.1 data mining wisdom pyramid . . . 3
2.2 schematic representation of a perceptron . . . 8
2.3 unfolding a sequence to vector model . . . 13
2.4 unfolding a vector to sequence architecture . . . 14
2.5 unfolding a sequence to sequence model . . . 15

4.1 basic LSTM architecture . . . 22

5.1 cycle of operators for a classical GA . . . 27
5.2 roulette wheel selection . . . 28
5.3 single-point crossover of parents P1 and P2 to offspring C1 and C2 . . . 30
5.4 bit-flipping mutation from parent P to offspring C [6] . . . 31

6.1 examples of activation functions . . . 37

8.1 exemplary plot of recorded signals over time at the site Seestadt Aspern . . . 48
8.2 exemplary plot of recorded signals over time at the site Fehring . . . 49
8.3 resized exemplary plot of the signals shown in Figure 8.2 to 400 timesteps . . . 50
8.4 the signals shown in Figure 8.3 where taken and rescaled between zero and one. . . . 51
8.5 deviation in fitness of models trained with the same data and same hyperparameters 52
8.6 deviation in fitness of models trained with different folds of the same dataset and same hyperparameters . . . 53
8.7 average fitness of the population over the generations . . . 57
8.8 values of the HPO the individuals take at each generation . . . 58
8.9 values of the HPO executed a second time the individuals take at each generation . . 59
8.10 reconstructed and real signals for a non-anomalous point of the site Fehring . . . 60
8.11 reconstructed and real signals for a anomalous point of the site Fehring . . . 61
8.12 two encoder-decoder structures where trained with the same hyperparameters of the HPO visualized in Figure 8.8 and on the same data and afterwards applied on the whole dataset of the site Fehring to get the reconstruction error for each point . . 62

8.13 histogram of the data gained at performing the reconstruction shown in Figure 8.12 62
8.14 reconstruction error of a ML model trained on the hyperparameters held by two individuals of the first generation when applied on the data of the site Fehring. 63
8.15 first example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples and trained models with optimized hyperparameters 64
8.16 second example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples 65
8.17 first example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples with models trained with random hyperparameters 66
8.18 second example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples with models trained with random hyperparameters 67
8.19 results of the HPO done on the enlarged domain for the site Fehring 68
8.20 outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples; models where trained on hyperparameters found by a HPO on the enlarged search space 69
8.21 results of the HPO done on the extended domain for the site Fehring 70
8.22 outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples; models where trained on hyperparameters found by the HPO on the extended search space 71

List of Tables

8.1 domains of the HPO for the normal and extended searchspace 53
8.2 results for the hyperparameters of different HPO-runs 72

List of listings

8.1 architecture of the LSTM-VAE 49
8.2 random crossover 55
8.3 mean crossover 56

Chapter 1
Introduction

This thesis addresses the issue of applying machine learning to real-time series data from industrial processes with the goal of producing added value. The key issue of this topic is that the data is acquired as real-time series data. One of the special characteristics of this type of data is that the sequence of the data needs to be maintained. Data collected in this setting is in most cases highly biased. In the case of outlier detection, this means that more non-anomalous than anomalous examples are present in the data. Often only unlabelled data is available, which limits the number of machine learning (ML) techniques that can be applied [1].

As a ML-model a long short-term memory (LSTM) based variational autoencoder (LSTM-VAE) is used, which was developed during a prior thesis (see [2]). This approach was chosen because ML-models that are applied on time-series data have to deal with the problem of vanishing/ exploding gradients [3] due to the nature of learning in deep recurrent architectures using gradient descent-based techniques and backpropagation.

Each ML-algorithm has several settings, hyperparameters, which need to be set before execution. The relationship of hyperparameters and the performance of machine learning algorithms is still unclear [4]. It is assumed that some few hyperparameters have bigger impact on the performance than others [5]. Optimizing this and finding for this values of good performance can be enough to reach a region of the search space where the overall performance of the ML-model does not vary much. Genetic algorithms, which belong to the subgroup of meta-heuristics, explore the search space by a directed search. However, its nature does not guarantee convergence to a global optimum. Despite this, it is very likely that meta-heuristics find good regions of the search space [6].

When optimizing multiple hyperparameters, hyperparameter optimization (HPO) becomes multidimensional and multiple configurations are possible. A common approach to find hyperparameters is applying grid search. Using this technique all possible configurations of the values on a specified grid are tried out one by one, which is an exhaustive search and has exponential runtime. The evaluation of ML-models is computationally expensive and the budget of possible evaluations limited. Another commonly used technique is that the hyperparameters are set manually, which requires a lot of expert knowledge [4].

The question of how to choose a subset of hyperparameters for optimization and how to determine the search space is still only discussed in the literature . The same is the case for the question of how to choose the right encoding for HPO applied on ML and how to design the genetic operators, which are the crucial part of every genetic algorithm.

1.1 Goals

This situation motivates an evaluation of the potential of using optimized hyperparameters on a variational autoencoder including LSTM layers when applied to time series in a physical system. As an optimization technique, a genetic algorithm is used. It should be discussed if genetic algorithms are suitable for this task. Further, the question about finding the right representation should be addressed as well as a literature review about existing solutions for LSTM hyperparameter optimization. Also, possible variations of the operators of genetic algorithms should be discussed. The goal is to show if different hyperparameter settings have an impact on the reconstruction of the encoded and decoded input signal and if there is potential to use this technique for outlier detection.

1.2 Organization

This work is structured as follows: Part 2 introduces the basic concepts of machine learning, Part 3 is discussing the theory related to time series forecasting, while Part 4 focuses on the machine learning architecture LSTM and the architecture LSTM-VAE. Part 5 introduces the optimization technique genetic algorithm (GA) while Part 6 combines the topics of GA and LSTM. Part 7 gives an overview of statistical approaches for outlier detection. Part 8 shows the application of a GA for HPO on real data. Part 9 summarizes and concludes this work and proposes possible directions for further work.

Chapter 2
Machine Learning Basics

2.1 What Machine Learning is

There is no single answer to what machine learning is. Commonly machine learning is presented as the idea of imitating the learning process of humans by computers. Human beings and some other biological systems are the only known example of robust learning, therefore trying to copy their behaviour is often a starting point for possible learning methods [7]. One downside of human-like-learning is that it is a slow process because the encoding of it is unknown. Additionally, there is no possibility to copy learned information [8]. One definition formulated by Simon in [8]: ”Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.” From that definition, learning can only be achieved by a repetitive process [8].

According to the authors of [9] computers would need to incorporate some kind of self-reflection to act and react intelligently. Humans can self-observe their behaviour and learn from that. This is the foundation for the ability of self-reflection [9].

The question is how much can computers understand? Extracting information that meets certain criteria out large data sets is known as data mining. This can be visualized with the data mining wisdom pyramid (see Figure 2.1) [10]. According to this definition, the hierarchy of processing raw data is: data, information, knowledge, understanding and ”ultimate” wisdom. The idea of the

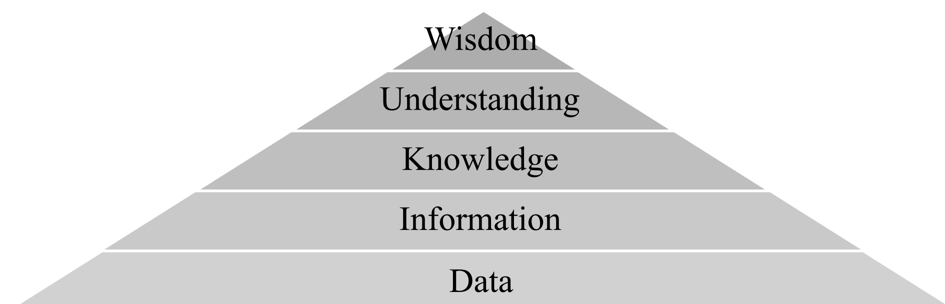


Fig. 2.1: data mining wisdom pyramid [10]

pyramid is that raw data is pre-processed to easier extract attributes or features needed for the application. By building a model that executes regression or classification knowledge is gained. This knowledge is then analysed and so a deeper understanding of the work principles is obtained. In this context as wisdom is defined transferring knowledge to other application domains [9].

These principles cannot be applied to all problems that humans can solve. Some problems, particularly those, that can be described by formal rules, can be solved simply and efficiently by computers. In contrast, computers have great difficulty in solving problems that are easy for humans to solve but cannot be described by formal rules. Human beings approach these problems intuitively. Examples for tasks of this group are: recognizing spoken words or faces in images. To solve a task a computer needs an algorithm that is designed for that task [11].

According to [12] an algorithm can be very abstract defined as: ”[...] mechanically executable, ordered (possibly non-terminating) list of (multiple, interacting) steps for the application of computable functions on a well-defined domain of input values (be these simple or streams of infinite and complex nature).”

For some task no algorithm is available. The nature of the inputs is known as the expected outputs, however, the required sequence of commands for the transformation is unknown. The algorithm for that task should be extracted by the computer providing enough examples that can be discovered. In most cases, only an approximation of an algorithm is extracted from the input data and this can be enough for the task of detecting certain patterns and regularities. With the discovered patterns, predictions on outcomes of inputs can be done [13]. Each piece of information included in the input data is called a feature. Machine learning can be described as a class of algorithms that extracts ”non-hard coded” knowledge from the inputs. Therefore, the performance of machine learning algorithms depends heavily on the representation of data that is given to them as input. The machine learning algorithm learns the mapping between inputs and outputs but cannot influence in any way how the features are defined [11].

A possible approach to solve this problem is called representation learning. If this variation of machine learning is used the input-output mapping is learned alongside with the representation itself. An example for representation learning are autoencoders (see Section 2.5.3.2) [11].

To solve more complex mappings so-called *deep learning* is used [11]. The term deep learning describes a subgroup of machine learning techniques that have at least three adaptive non-linear layers to convert a given input to an output [14].

This concept allows the computer to build more complex structures out of simpler concepts by breaking down the desired complex mapping into a series of simpler mappings. Each mapping is done by a different layer of the model. In deep learning models, layers are distinguished by their position in the network between visible and hidden layers. In visible layers, the variables can be observed, as in the input layer. The hidden layers extract abstract features from the data and their values are not given in the data. The last layer is an output layer, that outputs the solution to the given task. The structure of each layer is not specified, beside some parameters restricting

the dimensionality [11]. ML algorithms can be divided into two classes, based on the degree of supervision of learning. When *supervised learning* is used, the goal is to learn the input-output mapping and given a labelled set of input-output pairs. In *unsupervised learning* the outcome is not known and only the input data is available [13, 15]. As summed up in [13], ML can also be explained as the task to program computers in a manner so that they can determine the parameters of a model optimizing the performance criterion. So ML can be understood as an optimization problem [13].

Every machine learning algorithm has two sets of parameters: model parameters and hyperparameters. The first group, model parameters, are parameters that are initialized and updated as learning proceeds. On the other side, hyperparameters cannot be estimated by learning, but have to be set before training and have an impact on the learning procedure and the model architecture [16]. Every learning algorithm has a large set of hyperparameters. This set needs to be dimensioned and selected for each algorithm. These parameters can be set empirically or by using additional data (see Section 6.2 and Chapter 6) [11].

2.2 Classification vs. Regression

Two of the most common tasks in supervised machine learning are classification and regression [11].

2.2.1 Classification

As *classification* the problem assigning a class to each item is described [17]. This task is solved by the learning algorithm by producing a function f where

$$f : \mathbb{R}^n \rightarrow \{1, \dots, k\}, \tag{2.1}$$

such that it is a mapping from an element in $\mathbb{R}^n$ to an element in a finite set. If $y = f(x)$, the input to the model x produces an output y, which assigns x to a categorical output y. Another variation of a classification task is that the output of f is a probability distribution that describes how likely it is that the input of x is a member of each instance of the set $\{1, \dots, k\}$ [11].

2.2.2 Regression

The term *regression* describes the problem of predicting a real value for each input [17]. This task is solved by the learning algorithm producing a function [11]

$$f:\mathbb{R}^n \rightarrow \mathbb{R}, \tag{2.2}$$

which is a mapping from an element in $\mathbb{R}^n$ to a real number [11].

2.3 Error Measurement

In literature mainly a distinction between two classes of measures for forecast accuracy of time-series predictions is made. Both measures are comparing the target value to the value of the prediction [18].

2.3.1 Scale Dependant Errors

If a scale dependant error measure is used, the error is on the same scale as the data. The limitation of this technique is, that it should not be used for comparisons between data sets on different scales [18], but they can be used to compare different methods applied on the same set of data [19]. Some examples for scale dependent error measures [19]:

1. mean square error (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i^{obs} - y_i^{pred}\right)^2, \tag{2.3}$$

2. root mean square error (RMSE):

$$RMSE = \frac{1}{n}\sqrt{\sum_{i=1}^{n}\left(y_i^{obs} - y_i^{pred}\right)^2}, \tag{2.4}$$

3. mean absolute error (MAE):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i^{obs} - y_i^{pred}|, \tag{2.5}$$

whereby y_i^{obs} is the current observation and y_i^{pred} is the predicted value. Because the RMSE is on the same scale as the data, it is often preferred to the MSE. Both of those error measures are more sensitive to outliers in the data than MAE [19].

2.3.2 Percentage Errors

Percentage errors on the other hand are scale-independent. They are often used to compare performance between different scaled datasets with relative errors. This class includes [19]:

1. root mean square percentage error (RMSPE):

$$RMSPE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}[\frac{y_i^{obs} - y_i^{pred}}{y_i^{obs}}]} \times 100, \tag{2.6}$$

2. mean absolute percentage error (MAPE):

$$MAPE = \frac{1}{n}[\frac{y_i^{obs} - y_i^{pred}}{y_i^{obs}}]. \tag{2.7}$$

The disadvantage of these error measures is, that they are undefined if $y_i^{obs} = 0$. On the other hand the advantage of percentage error measures is that they deliver the relative error and so errors of data on different scales can be compared [19].

2.4 Artificial Neural Networks

The history of *artificial neural networks* (ANN) goes back to the first attempts understanding the structure and functionality of the human brain. McCulloch and Pitts developed in 1943 a computational model for a basic neuron (see [20]). They discovered that one of the main characteristics of nervous activity, neural events and relations between them is that they are active or not and can therefore be treated using propositional logic [20].

From the view of neurophysiology, each neuron consists of the soma, the cell body, and the axon. The nervous system is formed by a net of neurons. The synapses are always between the soma of one and the axon of another. Every neuron has at any point in time some threshold. The excitation needs to be higher than this threshold to initiate an impulse. It has been observed that this excitation needs to take place in a sufficient number of neighbouring neurons that an impulse is triggered [20]. Further, they showed that in principle any computable function can be computed with a high enough number of neurons, appropriate synapses, each synapse associated with a weight. This paper is seen by many experts as the starting point of the field of neural networks and artificial intelligence [21].

Despite all the biological and neuroscience background of this research area, the goal of modern artificial neural networks is not to model the brain function but achieving statistical generalization [11]. Many variations of ANNs have been developed over the last decades. It can be

distinguished between ANNs, which contain recurrent connections (see Section 2.4.3) and ANNs without recurrent connections (see Section 2.4.2) [22].

2.4.1 Perceptron

The basic processing element of an artificial neural network is called a perceptron. Each perceptron has d input units x_i, where $i = \{1, \dots, d\}$. As illustrated in Figure 2.2 alongside the inputs, every perceptron has a bias unit denoted with x_0 and a bias weight w_0 [13].

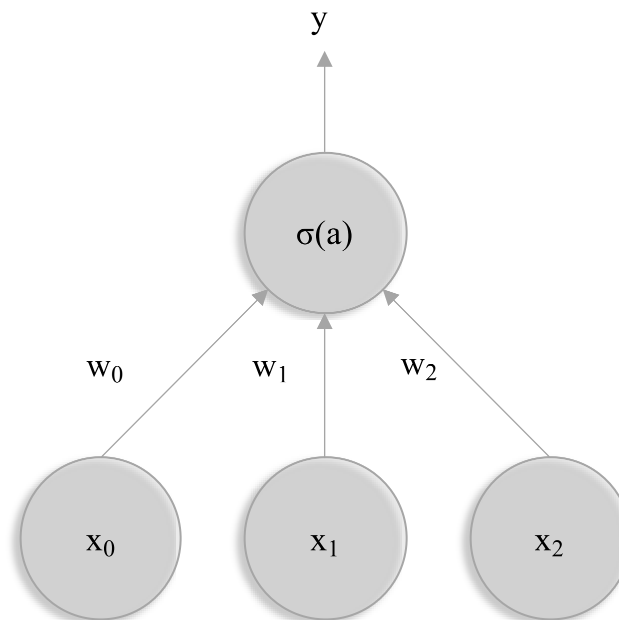


Fig. 2.2: Schematic representation of a perceptron with the inputs x_1 and x_2 alongside with the associated weights w_1 and w_2. The bias is represented by x_0 and the bias weight by w_0. The inputs are passed through the activation function $\sigma(a)$ to produce the output of the perceptron y.

There are two types of inputs that the perceptron can receive, either inputs from the environment or inputs that are the outputs of other perceptrons. Each input is weighted via a weight w_i. The output can be formed in different ways. The simplest output is a weighted sum of the inputs, which can be written as a dot product [13]:

$$y = \sum_{i=1}^{d} w_i x_i + w_0 = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}, \tag{2.8}$$

whereby $\boldsymbol{w} = [w_0, w_1, \dots, w_d]$ and $\boldsymbol{x} = [1, x_1, \dots, x_d]$. The vectors are augmented in a way that the bias weight and the bias input are included [13].

A perceptron implements the learning of the weights $\boldsymbol{w}$, such that for the input $\boldsymbol{x}$, the difference between the output of the perceptron $\boldsymbol{y}^{pred}$ and the corresponding label $\boldsymbol{y}^{obs}$ is minimized [13].

If the perceptron receives only one input from the environment, that means $d = 1$, it models the equation of a line [13]:

$$y = w_1 x + w_0, \tag{2.9}$$

whereby w_1 represents the slope and w_0 is the offset. Such models can be used for linear fitting. The perceptron defines a hyperplane and divides the input space into two half-spaces, to do so it uses a *activation function* on the output of the computations which are performed on the inputs. An example for a simple activation function is the binary step function [13]:

$$\sigma(a) = \begin{cases} 1 & \text{if a} \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

When the binary step function is applied on a regression task, if the output of the threshold function is 1, the input belongs to class 1, otherwise to class 2. Also other functions as the sigmoid function or the hyperbolic tangent can be used as activation function. If the task includes $K \geq 2$ outputs, K perceptrons are used, whereby each perceptron is a local function with inputs and weights. A perceptron with only one layer can only learn linear functions of the inputs and therefore can not be used for nonlinear problems such as nonlinear regression [13].

2.4.1.1 Training a Perceptron

A common learning technique for neural networks is the so-called *online learning*. Samples are presented one by one to the network and the weights are adjusted over time. A popular initialization technique is initializing the weights with random numbers. In the process of training the weights are slightly adjusted in the direction of the minimum of the objective after each iteration without forgetting the previous learned. The speed of learning is controlled by the hyperparameter called *learning rate*, which controls how much of the previously learned is kept and how much the weights are impacted by the adjustment in the direction of the minimum. If the learning rate is too high the algorithm tends to oscillate in the search space, because the weights are adjusted too rigorous. Otherwise, if the learning rate is too low could lead to the case that the algorithm converges to a local optimum because it examines only a small part of the search space. If the error function is differentiable, gradient descent can be used for determining the direction of the next update towards the minimum. After passing through the network all samples ones and adjusting the weights accordingly, one *epoch* is finished. The number of epochs is the hyperparameter that determines the number of training iterations [11].

2.4.2 Multilayer Perceptron

The *multilayer perceptron* (MLP) is a deep learning model. Each input example $\boldsymbol{x}$ is combined by a label $\boldsymbol{y}$, therefore it belongs to the class of supervised learning techniques. This kind of neural network is also called a (deep) feed-forward network because the information flow through the network is only evaluated in one direction; from $\boldsymbol{x}$ to define $\boldsymbol{y}$. There are no recurrent connections used in this type of architecture. Recurrent connections are implemented in other models to feed-back the output of a layer to earlier located computation steps (see Section 2.4.3). Further, MLPs implement at least one hidden layer containing at least one hidden unit. Each hidden layer has a vector containing the values of the hidden units denoted as $\boldsymbol{h}$ and also referred to as the *hidden state vector*. The goal of an MLP is if a input vector $\boldsymbol{x}$ is given to approximate some function g by defining the mapping [11, 22]:

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{w}). \tag{2.11}$$

The MLP learns the weights $\boldsymbol{w}$ that result in the best function approximation of g. Multilayer perceptrons are formed out of small processing units, called nodes or neurons, which are connected by weighted connections. After presenting some input $\boldsymbol{x}$ to the network it is activated. The activation then spreads through the weighted connections to the units in the hidden layers, until it reaches the output layer where the output of the network is computed. Each perceptron in a hidden layer receives input from several other upstream neurons and computes its own output through the activation function. The output only relies on the current input and is not dependent on any past inputs because of the absence of recurrent connections [11, 22].

More precisely, when an MLP is activated by an input vector $\boldsymbol{x}$ each of the hidden units in the first layer $\boldsymbol{h}_1$ calculates a weighted sum of the input units. The weights of the network are stored in a matrix W. The weight of the connection between neuron i and neuron j is denoted as $w_{i,j}$. Also, a vector of biases is needed, denoted as $\boldsymbol{b}$, where the entry b_i represents the bias weight of the i-th neuron. For the hidden unit k with the associated bias weight w_{0k} the sum a_k is calculated as [13]:

$$a_k = \sum_{i=1}^{n} (w_{ik} x_i) + w_{0k}. \tag{2.12}$$

The activation function σ is applied on each weighted sum a_k to receive the activation b_k [11, 22]:

$$b_k = \sigma(a_k). \tag{2.13}$$

The output of the first hidden layer is then used as input of the next hidden layer and the output of the last hidden layer is used to compute the output of the model in the output layer. This sequence of operations is called *forward pass* [11]. The activation of the neurons in the output layer determines the output vector $\boldsymbol{y}$. It is calculated the same way as for hidden neurons. The number of neurons in the output layer and the activation function in the output layer depends on the task that should be solved by the MLP [22].

In most cases when using MLPs a nonlinear function is needed to describe the features. Therefore a nonlinear activation function is used. Using a linear activation function is only suitable for MLPs with one hidden layer because an MLP with more than one hidden layer and a linear activation function is equivalent to some other MLP with only one hidden layer. Combining linear operators is itself a linear operator. Examples of nonlinear activation functions are the sigmoid function, the rectified linear unit (ReLU) or the hyperbolic tangent function. If the activation function is differentiable the network can be trained using gradient descent [22].

2.4.2.1 Training a MLP

Training an MLP is a very similar process to the training of a perceptron. Since MLPs are in most cases differentiable a gradient-based method is preferably used. The aim of gradient descent is to find the derivative of the loss function with respect to each of the network weights. The loss function depends on the task the MLP should solve. For an efficient calculation of the gradient, a technique called *backpropagation* can be considered, which is the repetitive application of the chain rule for partial derivative [22]. The error is computed using an error function and comparing the outputs of the network with the labels of the examples. The information about the error is then propagated back through the layers to compute the gradient [11]. If the convergence of gradient descent is too slow for an application other optimization methods using conjugate gradient and second-order methods can be considered [13].

Another version of gradient descent is the stochastic gradient descent. Using this algorithm, the samples are not presented one by one but *mini-batches* are built out of the training data. This extension was invented because learning large datasets which is necessary for good generalization can get computationally expensive. To reduce the number of forward passes the algorithm needs for training, more training samples are combined to a mini-batch

$$B = \left\{ x^{(1)}, \dots, x^{(m)} \right\}, \tag{2.14}$$

which is constructed by drawing uniformly m examples from the training set $\boldsymbol{x}$. The $mini-batch-size$ is typically chosen to be from one to a few hundred examples and is a hyperparameter that can be optimized. The weights of the ANN are only updated after each mini-batch [11].

2.4.3 Recurrent Neural Network

Recurrent neural networks (RNN) are a subgroup of artificial neural networks. In addition to the feed-forward connections, an RNN has self-connections or connection to neurons in upstream layers. [13]. In contrast to an MLP, which can only map from an input vector to an output vector, an RNN can map from the entire history of previous inputs to the output. The recurrent connections

act as a memory of previous inputs and influence the output of the network. The forward pass in an RNN is similar to the forward pass in an MLP, except that the neurons in the hidden layers get additional inputs from hidden layer activations of the previous time step. An input vector $\boldsymbol{x}$ with length T is presented to a RNN with I input units, H hidden units and L output units. The entry x_i^t is the value of the input i at time t and a_j^t and b_j^t are the inputs of the node j at time t. For each hidden unit h at time step t the activation is computed as [22]:

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{j=1}^{H} w_{jh} b_j^{t-1}. \tag{2.15}$$

After calculating the weighted sum, the activation function is applied in the same way as for an MLP [22]:

$$b_h^t = \sigma_h \left(a_h^t \right). \tag{2.16}$$

The equation 2.15 requires initial values for all b_i^0, corresponding to the RNN's state before receiving any input. There are different approaches how to chose these initial values. One possibility is to set all of them to zero. The network's output can be calculated in the same way as at the MLP. The output layer has no recurrent inputs because it is the last layer in the network [22, 11].

2.4.3.1 Training a RNN

Similar to the MLP the RNN uses partial derivatives of the loss function to determine the derivatives. Because of the dependence on earlier time steps, normal backpropagation cannot be used. Most commonly used is an algorithm called *backpropagation through time* [11]. This is further discussed in Section 3.3.1.

2.5 Architectures

An approach to visualize the architectures of algorithms is to use *computational graphs*. The *unfolding* of a recurrent or recursive computation over time into a computational graph, leads to a graph with repetitive structure. The parameters of the network are shared across the chain of events and the shared parameters do not change their value over time [11].

