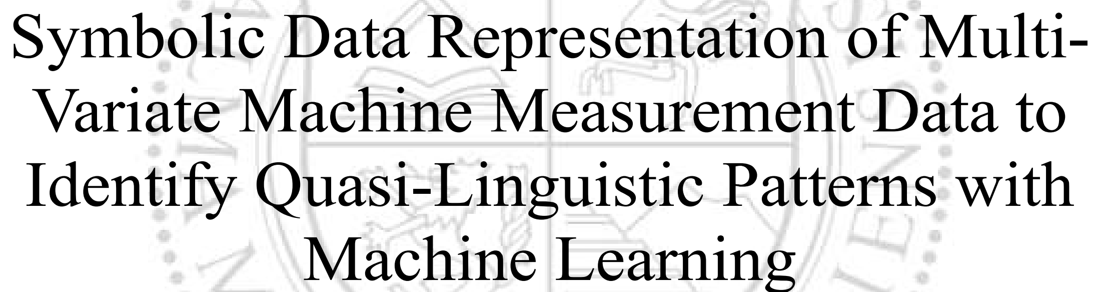
Chair of Automation

Master's Thesis



Symbolic Data Representation of Multi-Variate Machine Measurement Data to Identify Quasi-Linguistic Patterns with Machine Learning

Philip Nuser, BSc

November 2023

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 14.11.2023

Unterschrift Verfasser/in
Philip Nuser

Danksagung - Dedication

First of all, I want to express my deep gratitude to Professor Paul O'Leary, the head of the Chair of Automation, not only for the possibility and support writing this thesis, but also for his exceptional way of teaching. His ingenious mathematical and technical understanding combined with his broad knowledge across various fields were exceptionally inspiring for my studies.

In addition, I want to acknowledge the team of the Chair of Automation, especially Anika Terbuch, for always being welcoming and supportive in every situation.

Special thanks also to my fellow students and friends, with whom I shared a great time and many unforgettable experiences during my studies.

Furthermore, I want to thank my family for always being supportive in every way possible and allowing me to make all my dreams come true. Without their selfless support none of this would have been possible.

Kurzfassung

Diese Masterarbeit untersucht die Erkennung von Anomalien in multivariaten Zeitreihen-Daten mit Sprachmodellen aus der Computerlinguistik. Die Grundlage bildet die Umwandlung der numerischen Maschinendaten in tokenisierte Daten, ähnlich zu Text. Der Prozess der Tokenisierung wird durch Diskretisierung der Daten und Zuweisung eindeutiger Token zu den diskreten Werten realisiert. Die so erhaltenen symbolischen Zeitreihen wurden dann mit zwei unterschiedlichen Ansätzen auf Anomalien untersucht.

Der erste Ansatz basiert auf N-Gramm Sprachmodellen. Ein N-Gramm ist eine Sequenz von Wörtern der Länge n. Die Anzahl der N-Gramme im Datensatz wird berechnet und mit einem statistischen Maß zur Beurteilung der Relevanz von Termen in einem Textkörper, dem Tf-idf-Maß, gewichtet. Dieses Maß dient als Grundlage zur Erkennung von Anomalien. Die Idee dahinter ist, dass N-Gramme, welche selten im gesamten Textkorpus vorkommen, auf außergewöhnliches Verhalten hindeuten.

Der zweite vorgestellte Ansatz nutzt maschinelles Lernen für die Erkennung von Anomalien im tokenisierten Datensatz. Dafür wurde ein Transformer-Modell programmiert, welches normalerweise zur Sprachmodellierung benutzt wird. Das Modell erhält eine Symbolsequenz, in der zufällige Einträge durch einen Masken-Token ersetzt werden, und versucht, die originalle numerische Sequenz wiederherzustellen. Weicht die Rekonstruktion stark vom Original ab, sind Anomalien im Datensatz zu erwarten. Beide Methoden wurden erfolgreich an einem Datensatz, der von Sensoren einer Maschine zur Verbesserung der Bodenbeschaffenheit für Gebäudefundamente stammt, angewandt. Die Auswertung der Ergebnisse hat gezeigt, dass eine Anomalieerkennung mit den entwickelten Ansätzen möglich ist und rechtfertigt besonders die Weiterentwicklung des künstlichen neuronalen Modells.

Abstract

This thesis is an exploratory work of unsupervised anomaly detection in multi-variate time-series machine data with methods originating from natural language processing. The foundation is laid by tokenizing the time-series data, i.e., converting the numeric machine data into symbolic data similar to textual data. The process of tokenization is realized by discretizing the data and assigning unique tokens to the discrete values. The symbolic sequences obtained are then inspected for anomalies with two different approaches.