2.5.1 Sequence-Vector Model

Sequence to vector models take a sequence $x = \left\{ x^{(t)}, x^{(t-1)}, \dots x^{(1)} \right\}$ and produce as output a fixed sized vector $\boldsymbol{b}$. Such models are implemented to summarize a sequence and output a fixed size representation that can be used as input of another model. An example of this architecture is

shown in Figure 2.3. The RNN produces a fixed-sized output $\boldsymbol{b}$ at the end of the sequence. The loss $\boldsymbol{L}$ measures the distance between the output $\boldsymbol{h}$ and the labels $\boldsymbol{y}$. An example of a sequence to vector model is an RNN, which take an input sequence and tries to predict the next entry in the sequence. This architecture processes the input sequence x by incorporating it into state $\boldsymbol{h}$ which is passed forward through time. The black square in Figure 2.3 on the self-connection in the left diagram indicates the delay of one unit [11].

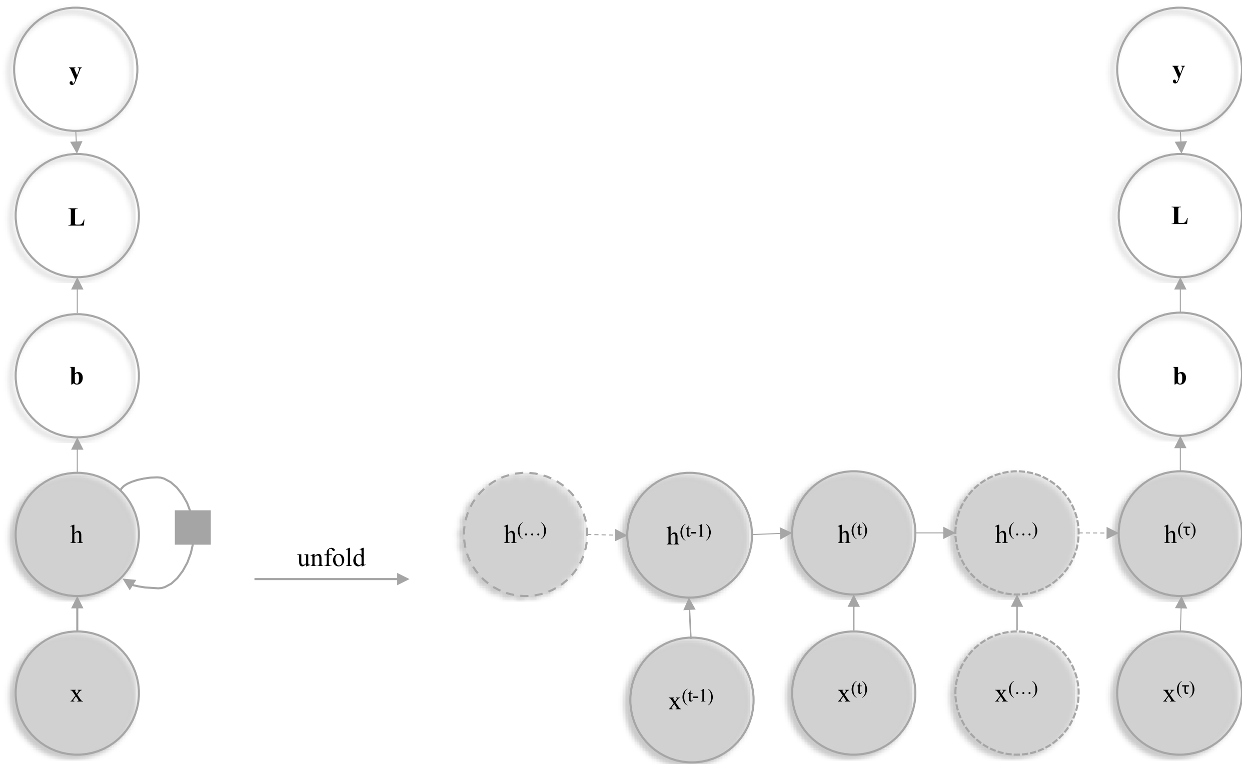


Fig. 2.3: Unfolding a sequence to vector model (cf. [11]): the grey instances are representing sequences and the white vectors. The sequence x is the input to the model that is followed by the hidden state h which has a self-connection with a delay of one time step, represented by the square on the connection. L is measuring the difference between the output of the model b and the label y. This network can also be unfolded into an unfolded computational graph where each node of the sequence is linked to a particular time step.

2.5.2 Vector-Sequence Model

A vector to sequence model maps a fixed-sized vector $\boldsymbol{x}$ into a distribution over sequences y. An example of this type is an architecture build for image captioning. The input is an image and the output a sequence of words that are describing the image. In Figure 2.4 an example of an RNN is shown that takes a fixed-sized vector $\boldsymbol{x}$ and produces out of it a sequence output y. Each element

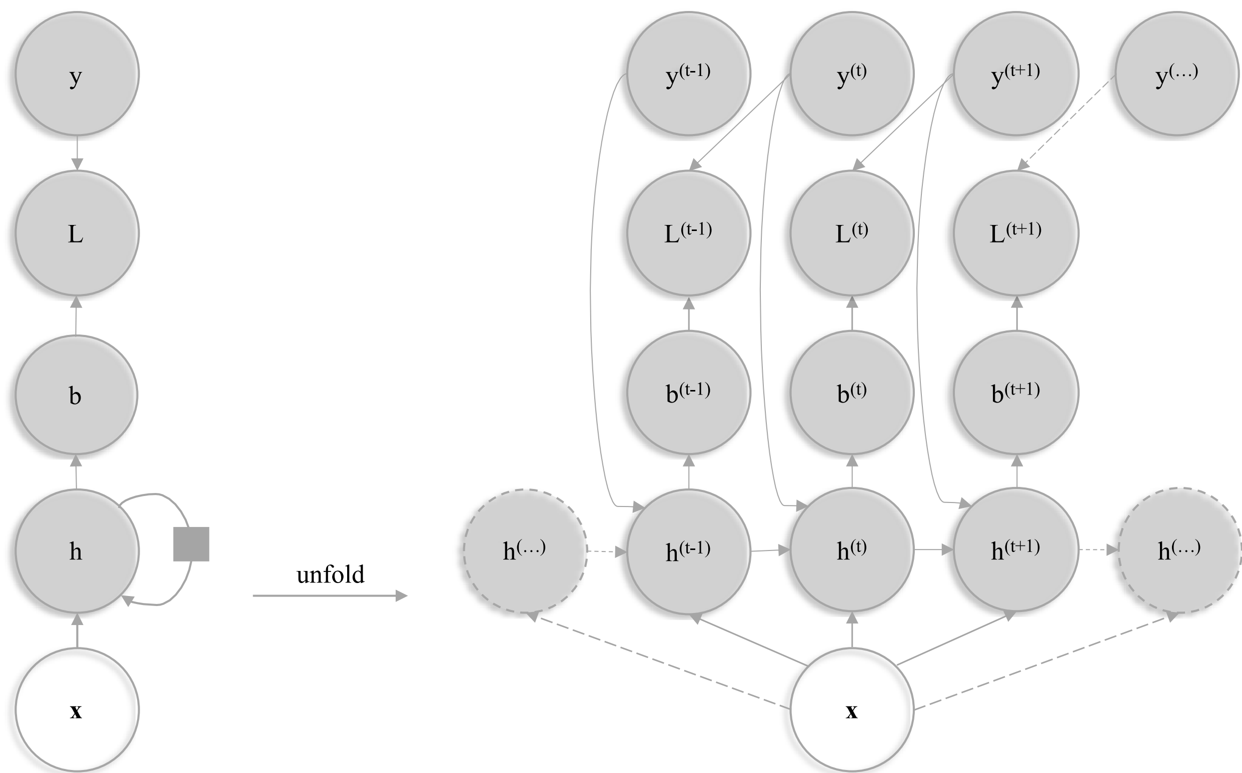


Fig. 2.4: Unfolding a vector to sequence architecture (cf. [11]): A fixed-sized vector x is used as input. The information is passed on to the hidden state sequence h and then produces an output sequence b. L measures the difference between the expected output y and the archived output b. Note that the entry of y at a certain time step is used as input for the current time step and as the target of the previous.

$y^{(t)}$ is at the same time the input for the current time step and the target of the previous time step [11].

2.5.3 Sequence-Sequence Model

Sequence to sequence models are recurrent networks that map an input sequence x to a corresponding sequence of output values b. These models include networks that produce an output at each time step and have recurrent connections between hidden units as well as models that only have recurrent connections from the output of one time step to the hidden unit of the next time step [11].

An example of a sequence to sequence model is the architecture shown in Figure 2.5. This RNN maps an input sequence x to a sequence of output values b. L is the loss that measures the distance between the elements in b and the corresponding target values in y [11].

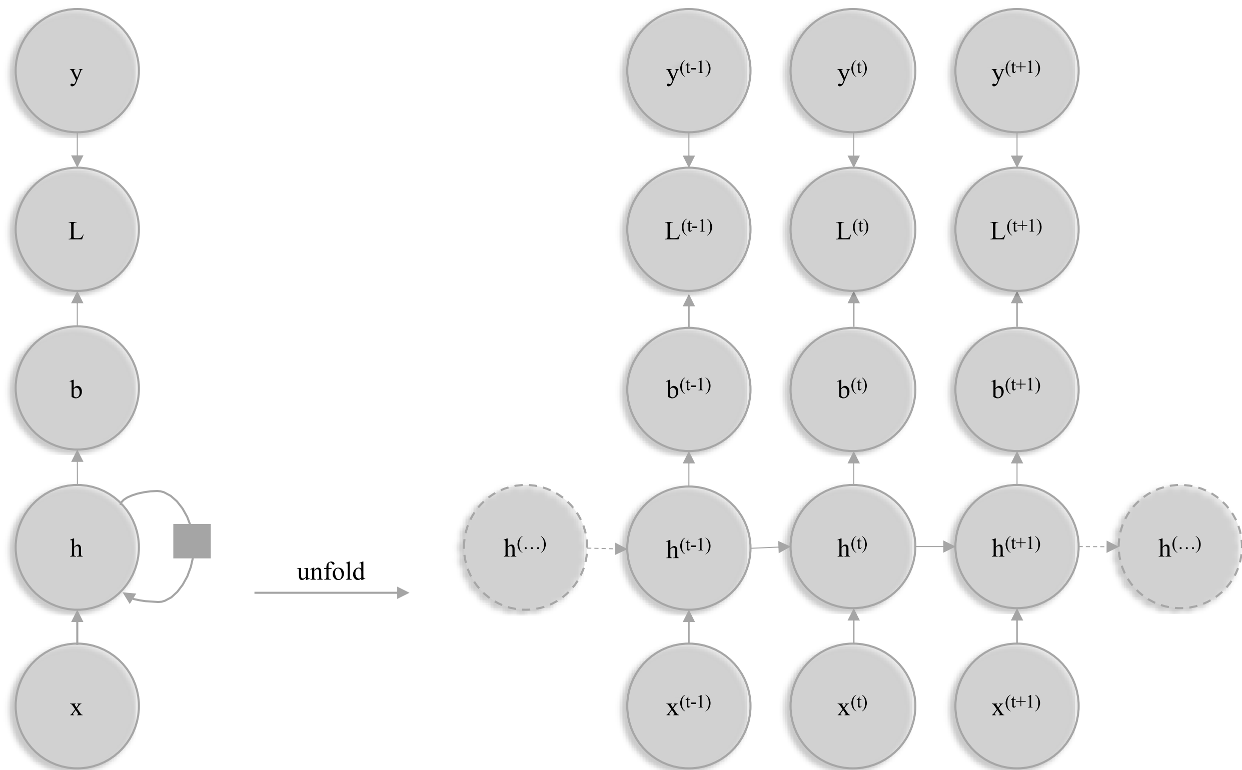


Fig. 2.5: unfolding a sequence to sequence model (cf. [11]): A sequence *x* is used as input. The information is passed on to the hidden state sequence *h* and then produces an output sequence *b*. *L* measures the difference between the expected output *y* and the archived output *b*.

2.5.3.1 Encoder-Decoder Architecture

The encoder-decoder architecture is a special case of sequence to sequence learning. With this type of architecture an input sequence can be mapped to an output sequence, but the two sequences are not necessarily the same length. This is archived by an encoding step followed by a decoding step. In the encoding step, a sequence to vector representation is generated and in the decoding step, the vector is mapped to a sequence [11].

2.5.3.2 Autoencoder

An *autoencoder* is a neural network model that can be seen as a special form of MPLs for unsupervised learning. An autoencoder is the combination of an encoder function and a decoder function. The encoder maps the input data into so-called *latent variables* in the *latent space* [23]. Latent variables are variables that are part of the model but cannot be observed from outside and are therefore not part of the input dataset [24]. The decoder function converts the representation found by the encoder back into the original domain. In most applications the architecture of autoencoders is designed in a way, that they are not able to produce perfect copies of the input but instead have to learn how to produce an approximation of it [11]. The difference between the input of the encoder

and the output of the decoder is the residual from which the reconstruction loss can be derived. The target of the autoencoder during training is to minimize this residual [25].

The autoencoder is a possible implementation of a encoder-decoder architecture, therefore it consists of a encoder with a encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder that produces the reconstruction $\boldsymbol{r} = g(\boldsymbol{h})$, whereby $\boldsymbol{h}$ is the vector of the latent variables for the input $\boldsymbol{x}$ [11]. The number of output nodes of the decoder is the same as the number of input nodes of the encoder because the network is trained to reproduce the inputs. Further, the number of neurons in the output layer of the encoding network should be the same as the number of neurons in the input layer of the decoding network. If this number is lower than the number of input neurons this layer where the encoding and decoding step are connected is often referred to as a bottleneck. In this case, the network is trained to find the best representation of the input on the reduced number of neurons. While doing this it is performing a dimensionality reduction. In the following decoding step, the reduced representation is reconstructed in the best possible way [13]. Such autoencoders are called *uncomplete autoencoder*. If the autoencoder learns the incomplete representation it is forced to extract the most prominent features. The learning can be described as minimizing a loss function L which is measuring dissimilarities between the reconstructed representation $\boldsymbol{r}$ and the input of the encoder [11]:

$$L(\boldsymbol{x}, \boldsymbol{r}) \,. \tag{2.17}$$

Another variation of the autoencoder is a *denoising autoencoder*. They are inspired by the human ability to recognize objects even if they are partially hidden. This type of autoencoders is trained by adding noise to the training data. The model is trained to recover the original, noise-free, image [23].

2.5.3.3 Variational Autoencoder

A *variational autoencoder* (VAE) is in contrast to the autoencoder, which is a discriminative model, a generative model. A discriminative model aims to learn a predictor given the observation. A generative model learns a joint distribution over all the variables. The aim is to find a generalization of the data generation process. To fulfil this task generative models learn the mapping between the observed space of inputs with a typical complicated unknown distribution $P_d(x)$ and the latent space which has in most cases a relatively simple distribution P [24]. The goal is to learn the simpler distribution P that it is as similar as possible to $P_d(x)$ [26].

The VAE consists of the autoencoder of two coupled parts: the encoder and the decoder. Each of the parts is independently parametrized. The encoder delivers an approximation of its posterior over latent random variables to the decoder. The encoder is a (stochastic) function of the input variables and uses a set of parameters to model the relationship between the input and the latent variables. Every data point in the input dataset should have at least one setting of the latent variable. VAE is often designed in a way that the output of the encoder is a Gaussian distribution described

by standard deviation and mean [26]. The decoder is activated by random variables. The distribution of the random variables has been learned from the data. An autoencoder (see Section 2.5.3.2) produces a point in the latent space, whereby a VAE produces a distribution over possible values in the latent space [21].

Chapter 3
Time Series

3.1 Definition

A time series is a set of observations $\mathscr{M}$ whereby each observation $x_t \in \mathscr{M}$ was recorded at a specific time t [27]. A time series that contains only records of one variable is called *univariate*. If it contains measurements of more than one variable it is a *multivariate* time series [28].

3.2 Components

Time series data is affected by four main components which can be separated from the data. These four are: the trend as well as cyclical, seasonal and irregular components. The trend is the general tendency of the time series. The tendency can be weather the series decreases, increases or stagnates over a long period of time. In other words, the trend is the long term movement of the time series. The seasonal component is a specific repetitive yearly pattern. As the name indicates, cyclic components are medium-term changes caused by effects that occur in cycles. Not repeating random variations which do not follow a particular pattern in time are the irregular component [28].

3.3 Characteristics of Time Series in ML

One common approach to do time series modelling is using statistical methods. Some of the most commonly used are all kinds of autoregressive models and moving average models or a combination of those. Another approach is using machine learning, especially ANNs, for this task. An advantage of ML models is that non-linear modelling can be done. The majority of time-series data recorded in a real-world environment show non-linear patterns. Another advantage is, that ML models are data-driven and self-adaptive [28].

3.3.1 Backpropagation Through Time

A common choice for time series forecasting are ANNs which implement some kind of recurrence. So, for the learning of these models, normal backpropagation cannot be used. Instead, another version called *backpropagation through time* (BPTT) is applied. Similar to the backpropagation used for MLP training, also BPTT consists of repeated application of the chain rule [22]. Computing the gradient of a network with recurrent connections can be seen as applying the classical backpropagation algorithm on a graph that is unrolled through time [11]. So an RNN with N time steps can be seen as an MLP with N layers. The first layer corresponds to the first time step. Calculating the gradients is done recursively, so it is started at time instance N, at the last layer, and propagates backwards in time until the first layer is reached. Therefore this algorithm is called backpropagation through time. Because of this property that a recursive network can be seen as a feed-forward multilayer network the downsides of normal backpropagation are also occurring in BPTT, namely vanishing or exploding gradients. This occurs because the derivative of the $r-1$, $r = \{1,\dots,N\}$, layer depends on the derivatives of the layer above and on the weights in the current layer. Both these terms are multiplied recursively and the number of therms involved grows after each step as the algorithm flows backwards. Depending on the used activation function the values of the derivatives can be smaller than one. If the parameters are small the gradients of the cost function with respect to the parameters of the lower layers can take very small values and vanish. This makes the training very slow. This can also happen in the other direction if the values of the parameters take very large values. This can lead to the effect that the gradients blow up. The deeper the network is, the more prominent this occurrence is. In time series learning, long sequences are often used so one should pay attention not to run into this problem [21]. A heuristic approach to solve this problem is the clipping of gradients at some maximum value. Another proposed solution was using LSTMs (see Section 4) instead of RNNs [29].

3.3.2 Time Series Forecasting

The task of predicting values of the future of a given sequence by using historical data is called time series forecasting [18]. The goal of time series modelling is to study past observations and develop an appropriate model to describe the structure of the underlying time series. This model is used for forecasting future values of the time series. Time series forecasting can be described as the act of predicting the future by understanding the past. There are a lot of statistical models to approach this forecasting task [28].

3.3.2.1 One-Step Ahead Forecasting

When one-step ahead forecasting is applied and the observations of the last n time steps are known $Y = \{y_1, \dots, y_n\}$, then the value y_{n+1} for the time step $n+1$, is predicted [18].

3.3.2.2 Multi-Step Ahead Forecasting

Given a historical time series containing n observations time steps $Y = \{y_1, \dots, y_n\}$ the task when using multi-step ahead forecasting is to predict the values of the h next observations $Y_f = \{y_{n+1}, \dots, y_{n+h}\}$, where $h \geq 1$ and h is called the forecast horizon [30].

3.3.3 Outlier Detection

A pattern that normally doesn't occur in that way in the dataset is called an *outlier*, or anomaly. These patterns can indicate that an unusual event happened. It can be also caused by errors in the measurement or during the procedure of data collection [1]. Identifying unlikely or rare events is called outlier detection [31]. When applying outlier detection the task is to determine for every time step if the data measured at that time step is an outlier or not. Given a time series $Y = \{y_1, \dots, y_n\}$, where y_i refers to the data measured at time step i, at time step i the last L observations are taken in account, $t = \{L, \dots, n\}$, to determine if the measurements at time step i contain anomalous data or not. The output should be binary and 1 indicating that the sample is anomalous and classified as an outlier [32]. To classify dissimilarities a so-called *anomaly-score* is commonly used, which measures the deviation from the data that is considered to be non-anomalous. If this score exceeds a certain pre-defined threshold, the sample is considered to be an outlier [33].

Outliers can also be found by visually inspecting the data but according to [34] this technique is on its own not reliable enough to identify outliers. Therefore algorithms have been developed for this task and also ML can be applied. Like other ML tasks, also outlier detection is divided into supervised, unsupervised and semi-supervised methods. If the data the ML-model used for the outlier detection is labelled and the labels are indicating if an error occurred in that specific sample, the outlier detection is called supervised. Then the model is trained with the labelled data to predict the labels of unseen examples [1].

Semi-supervised methods are those, where only labels for either anomalous or non-anomalous data are available. A model that performs a classification is trained in a supervised way using the available data and is then used to predict the labels of the unseen data [1].

As summed up in [35] when applying outlier detection in practical applications these three main problems occur:

1. **Imbalanced data**: anomalous data occurs less frequent in the data then non-anomalous data. This can lead to overfitting and lack of generalization in trained models.

2. **Labelling costs**: if data needs to be labelled by human experts, it is difficult to create big amounts of data for testing and training.
3. **Presence of unseen anomalies**: there are a lot types of anomalies in practical applications. Some are not known before they occur the first time.

It was found that autoencoders can be used for outlier detection. The time-series data that should be analysed is often high-dimensional. An autoencoder then performs dimensionality reduction and extracts the most representative features out of it. Because outliers are often non-representative features they can be found by reconstructing the compressed representation and calculating the reconstruction error. An autoencoder is often not able to reconstruct anomalous data, so the reconstruction error is high and this can indicate an outlier [35].

Chapter 4
Long-Short Term Memory

The network structure called *long-short term memory* (LSTM) is a variation of RNNs. Unlike RNN, LSTM can store and retrieve information over long time periods. [29]. According to Hochreiter and Schmidhuber [3], the founders of the LSTM, the model was introduced due to the problem that backpropagation takes a very long time and to solve the problems occurring because of exploding/ vanishing gradients when learning long time sequences. While RNN can use the recurrent connections to store activations from the recent past (short term memory), LSTM can learn to bridge long time lags [3]. This architecture is widely used for learning of time series data, because of their ability to learn long-term dependencies [18].

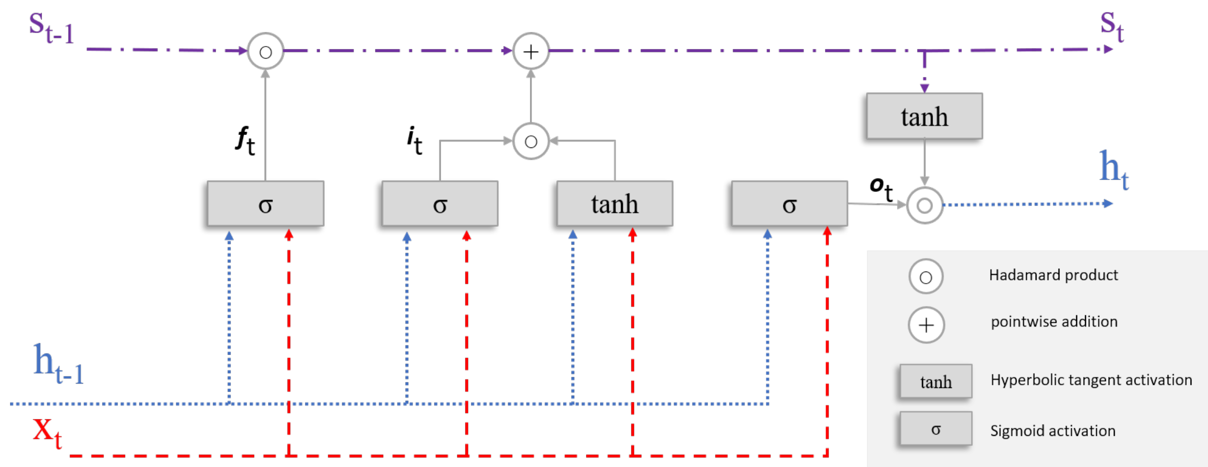


Fig. 4.1: Basic LSTM architecture and information flow when the architecture is activated by an input x_t at time step t. The information flows through the gates f_t, i_t and o_t. The cell state at time step t is denoted with s_t and h_t denotes the hidden vector at time t.

4.1 Basic Architecture Description

To control the outflow of information in the system memory and the inflow of information the LSTM architecture incorporates so-called *gates*, which are non-linear elements. These gates consist of a sigmoid activation function and a multiplier. Algorithmically seen the gates are equivalent to weighting the information flow between zero and one, whereby the value depends on the value of the involved variables. This weighting ensures that the model is and stays differentiable. The weight is the activation of the sigmoid activation function when this value is used as input to it [21]. Each LSTM has three binary gates which are represented by the vectors $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, $\boldsymbol{o}_t$, the forget gate, the input gate and the output gate at time step t. The gates control, whether the memory cell is updated, set to zero, or whether the local state is incorporated in the hidden vector [29].

In addition to the hidden state vector, which is also used in RNNs, the LSTM incorporates a cell state, which is represented by a cell state vector $\boldsymbol{s}_t$ at time step t. The key element of the LSTM structure is the cell state, also called the memory cell, which can be imagined as a conveyor belt. It runs through the whole structure [18, 29].

As visualized in Figure 4.1 the cell state is influenced by the following three terms (from left to right in Figure 4.1):

1. In the first step the LSTM decides which piece of information is thrown away. This decision is made by the forget gate $\boldsymbol{f}_t$;
2. The second term provides information that should be stored in the cell state. This step has two folds: the input gate $\boldsymbol{i}_t$ decides which values are updated and the hyperbolic tangent activation creates new candidate values;
3. Then the cell state at the previous time step $\boldsymbol{c}_{t-1}$ is updated to the new cell state $\boldsymbol{c}_t$.

The output gate decides which parts of the cell state are incorporated into the hidden vector at time t $\boldsymbol{h}_t$. It is based on the cell state but a filtered version of it [18].

To overcome the problems with BPTT, LSTM uses a constant error flow and truncates gradients where it is possible within special multiplicative units. These are the units that learn to close and open the gates to ensure constant error flow. In LSTM networks long-term information with significant delays is approximated [18].

4.2 Combination of LSTM and VAE

As mentioned in Section 3.3.3 autoencoders were used in the past successfully for outlier detection. Also the generative type of autoencoders, variational autoencoders, were applied to this task. An emerging idea for outlier detection in time series is the use of a hybrid model consisting of VAE incorporating LSTM layers. In literature this combination is often referred to as *LSTM-based variational autoencoder* (LSTM-VAE) [25]. This was for example used in [25], [36], [37] and [32].

Developing a reliable outlier detection algorithm can be quite a challenging task. How the outliers look is in some cases not known before they occur the first time. So in the best case, the algorithm should be able to categorize prior unseen outliers as such. In most cases, the available data sets used for training are often unbalanced and contain way more non-anomalous data then anomalous data that represent outlier. An advantage of this hybrid model is, that it can be trained in an unsupervised way with unlabelled data [32].

To overcome these constraints of unbalanced amounts of data the autoencoder is trained only with non-anomalous data. The key idea is that the model cannot reconstruct patterns in unseen data that contains anomalies and so the reconstruction loss on this data is higher than when reconstructing data without anomalies. One possibility for outlier detection in the latent space is to sum up the log-likelihoods of the current observation and comparing it to a given distribution. If the value of the summation is below a predefined threshold it is determined as an outlier [25]. Another possibility is to measure the magnitude of the reconstruction loss. If the samples are normal the VAE can reconstruct them with a reasonably low reconstruction loss. Therefore a predefined threshold is needed as in conventional outlier detection that the outlier score can be used and a sample is labelled as anomalous if the threshold is exceeded [33].

However, it can occur that outliers in the original domain do not occur as outliers in the latent space due to the compression done during the encoding. Another method of detection is by reconstructing the current observation and measuring the reconstruction loss. If the reconstruction loss is above a predefined threshold, the observation is marked as outlier [25].

The LSTM part of this hybrid approach is used to model the time series data with its long-term dependencies [25].

Chapter 5
Genetic Algorithm

5.1 Description

Genetic algorithms (GA) belong to the class of so-called evolutionary algorithms which are inspired by the principles of evolution and inheritance. The naming associated with this class is heavily inspired by biology. The underlying idea is the ”survival of the fittest”. At each iteration, a set of potential solutions, referred to as *population*, is produced. Each population $P(t) = \{x_1^t, \ldots, x_n^t\}$ consists at time step t of n individuals x_i, where $i = \{1, \ldots, n\}$. Each individual represents a possible solution to the problem which the algorithm tries to optimize [38].

Further, GA assumes that every distinct possible solution can be described by a set of parameters, called chromosomes in the GA jargon [39]. The individuals consist of one or more chromosomes, each formed by one or more genes, entries in the encoding. The value of a single gene is called an allele [6]. All possible solutions present in the current generation are evaluated to provide some measure of the quality how well they solve the problem. This measure is called the *fitness* of the individual and ranks the individuals depending on how good their represented solution fits the problem. This is done by the underlying fitness function that reflects the problem that should be optimized. This value is a quantitative information that guides the search [6, 38].

In literature the highest fitness is often considered to be the best, in other words, the optimization task is formulated as a maximization problem. However, in the Matlab implementation of genetic algorithms, it is the other way around and the lowest fitness value is considered to be the best (see [40]). Because the coding part of this work is done with Matlab (version 2020b), the Matlab notation will be used in this work. So, in this case the fitness function corresponds to a minimization problem.

The process of creating the next population contains the following steps: firstly some individuals from the current population are selected following the applied selection rule (see Section 5.4.1). These individuals form the *mating pool*. After this, some members of the mating pool are transformed through the genetic operators (see Section 5.4). These transformed individuals are the population of the next time step. Intuitively, the genetic algorithm produces iteratively better individuals, while creating new generations by applying simple operations. The algorithm converges,

hopefully, after a number of generations. If the search of the genetic algorithm was successful, the best individual of the last generation represents a solution near the global optimum. The initial population is created randomly or with the help of some construction heuristic [6, 38]. In general, all algorithms that obtain a population, try to archive a better population in every iteration by some local perturbations, called mutations (see Section 5.4.3), and the combination of different members, called crossover (see Section 5.4.2) [41].

The combination of these genetic principles into an algorithm can be used to search an optimal solution to a problem without using any derivative information as required for many other search algorithms [6, 42]. A search algorithm has several possible solutions and the task is to find the best solution in a fixed number of time steps. If there is only a small number of possible solutions in the search space, this can be done by trying out all possible solutions, this approach is called exhaustive search. As the search space increases and with it the number of possible solutions is growing, exhaustive search becomes very quickly impractical because it would take to long to evaluate all possible solutions. Traditionally search algorithms evaluate one solution at a time step and are hoping to find the optimal solution. Whereby evolutionary-based algorithms evaluate a larger number of individuals and perform a directed search [6].

5.2 Representation

Every problem that can be formulated as an optimization problem, can be solved by applying some kind of evolutionary algorithm. To apply some evolutionary algorithm to a distinct problem it requires a data structure that can encode all possible solutions. This representation is commonly chosen by humans intuition. It should allow applying the genetic operators on the encoded solutions by maintaining the property that small changes in the parents lead to small changes in the offspring. Large changes in the parents' genes lead to much bigger changes in the next individuals of the next generation [6].

5.3 Initialization

The creation process of the first generation differs from all subsequent generations. Unlike the following generations, which are obtained by applying genetic operators, the first one is the initialization of the algorithm. A bad initial guess can lead to the optimum of the objective function not being found. If the search landscape of a given problem is not known in the beginning, it is hard to determine if a guess is "good" or "bad", therefore often an initialization with random numbers is chosen. The risk by doing so is that if the dimension of the search space increases, the chances that a random guess is in a promising region decreases [43].

5.4 Genetic Operators

The population of the next iteration is created by applying the genetic operators selection, crossover and mutation to the previous population [38]. In the traditional form of the genetic algorithms, the solutions are encoded by a fixed-length bit-string representation. In this encoding, every position of the string represents a particular feature of the solution or the whole string can represent a number. However, this encoding is not natural for many problems and in some cases, corrections are required after applying genetic operators [6].

As described in [6] and visualized in Figure 5.1 every iteration of the genetic algorithm consists of the following steps:

1. **Evaluation**: The fitness of the individuals is evaluated;
2. **Selection**: In the second step the individuals who are candidates for the reproduction are chosen;
3. **Reproduction**: The offspring are formed by the selected individual out of the candidates by using one or both of the genetic operators crossover and mutation;
4. **Replacement**: The new population is formed by replacing the individuals from the previous according to the replacement strategy.

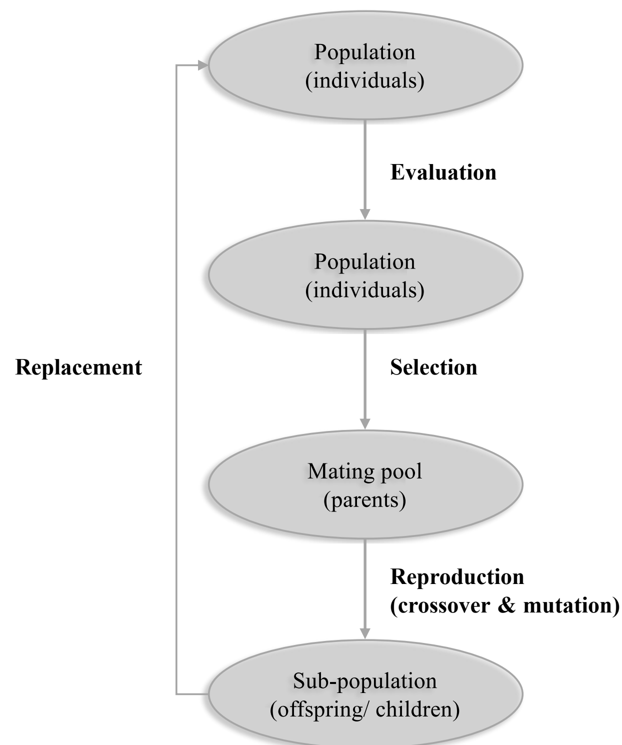


Fig. 5.1: cycle of operators for a classical GA

These steps are repeated until the stopping criterion, which could be convergence, the number of maximal generations or no better solution was found for a predefined number of generations is met. Also a combination of two or more stopping criteria can be applied [6, 44].

5.4.1 Selection

Selection typically ensures that the population size stays constant over all generations. The first step of selection is selecting the individuals for reproduction. In most selection schemes the individuals with the best fitness are favoured. The degree of this favouring is called the *selection pressure*. The aim of the selection pressure is to produce new solutions in regions of the search space of previously found good solutions. This should lead to the effect that the mean fitness of a population gets better as the algorithm proceeds. Therefore, the convergence of GA is highly influenced by the selection pressure. A higher selection pressure leads to high convergence rates. If it is too low the algorithm will need a long time to find the optimal solution. On the other hand, if it is too high it increases the risk that the algorithm converges through a local optimum [6, 45].

5.4.1.1 Roulette Wheel Selection

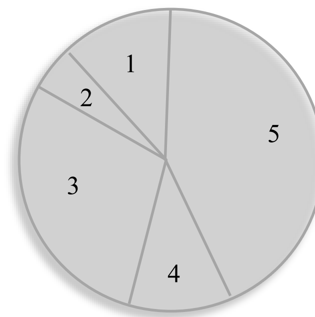


Fig. 5.2: roulette wheel selection of five individuals, better individuals are favoured in the selection process by assigning them bigger slots on the wheel

Roulette wheel selection is the selection technique used in traditional GA approaches. In this selection scheme individuals are chosen to be members of the mating pool depending on their relative fitness. The roulette wheel selection works in that way, that every individual of the population gets a slot in an interval weighted proportional to its fitness value. A better fitness value leads to a bigger slot. A random target value is sampled in the interval and it is evaluated in which slot it falls into. The roulette wheel can be imagined as shown in Figure 5.2. In the example illustrated the individual five has the best fitness and individual two the worst. The individual that is associated

with the slot the target value lies is selected as a candidate for the mating pool. This is repeated n times, if n is the number of individuals in the population [6].

This technique has only a moderately strong selection pressure since it is not guaranteed that the fittest individuals of the population are selected, although the chances are high. On the other side, the diversity in the population decreases, in most cases, very quickly. If some few individuals have very good fitness, then the chances of the other individuals to be selected is very low [6].

5.4.1.2 Rank Selection

This selection scheme selects individuals for the mating pool depending on their rank in the population. The best individual gets the fitness value 1 and the worst individual gets the fitness value n, if n is the number of individuals in the population. This leads to a much lower convergence compared to roulette wheel selection because the genetic diversity is better preserved over a longer time. It also keeps selection pressure high if the variance in the fitness values of the fitness function is low. The parents can be selected with different approaches. One possibility is selecting two random pairs out of the individuals. As parents the individuals with a lower rank in each of the random pairs are selected [6].

5.4.1.3 Tournament Selection

When using the tournament selection, the selection pressure is created by holding a tournament competition among m individuals, while $m \leq n$. The winner of the tournament is the individual with the better fitness value. The winner of the competition is added to the mating pool. This is repeated until enough individuals are in the mating pool to create the next generation [6].

5.4.1.4 Elitism

Elitism can be used in combination with other selection methods and forces the GA to perceive the best individuals and passes it on to the next generation. If elitism is not used the best performing individuals can be destroyed after applying crossover and mutation [46].

Another selection scheme is the combination of rank selection with elitism. If this selection scheme is applied, the individuals of a population are ranked and then a fixed number of the best individuals are selected to form the mating pool [46].

5.4.2 Crossover

Crossover is a genetic operator that creates a new individual, called offspring, by combining parts from at least two individuals which are referred to as parents [38]. Commonly, individuals are selected probabilistically, according to their fitness value, to form the parents of the individuals of the next generations. Another possibility is to select randomly two parents out of the mating pool. If a bit-string representation is used as encoding, the crossover step can be done by a so-called bit-string crossover. In this operator parents, two individuals encoded as strings, are used. Two new individuals are created by swapping a sequence of the two strings. Crossover creates new individuals that are hopefully better than their parents [6].

5.4.2.1 Single-Point Crossover

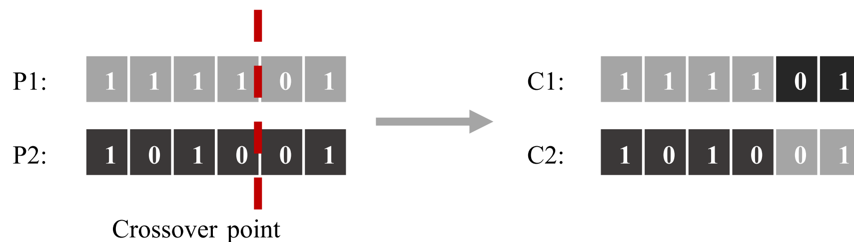


Fig. 5.3: single-point crossover of parents P1 and P2 to offspring C1 and C2

The classical implementations of the GA use the so-called single-point crossover. For this procedure, two parents are used. That is the simplest way to implement the crossover operator. As shown in Figure 5.3, a random crossover point along the length of the parent individuals is chosen. The sections after the crossover point are exchanged between the two parental individuals [6].

5.4.2.2 K-Point Crossover

If a k-point crossover is applied more than one cut point is used to create new individuals. Also more than two parent individuals can be used. In most cases, this reduces the performance of the GA because building blocks, short sequences of alleles that lead to a good fitness value, are more likely disrupted. On the other hand, the changes that the search space is explored more thoroughly are higher than when a single-point crossover is used [6].

5.4.2.3 Uniform Crossover

Another version of a crossover without cuts is the uniform crossover. When applying this technique it is randomly chosen for each gene separately if it is taken from the first or second parent [6].

5.4.3 Mutation

The mutation is one of the genetic operator. It creates a new possible solution by a small change in a single individual [38]. It aims to prevent the solutions to get stuck in local optima. Mutations should recover the lost gene material as well as bring some random portion into the genetic information. This operator helps to explore the search space, which is not reached by using crossover without a mutation step. If a binary representation is used, mutation can be simply done by inverting a bit by

Fig. 5.4: bit-flipping mutation from parent P to offspring C [6]

a very small probability. This kind of mutation is called bit-flipping mutation and is demonstrated in Figure 5.4 [6].

An important parameter of the GA is the mutation probability. If mutation occurs too often, mutation probability is too high, the GA becomes a random search [6]. On the other hand if the mutation rate is too low this can lead to stagnation of the algorithm [45].

5.4.4 Replacement

Replacement is the last step in the process of forming a new population. If a new individual is created, the algorithm needs to decide which individuals to remove, to obtain a fixed-sized population. There are again multiple replacement strategies [6].

One approach is to produce n new individuals and the newly formed individual replace the prior generation. There are some other methods for example the $\lambda + \mu$ strategy where not the whole population is replaced, but there are only λ new individuals out of a population of μ individuals produced [6].

5.5 Hyperparameters of the Genetic Algorithm

When using a genetic algorithm there are some hyperparameters to set. The two most crucial ones for the performance and the runtime are the population size and the number of generations. GA is a meta-heuristic with a stochastic nature. Because if these properties there are no methods known that predict when the genetic algorithm will converge. The problem of the right population size can be thought of as the problem of human decision making and the number of group members involved. If there are only a few persons involved, it is hard to come up with a solution for a very complex problem. On the other hand if the group is too big, the initial results will be worse because

of disagreements of the group members but the final result will be most likely a more successful output [47]. The computationally most demanding step of a GA is the evaluation of the individuals. The upper bound F of this step is given with

$$F = GN, \tag{5.1}$$

where N is the population size and G is the number of maximal generations. There is a trade-off at determining the population size between finding better solutions and the fact that increasing the population size leads to a significant higher number of fitness function evaluations [47].

The next question that occurs when dealing with genetic algorithms is if to choose a higher number of generations or a higher number of individuals per population. The number of individuals needed for a sufficient result also depends on the genetic operators used. For example it was observed that more disruptive crossover techniques, for example n-point crossover or uniform crossover, are leading to better results when smaller population sizes are used. In comparison, less disruptive crossover techniques as 1-point or 2-point crossover does not perform that well when the population size is small. These less disruptive versions on the other hand lead to better overall results when applied on larger population sizes. But also this assumption is only a tendency and depends on the task that should be solved so it cannot be generalized. But it has been shown that when larger populations are used the fitness improves slower but overall leads to better results [47].

5.6 Why GA works

Several attempts try to explain why the idea of GA works. One of them is the *building blocks hypothesis*. Building blocks are considered to be good combinations of values that lead to good fitness values. The name is derived from using bit-string encoding, where bit values build blocks. The building block hypothesis assumes that genetic algorithms discover, emphasize and recombine these good value combinations [46]. Through mutation, several of these building blocks can be combined in one individual by crossover and lead to a better fitness value [6].

An alternative hypothesis for the explanation of the performance of GA is the *macro-mutation hypothesis*. Crossover is seen as a ”macro-mutation”, a mutation of many bits. This can lead to building blocks being passed on by mutation [6].

Also in the *adaptive mutation hypothesis*, crossover is seen as mutation. It is interpreted that it automatically adapts to the stage of convergence the algorithm is in. If it is near convergence the individuals of a population have similar alleles and so it crates offspring in the same region of the search space as the parents are in [6].

5.6.1 Convergence of Genes

When using genetic algorithms it can be observed that not all genes need the same number of generations to be solved. This happens because not all genes equally impact the individuals fitness. Because fitness is what guides the search, the algorithm focuses on fixing the genes which are contributing a larger part to it. If these important portions are solved the GA moves on and tries to improve the fitness further by fixing the genes with lower impact [48].

Chapter 6
LSTM Hyperparameter-Selection using Heuristics based on GA

6.1 Hyperparameter Optimization

Before starting a machine learning algorithm one needs to set the hyperparameters. One option is to use the default values of the software package that is used, if there are any available, default values proposed in the literature or find out a good value by trial and error. Another option is to optimize these values before executing the original program. If the hyperparameters should be optimized, it needs to be defined which one to optimize and which can be set to a default value that works well over a range of datasets. This decision impacts on one hand the quality of the results and on the other hand the convergence and speed of the optimization [49].

Machine learning transforms a problem that should be solved into an optimization problem and different optimization methods are used to solve the underlying problem. The corresponding optimization function contains several hyperparameters. Those are set before starting the learning process. These parameters affect how the learned model is fitted to the data. Before starting the training of the machine learning algorithm the task is to identify the best possible set of hyperparameters in a reasonable amount of time. This is called *hyperparameter optimization* (HPO) or hyperparameter tuning. The relationship between hyperparameters and the performance of ML models is still not completely understood. In practice, the problem of finding a good set of hyperparameters is addressed by training several models with different hyperparameter settings and then choose the best of it [4]. Before applying a machine learning algorithm to a new dataset, the appropriate set of hyperparameters for this specific set has to be found because of the inability to reach the stage of generalization that a found set of hyperparameters maximizes the performance of the applied ML algorithm on all datasets [50, 44].

According to [16] the hyperparameter optimization process is formed by four main components:

1. an estimator: regressor or classifier and the associated objective function;
2. a search space;
3. a method to find the hyperparameter combinations;
4. an evaluation function to compare the performance of the different combinations of the hyperparameters.

The domain of hyperparameters is not always continuous. For example, there can be hyperparameters that take discrete, binary or categorical values. Further, the hyperparameter space can contain conditional hyperparameters, that is if there are dependencies between the values of the hyperparameters that are tuned. Also, the hyperparameters of the model depend on the specific kind of learning algorithm used [16].

6.1.1 Importance/ Impact of LSTM hyperparameters on the Performance

6.1.1.1 Training Parameters

According to [5] one of the most important hyperparameters for an LSTM is the learning rate. It was found that three thirds of the performance on the test set, in comparison to all other hyperparameters, are impacted by the learning rate. The range of values for the learning rate where the performance doesn't vary much is for each dataset quite large [5].

Another hyperparameter that influences the quality of the training process is the number of epochs. When training a neural network over more iterations, the error on the training gets lower at each iteration. But there is a point where the error on the validation set starts to rise again because the model overfits on the training data. At this point the training should be stopped [11].

6.1.1.2 Architecture Parameters

In general, when designing a neural network architecture the number of nodes and their location in the layers, as well as the connections between them has a significant impact on the overall performance of the network [51]. A too high number of neurons can lead to slow training and overfitting and a too low number of neurons limits the networks ability to map complicated relationships [52].

Another important hyperparameter of LSTMs is the hidden layer size. As expected the performance of the network increases with increasing hidden layer size. The downside of increasing the hidden layer size is that the time needed for training increases because the computation of each epoch increases and the convergence speed decreases [5]. All these parameters, regarding the architecture of the model, should be chosen in a way that on one side they provide enough capacity to model the objective function and on the other side not to over-fit [16]. If overfitting occurs while training, the network does not learn how to approximate the function but memorizes each training example instead. The problem with this is, that also the noise present in the data is treated like it would be part of the data and so the generalization skills of the network are lost. Also, the right stopping time of the training is an important parameter to prevent the network to overfit the data [51].

6.1.1.3 Activation Function

The activation function is used to model the nonlinearity in the ML-model [16]. As mentioned in the architecture description in Section 2.4.2 and the basic architecture description of LSTMs in Section 4.1 activation functions are used to model the non-linearities of a deep neural network. The type of the non-linear activation function can vary and is therefore a hyperparameter of the model [53]. Commonly used types of activation functions are [53]:

1. sigmoid:
$$sig(x) = \frac{1}{(1+e^{-x})}, \tag{6.1}$$
2. hyperbolic tangent:
$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{6.2}$$
3. softsign:
$$softsign(x) = \frac{x}{(1+|x|)}. \tag{6.3}$$

This activation functions are visualized in Figure 6.1. The activation functions softsign and hyperbolic tangent have both the range from -1 to 1 but the tails of the softsign are quadratic polynomials and the tails of the hyperbolic tangent are exponentials. Therefore the softsign approaches the tangent much slower [53]. However, when using an LSTM the activation functions are already fixed in the network architecture (see Chapter 4 and Figure 4.1) so they are not optimized, but in other ANN they can be a potential hyperparameter that can be tuned.

6.2 Classical Techniques for Choosing Hyperparameters

There is a distinction between two groups of approaches to approach this task: manual search (see Section 6.2.1) and automatic search (see Section 6.2.2) [4].

6.2.1 Manual Search

If manual search is used hyperparameter settings are tried out per hand. This requires the user's expertise and background knowledge. As the number of parameters to optimize and their range of values increases this method becomes difficult to handle [4].

Promising regions in the search space can be found by using manual search. The goal of this technique is to develop the intuition of how to choose the hyperparameters best for the underlying task [54].

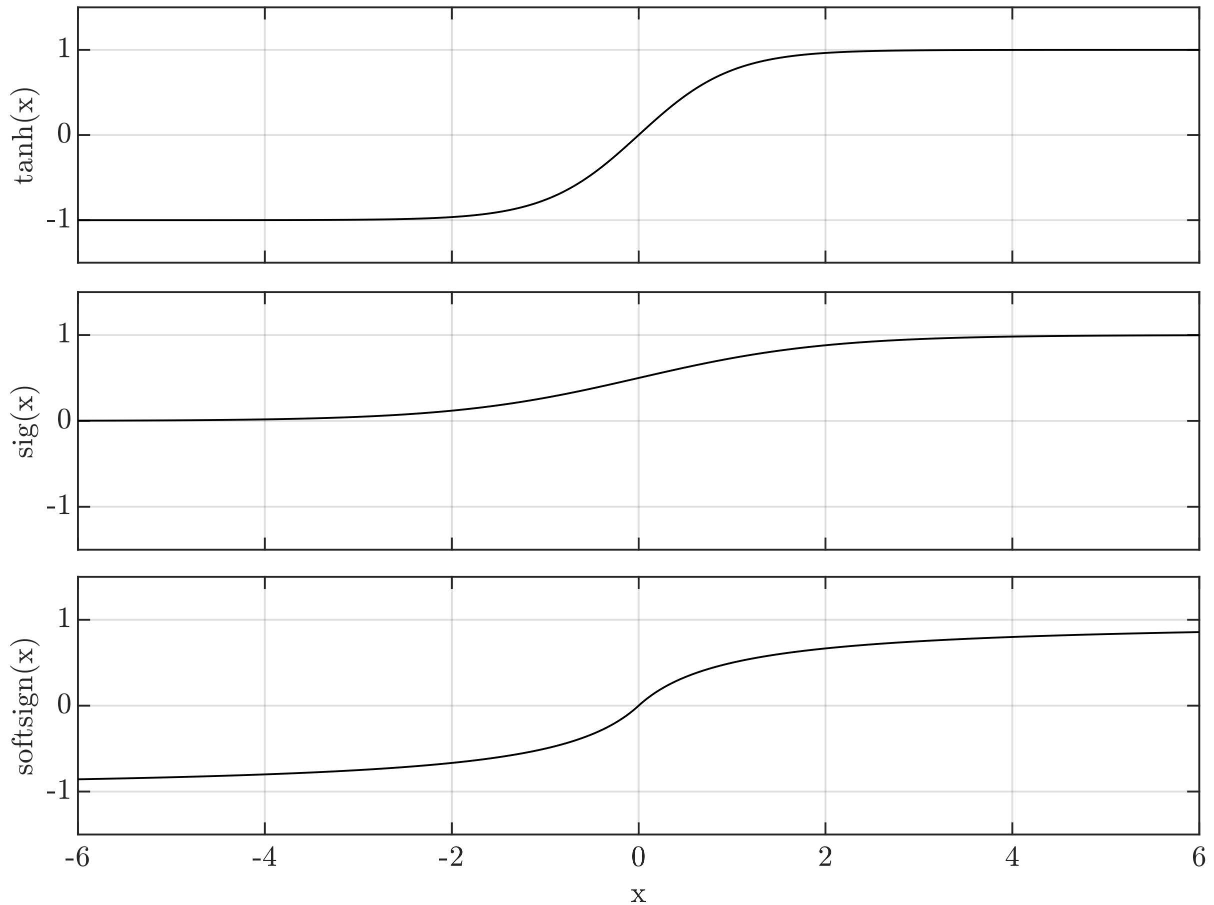


Fig. 6.1: examples of activation functions

6.2.2 Automatic Search

To be able to do hyperparameter search over a higher dimensional space automatic search algorithms, such as grid search and random search, were introduced [4]. When using these two techniques every configuration is evaluated and treated independently [16].