The first method is based on word n-gram language models. A n-gram is a sequence of words of length n. The counts of those n-grams in the data set are computed and weighed with a measure for the importance of a word to a document, the term frequency-inverse document frequency measure, to derive anomaly scores for each sequence.

The second approach presented utilizes a machine learning model, more specific a masked language model with a transformer architecture at its core. Random tokens in the input sequences get masked and the transformer is trained to recreate the numeric sequences. When an input sequence that has not been used for training outputs a diverging numeric sequence, anomalies in this sequence are expected.

Both anomaly detection methods were programmed and successfully applied to an unlabeled data set originating from instrumented machinery used for ground improvement of building foundations. The results indicate that both approaches are principally functional and strongly justify continued work in this area, especially on the machine learning model.

Acronyms

AI	Artificial Intelligence
ASCII	American Standard Code for Information Interchange
ANN	Artificial Neural Network
BOW	Bag-of-Words
BPTT	Back-Propagation Through Time
BPE	Byte Pair Encoding
CBW	Continous Bag-of-Words
CLM	Causal Language Model
FNN	Feedforward Neural Network
GPU	Graphics Processing Unit
HML	Hybrid Machine Learning
KPI	Key Performance Indicator
LLM	Large Language Model
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MBE	Mean Bias Error
ML	Machine Learning
MLM	Masked Language Model
MVTS	Multi-Variate Time-Series
NLP	Neural Language Processing
NMT	Neural Machine Translation
NSP	Next Sentence Prediction
OOV	Out-of-Vocabulary
PAA	Piece-wise Aggregate Approximation
POS	Part-of-Speech
RMSE	Root Mean Squared Error
RSE	Relative Squared Error
RNN	Recurrent Neural Network
SSE	Sum of Squared Errors
TF-IDF	Term Frequency - Invers Document Frequency
TPU	Tensor Processing Unit
VAE	Variational Auto Encoder

Contents

1 Introduction ... 1

2 Machine Learning Basics ... 3
- 2.1 Tasks ... 3
  - 2.1.1 Supervised Learning ... 4
  - 2.1.2 Unsupervised Learning ... 5
- 2.2 Evaluation Metrics ... 6
  - 2.2.1 Metrics for Classification ... 6
  - 2.2.2 Metrics for Regression ... 9
- 2.3 Artificial Neurons ... 11
  - 2.3.1 Perceptrons ... 11
  - 2.3.2 Gradient Descent ... 12
  - 2.3.3 Activation Functions ... 13

3 Deep Learning ... 15
- 3.1 Artifical Neural Networks ... 15
  - 3.1.1 Feedforward Networks ... 15
  - 3.1.2 Back-Propagation ... 17
  - 3.1.3 Recurrent Networks ... 18
- 3.2 Transformers ... 20
  - 3.2.1 Vanilla Architecture ... 21
  - 3.2.2 Attention ... 22

4 Natural Language Processing ... 25
- 4.1 Brief Linguistics ... 25
  - 4.1.1 Challenges in Language Processing ... 26
  - 4.1.2 Lexical Resources ... 27
  - 4.1.3 Features for Textual Data ... 27
- 4.2 Tokenization of Textual Data ... 28

4.2.1 Word-Based Tokenization 28
4.2.2 Character-Based Tokenization 29
4.2.3 Subword-Based Tokenization 29
4.3 N-Gram Language Models 30
4.3.1 Bag-of-Words 30
4.3.2 Term Frequency-Inverse Document frequency 31
4.3.3 Bag-of-n-grams 32
4.4 Neural Language Models 32
4.4.1 Word Embedding 33
4.4.2 RNN Models 35
4.4.3 Large Language Models 37

5 Application on Multi-Variate Time-Series Data 39
5.1 Underlying Process 39
5.2 Tokenization of MVTS Data 41
5.2.1 Time domain of MVTS Data 43
5.2.2 Discretization of MVTS Data 44
5.2.3 Wording of MVTS Data 45
5.3 N-Gram-based Anomaly Detection 48
5.3.1 Process Description 49
5.3.2 TF-IDF Scores 49
5.4 Transformer-based Anomaly Detection 53
5.4.1 Transformer Model 53
5.4.2 Data Preprocessing for Machine Learning 54
5.4.3 Training 55
5.4.4 Validation 56

6 Summary and Conclusion 58

List of figures 59

List of tables 61

A Appendix A 62
A.1 Matlab Tokenizer Code 63

Bibliography 70

Chapter 1
Introduction

This thesis is an exploratory work of detecting anomalies with linguistic methods in multi-variate time-series (MVTS) data. The basis of this work is laid by converting the numeric values of a time-series into symbols. The symbolic data is than handled with two natural language processing (NLP) approaches.