6.2.2.1 Grid Search

When grid search is applied every possible configuration of a predefined, finite set of hyperparameters is tried out and evaluated. The idea behind grid search is applying an exhaustive search over a grid of possible configurations. The output of grid search is the set of values for the hyperparameters with which the best performance was archived [54, 4, 16].

Assuming the hyperparameter optimization should be done over a set of K parameters, grid search requires that a set of possible values, $\left\{L^1 \dots L^k\right\}$ is chosen for every variable. The number of trials is then given as $\prod_{k=1}^{K} L^{(k)}$. This product over k, and so the number of needed trials, grows

exponentially with the number of optimized parameters. Grid search suffers from the *curse of dimensionality* and because of this, it leads to poor results in practical applications. One of the advantages of grid search is that it is very easy to implement and to parallelize and it typically finds better sets of hyperparameters than purely manual search [54].

Grid search cannot explore regions that provide good results further by using a smaller step size in good regions. Therefore the procedure of grid search has to be repeated in good regions by narrowing the search space and reducing the step size, the distance between the evaluation points on the grid. Because of the predefined grid, it is not guaranteed that it detects the global optimum of continuous parameters [16].

6.2.2.2 Random Search

Random search proceeds similar to grid search, but instead of trying all possible combinations on a grid, it samples random solutions in the predefined search space as long as the predefined budget for different configurations is exhausted [16]. Random search performs better than grid search if some of the optimized parameters have a greater impact on the performance than others. If the resources for a random search are sufficiently high, it will find a hyperparameter setting arbitrary close to the optimum [41].

6.3 Hyperparameter Optimization Using Genetic Algorithms

Modern machine learning models have large sets of hyperparameters with complex search spaces. To find a good configuration in a reasonable amount of time, meta heuristics, to which group genetic algorithms belong, have gained popularity in this field [16]. Compared to grid search which is still a widely used technique, GA can find good solutions after fewer evaluations of the objective function but on the other hand, the accuracy of grid search is better under higher computation time [44]. Genetic algorithms identify the best parameter settings in each generation and pass the combinations with the best performance more likely on to the next generation until the best-performing combination is found [16].

To apply genetic algorithms to an HPO problem the hyperparameters need to be encoded in the individuals. One common approach to performing the encoding is to encode every hyperparameter in a chromosome. The value of the chromosome is then the actual input value of the hyperparameter for the ML-algorithm. One possible encoding for the chromosomes is that every chromosome consists of several genes that are binary digits [16].

When using GA for HPO the main limitation is that the algorithm itself again introduces additional hyperparameters that need to be configured. The hyperparameters of the GA are including the fitness function, the population size, crossover rate and the mutation rate [16]. When hyperpa-

rameter search is applied on deep ML architectures evaluating the different hyperparameter settings is often very costly in terms of computational power. There is always a trade-off between optimization performance and runtime. To overcome the enormous long runtimes of optimization runs the HPO is often applied on a small subset of the data, training only a few iterations or using only a few cross-validation folds [41]. When genetic algorithms are applied in the context of machine learning most of the computational time is used for the evaluation of the fitness function [48].

Another problem occurs while evaluating the fitness of the individuals in the generation. The randomly chosen weights at the start of the training lead to different outcomes with the same parameter settings. So a configuration could be considered to be better, but the difference in the fitness is only caused by the noise. To reduce the effects of the noise, the models are trained several times with different random initial weights. This of course leads to an increased evaluation time of the models. So a trade-off between the run time and the presence of noise needs to be found [51]. According to [55] even this procedure is not sufficient to get reliable results. It was shown that when using random initialization of the weights it is possible getting for the same hyperparameter settings learning curves that do not fall into the same distribution [55].

Chapter 7
Statistical Approaches for Outlier Detection

The most commonly used methods in machine learning for detecting outliers are to calculate some kind of outlier score and then use a certain threshold to detect outliers. These methods are often based on statistical methods, which can be biased if the data they are applied to contains outliers. This causes that for the threshold a too high value is chosen and this leads to several outliers not being detected [56] or observations that are non-anomalous being labelled as outliers [34].

According to [56] the most commonly used techniques to get a threshold for outlier detection are standard deviation, mean absolute deviation and interquartile range. But there is the assumption that none of these techniques are suitable for a dataset that contains outliers that deviate strongly from the non-anomalous data [34]. All of these techniques have in common that they are not suitable for applying to non-normal distributed datasets or small datasets. This occurs because depending on the distribution and the number of samples the percentage of expected outliers changes [57]. Outlier detection methods can be in general divided into two groups, formal tests and informal tests and most of them are designed for being applied on a well-behaving distribution. Which test is selected depends on the type of outliers, single outliers or multiple outliers, that should be detected and on the type of distribution the non-anomalous data follows. However, most data recorded under real-world conditions does not follow the well-known distributions, for example the normal distribution, gamma distribution, exponential distribution, or the distribution is unknown. Informal tests provide an interval in which the data is considered to be normal, above and beyond it is categorized as an outlier [57].

Another limitation is if an outlier detection method is sensitive to *masking*. Masking denotes the effect that an outlier ”masks” another outlier. That is, if the second outlier can be detected as an outlier by itself but not if the first outlier is also present in the dataset [57]. The opposite effect of masking is called *swamping* and describes that good data points are marked as outliers [58].

7.1 Outlier Detection for Normal Distributed Data

Most of the standard outlier detection techniques assume the underlying data to be normally distributed. If a normal distribution is assumed, it is also assumed that the distribution is symmetric [59].

7.1.1 Standard Deviation (SD)

This technique defines the threshold by adding to the mean μ a multiple a of the standard deviation σ [56]. The standard deviation can be seen as an indicator of the distance from the mean. A low standard deviation is an indicator that the distance between the data points and the mean is low and a high standard deviation means that the data points are spread out. If the SD-method is used as an outlier detection method it is based on comparing some kind of residual to a value. This value represents the distance to the mean by a certain number of standard deviations [60]. A common choice for α is 3 if only a few outliers are expected or for a more aggressive approach $\alpha = 2$ [56]. If a data set is normally distributed the following values can be approximately assumed [61]:

1. 68 % of the values are in the interval $[\mu - \sigma; \mu + \sigma]$,
2. 95 % of the values lie the interval $[\mu - 2\sigma; \mu + 2\sigma]$,
3. 99.7 % are in the the interval $[\mu - 3\sigma; \mu + 3\sigma]$.

This is also known as the 68-95-99.7 rule. With this technique, a lower threshold T_{min} and an upper threshold T_{max} can be calculated as:

$$T_{min} = \mu - \alpha\sigma, \tag{7.1}$$

$$T_{max} = \mu + \alpha\sigma. \tag{7.2}$$

The data points between this lower bound (Equation 7.1) and upper bound (Equation 7.2) are considered to be non-anomalous, all others are marked as outliers. When using SD the underlying distribution is assumed to be normal. The mean and SD may be strongly affected by outliers [56]. This method can lead to wrong results because the standard deviation is afflicted by outliers and its value gets increased by them [60].

7.1.2 Median Absolute Deviation (MAD)

This technique was developed because the median [1] is more robust towards outliers then the mean. The thresholds can be calculated as follows:

[1] The median of a set X is denoted as $\tilde{X}$ in this work.

$$T_{min} = \tilde{X} - \alpha MAD, \tag{7.3}$$

$$T_{max} = \tilde{X} + \alpha MAD. \tag{7.4}$$

The median absolute deviation of a dataset X, $X = \{x_1 \dots x_n\}$, is calculated as:

$$MAD = b\tilde{S}, \tag{7.5}$$

whereas for each x_i of X the entry s_i of S is calculated an as:

$$s_i = |x_i - \tilde{X}|. \tag{7.6}$$

The parameter b depends on the underlying distribution of the data. For a normally distributed dataset the scale $b = 1.4826$ is suggested, which is not taking into account that outliers do not always follow a normal distribution [62]. If the data follows another distribution, b changes and can be calculated as [63]:

$$b = \frac{1}{Q_{0.75}}, \tag{7.7}$$

whereby $Q_{0.75}$ indicates the 0.75-quantile of the underlying distribution [63].

Similar to the standard deviation method presented in the previous section this method also requires the user to set α which subsequently impacts the rejection criterion, which is bounded by T_{min} and T_{max} [63]. In [63] the value $\alpha = 2.5$ is recommended for outlier detection as a reasonable choice, but it is also pointed out that the choice should be somehow justified [63].

In addition, this technique is affected by the presence of outliers. If more than 50% of the data consists of anomalous data, the median is located within the anomalous data [56].

7.1.3 Interquantile Range (IQR)

The difference between the first quantile Q_1, which contains 25% of values and the third quantile Q_3 that holds 75% of values is called the interquantile range. This can be used to calculate the thresholds:

$$T_{min} = Q_1 - c * IQR, \tag{7.8}$$

$$T_{max} = Q_3 + c * IQR. \tag{7.9}$$

A common value for the parameter c is 1.5 [56]. This is based on the assumption of an underlying normal distribution. Assuming this, Q_1 corresponds to a -0.675σ, that means that the interval that contains 25% of the values is bounded by the mean and the value of -0.675σ. The same consideration applies also to Q_3 and 0.675σ. The interquantile range is the population spread between Q_1 and Q_3, which leads to the interval bounded by -0.6745σ and 0.6745σ. To reach that

95% of the values lie in the interval, the value for c can be determined by substituting the values for Q_1 and Q_3 and solving for the c. It leads to a value for c that is often approximated as 1.5 [64].

7.1.4 Z-Score

To use the Z-score the data should follow a normal distribution because this technique is derived from the property of the normal distribution, indicated with N, that if $X = \{x_1, x_2, \dots, x_n\}$ follows $N\left(\mu, \sigma^2\right)$, then $\frac{X-\mu}{\sigma}$ distributed as $N(0,1)$. Following this property the Z-score z_i of each measurement x_i can be calculated as

$$z_i = \frac{x_i - \overline{x}}{s}, \tag{7.10}$$

where $\overline{x}$ is the mean and s the standard deviation of X.

A commonly used threshold for outlier detection is if $|z_i|$ exceeds 3 the measurement is marked as outlier. Both, σ and μ are quite sensitive to outliers [34, 57]. However, due to its nature the Z-score is not an appropriate technique for outlier detection in small datasets. The largest value of the z-score z_{max} of a data set with n samples is bounded by the equation:

$$z_{max} = \frac{n-1}{\sqrt{n}}. \tag{7.11}$$

The largest negative z-score z_{min} is given by $z_{min} = -z_{max}$. So, the empirically commonly chosen threshold for outlier detection is to mark observations as outliers if their z_i exceed 3, because of Equation 7.11 this can only be exceeded with a data sample containing at least 18 samples [65].

7.1.4.1 Modified Z-Score

The modified Z-score was introduced because the mean and standard deviation of a sample can be strongly affected by the presence of a few outliers and even by a single strong anomalous measurement. To reduce this effect, instead of the mean and the standard deviation, the median absolute deviation is used. The modified Z-score for x_i is calculated as [34, 57]:

$$M_i = \frac{0.6745 x_i - \overline{x}}{MAD}, \tag{7.12}$$

whereby the constant 0.6745 represents the 0.75-quantile of normally distributed data sets with a large sample size n. A commonly used empirically chosen threshold for outlier detection is to mark observations with $M_i > 3.5$ as outliers [34].

7.1.5 Two-Stage Threshold

This threshold technique was introduced in [56] to overcome the problem with too strongly deviating outliers and give a better measure in this case. Therefore in [56] it was proposed to calculate the threshold in two steps. For this method SD, MAD or IQR can be used. In the first step, called the ”cleaning step”, the threshold is calculated and all the data detected as anomalous is removed from the dataset before the threshold is calculated a second time. This procedure is more robust if the data is very noisy compared to calculating the threshold only once [56].

7.2 Outlier Detection for Non-Normal Distributed Data

If a normal distribution is assumed and the data follows another distribution, observations can be considered to be non-anomalous even if they are outliers but are drawn from another distribution [34]. One approach to apply outlier detection to highly skewed data is to transform it into a normal distribution and in the next step applying methods that work well for normal-distributed data. However, this distribution could affect the accuracy of the following outlier-detection step [57]. The skewness of a distribution measures its shape and potential asymmetry. A symmetric distribution has a skewness of zero. A distribution with a larger tail to the right than to the left has a positive skewness, whereby a distribution with a tail to the left has a negative skewness [66].

7.2.1 Boxplot

The advantage of the boxplot is that it does not assume a distinct distribution and does not use the mean or standard deviation in its statistics. It depends on the IQR of the data and defines inner fences and outer fences. For the inner fence, c takes the value 1.5 and for the outer fence, c takes the value 3. Observations that lie between the inner fence and the nearest outer fence are called to be outside. This interval is also referred to as whisker [57].

Assuming normally distributed data and calculating the percentage of data points that lie outside the inner fence by substituting $c = 1.5$ into 7.8 and $Q_1 = -0.675\sigma$ as well as $IQR = 1.35\sigma$ leads to -2.7σ. Because the boxplot is symmetric, the inner fence is represented by the interval $[-2.7\sigma; 2.7\sigma]$, which corresponds to approximately 99.65% of the data left of the median lying in the interval $[-2.7\sigma; median]$. That means on each side only 3.5% of the data lies outside the inner fence. So, in total it is expected that 7% of the data is classified as being outside [64, 67].

However, the boxplot is not depending on the distribution as it delivers the best results when applied on datasets containing a high sample size that are fairly following the normal distribution having a the degree of skewness that is not too high. Despite all the downsides it is resistant to extreme values in the data [57]. If the skewness in the data is too high for this technique too many

observations are classified as outliers. This occurs because the calculation of the cut-off values are based on principles of normal distribution [67].

7.2.1.1 Skewness-Adjusted Boxplot

This variation of the boxplot was introduced in [68] and does not assume symmetric distribution. This approach should be more suitable for skewed data. The calculation of the whiskers depends on a more robust measure of skewness, the medcouple [68].

The medcouple was introduced as a measure for skewness in [69] because the commonly used Pearson's moment coefficient of skewness is very sensitive to outliers in the data and uses the moment coefficient of skewness for a univariate dataset. The medcouple MC_n of the descending sorted dataset X with n samples and the sample median m_n is defined in [69] as:

$$MC_n = \underset{x_i \leq m_n \leq x_j}{median}\, h(x_i, x_j). \tag{7.13}$$

There are two kernel functions, one for the normal case, (see equation 7.14) where $x_i < m_n$ and $x_j > m_n$ and one for the special case where $x_i = x_j = m_n$ (see equation 7.15).

$$h(x_i, x_j) = \frac{(x_j - m_n) - (m_n - x_i)}{x_j - x_i}. \tag{7.14}$$

Following equation 7.13 the values for the MC always lie between -1 and 1. For a right-skewed distribution MC takes a positive value and for a left-skewed distribution a negative, whereas symmetric distribution has a medcouple of zero [67]. The distance between x_i or x_j to the median is measured by the kernel function 7.14. If the result of the kernel function is a positive value, then x_i lies closer to the median, if it is negative x_j. The value of the kernel function is -1 for all pairs where x_i is the median and 1 for all pairs where x_j is the median.

Comparing the original publication where the skewness-adjusted boxplot was introduced in 2004 (see [68]) with later versions of the same authors (see [67]) and the introduction of the medcouple (see [66]) one notices that the definitions vary. In [68] only the normal case where $x_i < m_n, x_j > m_n$ is discussed and the special case where $x_i = x_j = m_n$ was introduced for the adjusted boxplot in later publications (see [67]).

In [69] where the medcouple was introduced and in [66] an additional calculation rule $h_1(x_i, x_j)$ for the kernel function in the case where $x_i = x_j = m_n$ is employed.

$$h_1(x_i, x_j) = \begin{cases} 1 & i > j \\ 0 & i = j \\ -1 & i < j \end{cases} \tag{7.15}$$

Instead of using a c of 1.5 in equation 7.8 and equation 7.9 it is purposed in [67] to replace c for a lower boundary with $h_l(MC)$ and for the upper boundary $h_r(MC)$, whereby h_l and h_r are functions. To obtain the properties of the boxplot for symmetric distribution it is required that $h_l(0) = h_r(0) = 1.5$. It was also found in [67] that over all skewed distributions the best fit is to mark observations that lie outside the following interval as outliers:

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

The interval of equation 7.17 is defined for a right-skewed distribution. To get the interval of an left-skewed distribution it is only necessary to switch the boundaries [70, 67]:

$$\left[Q_1 - 1.5\, e^{-4MC}\, IQR; Q_3 + 1.5\, e^{3MC}\, IQR\right] \tag{7.17}$$

As pointed out in [70] in this method the whiskers are adjusted in that way so that in comparison with the normal boxplot fewer data points fall outside the interval.

Chapter 8
Approach for HPO Using GA

8.1 Used Time Series Data

The developed model was applied to time-series data recorded during geo-drills at different construction sites. The process is more in depth described in [2]. The raw data consists of different sensor signals gained during the process. For this application four of the signals were chosen and analysed: the depth of the drilling, the vibrator frequency, the vibrator temperature and the weight. The execution of the drilling process is dependant on external factors at the site, so the patterns in the data change from site to site. As visible in Figure 8.1 and Figure 8.2 for example the depth of the drills varies from site to site. Because of these changes, an optimization was done for each site separately.

8.1.1 Preprocessing

The processing time for a drilling process is not the same for each point, so also the number of recorded measurements varies. For the LSTM layers of the ML-model, a fixed number of timesteps is needed. To fulfil this condition, the signals are compressed in one of the preprocessing steps to this length called the time series length. This resizing is shown in Figure 8.3 for the signals of Figure 8.2.

The second step of the preprocessing is rescaling. This is done because the input signals are on different scales and the reconstruction error is calculated as the sum of all differences between the input signals and reconstructed signals, the range of the signals is defined to be between zero and one.

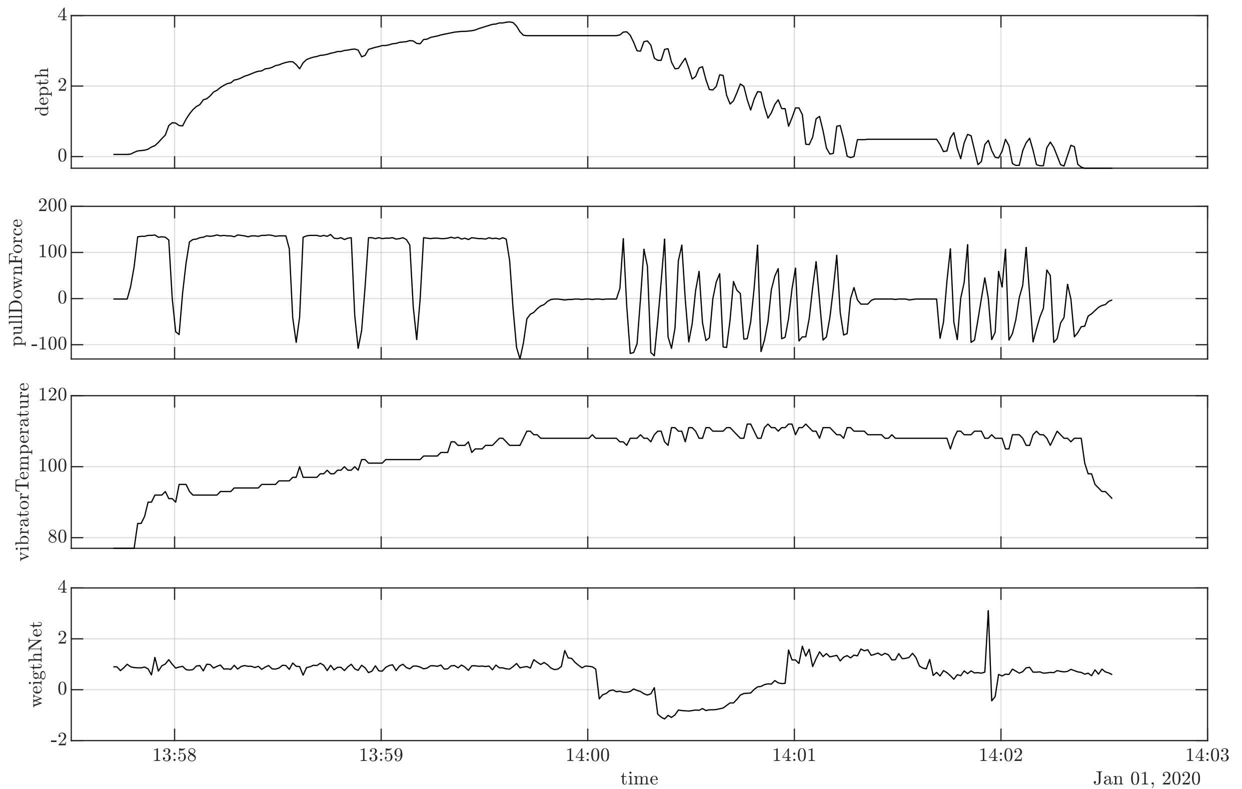


Fig. 8.1: exemplary plot of recorded signals over time at the site Seestadt Aspern

8.2 Optimized ML-Model

The hyperparameter optimization was applied to an LSTM-VAE (see Section 4.2). This ML model was developed in a prior master thesis (see [2]) and slightly modified and extended for hyperparameter optimization. The model mainly consists of two parts: the encoder and the decoder. The code for these two parts is shown in listing 8.1. The variable *numNeuornsEncoder* contains the number of neurons used in the LSTM layer of the encoder and the variable *numNeuronsDecoder* of the LSTM layer of the decoder. The *inputDim* represents the number of signals of the input data on which the model is applied.