The first one is based on a rather classic bag-of-ngrams model. A n-gram is a sequence of n adjacent words. Put them in an unordered collection and you get a bag-of-ngrams. Earlier language models were purely based on the statistical probabilities of the occurrence of such n-grams[1]. So the idea is, if we create symbolic sequences, we can also create a bag-of-ngrams for each sequence. Furthermore, we can use the frequencies of the n-grams to derive anomaly scores. A n-gram that is rarely seen across all sequences is likely to indicate an anomaly in the data set. A very useful measure in this context is the term frequency-inverse document frequency (TF-IDF) measure[2].

Word n-gram models were outperformed almost 20 years ago by deep learning models[3]. These machine learning models use multiple layers of artificial neurons to mimic the architecture of the human brain. Machine learning models have evolved over time and today the transformer is considered to be the state-of-the-art architecture for NLP tasks. A key component of the transformer model is the attention mechanism. This invention allows the model to pay attention to multiple inputs at once and therefore understand complex relationships in the data[4]. A very famous example for a transformer is the Generative Pre-Trained Transformer (GPT) series from OpenAI[5].

The second approach presented utilizes a transformer model that is very similar to BERT, a language model presented by Google in October 2018. Its training strategy manages to learn without any labeled data. This fact comes in very handy, because the model is applied to an unlabeled multi-variate time-series data set. That means the model shall learn to detect anomalies in an unsupervised manner without providing any information how an anomaly looks like or where it is found in the provided data[6].

The data for the application of both models originates from a ground improvement process. It is composed of 9 different channels, each corresponding to a different sensor record sampled at 1 Hz. Previous research performed by the Chair of Automation in Leoben to detect anomalies in this

data set was mainly based on long short-term memory (LSTM) networks[7]. This work is breaking new ground.

Chapter 2
Machine Learning Basics

This chapter provides a brief overview of fundamental concepts used in *machine learning* (ML).

Over a century before programmable computers where even built, Ada Lovelace contemplated the possibility of machines achieving intelligence [8]. Another milestone dealing with these questions was published by Alan Turing in 1950 in his essay ”Computing Machinery and Intelligence”. He introduced the concept of the Turing Test, which is a test of a machine’s ability to mimic human communication skills. Rather than addressing the question of whether machines can think, he focuses on a more practical question: Can a machine imitate human intelligence well enough to pass for a human in a conversation? This imitation game forms the basis of the Turing test. He also explores the limitations and potential of artificial intelligence and touches the idea of machine learning and adaption over 70 years ago [9]. Machine learning is a subset of *artificial intelligence* (AI). A broadly used definition for machine learning was coined by Tom Mitchell in 1997:

> A computer program is said to *learn* from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [10, p.2].

My personal opinion is that the term experience is used rather sloppy in this context. Human experience involves our senses and our mind, whereas machine learning is mostly based on data sets. For example, how would you share your experience of eating a banana with someone who has never eaten a banana before. You can describe your experience with words, but the person you are sharing your experience with hasn’t automatically made the experience himself. However, where machine learning models really excel is in the generalization of data.

2.1 Tasks

For instance, consider a computer program assigned with the task T of playing chess. Its performance P could be measured by the percentage of games it wins against its opponent. Training experience E is gained by playing practice games against itself. This example was not only chosen because I really enjoy the game of chess, rather because computers outperformed humans in this

task already more than two decades ago. In 1996 an IBM supercomputer called Deep Blue was the first machine to beat the then-reigning world chess champion Garry Kasparov in a game of chess. One year later in a rematch it won a 6-game match against Kasparov, being the first defeat of a reigning world chess champion by a computer under tournament conditions [11]. However, playing chess is just one of the possible tasks machine learning can master. It has become a common tool in almost any task that requires information extraction from large data sets [12].

In the following, common machine learning tasks are presented. Note that machine learning has been applied in far more applications, but to mention them all would go beyond the scope of this thesis. The tasks are separated into two main groups, supervised and unsupervised learning. However, there have been methods developed where both labeled and unlabeled data is used is called *semi-supervised learning*. There are different concepts, for example *minimum entropy regularization*, allowing the incorporation of unlabeled data into standard supervised learning.[13].

Reinforcement learning presents a different approach. Unlike supervised and unsupervised learning, reinforcement learning algorithms learn by interacting with an environment rather than a fix data set. In this paradigm, an agent learns to make decisions by taking actions in an environment and receiving rewards or penalties in return. The agent's objective is to learn a policy, which is a mapping from states to actions that maximizes the cumulative reward over time. This involves a balance between exploration, where the agent tries out new actions to gather information, and exploitation, where the agent makes the best decision based on current knowledge. It's particularly suitable for tasks where the optimal solution can only be found through trial-and-error and where delayed rewards are involved. Chess engines are a good example of reinforcement learning[14].

2.1.1 Supervised Learning

Machine learning tasks can differ significantly in the data they are provided with and in the way they are trained. The most straightforward and widely used tasks are based on *supervised learning*. Algorithms are exposed to a data set that includes not only features, but also associated labels or targets. Take the iris data set[15], a typical test set for many classification techniques, as an example. Each iris plant in the set is labeled with its species. A supervised learning algorithm can analyze this data set and learn how to categorize iris plants into three distinct species using their measurements. This task is called classification. Together with regression they form the most common tasks in supervised machine learning [8].

2.1.1.1 Classification

A classification task learns to map inputs to one of k categories. Usually the learning algorithm is asked to output a function [8]:

$$f : \mathbb{R}^n \to \{1, \ldots, k\}. \tag{2.1}$$

If $y = f(x)$, the input described by vector x creates a numeric code y that maps to a category. Other variants, for example, output probability distributions over classes, that describe how probable it is that an input belongs to each of the categories [8].

An important step for classification is *feature extraction*. Mostly this is done by a human, but it can also be done by an algorithm. Take the iris plant classification task mentioned in the beginning of this section for example. Features of the plants have already been extracted by botanists. In this case sepal length and width, and petal length and width [16].

In a more difficult problem of classifying images directly, the data is most likely not preprocessed by a human. The task might be to classify the image as a whole, e.g., does it contain a dog or not? This is called *image classification*. If the images contains hand-written numbers and letters to classify, its called handwriting recognition. Even harder problems deal with object detection or object localization in images. A special case of this is face detection for example. If a face is found in an image, one can then possibly estimate the identity of the person by face recognition [16].

2.1.1.2 Regression

In contrast to classification tasks, which assign an input to a specific class, regression tasks predict a numerical value when given some input. To accomplish this task, the learning algorithm is asked to produce a function [8]:

$$f : \mathbb{R}^n \to \mathbb{R}. \tag{2.2}$$

Regression tasks include the forecasting of time-series and can for example be used by insurance companies to predict the expected claim amount that an insured person will make and therefore set insurance premiums. Another example is the prediction of future prices of securities. [8].

2.1.2 Unsupervised Learning

In contrast to supervised learning stays *unsupervised learning*. This learning method is termed unsupervised because the input data doesn't come with predefined labels. Clustering is typically used to identify distinct groups within the data. For instance, consider an unsupervised learning technique that processes a collection of images depicting handwritten digits. It might identify 10 unique clusters in the data, which could potentially correspond to the 10 individual digits from 0 through 9. However, since the training data lacks labels, the model can't provide any semantic interpretation of the clusters it has identified [17].

2.1.2.1 Clustering

Clustering describes the process of grouping a set of objects into different groups so that similar objects end in the same group and dissimilar objects are separated into different groups[12].

A widely used clustering algorithm is *k-means clustering*. It directly decomposes a data set $\chi = \{x_1, x_2, \ldots, x_n\}$ into a set of k disjoint clusters $C = \{c_1, c_2, \ldots, c_k\}$. Each cluster has a representative, the *centroid*, which is computed as the mean vector of all objects assigned to the corresponding cluster. The goal of the clustering operation is to reduce the distortion between the data objects and the centroids. If distortion is measured using Euclidean distance, the objective is the minimization of the *sum-of-squared error* between the objects and the centroids, $\mu = \{\mu_1, \ldots, \mu_k\}$[18]:

$$SSE(C) = \sum_{c=1}^{k} \sum_{x_i \in C_c} \|x_i - \mu_c\|^2 \quad \text{where} \quad \mu_c = \frac{\sum_{x_i \in C_c} x_i}{|C_c|}. \tag{2.3}$$

A similar version of k-means clustering where an object is not bound to one cluster, but receives probabilities for each cluster is called *fuzzy clustering*. Other clustering methods include *hierarchical clustering*, *kernel clustering* and *spectral clustering* to name a few[18].

2.1.2.2 Anomaly Detection

Anomaly detection is the type of task, where the algorithm sifts through the data with the goal of finding patterns that do not confirm with the expected behaviour. Finding anomalous data can be of critical interest in various application domains, such as fraud detection for credit cards, health care, insurance, intrusion detection for cyber-security, and fault detection in safety critical systems[19].

2.2 Evaluation Metrics

To train a machine learning algorithm, its performance has to be evaluated. In practice this is done by choosing a *loss function* that measures the difference between the predicted and the actual output. The loss can then be minimized by adjusting the algorithms parameters. It is important to choose a loss function that is suitable for the task at hand. Some metrics for designing a loss function for classification and regression tasks are presented in the following[20].