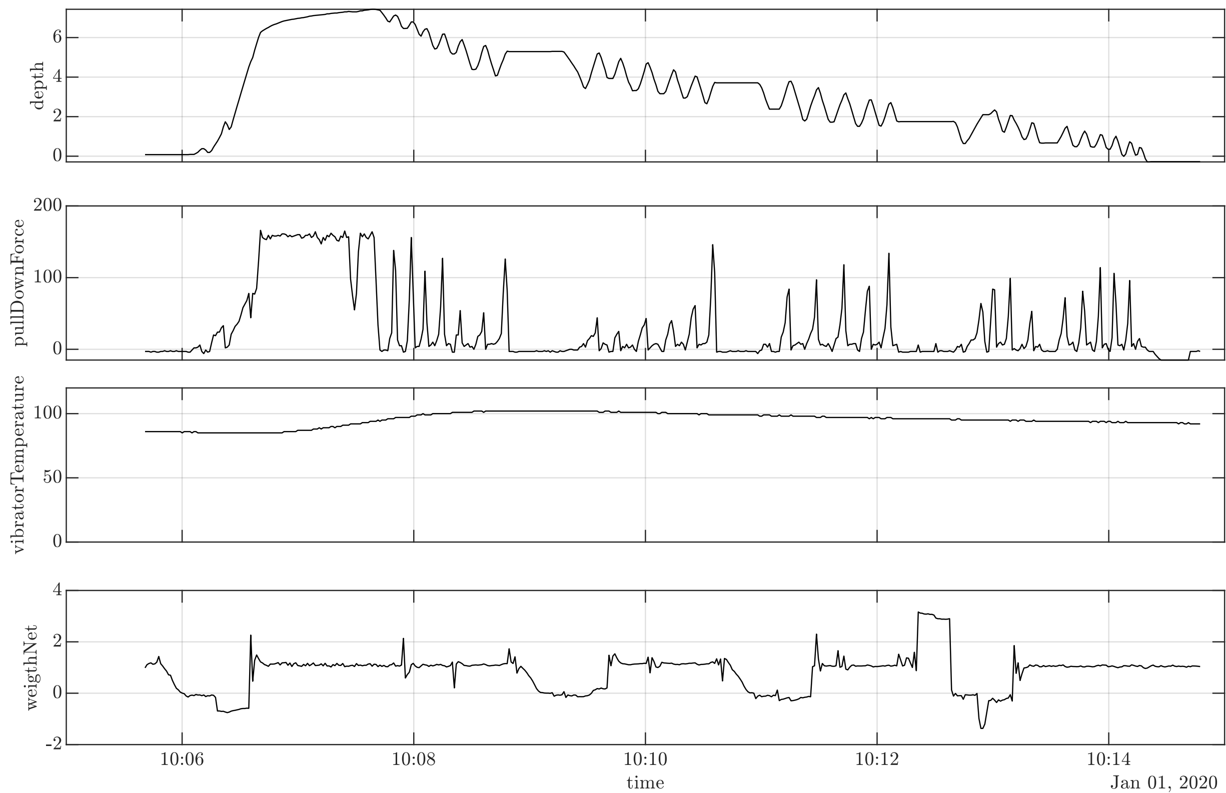


Fig. 8.2: exemplary plot of recorded signals over time at the site Fehring

```
1     % dimension of the latent space
2     latentDim = 2;
3     % dimension of the input features
4     inputDim=size(XTest,1);
5     % layers of the encoder
6     encoderLG = layerGraph([
7        sequenceInputLayer(inputDim, 'Name', 'inputEnc')
8        lstmLayer(numNeuronsEncoder,'Name', 'lstmE1')
9        fullyConnectedLayer(2*latentDim, 'Name', 'fcE')
10    ]);
11    % layers of the decoder
12    decoderLG = layerGraph([
13       sequenceInputLayer(latentDim, 'Name', 'inputDec')
14       lstmLayer(numNeuronsDecoder,'Name', 'lstmD1')
15       fullyConnectedLayer(inputDim, 'Name', 'fcD')
16    ]);
```

Listing 8.1: architecture of the LSTM-VAE

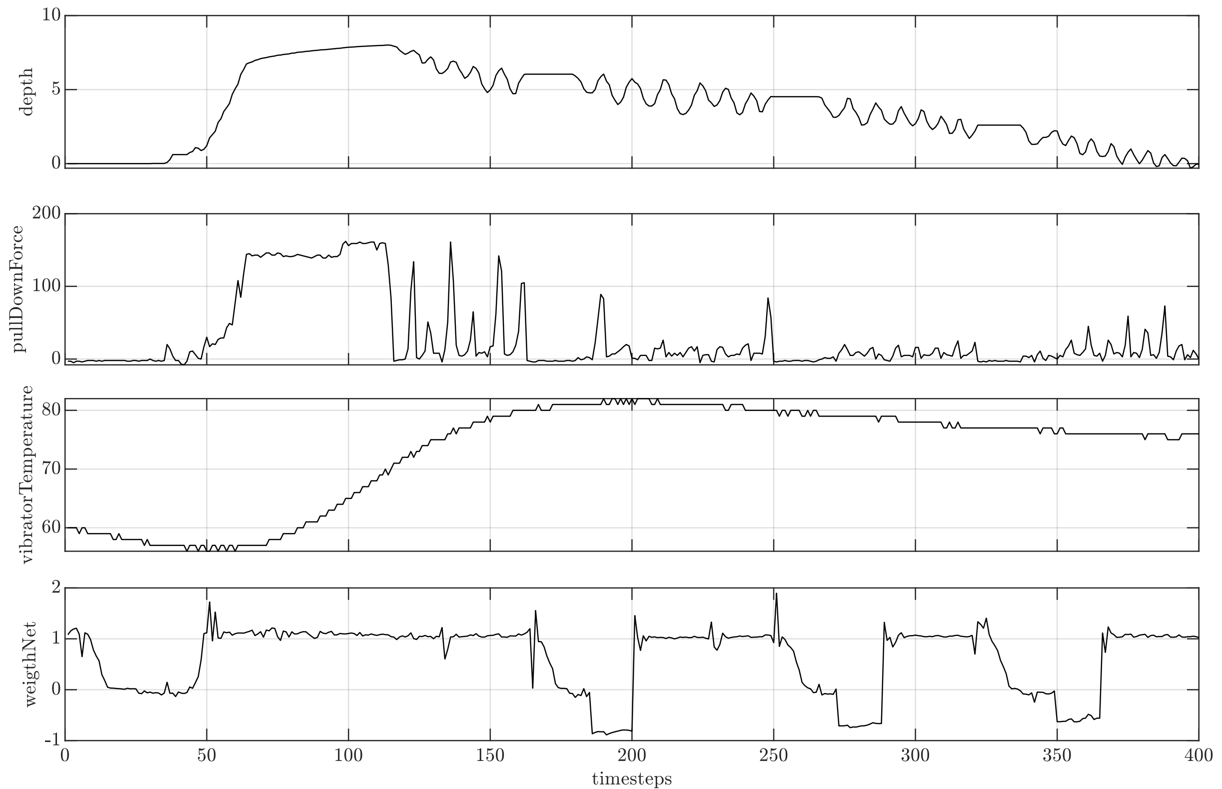


Fig. 8.3: resized exemplary plot of the signals shown in Figure 8.2 to 400 timesteps

The output of the model is the reconstruction error on a provided test set. The dataset is highly imbalanced in the number of positive examples and anomalous examples, the learning is done on a subset of positive (non-anomalous) manually selected examples.

The data is unlabelled and autoencoders belong to the group of unsupervised learning techniques, so does the LSTM-VAE, it was chosen not to label the entire data because certain anomalies are very hard to identify manually only by manual inspection. Further, certain anomalies are unknown before they occur the first time.

8.3 Applied Genetic Algorithm

For the special task of hyperparameter optimization of an LSTM-VAE a quite selective approach of the genetic algorithm was chosen. To evaluate the fitness of each individual a ML-model has to be trained with different parameters. The long computational time of the machine learning model leads to a limited budget of possible evaluations. It can be observed that also the chosen hyperparameters influence the training time. Whereby the models are trained using the GPU and all other manipulations were done using the CPU. Two of these folds were used for training and one for testing. So each hyperparameter configuration was trained three times for the HPO. That leads to

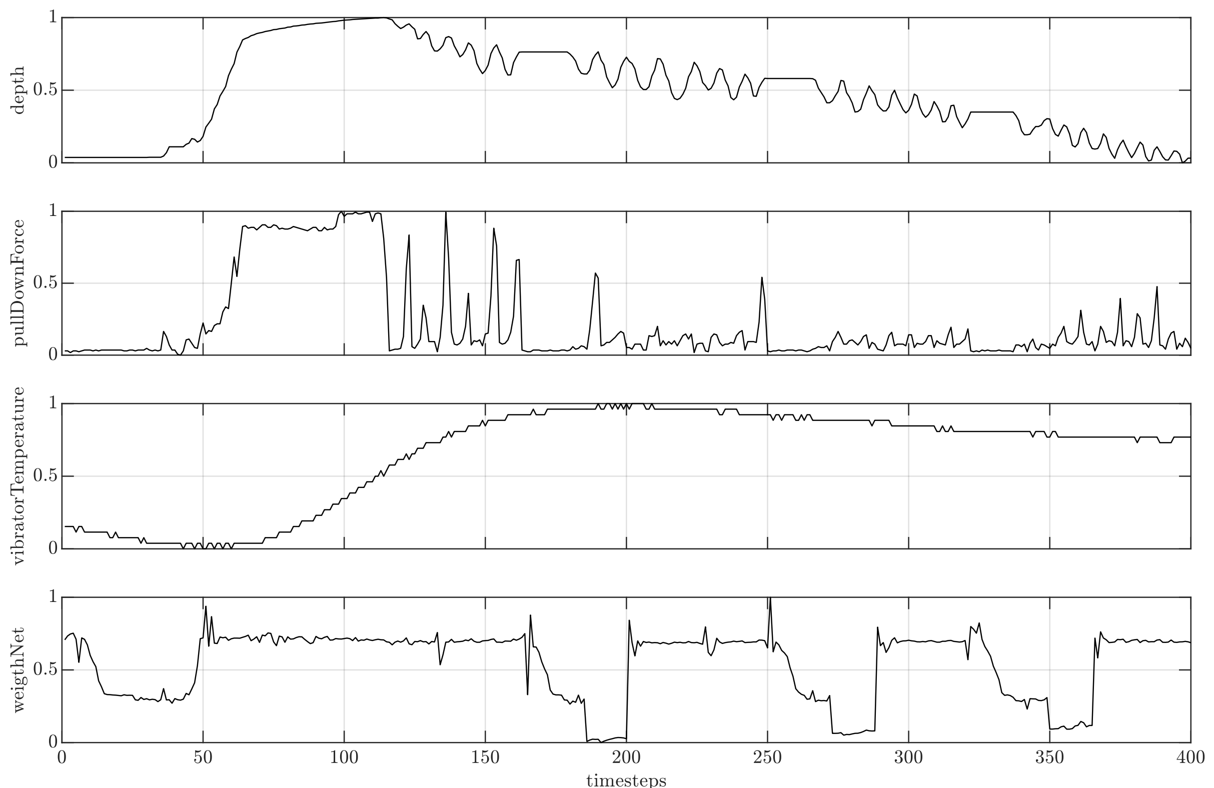


Fig. 8.4: the signals shown in Figure 8.3 where taken and rescaled between zero and one.

the training of up to 60 machine learning models in one generation consisting of 20 individuals. The approach with training three folds and three repetitions was chosen because of the effect of the random seed on the model. The weights of the machine learning model are initialized randomly in the beginning and that leads to a discrepancy in the performance of the model as shown in Figure 8.5. The fitness deviates even if the same data and the same hyperparameters are used. To illustrate this 200 models were trained with the same data and the same hyperparameter settings and the archived fitness values were plotted in Figure 8.5.

This effect is even stronger if random folds of the same data are generated and then the model is trained and evaluated on them. This is visualized in Figure 8.6. For a better generalization, each generation was trained with a different permutation of the data. That lead to different subsets used as folds.

8.3.1 Termination Condition

As the termination condition a combination of three arguments was chosen. The first one is the number of maximal generations and the second one tests if the average fitness of the generation is better than one of the average fitness values of the last four generations. The third argument is

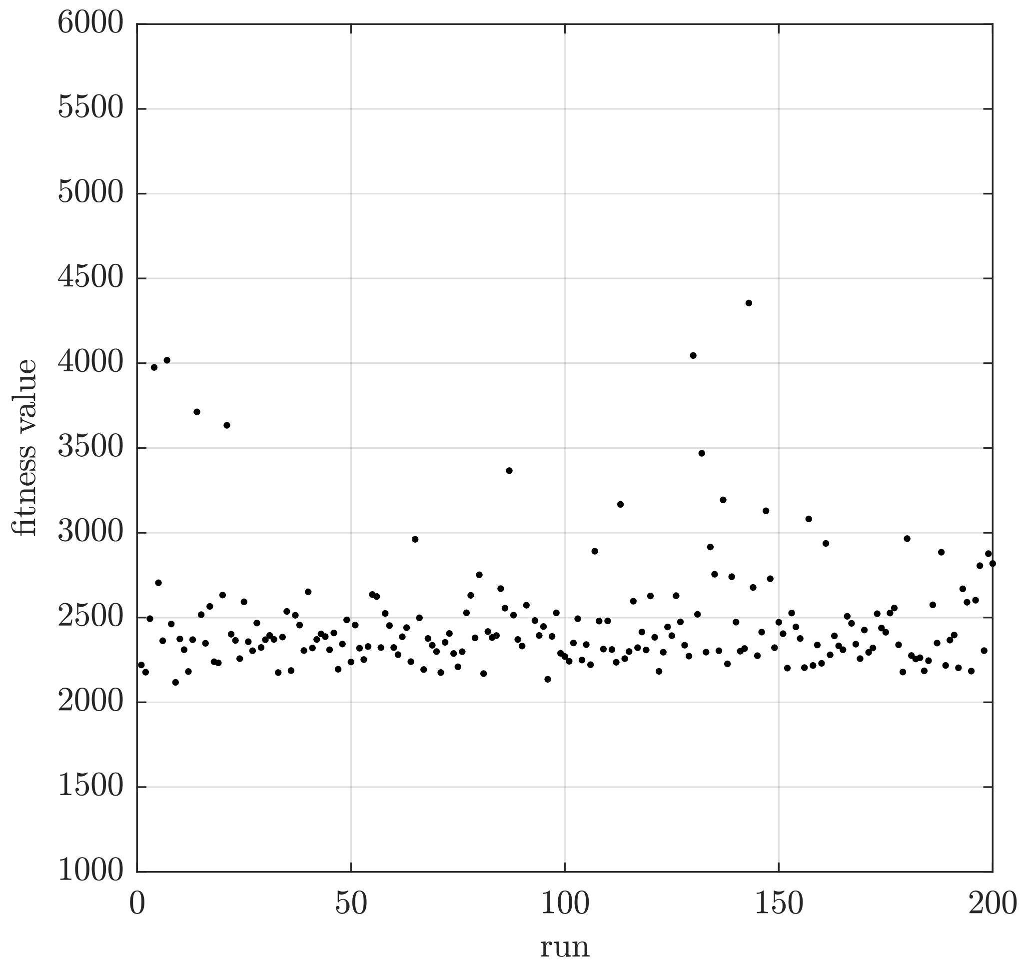


Fig. 8.5: deviation in fitness of models trained with the same data and same hyperparameters

that the algorithm terminates if the generation only consists of individuals that all hold the same hyperparameters.

8.3.2 Fitness Function

The fitness function is formed by the ML-model the HPO is done for. It is realized as a function where different encoded individuals can be passed as input parameters alongside the folds of the train data and test data. As a fitness measure the reconstruction error on the test set is used. The input data and also the reconstruction consists of four signals, the summed up absolute values of the difference between the preprocessed input data and the reconstructed data of the test set is used as fitness value.

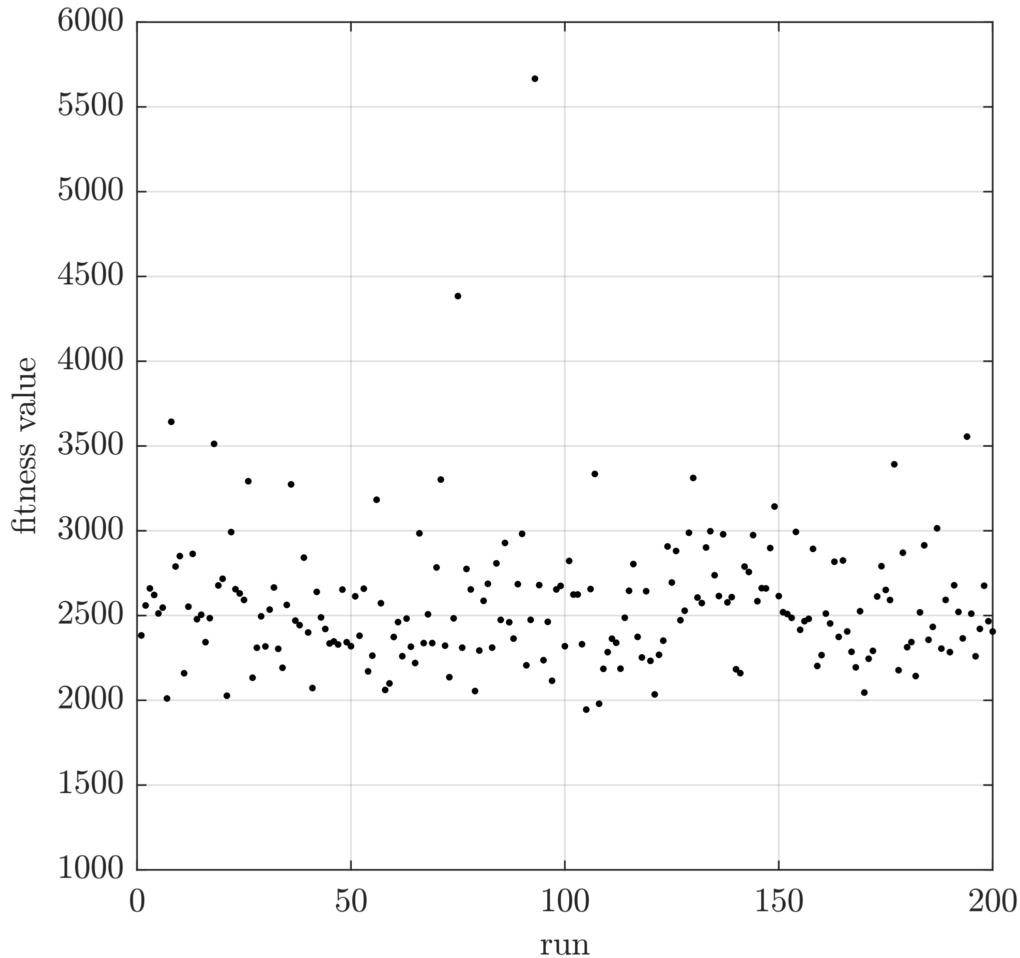


Fig. 8.6: deviation in fitness of models trained with different folds of the same dataset and same hyperparameters

8.3.3 Chromosomes

The hyperparameters are encoded as integers and a range of allowed values for each parameter was defined. For the optimization the following hyperparameters (see Table 8.1) are used alongside the defined domains of the values.

variable name	normal searchspace	enlarged searchspace
trainedEpochs	[1;100]	[1;200]
numberOfNeuronsEncoder	[1;100]	[1;200]
numberOfNeuronsDecoder	[1;100]	[1;100]
learningRate	[0.005,0.01] = [5,1000] 10^{-5}	[0.001, 0.1]=[1; 10000] 10^{-5}
miniBatchSize	[2;53]	[2;53]

Table 8.1: domains of the HPO for the normal and extended searchspace

As shown in Table 8.1 the HPO was firstly done on a smaller search space and then the domain was extended for the variables $trainedEpochs$, $numberOfNeuronsEncoder$ and $learningRate$.

As discussed in Section 6.1.1 the learning rate was found to be the most important hyperparameter and therefore it was tried out to optimize it on a larger domain. Because the learning rate is a decimal between zero and one, it is shifted to the integer domain and then multiplied by 10^{-5} before beeing applied to the ML-model. The upper bound of the mini-batch size is set to 53 because for the run 80 training examples were used, divided into three folds and two out of them used for training.

The genetic algorithm was also programmed for a bit-string-representation. But because of the interval the values should lie in and the number of bits they need to be encoded, it is necessary that after every genetic manipulation is checked if the produced offspring is a valid number. This extends the runtime of the genetic operators and because except one, all hyperparameters are integers, it was chosen to use the integer-representation instead of the bit-string-representation.

8.3.4 Genetic Operators

A similar approach for designing the genetic operators was presented in [71] also using integer encodings. The genetic operators are designed in a way that the integer property of the chromosomes is ensured. The first step is to create the initial population. This is done by creating individuals with randomly initialized genes. Then as long as the termination criterion is not fulfilled the following genetic operators are applied to the population:

8.3.4.1 Evaluation

The first step of the genetic algorithm is that the individuals are evaluated to determine their fitness. For each individual, each hyperparameter setting, three models are trained with the same parameters on different folds of the data. As fitness of the model, the mean of the three runs is taken. If individuals are holding the same hyperparameters in the population, each configuration was evaluated only once per population, because of the long runtime that is caused by training three ML-models per individual. The fitness evaluation is with no doubt the dominant term of the runtime-function.

8.3.4.2 Selection

For the selection, an approach with quite high selection pressure is used. Elitism is employed by transferring the $k = 3$ best individuals of the population to the next population without applying crossover to them.

All individuals that have a fitness better, smaller, as the median of all fitness values of the population form the mating pool. Additionally, the other individuals are added with a small pre-defined probability to the mating pool to maintain genetic diversity.

8.3.4.3 Crossover

The next genetic operator applied to the population is crossover. This process is repeated $p-k$ times if p is the number of individuals in each generation because each iteration produces one new individual. The crossover-step consists of two possible crossover operators that are chosen randomly. Firstly two individuals are selected randomly out of the mating pool. If the first variation of crossover is used as shown in Listing 8.2 then each chromosome is randomly selected if the child inherits the value of the first or the second parent.

```
1  % number of variables
2  numVar = length(parentA);
3  for i=1:numVar
4     % random number [0,1] if <= 0.5 component from first parent
5     % otherwise from second
6     randNr=rand();
7     if(randNr <= 0.5)
8     child(i)=parentA(i);
9     else
10    child(i)=parentB(i);
11    end
12 end
```

Listing 8.2: random crossover

If the second variation of crossover is used (see Listing 8.3), for each allele the mean of the alleles of the two parents is taken as the value of the offspring and rounded to the next integer to obtain the integer property of the entries.

```
1    numVar=length(parentA);
2
3    child= (parentA+parentB)./2;
4    child=round(child);
```

Listing 8.3: mean crossover

This approach with the combination of two crossover functions was chosen because of the relatively low number of individuals in comparison of possible combinations in the search space that more new values for each parameter are produced and the search space is explored more.

The first trials were done using only random crossover and that lead to the effect that the values each parameter can take were the same as in the random construction from the first generation, except if a mutation occurs, which is applied with only a very small probability that the search does not turn into a random search.

8.3.4.4 Mutation

The mutation is applied with a relatively small pre-set probability randomly on the newly created population. If a chromosome is mutated, the allele is set to a random value of the parameter's domain. The mutated population is the starting point for the evaluation at the beginning of the next generation.

8.3.5 Algorithm Applied to Real Data

After fulfilling the termination condition the winning individual of the optimization is the individual that has to hold the hyperparameter setting that performed best in the last evaluation.

All the following plots in this section were obtained by an optimization done on the dataset of the site Fehring and the parameters listed in Section 8.3.3, a population consisting of 20 individuals and as many generations until the termination condition (see Section 8.3.1) was fulfilled.

8.3.5.1 Convergence

The average fitness of a genetic algorithm at the beginning decreases exponentially, this is shown for one exemplary run in Figure 8.7.

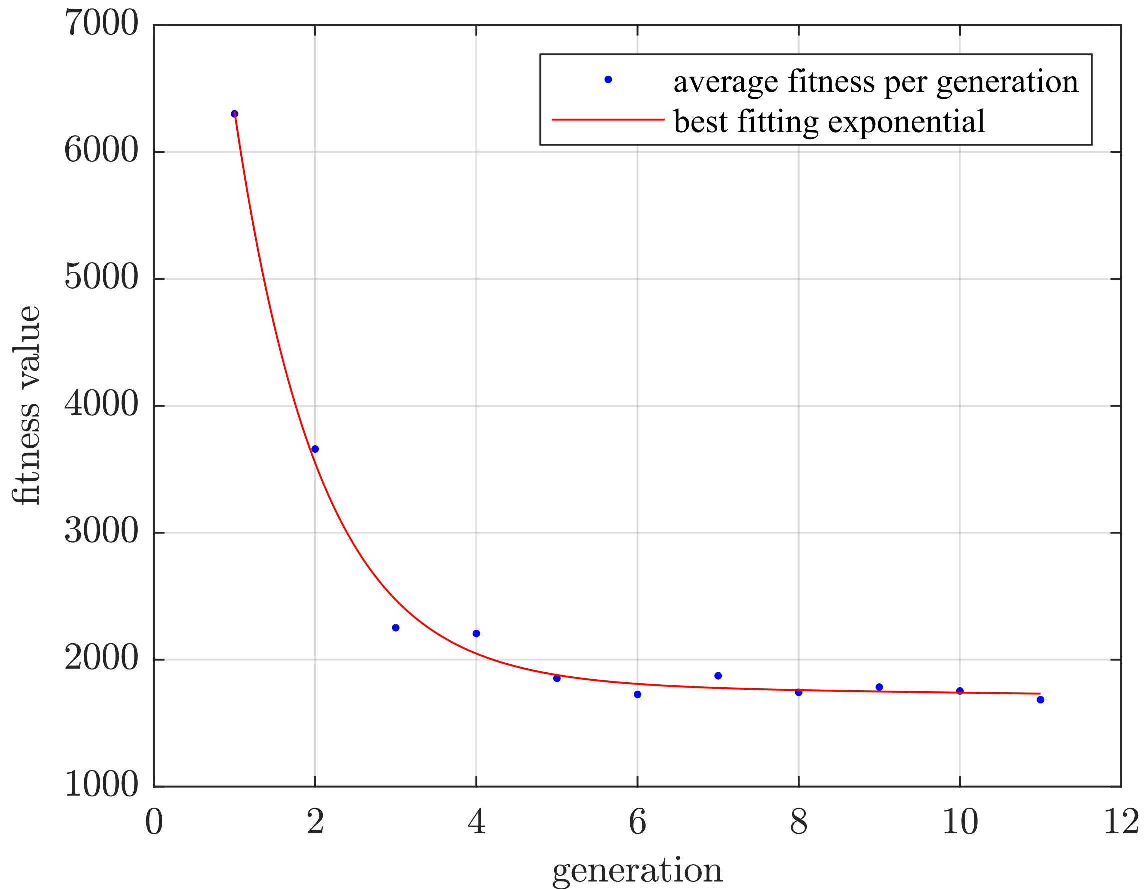


Fig. 8.7: average fitness of the population over the generations

Because the genetic algorithm performs a directed search, the number of values the individuals of a population hold is decreasing from population to population. For one optimization run it is shown in Figure 8.8

If the HPO is executed two times, compare Figure 8.8 and Figure 8.9, on the same data with the same settings it does not lead to the same results, although the values are in a similar range they vary. The resulting values for the hyperparameters of the first run were [49,82,60,878,3] and the second run of the HPO returned [76, 49, 56, 978, 7]. The only real dissimilarity between these two results is, that in the first result the encoder has more neurons than the decoder and in the second result, it is the other way around. In the master thesis of [2] the author assumed that the encoder should have more neurons as the decoder. He based his assumptions on observations of the shape and density of distributions in the latent space. In most cases (see Table 8.2), this assumption can be also done with the experiments done here, but there are also cases as that the second exemplary HPO (see Figure 8.9), where it does not hold.