2.2.1 Metrics for Classification

Let us assume we have a binary classification problem. There are four possible outcomes as shown in the *confusion matrix* in Table 2.1. If the predicted class is positive and the true class is also positive, we call it a *true positive* - TP. If the true class is negative, it is a *false positive* - FP.

In the case of a negative predicted class and a negative true class, we speak of a *true negative* - TN. If it would have been a negative predicted with a positive true class, it would be called a *false negative* - FN. Note that the false negative and false positive test results are not always equally as bad. Consider an authentication task, where a user wants to log into a private system by voice. A false negative, when an authorized person is denied access to the system, could potentially cause less harm, than a false positive, when an unauthorized person is granted access to the system[21].

		Predicted Class	
		positive	negative
Actual Class	positive	*true positive*	*false negative*
	negative	*false positive*	*true negative*

Table 2.1: A binary confusion matrix.

The concept of confusion matrices can be expanded in the case of $K > 2$ classes, to a $K \times K$ *class confusion matrix*. The test entries (i, j) contain the instances that belong to C_i, but were assigned to C_j. Table 2.2 shows a 5×5 confusion matrix. Like in the binary case, the off-diagonal entries are ideally 0[21].

		Predicted Class				
		1	2	3	4	5
Actual Class	1					
	2					
	3					
	4					
	5					

Table 2.2: A mutli-class confusion matrix with 5 different classes.

Different classification measures can be derived from the test results:

1. **Error**: The *error* is the number of total false classifications over the number of total classifications N. We can also say it is the rate of false classifications[21]:

$$error = (FP + FN)/n. \tag{2.4}$$

2. **Accuracy**: The proportion of total true classifications over the number of total classifications is the *accuracy*. A good classifier has an accuracy near 1[21]:

$$accuracy = (TP + TN)/n = 1 - error. \tag{2.5}$$

3. **Recall**: The *recall*, also known as the *sensitivity* or *true positive-rate*, is a measure of how many of the actual positives are classified right[21]:

$$recall = TP/(TP+FN). \tag{2.6}$$

4. **False positive-rate**: The *false positive rate* or short FP-rate is the rate at which a classifier predicts a positive class when it actually wasn't[21]:

$$FP\text{-}rate = FP/(FP+TN). \tag{2.7}$$

5. **Precision**: The *precision* measures the ratio of true positives to the total number of positive predictions. A model with high precision has fewer false positives and is therefore accurate in predicting positive classes[21]:

$$precision = TP/(TP+FP). \tag{2.8}$$

6. **Specificity**: The proportion of true negatives to the total number of negative predictions is called *specificity*. A classifier with high specificity has fewer false positives, which means it is more accurate in predicting negative classes[21]:

$$specificity = TN/(TN+FP) = 1 - FP\text{-}rate. \tag{2.9}$$

7. **Receiver operating characteristics**: Most classifying models calculate class probability scores before assigning an input to a class. Assume the classifier returns the probability $\hat{P}(C_1 \mid x)$ for a given input x to belong to the positive class C_1 and $\hat{P}(C_2 \mid x) = 1 - \hat{P}(C_1 \mid x)$ to belong to the negative class. We can classify to the positive class if $\hat{P}(C_1 \mid x) \geq \theta$. If we choose θ close to 1, we will rarely have false positives, but also a small number of true positives. By increasing θ to increase the number of true positives, the risk of introducing false positives also increases. If we evaluate the true positive and the false positive rate values for different values of θ, we get the *receiver operating characteristics* (ROC) curve[21] An exemplary ROC curve of the iris classification task is shown in Figure 2.1.
The ROC-curve allows a quick visual interpretation of the classifier. An ideal curve is close to the upper-left corner. If we want a single value for the performance, we can calculate the *area under the curve* (AUC). Ideally the value for AUC is 1[21].
8. **F-score**: The *F-score* in its origin formulation F_1 is the harmonic mean of precision and recall[23]:

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} = \frac{2TP}{2TP+FP+FN}. \tag{2.10}$$

A derivation of the F-measure has been developed by implementing the parameter β, which controls the balance between precision and recall[24]:

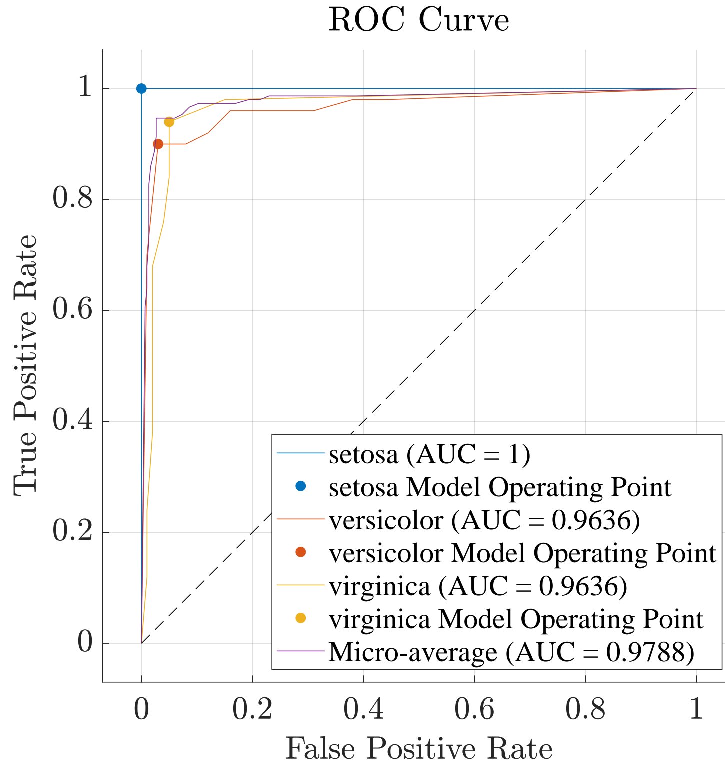


Fig. 2.1: The receiver operating characteristics (ROC) curve from a multiclass iris plant classifier. The area under the curve values for the classes can be seen in the legend. The model operating points are also plotted. Note the AUC score for the setosa class, indicating a perfect true positive and false positive-rate[22].

$$F_\beta = \frac{(\beta^2 + 1) \cdot precision \cdot recall}{\beta^2 \cdot precision + recall}. \tag{2.11}$$

When $\beta > 1$, the F-measure becomes more recall-oriented, if $\beta < 1$ more precision-oriented, for example $F_0 = precision$[25].

2.2.2 Metrics for Regression

In the following common regression loss measures are presented. All of them are based on functions of the residuals, the difference between the observed value y and the predicted value $\hat{y}$. The number of samples is denoted with N[26].

1. **Mean Absolute Error**: A computationally cheap loss function is the *mean absolute error* (MAE). It is not sensitive to outliers, which can be useful in some applications. MAE is

simply the aggregation of all absolute residuals[26]:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|. \tag{2.12}$$

2. **Mean Squared Error**: A loss that is more sensitive to outliers is *mean squared error* (MSE). The calculation is similar to MAE, but the residuals get squared before summation[26]:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{2.13}$$

3. **Mean Bias Error**: In contrast to MAE and MSE, the *mean bias error* (MBE) can also be negative. If so, it indicates that the predictions are underestimated, if positive the predictions are overestimated. It's important to be cautious because positive and negative biases can cancel each other out[27][26]:

$$MBE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i). \tag{2.14}$$

4. **Relative Absolute Error**: Relative errors are independent of scale, and can therefore compare models that are measured in various units. The *relative absolute error* (RAE) is calculated by dividing the absolute error by the difference between the mean of all actual values $\hat{y}_i$ and the actual value[26]:

$$RAE = \frac{\sum_{i=1}^{N}|y_i - \hat{y}_i|}{\sum_{i=1}^{N}|y_i - \overline{y_i}|} \quad \text{with} \quad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i. \tag{2.15}$$

5. **Relative Squared Error**: An error that is not affected by varying scale of the prediction targets is the *relative squared error* (RSE). It calculates as the sum of total squared errors over the sum of the actual values minus the mean of all actual values[26]:

$$RSE = \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y_i})^2} \quad \text{with} \quad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i. \tag{2.16}$$

6. **Mean Absolute Percentage Error**: A metric often used to assess the accuracy of a forecast system is the *mean absolute percentage error* (MAPE). It is calculated as the average of absolute residuals divided by the actual values as percentage[26]:

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{y_i} \quad \text{with} \quad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i. \tag{2.17}$$

7. **Root Mean Squared Error** (RMSE) Is useful as a loss function, when huge errors in the system want to be cancelled out. It is the root of the MSE[26]:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}. \quad (2.18)$$

2.3 Artificial Neurons

2.3.1 Perceptrons

McCulloch and Pitts laid the foundation for machine learning with Artificial Neural Networks (ANN) in 1943. They analysed how networks of neuron process information and how they can be coupled together to create logical functions[28]. The development of McCulloch and Pitts neuron models led to the invention of *perceptrons*. The term was coined by F. Rosenblatt in 1958. [29]