The dissimilarities of two HPO runs could be explained on one hand with the random initialization of the first population, the uncertainties regarding the random seed and learning in general,

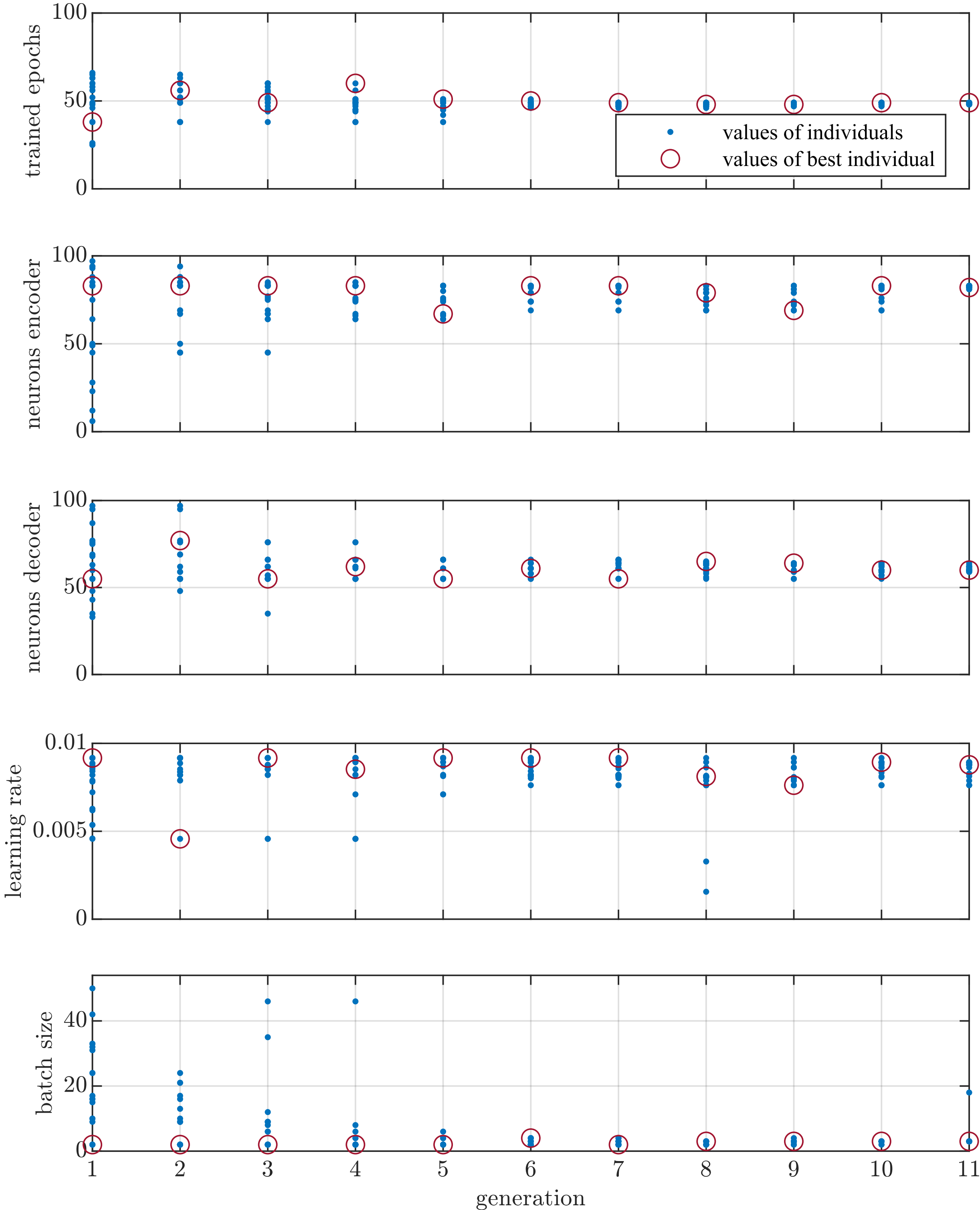


Fig. 8.8: values of the HPO the individuals take at each generation

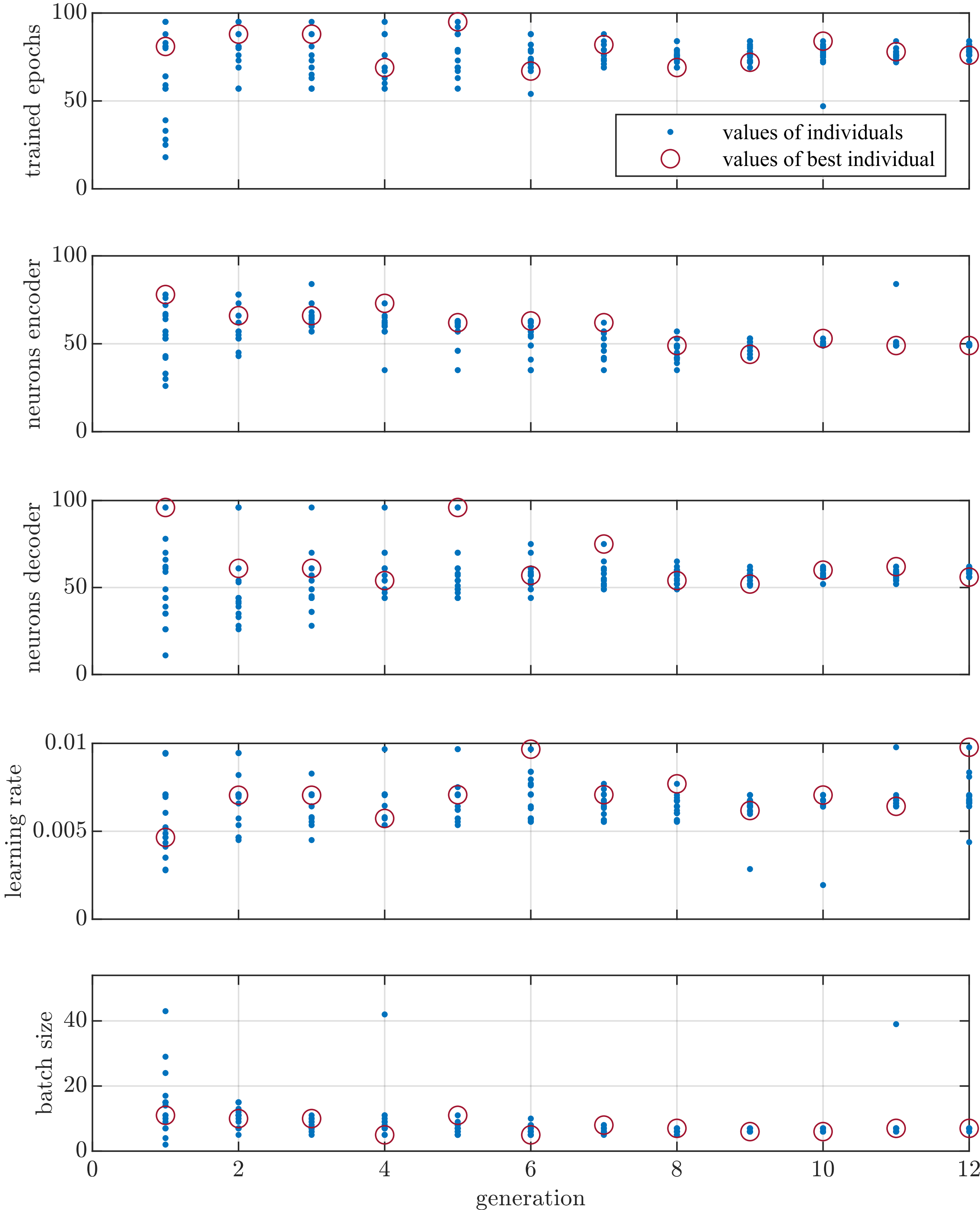


Fig. 8.9: values of the HPO executed a second time the individuals take at each generation

as well as the random portion of the genetic algorithm with the random crossover and the random probability that an individual with a too high fitness is added to the mating pool and potentially chosen for crossover. Another possible explanation is that some hyperparameters, for example the learning rate, as discussed in Chapter 6 can be imagined as a bathtub function and there are larger regions of values with good performance.

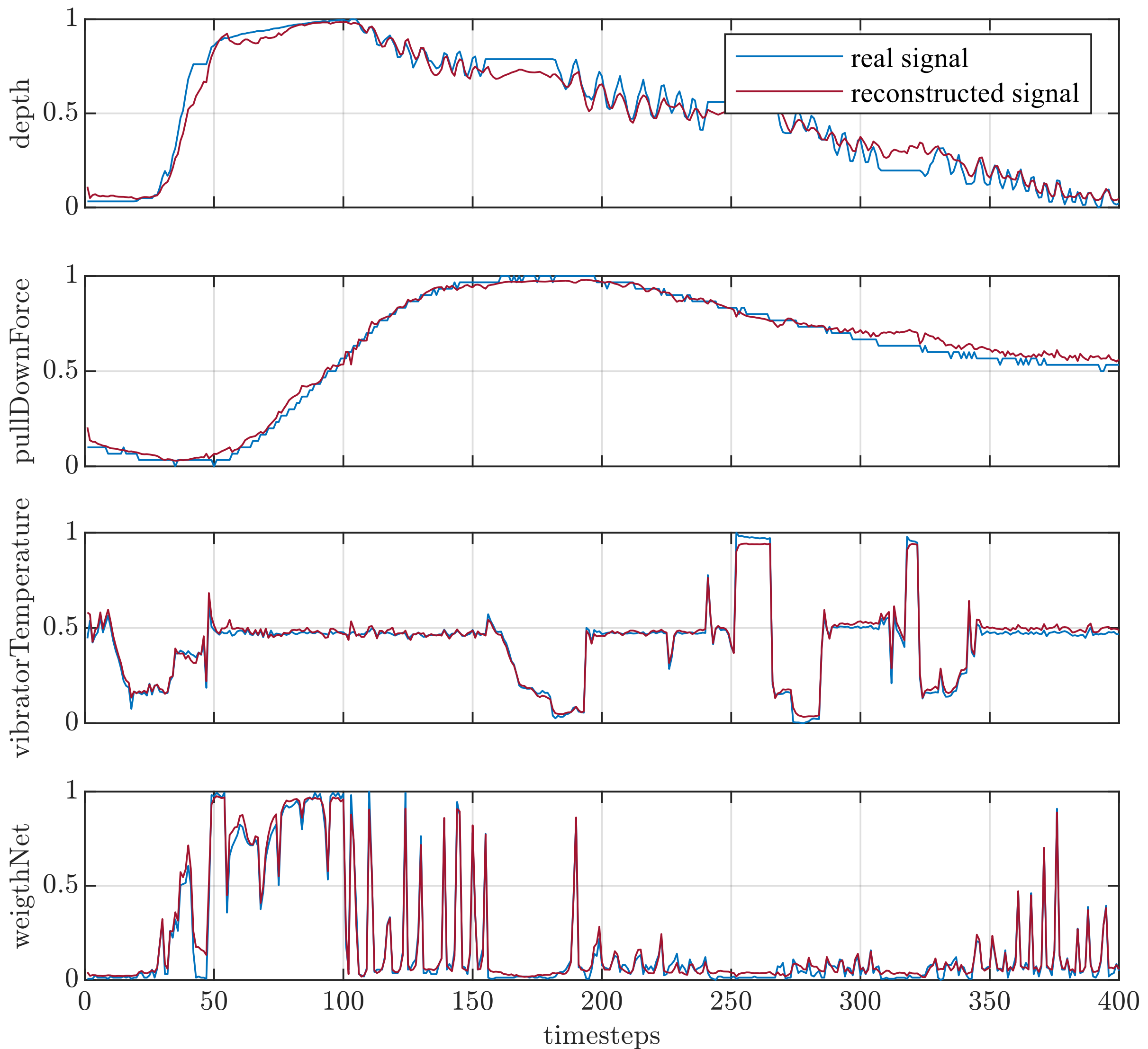


Fig. 8.10: reconstructed and real signals for a non-anomalous point of the site Fehring

After training an encoder-decoder structure found by the HPO, all data, all drilling points of a site, are encoded and decoded to obtain the reconstruction error of each sample. For a non-anomalous sample it is considered to be relatively low and for an anomalous sample, an outlier, accordingly higher. As example the results of an non-anomalous sample (see Figure 8.10) and an anomalous (see Figure 8.11) sample where visualized.

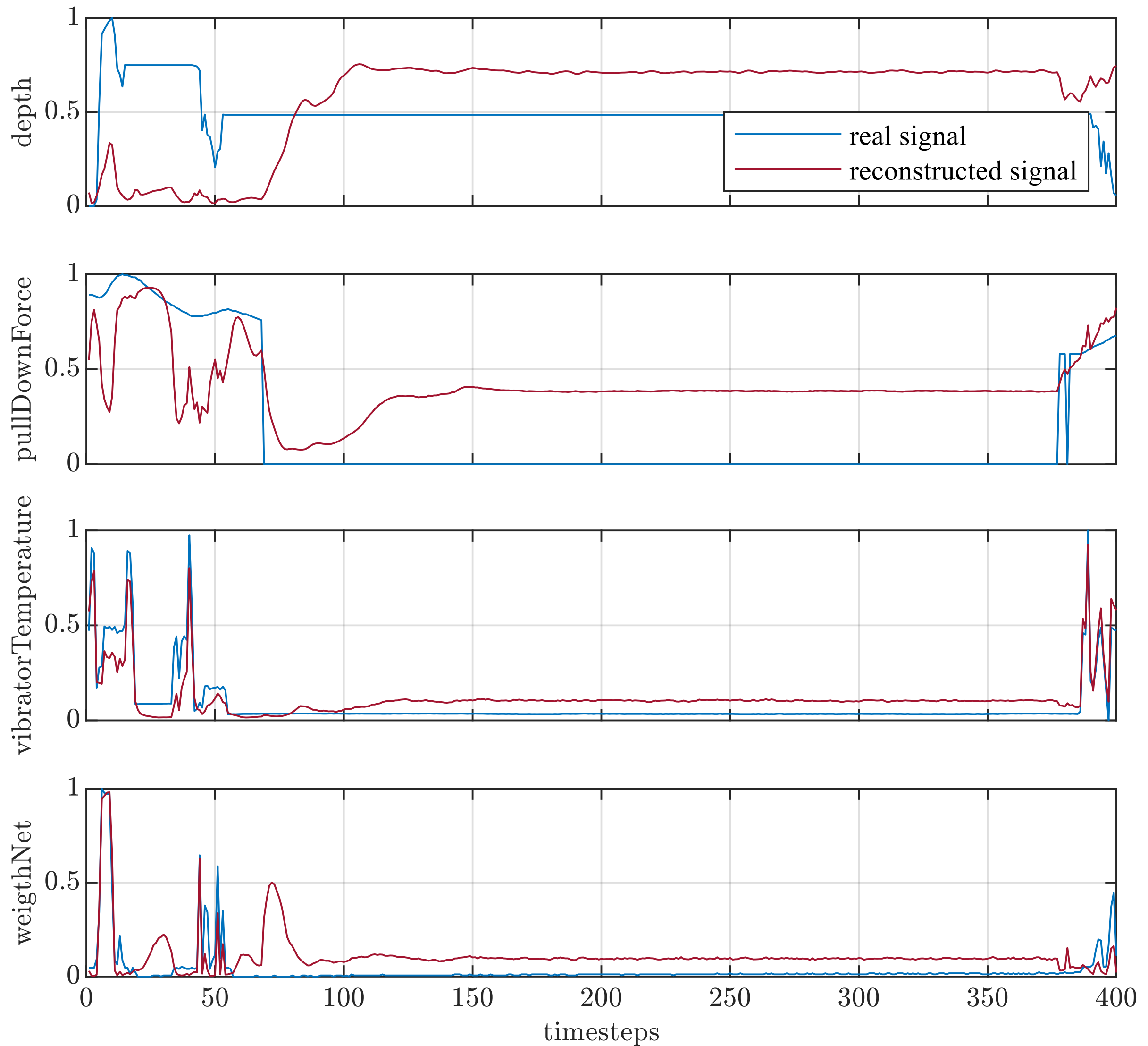


Fig. 8.11: reconstructed and real signals for a anomalous point of the site Fehring

As shown in Figure 8.12 also training the actual ML-model, which is later on used to calculate the reconstruction error, inherits all the uncertainties regarding ML-models. Training it multiple times with the same hyperparameters found by the optimization will lead to slightly different results - also in the classification results of what is an outlier and what not.

For comparison in Figure 8.14 the reconstruction error of a model trained with the optimized hyperparameters and two individuals of the first generation is shown. By comparing it with Figure 8.12 it can be seen that the fewer points are outlying and the overall reconstruction error is higher than when data is applied to the optimized model.

If the reconstruction error of the optimized parameters shown in Figure 8.8 is visualized in a histogram (see Figure 8.13), one can see that the reconstruction error does not follow a normal dis-

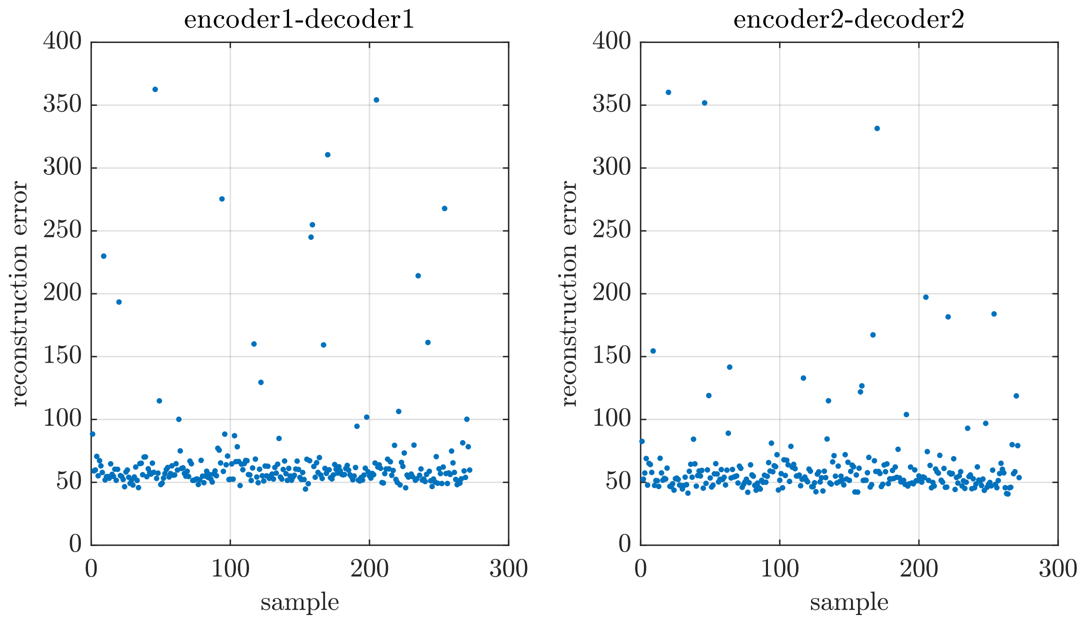


Fig. 8.12: two encoder-decoder structures where trained with the same hyperparameters of the HPO visualized in Figure 8.8 and on the same data and afterwards applied on the whole dataset of the site Fehring to get the reconstruction error for each point

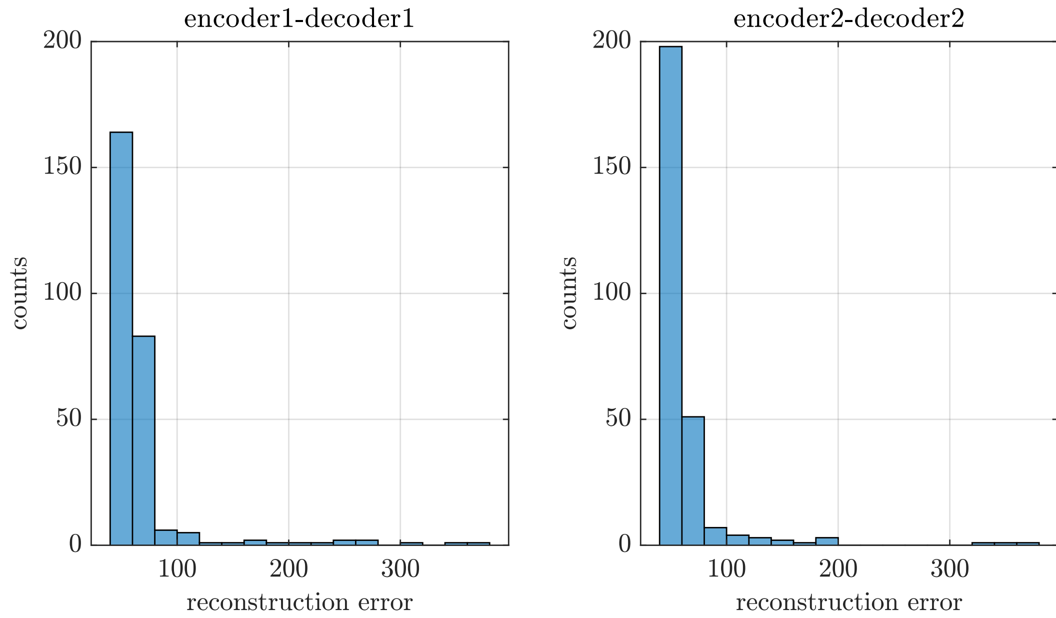


Fig. 8.13: histogram of the data gained at performing the reconstruction shown in Figure 8.12

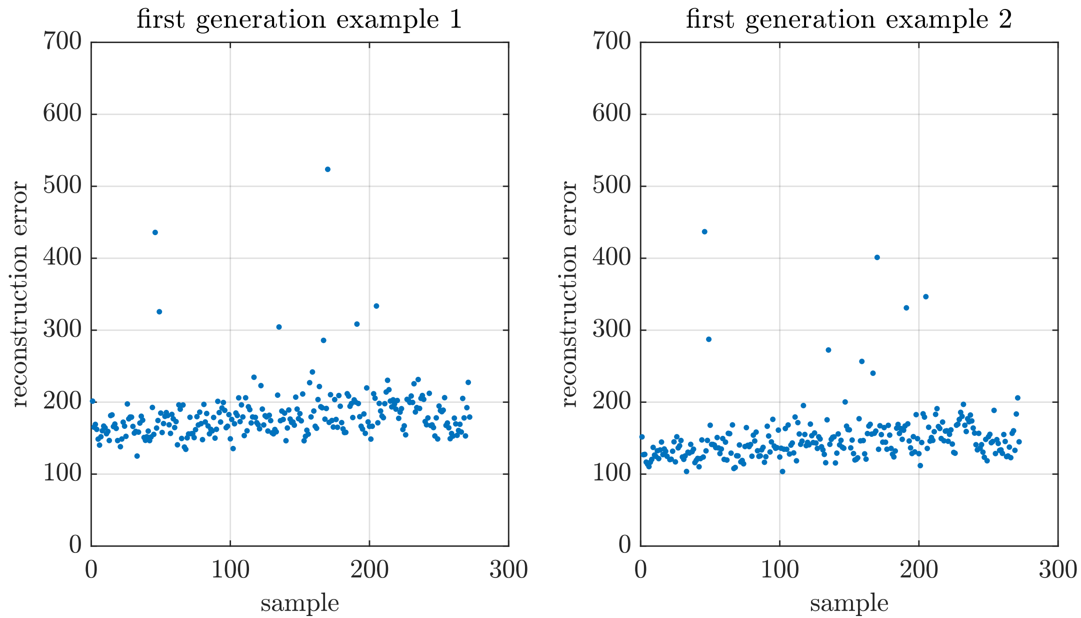


Fig. 8.14: reconstruction error of a ML model trained on the hyperparameters held by two individuals of the first generation when applied on the data of the site Fehring.

tribution. Because of this most statistical methods for outlier detection cannot be applied. Therefore for the outlier detection the skewness-adjusted boxplot (see Section 7.2.1.1) is used. This method is applied only single-sided, only a too high reconstruction error is flagged as an outlier. If the reconstruction error is too low, it means that the reconstruction fits the real recorded signals well. That should not be marked as an outlier.

Figure 8.15 shows the classification done by using the skewness-adjusted boxplot. Additionally, ten samples that are considered to be non-anomalous and ten samples assumed to be outliers were picked manually and marked in the plot.

The outlier detection was applied on the reconstruction errors of the two trained ML-models shown in Figure 8.12 which were trained using the HPO visualized in Figure 8.8. The results of the first model is shown in Figure 8.15 and of the second model in Figure 8.16.

When comparing them one can see that in the first figure one of the examples manually labelled as an outlier is classified as normal, whereby in the other figure all examples are classified as they were classified manually.

The same procedure was applied on two models with hyperparameters randomly initialized in the first generation of the GA. The results of this outlier detections are shown in Figure 8.17 and Figure 8.18. When comparing the outlier detection done on the models with non-optimized hyperparameters (Figure 8.17 and Figure 8.18) and the outlier detection done on models with optimized outlier detection, one can see that in the optimized models more of the manually labelled outliers are identified. Also, the boundaries between the two classes are clearer and the reconstruction error of the normal data is less scattered.

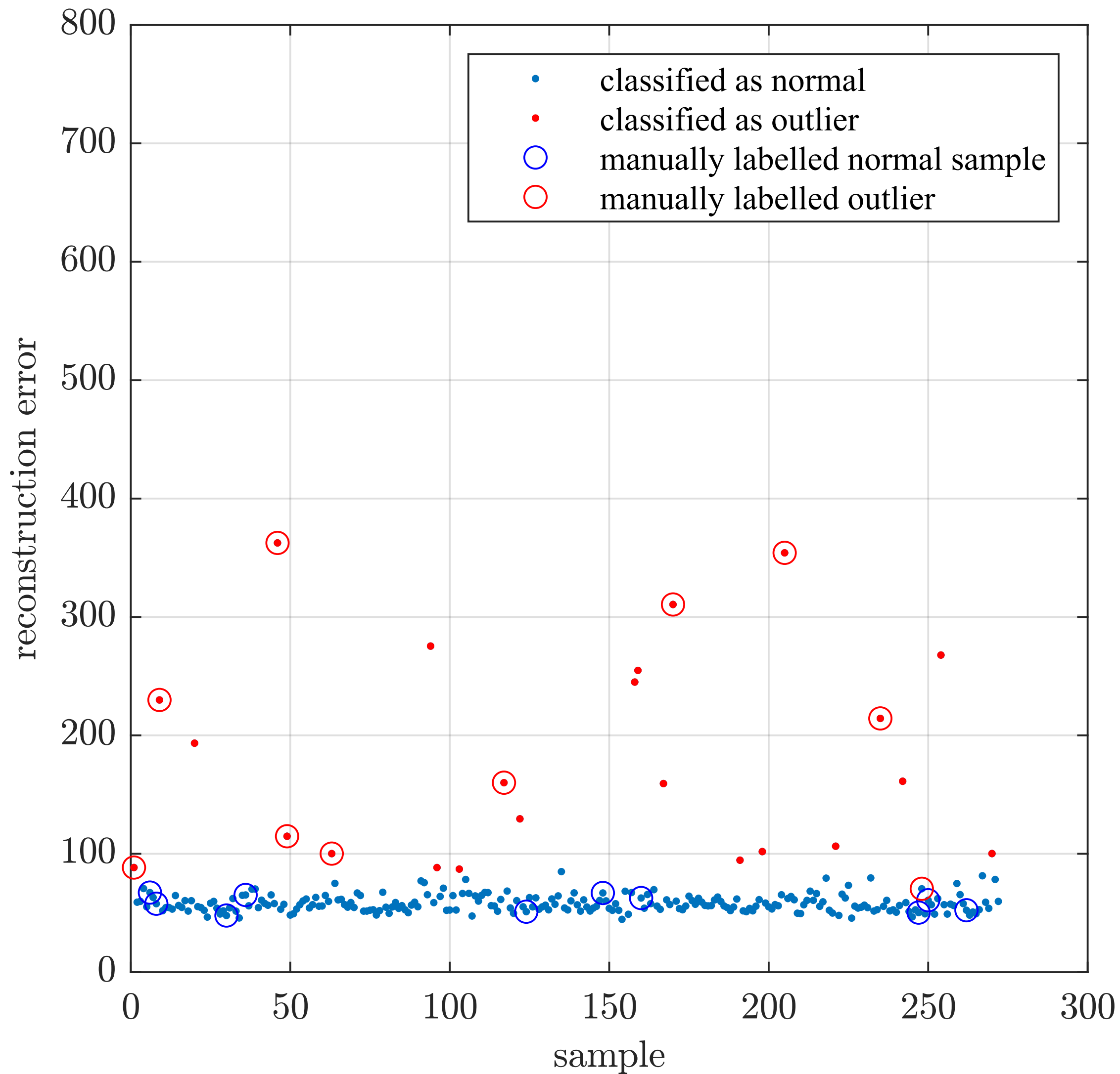


Fig. 8.15: first example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples and trained models with optimized hyperparameters

8.3.5.2 Enlarged Searchspace

Some trials were also done on a larger domain of the values as listed in Table 8.1. This was done because in [2] it was assumed that a higher number of neurons in the encoder leads to even better generalization and to better results.

Also on a trained model of this type an outlier detection with the manually selected samples was done and leads to the results shown in Figure 8.20. When comparing to the outlier detection done on the normal domain, it can be seen that the overall reconstruction error is lower. However, also the reconstruction error of the outliers is lower and the classes are less well distinguishable. This could be explained by the bigger search space which was not explored as well with the same

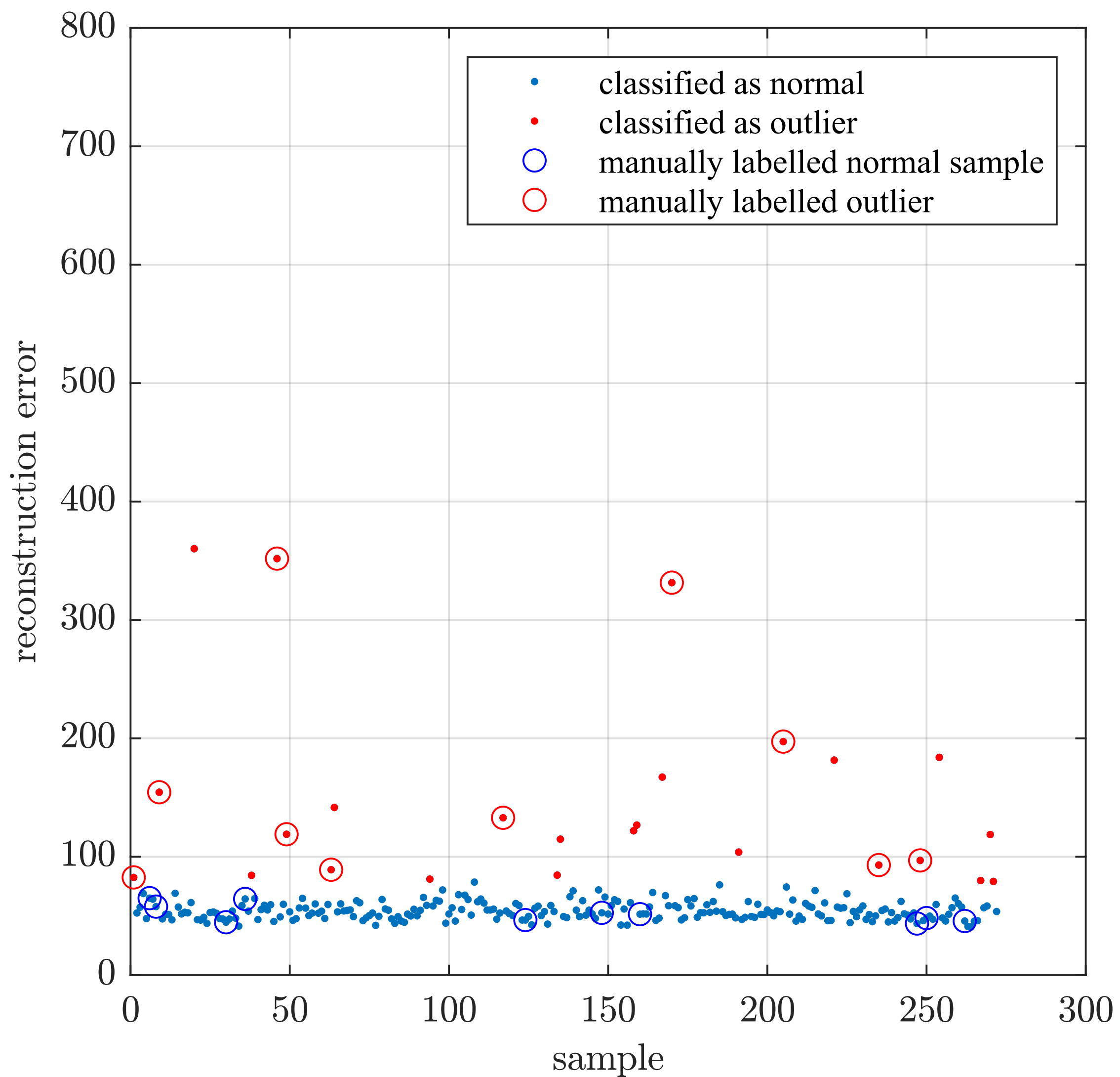


Fig. 8.16: second example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples

number of solution candidates. This can be seen also in the visualization of the exploration of the search space in Figure 8.19 and the gaps in the first generation, where no random solution was sampled on. So, for this variation, bigger population size should be considered.

8.3.5.3 Extended Optimization

The model was then extended to a optimization containing four additional parameters to the five used in the other variations (see table 8.1) which are:

6. additional LSTM layer encoder [1; 10]

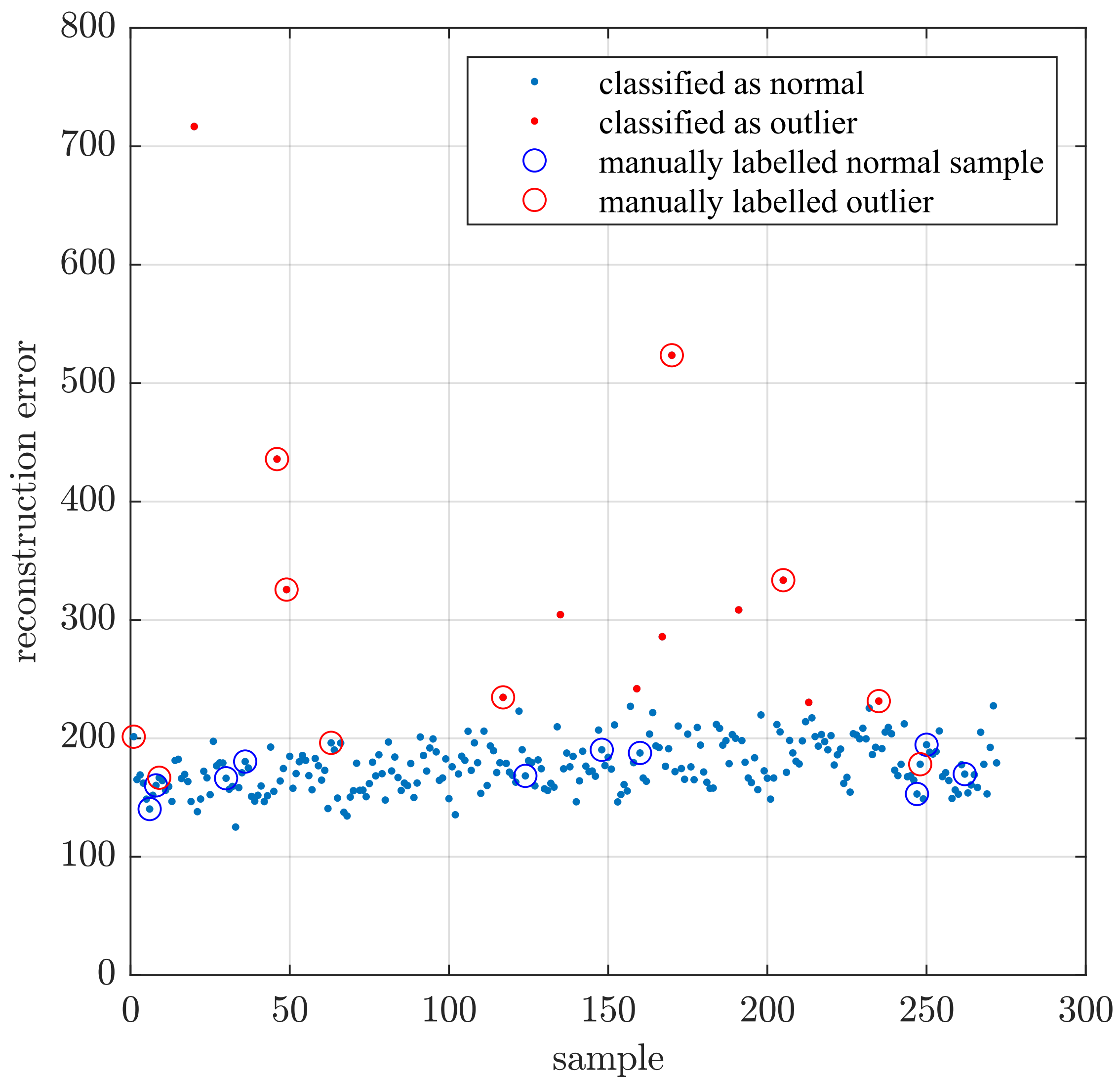


Fig. 8.17: first example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples with models trained with random hyperparameters

7. number neurons additional layer encoder [1; 100]
8. additional LSTM layer decoder [1; 10]
9. number neurons additional layer decoder [1; 100]

If the variable *additionalLSTMlayerEncoder* takes a value greater than five, an additional LSTM layer is used in the encoder and the same principle is applied for the decoder. Then the next parameter defines the number of neurons for this additional layer. The same applies to the next two parameters regarding the decoder. The domain of the additional LSTM layer was chosen in this way, that the crossover presented in Section 8.3.4.3 works. If the domain [0;1] would be used, where 1 would indicate that the additional LSTM layer is used and one parent would hold

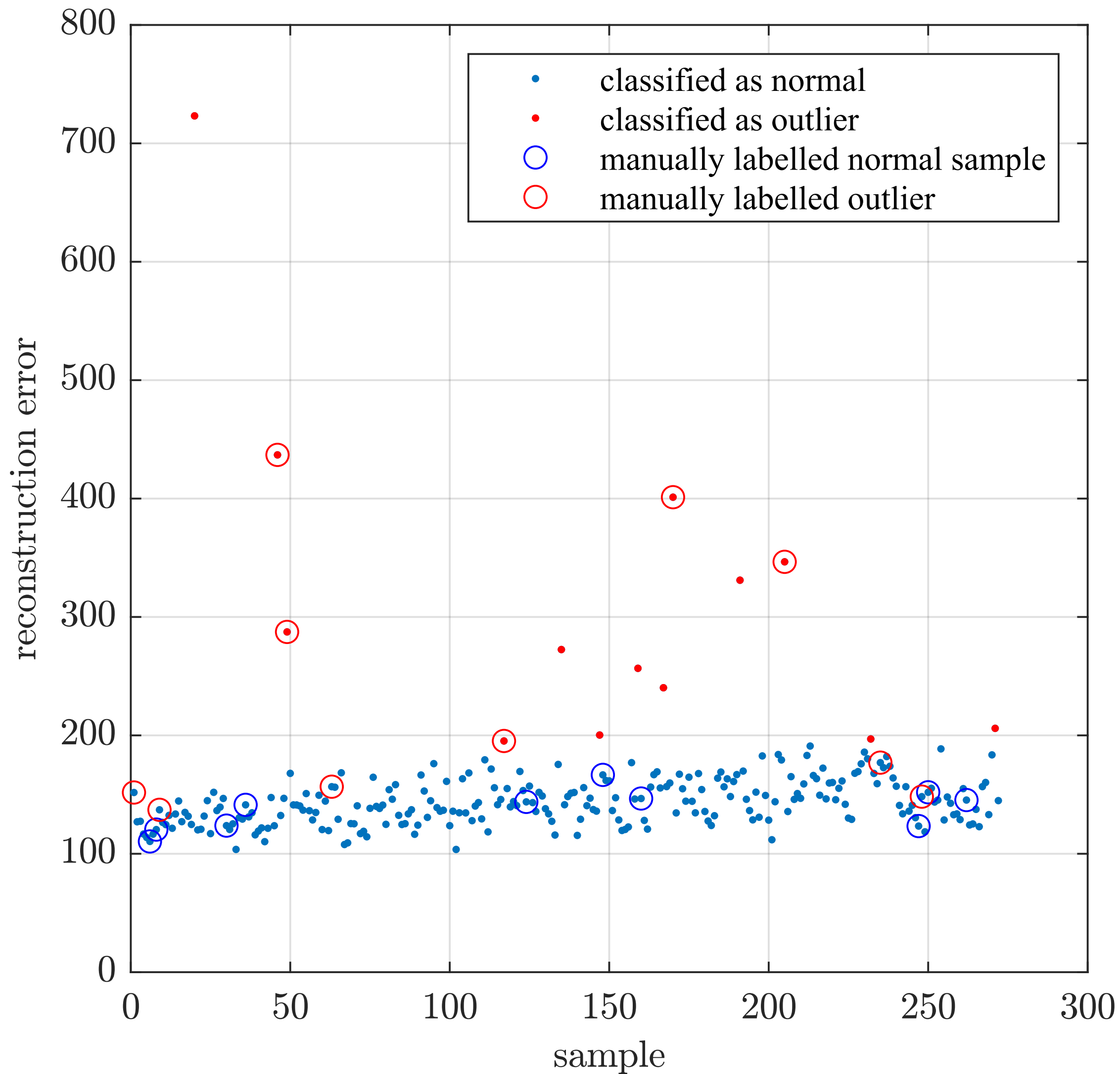


Fig. 8.18: second example of outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples with models trained with random hyperparameters

the value 0 and the other the value 1, calculating the mean and round it to the next integer, would favour choosing the LSTM-layer.

In [2] the assumption was made that the model generalizes better if the encoder has more and the decoder fewer layers. By including the conditional layers into the HPO this could not be concluded for the HPOs done in this thesis. Either non of the layers were chosen or an additional layer for the decoder with a higher number of neurons than the first LSTM-layer, which has converged to a much lower number as normal, was chosen. Only a few trial runs were done and further investigation is needed to see a tendency if this was caused by the random seed, a too low number of training examples to deal with more degrees of freedom in the HPO.

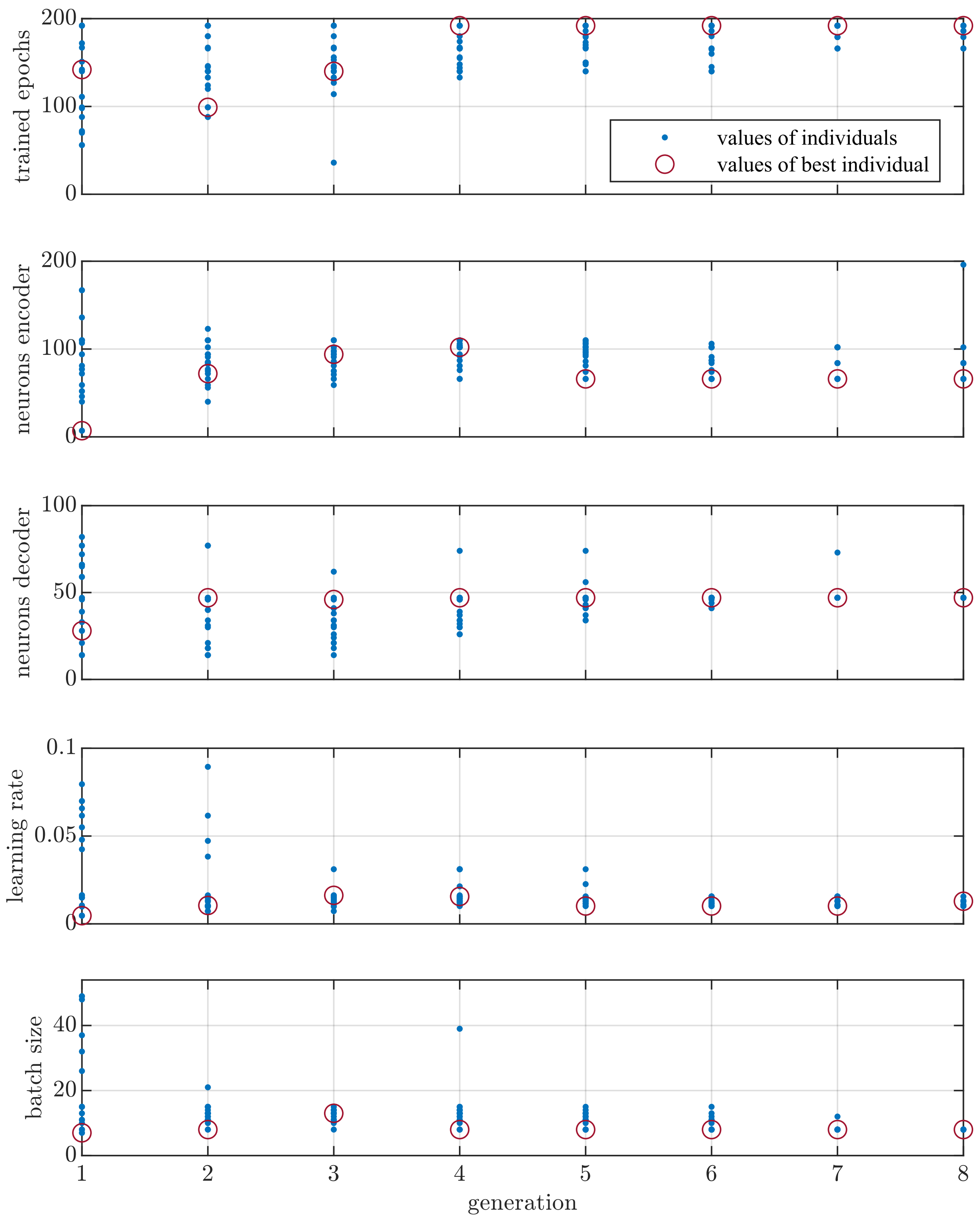


Fig. 8.19: results of the HPO done on the enlarged domain for the site Fehring

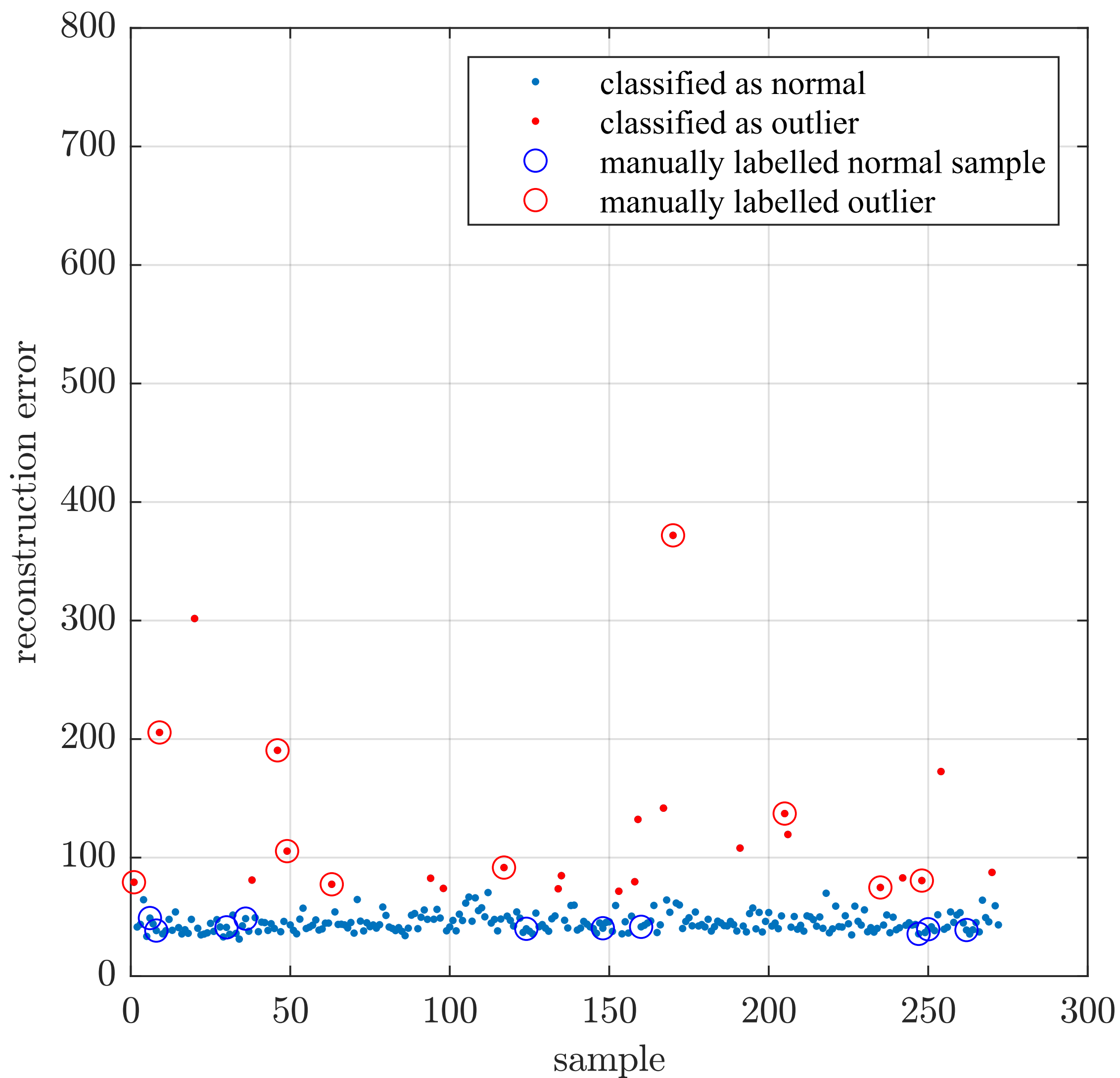


Fig. 8.20: outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples; models where trained on hyperparameters found by a HPO on the enlarged search space

As example the values the individuals take during one HPO done of the Fehring dataset using the extended model is shown in Figure 8.21.

Also, the reconstruction error of an ML-model trained with the values of the best individual of the last generation, as well as the outlier detection using the skewness-adjusted boxplot and the position of the manually labelled examples are shown in Figure 8.22. The reconstruction error of the examples that are classified as normal is higher than in the other two variants of the HPO. Also, the separation between the two classes is not so clear, because the reconstruction error of

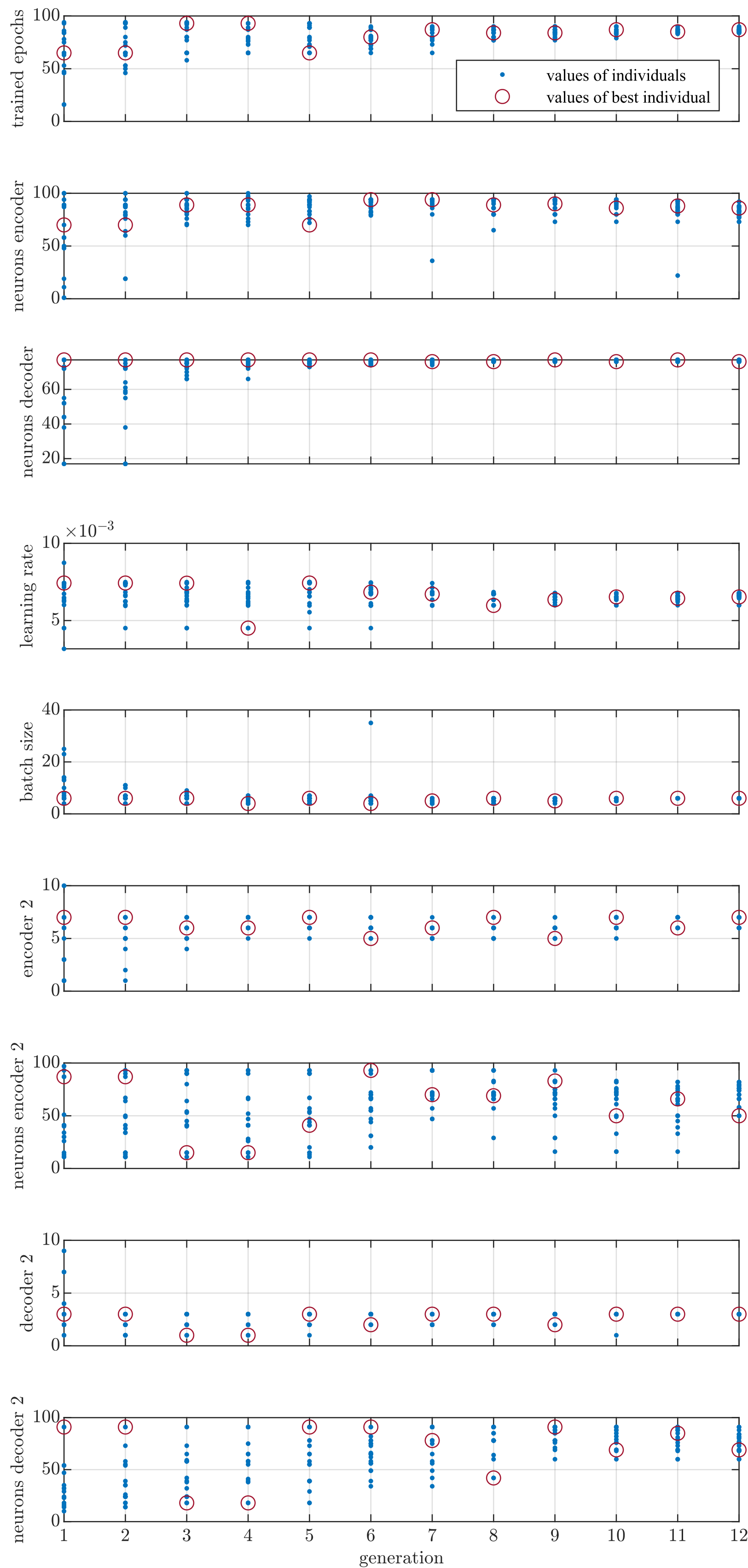


Fig. 8.21: results of the HPO done on the extended domain for the site Fehring

the outliers is lower. So, it happens that some of the manually labelled outliers are categorized as good.