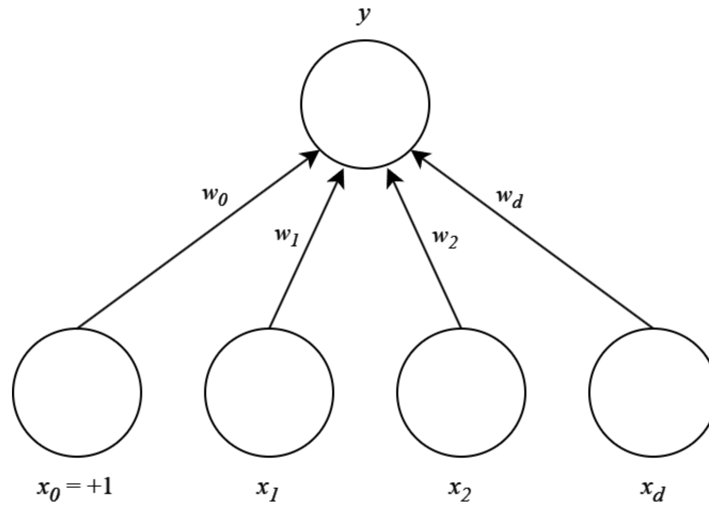


Fig. 2.2: The architecture of the perceptron.

Each perceptron has d inputs, x_i, with $i = \{1,\dots,d\}$. The inputs that are either coming from the environment or from the outputs of other perceptrons are weighted via a *connection weight*, $w_j \in \mathbb{R}$. In the simplest case the output y is a weighted sum of the inputs[21]:

$$y = \sum_{j=1}^{d} w_j x_j + w_0. \quad (2.19)$$

The concept of a *bias* is implemented here with the intercept value w_0. Augmented vectors, $w = [w_0, w_1, \dots, w_d]^T$ and $x = [1, x_1, \dots, x_d]^T$, are implemented to allow writing the output, y, of the perceptron as a dot product [21]:

$$y = w^T x. \quad (2.20)$$

When $d = 1$ we obviously get the equation of a line with slope w and w_0 as the intercept[21]:

$$y = w_1 x + w_0. \tag{2.21}$$

This means, that such a perceptron could be used to create a linear fit. If there are two inputs, we create a plane; if there are more than two inputs we create a hyper plane. This can be used to divide the input space into two half-spaces. To do so a binary step function, also called Heaviside step function is used[21]:

$$H(y) = \begin{cases} 1 & \text{if} \quad y \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.22}$$

If the function generates $s = 1$, the input belongs to class 1; otherwise it is assigned to class 2[21].

By using $K \geq 2$ outputs, there are K parallel perceptrons, each of them has a weight vector w_i. This model performs a linear transformation from a d-dimensional space to a K-dimensional space[21].

2.3.2 Gradient Descent

In 1986, Rumelhart et. al. demonstrated that parallel perceptrons could be trained using *gradient descent* when the error function is differentiable. This method involves iteratively adjusting the connection weights between neurons to minimize output errors[30]. The gradient of a differentiable convex function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is represented by $\nabla f(w) = \frac{\partial f(w)}{\partial w[1]x}, \ldots, \frac{\partial f(w)}{\partial w[d]x}$, which is the vector of partial derivatives of f. Gradient descent is an iterative optimization algorithm used to find the local minimum of a differentiable function. The algorithm starts with an initial value of w. At each iteration, we take a step in the negative direction of the gradient at the current point. The formula for an update step is [31]:

$$w^{(t+1)} = w^{(t)} - \eta \nabla f(w^{(t)}) \tag{2.23}$$

The *learning rate* η is a hyperparameter that defines how strong weights are adapted. If the learning rate is set too high, the algorithm tends to oscillate around the search space. In the other case of η being too low, we risk of remaining in a local optimum[21].

Learning often starts with randomly initialized weights. Iteration for iteration the weights adapt to minimize the error function. When the error function is not performing over the whole data set, but on instances of it, it is called *online learning*. A full traversal of the training set is called an *epoch*. A different training strategy is *batch learning*, where the weights are not updated after each instance, instead the data is divided into mini-batches that decide the iterative change of weights[21].

2.3.3 Activation Functions

The perceptron output function is always a binary step function. In modern neural networks different output functions, called *activation functions*, are mostly used. A perceptron with any other activation function than the binary step function is defined as *artificial neuron*. Activation functions act as a limiter of the amplitude of a neuron squashing the output into the range from 0 to +1 or -1 to +1[32]. To exploit the benefits of a deep neural network, non-linear activation functions have to be used. This is due to the fact that compositing linear functions always generates some new linear function. But a very complex non-linear function can be created by the composition of relatively simple non-linear functions. So stacking non-linear layers allows the model to make use of more complex features in the data and therefore enhance the power for its given task[32].