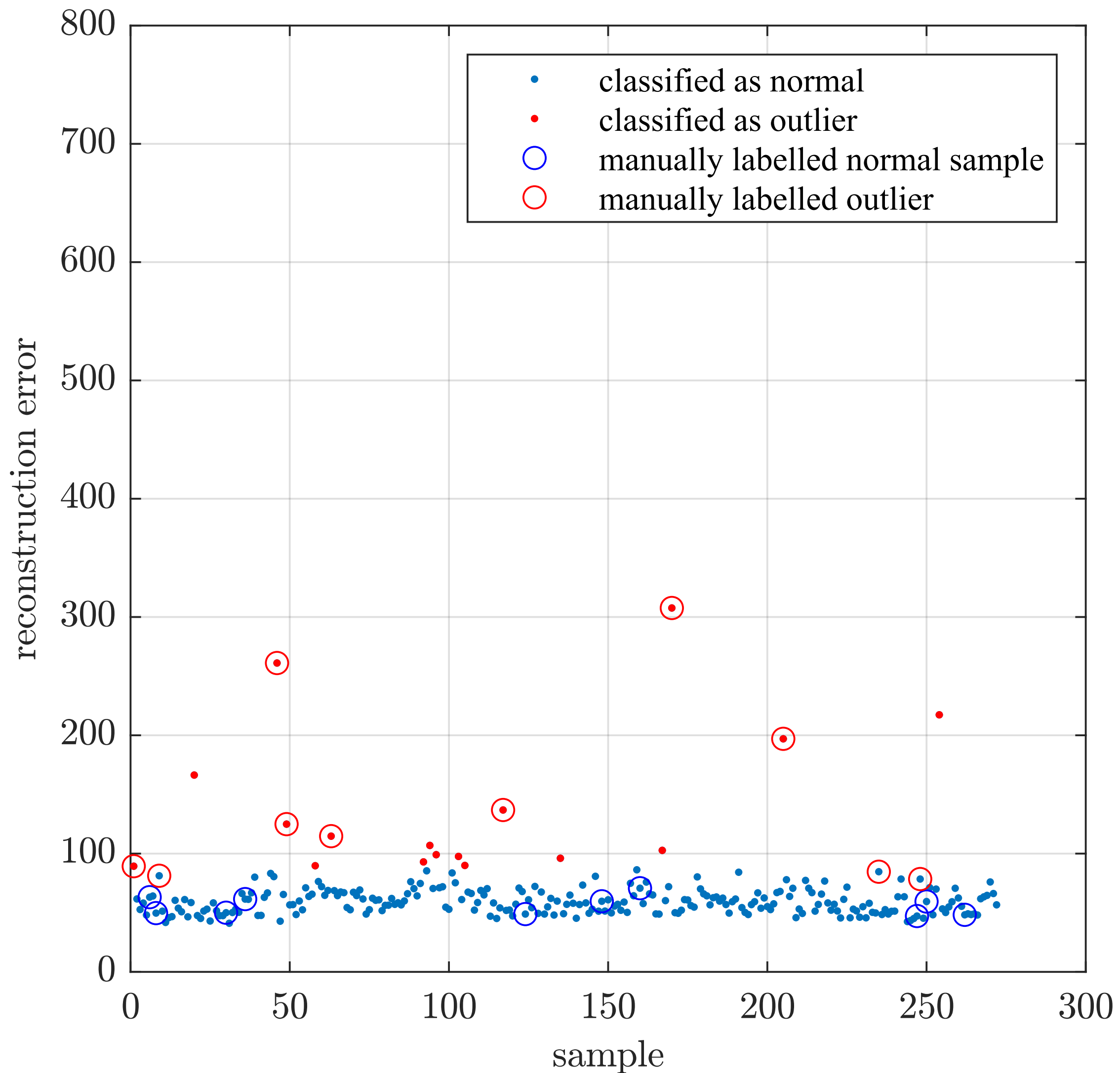


Fig. 8.22: outlier detection using the skewness-adjusted boxplot and manually selected anomalous and non-anomalous samples; models where trained on hyperparameters found by the HPO on the extended search space

8.3.5.4 Overview of the Results of all HPOs

In table 8.2 an overview of the HPO done on the datasets of the sites Fehring and Aspern with the described setup in this chapter is given. The type describes which variation of the algorithm was used. Also, the number of files used for training varied. The results are given in the same or-

der as used in this chapter in general: [trainedEpochs, numberOfNeuronsEncoder, numberOfNeuronsDecoder, learningRate, miniBatchSize] and for the extended optimization as [trainedEpochs, numberOfNeuronsEncoder, numberOfNeuronsDecoder, learningRate, miniBatchSize, additionalLSTMLayerEncoder, numberNeuronsAdditionalLSTMencoder, additionalLSTMLayerDecoder, numberNeuronsAdditionalLayerDecoder]

index	site	type	number files	generations	result
1	Aspern	normal	80	13	[75, 54, 37, 756, 3]
2	Aspern	normal	80	12	[99, 46, 43,868, 3]
3	Aspern	normal	80	13	[99, 58, 46, 784, 3]
4	Aspern	normal	160	13	[68, 69, 40, 612, 5]
5	Fehring	enlarged	160	10	[174, 44, 37, 3754, 13]
6	Fehring	enlarged	160	8	[80, 46, 61, 569, 5]
7	Fehring	enlarged	80	13	[126, 68, 97, 2353, 12]
8	Fehring	normal	80	12	[49, 82, 60, 878, 3]
9	Fehring	normal	80	12	[76, 49, 56, 978, 7]
10	Fehring	normal	80	12	[69, 47, 96, 910, 7]
11	Fehring	extended & enlarged	80	12	[178, 28, 65, 2482, 22, 3, 134, 7, 83]
12	Fehring	extended & enlarged	80	11	[399, 32, 55, 3421, 12, 3, 247, 5, 286]
13	Fehring	extended	80	12	[87, 86, 76, 635, 6, 7, 50, 3, 69]

Table 8.2: results for the hyperparameters of different HPO-runs; the *site* indicates the dataset that the HPO was performed on, in the column *type* is specified which variation of the HPO was done, the third column contains the number of files that where used, the fourth column indicates the number of generations used for the optimization and the last column contains the results of the HPO in the format defined in Section 8.3.5.4

If comparing the results of all HPO-runs tabulated in Table 8.1 the same assumption can be made as in [2] that the encoder has in most cases a higher number of neurons than the decoder. It seems that there are different optimal sizes of the mini-batches depending on the dataset. When the hyperparameters are optimized for the data of the site Aspern, in three of four cases 3 is the result for the optimal batch-size for the normal domain. Whereby for Fehring it tends to be higher, around 7 for the normal domain. For the enlarged search space it seems that a higher mini-batch size is favoured.

The assumption often found in the literature that when using a smaller learning rate, more epochs need to be trained cannot be made from these results. The resulting value for the number epochs trained varies relatively strong, even on the same data.

It also cannot be assumed that more LSTM-layers in the encoder are favoured towards the model with only one layer. When comparing the HPOs with the indices 11-13, one time an additional layer in the decoder was used in the configuration hold by the best individual of the last generation, once one in the encoder and once non of the above. However, it can be assumed that if the HPO has more degrees of freedom the number of epochs trained needs to be higher. This may possibly

be explained by the reasonably low number of input features, the low number of training files or overall the structure and degree of complexity of the data itself.

Chapter 9
Summary and Conclusion

This thesis investigated the use of genetic algorithms for the hyperparameter optimization of a machine learning model for outlier detection to improve the detection of unusual events. To accomplish this, a genetic algorithm with fewer evaluations compared to a classical genetic algorithm was developed.

Most of the data in real industrial application is gathered without labels, so for this approach, only a subset of non-anomalous examples needs to be labelled and not the whole data. This is done because the machine learning model is trained in a highly biased way only using non-anomalous data and so the reconstruction error of outliers, which were not shown to the network in the training phase, is most likely higher because these unknown pattern can not be reconstructed as accurate as patterns of non-anomalous examples the network learned during the training phase.

The combination of machine learning with a meta-heuristic is afflicted by uncertainties. The weights, which are the values that are adjusted during the training of neural networks, are set by default at the initialization to random values. Further, also the first population which is the starting point of the HPO is initialized with random values. This leads to performance fluctuation and each setting of the hyperparameters needs to be evaluated on different folds of the data to get better generalization. The number of repetitions is a trade-off between generalization, run time and the possible risk of overfitting. When using an evolutionary algorithm, new hyperparameters are introduced that also need to be set. Also, the question arises of how big the portion of the data that is used for the HPO should be to guarantee some kind of generalization without increasing runtime too much.

Evaluating the performance of unsupervised learning techniques is not a trivial task and cannot be done straight forward by calculating for example a confusion matrix as often used in supervised learning. Therefore, to show the functional principle exemplary some measurements were manually labelled and compared to the outcome of the model. This leads to the assumption that LSTM-VAEs with optimized hyperparameters perform better than models trained with random hyperparameters. The results of these evaluation suggest that when using optimized hyperparameters the clusters of anomalous and non-anomalous data are better distinguishable compared to models trained on random hyperparameters. Also, the average fitness of the individuals per generation de-

creases exponentially, as expected for a genetic algorithm. Further investigations are needed to find the hyperparameters of the LSTM-VAE that have the biggest impact on the performance and also which subset of the features is most suitable as input for the ML model to archive the best possible performance in a reasonable runtime.

Overall, the thesis showed that genetic algorithms are a suitable method for the hyperparameter optimization of machine learning models. It could be considered for practical application because the runtime, which is often the limiting factor can be adjusted by training less generations and even then archiving good results.

Some important insights of this work are:

1. It has been shown that genetic algorithms are a suitable technique for the HPO of variational autoencoders. This is concluded because the reconstruction error of non-anomalous examples is lower when the ML-model is trained with optimized hyperparameters and the overall performance of the outlier detection is increased.
2. It could not be shown that a larger searchspace and more layers in the decoder and encoder could increase the performance.
3. Further, the discovery was made that a relatively low number of non-anomalous samples used for the optimization lead to sufficient results.
4. Not all chosen features contribute to the reconstruction error equally. Some features are reconstructed rather accurately, even if they are considered anomalous.
5. It is assumed that genetic operators that create new values for the parameters to be optimized in the course of the algorithm are beneficial, since the winning individual often contains values for the parameters that were not part of the first generation.

References

[1] Tsatsral Amarbayasgalan et al. “Unsupervised Anomaly Detection Approach for Time-Series in Multi-Domains Using Deep Reconstruction Error”. In: *Symmetry* 12.8 (Aug. 2020), p. 1251. ISSN: 20738994.

[2] Stefan Herdy. “Machine Learning in the Context of Time Series”. MA thesis. Chair of Automation, Montanuniversitaet Leoben, 2020.

[3] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 08997667.

[4] Jia Wu et al. “Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimizationb”. In: *Journal of Electronic Science and Technology* 17.1 (Mar. 2019), pp. 26–40. ISSN: 1674862X.

[5] Klaus Greff et al. “LSTM: A Search Space Odyssey”. In: *IEEE Trans. Neural Networks Learn. Syst.* 28.10 (July 2016), pp. 2222–2232. ISSN: 21622388.

[6] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Berlin: Springer-Verlag, 2008. ISBN: 9783540731894.

[7] “An overview of machine learning”. In: *Machine Learning: An Artificial Intelligence Approach*. Ed. by Jaime G Carbonell, Ryszard S Michalski, and Tom M Mitchell. San Francisco (CA): Morgan Kaufmann, 1983, pp. 3–23. ISBN: 9780080510545.

[8] Herbert A Simon. “Machine Learning: An Artificial Intelligence Approach”. In: ed. by Jaime G Carbonell, Ryszard S Michalski, and Tom M Mitchell. San Francisco (CA): Morgan Kaufmann, 1983, pp. 25–37. ISBN: 9780080510545.

[9] Sven Tomforde et al. “”Know Thyself” - Computational Self-Reflection in Intelligent Technical Systems”. In: *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. London: IEEE, Sept. 2014, pp. 150–159. ISBN: 9781479963782.

[10] “Introduction to Scientific Data Mining: Direct Kernel Methods and Applications”. In: *Computationally Intelligent Hybrid Systems: The Fusion of Soft Computing and Hard Computing*. Ed. by Seppo J. Ovaska. New York: Wiley Online Library, 2004, pp. 317–362. ISBN: 0471476684.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Cambrige and London: MIT Press, 2016. ISBN: 0262035618. URL: `http://www.deeplearningbook.org/`.

[12] Giuseppe Primiero. “Algorithmic Iteration for Computational Intelligence”. In: *Minds & Machines* 27.3 (Sept. 2017), pp. 521–543. ISSN: 15728641.

[13] Ethem Alpaydin. *Introduction to Machine Learning*. 3rd ed. Adaptive Computation and Machine Learning. Cambridge and London: MIT Press, 2014. ISBN: 9780262028189.

[14] Dong Yu et al. “Introduction to the Special Section on Deep Learning for Speech and Language Processing”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 20.1 (Oct. 2011), pp. 4–6. ISSN: 15587924.

[15] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. Cambridge and London: MIT press, 2012. ISBN: 9780262018258.

[16] Li Yang and Abdallah Shami. “On hyperparameter optimization of machine learning algorithms: Theory and practice”. In: *Neurocomputing* 415 (Nov. 2020), pp. 295–316. ISSN: 09252312.

[17] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. 2nd ed. Cambridge and London: MIT press, 2018. ISBN: 9780262039406.

[18] Alaa Sagheer and Mostafa Kotb. “Time series forecasting of petroleum production using deep LSTM recurrent networks”. In: *Neurocomputing* 323 (Jan. 2019), pp. 203–213. ISSN: 09252312.

[19] Rob J. Hyndman and Anne B. Koehler. “Another look at measures of forecast accuracy”. In: *International Journal of Forecasting* 22.4 (Oct. 2006), pp. 679–688. ISSN: 0169-2070.

[20] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *Bull. Math. Biophys.* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602.

[21] Sergios Theodoridis. *Machine Learning: A Bayesian and Optimization Perspective*. London: Elsevier, 2015. ISBN: 9780128015223.

[22] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Berlin: Springer-Verlag, 2012. ISBN: 9783642247965.

[23] Walter Hugo Lopez Pinaya et al. “Autoencoders”. In: *Machine Learning*. Cambridge: Academic Press, Jan. 2020, pp. 193–208. ISBN: 9780128157398.

[24] Diederik P. Kingma and Max Welling. *An Introduction to Variational Autoencoders*. New York: Now Publishers, 2019. ISBN: 9781680836233.

[25] Daehyung Park, Yuuna Hoshi, and Charles C. Kemp. “A Multimodal Anomaly Detector for Robot-Assisted Feeding Using an LSTM-Based Variational Autoencoder”. In: *IEEE Robotics and Automation Letters* 3.3 (Feb. 2018), pp. 1544–1551. ISSN: 23773766.

[26] Carl Doersch. *Tutorial on Variational Autoencoders*. [Online; accessed 21. Feb. 2021]. 2016. arXiv: `1606.05908 [stat.ML]`. URL: `https://arxiv.org/abs/1606.05908`.

[27] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. 3rd ed. Cham: Springer International Publishing, 2016. ISBN: 9783319298528.

[28] Ratnadip Adhikari and R. K. Agrawal. *An Introductory Study on Time Series Modeling and Forecasting*. Saarbrücken: LAP LAMBERT Academic Publishing, 2013. ISBN: 9783659335082.

[29] A. Karpathy, Justin Johnson, and Li Fei-Fei. *Workshop Track -iclr 2016 Visualizing and Understanding Recurrent Networks*. [Online; accessed 18. Dec. 2020]. 2016. URL: `https:`

//www.semanticscholar.org/paper/Workshop-Track-iclr-2016-Visualizing-and-Recurrent-Karpathy-Johnson/8390c96f0b2ff3b36b232f7f9918401e51632f4e#citing-papers.

[30] Souhaib Ben Taieb et al. "A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition". In: *Expert Systems with Applications* 39.8 (June 2012), pp. 7067–7083. ISSN: 09574174.

[31] Nico Görnitz et al. "Toward supervised anomaly detection". In: *Journal of Artificial Intelligence Research* 46.1 (Jan. 2013), pp. 235–262. ISSN: 1076-9757.

[32] Shuyu Lin et al. "Anomaly Detection for Time Series Using VAE-LSTM Hybrid Model". In: *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020*. Barcelona: IEEE, May 2020, pp. 4322–4326. ISBN: 9781509066322.

[33] Yuta Kawachi, Yuma Koizumi, and Noboru Harada. "Complementary Set Variational Autoencoder for Supervised Anomaly Detection". In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2018, Calgary, AB, Canada, April 15-20, 2018*. Calgary: IEEE, Apr. 2018, pp. 2366–2370. ISBN: 9781538646571.

[34] Boris Iglewicz and David Caster Hoaglin. *How to Detect and Handle Outliers*. Milwaukee: ASQC Quality Press, 1993. ISBN: 9780873892476.

[35] Tung Kieu, Bin Yang, and Christian S. Jensen. "Outlier Detection for Multidimensional Time Series Using Deep Neural Networks". In: *2018 19th IEEE International Conference on Mobile Data Management (MDM)*. IEEE. Aalborg, June 2018, pp. 125–134.

[36] Zijian Niu, Ke Yu, and Xiaofei Wu. "LSTM-Based VAE-GAN for Time-Series Anomaly Detection". In: *Sensors* 20.13 (2020). ISSN: 14248220.

[37] F. Zhang and H. Fleyeh. "Anomaly Detection of Heat Energy Usage in District Heating Substations Using LSTM based Variational Autoencoder Combined with Physical Model". In: *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. 2020, pp. 153–158.

[38] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer-Verlag, 1996. ISBN: 9783540606765.

[39] Kim-Fung Man, Kit Sang Tang, and Sam Kwong. *Genetic algorithms: concepts and designs*. 3rd ed. London: Springer Science and Business Media, 2001. ISBN: 978-1-85233-072-9.

[40] MathWorks Inc. *Coding and Minimizing the Fitness Function Using the Genetic Algorithm*. [Online; accessed 19. Dec. 2020]. 2020. URL: https://www.mathworks.com/help/gads/fitness-function-forms.html%3Bjsessionid=abaebde918f54091963e9fb6448a.

[41] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning*. Cham: Springer International Publishing, 2019. ISBN: 9783030053178.

[42] Chia-Feng Juang. "A hybrid of genetic algorithm and particle swarm optimization for recurrent network design". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.2 (Apr. 2004), pp. 997–1006. ISSN: 19410492.

[43] Borhan Kazimipour, Xiaodong Li, and A. K. Qin. "A review of population initialization techniques for evolutionary algorithms". In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. Beijing: IEEE, July 2014, pp. 2585–2592.

[44] Matthias Reif, Faisal Shafait, and Andreas Dengel. "Meta-learning for evolutionary parameter optimization of classifiers". In: *Mach. Learn.* 87.3 (June 2012), pp. 357–380. ISSN: 15730565.

[45] Artem Sokolov, Darrell Whitley, and Andre' da Motta Salles Barreto. "A note on the variance of rank-based selection strategies for genetic algorithms and genetic programming". In: *Genetic Programming and Evolvable Machines* 8.3 (Sept. 2007), pp. 221–237. ISSN: 15737632.

[46] Melanie Mitchell. *An introduction to genetic algorithms*. Cambridge and London: MIT press, 1998. ISBN: 0262133164.

[47] Yuri R Tsoy. "The influence of population size and search time limit on genetic algorithm". In: *7th Korea-Russia International Symposium on Science and Technology, Proceedings KORUS 2003*. Vol. 3. IEEE. Ulsan, 2003, pp. 181–187. ISBN: 8978686176.

[48] Fernando G. Lobo, David E. Goldberg, and Martin Pelikan. "Time Complexity of Genetic Algorithms on Exponentially Scaled Problems". In: *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. GECCO'00. Las Vegas: Morgan Kaufmann Publishers Inc., 2000, pp. 151–158. ISBN: 1558607080.

[49] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. "Tunability: importance of hyperparameters of machine learning algorithms". In: *Journal of Machine Learning Research* 20.1 (Jan. 2019), pp. 1934–1965. ISSN: 15324435.

[50] Steven R Young et al. "Optimizing deep learning hyper-parameters through an evolutionary algorithm". In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. 2015, pp. 1–5.

[51] A Fiszelew et al. "Finding optimal neural network architecture using genetic algorithms". In: *Advances in computer science and engineering research in computing science* 27 (2007), pp. 15–24. ISSN: 0973-6999.

[52] Fernando Itano, Miguel Angelo de Abreu de Sousa, and Emilio Del-Moral-Hernandez. "Extending MLP ANN hyper-parameters Optimization by using Genetic Algorithm". In: *2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018*. Rio de Janeiro: IEEE, 2018, pp. 1–8. ISBN: 9781509060146.

[53] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artifi-*

cial intelligence and statistics. Vol. 9. JMLR Proceedings. Sardinia: JMLR.org, May 2010, pp. 249–256.

[54] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *The Journal of Machine Learning Research* 13 (Feb. 2012), pp. 281–305. ISSN: 1532-4435.

[55] Peter Henderson et al. “Deep reinforcement learning that matters”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. Palo Alto, Feb. 2018. ISBN: 9781577358008.

[56] Jiawei Yang, Susanto Rahardja, and Pasi Fränti. “Outlier detection: how to threshold outlier scores?” In: *AIIPCC ’19: Proceedings of the International Conference on Artificial Intelligence, Information Processing and Cloud Computing*. New York: Association for Computing Machinery, Dec. 2019, pp. 1–6. ISBN: 978-1-45037633-4.

[57] Songwon Seo. “A review and comparison of methods for detecting outliers in univariate data sets”. MA thesis. University of Pittsburgh, 2006.

[58] Peter J. Rousseeuw and Mia Hubert. “Robust statistics for outlier detection”. In: *WIREs Data Mining and Knowledge Discovery* 1.1 (2011), pp. 73–79. ISSN: 19424795.

[59] Rand Wilcox. *Introduction to robust estimation and hypothesis testing*. Amsterdam and Boston: Academic Press, 2012. ISBN: 978-0-12-386983-8.

[60] Dhwani Dave and Tanvi Varma. “A Review of various statestical methods for Outlier Detection”. In: *International Journal of Computer Science & Engineering Technology (IJCSET)* 5 (Feb. 2014), pp. 137–140. ISSN: 22293345.

[61] Leonard J. J. Kazmier. *Schaum’s Outline of Business Statistics, Fourth Edition (Schaum’s Outlines)*. 4th ed. New York: McGraw-Hill Education, Sept. 2004. ISBN: 0071430997.

[62] Peter J. Rousseeuw and Christophe Croux. “Alternatives to the Median Absolute Deviation”. In: *Journal of the American Statistical Association* 88.424 (1993), p. 12731283.

[63] Christophe Leys et al. “Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median”. In: *Journal of Experimental Social Psychology* 49.4 (July 2013), pp. 764–766. ISSN: 00221031.

[64] Frederick Mosteller, John Wilder Tukey, and David C. Hoaglin. *Understanding Robust and Exploratory Data Analysis*. New York, NY, USA: John Wiley & Sons Inc, June 1983. ISBN: 978-0-47109777-8.

[65] Ronald E. Shiffler. “Maximum Z Scores and Outliers”. In: *The American Statistician* 42.1 (1988), pp. 79–80. ISSN: 00031305.

[66] Guy Brys, Mia Hubert, and Anju Struyf. “A Robust Measure of Skewness”. In: *Journal of Computational and Graphical Statistics* 13.4 (2004), pp. 996–1017. ISSN: 15372715.

[67] M. Hubert and E. Vandervieren. “An adjusted boxplot for skewed distributions”. In: *Computational Statistics & Data Analysis* 52.12 (2008), pp. 5186–5201. ISSN: 0167-9473.

[68] E Vandervieren and Mia Hubert. "An adjusted boxplot for skewed distributions". In: *COMPSTAT 2004: Proceedings in Computational Statistics*. Springer-Verlag. Prague, 2004, pp. 1933–1940. ISBN: 9783790815542.

[69] G. Brys, M. Hubert, and A. Struyf. "A Comparison of Some New Measures of Skewness". In: *Developments in Robust Statistics*. Heidelberg: Physica, 2003, pp. 98–113. ISBN: 9783642632419.

[70] Mia Hubert and Stephan Van der Veeken. "Outlier detection for skewed data". In: *Journal of Chemometrics: A Journal of the Chemometrics Society* 22.3-4 (2008), pp. 235–246. ISSN: 1099128X.

[71] Nikolaos Gorgolis et al. "Hyperparameter Optimization of LSTM Network Models through Genetic Algorithm". In: *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*. IEEE. July 2019, pp. 1–4. ISBN: 9781728149608.