Let us introduce another very common used activation function, the *sigmoid function* σ with its typical "s-shape". An example of the sigmoid function is the *logistic function* shown in Figure 2.3a. Given an input x and the slope parameter a, it computes as follows[32]:

$$\sigma(x) = \frac{1}{1+e^{-ax}} = 1 - \sigma(-x). \tag{2.24}$$

Note its symmetry property at the end of the equation. When $a = 1$, we call it the *standard logistic function*. The derivative of it computes as follows[32]:

$$\frac{d}{dx}\sigma(x) = \frac{e^{x}}{(1+e^{-x})^2} = \sigma(x)(1-\sigma(x)). \tag{2.25}$$

The hyperbolic tangent function tanh also has a sigmoid shape (Figure 2.3b) and the same symmetry property as the standard logistic function:

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = 1 - \sigma(-x). \tag{2.26}$$

The derivative of the tanh function ranges from zero to one, making it very suitable for backpropagation by reducing the vanishing gradient problem. It is plotted in Figure 2.3b. It computes as one minus the tanh function squared:

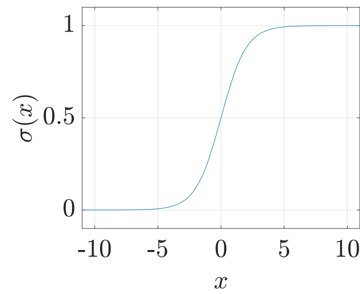
$$\frac{d}{dx}\tanh(x) = 1 - \frac{(e^{x}-e^{-x})^2}{(e^{x}+e^{-x})^2} = 1 - \tanh^2(x). \tag{2.27}$$

The ReLu function shown in Figure 2.3c assigns zero to all negative input values and outputs its input values for all other values:

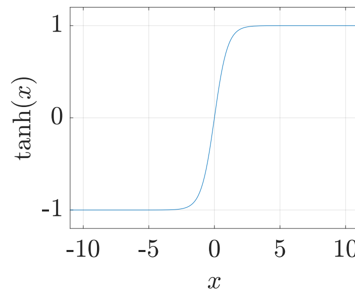
$$\mathrm{ReLu}(x) = \begin{cases} x & \text{if} \quad x \geq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{2.28}$$

The derivative of the ReLu function plotted in Figure 2.3f is the same as the Heaviside step function:

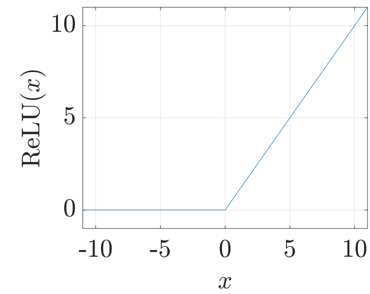
$$\frac{d}{dx}\mathrm{ReLu}(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{2.29}$$



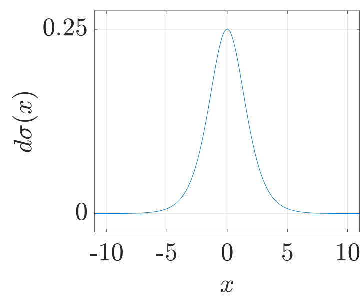
(a) The standard logistic function



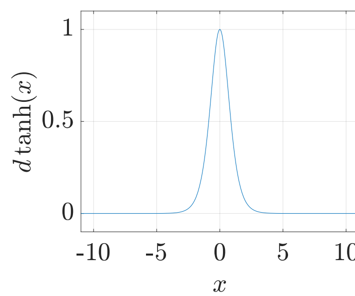
(b) The tanh function



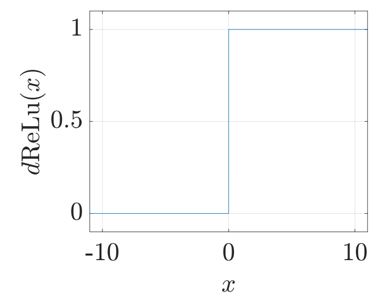
(c) The ReLu function



(d) The derivative of the standard logistic function



(e) The derivative of the tanh function



(f) The derivative of the ReLu function

Fig. 2.3: Different activation functions and its derivatives used in neural networks.

Chapter 3
Deep Learning

3.1 Artifical Neural Networks

Artificial Neural Networks (ANN) are machine learning models that are inspired by the architecture of the human brain. The networks use highly idealized neuron models, but the principle stays the same[33]. Many neurons in parallel are called a *layer*. A single layer of neurons can only model linear functions. This limitation can be overcome by stacking layers of neurons to create artificial neural networks[8].

3.1.1 Feedforward Networks

The name *feedforward network* (FFN) originates from the flow of information in the network. In contrast to *recurrent networks*, as presented in 3.1.3, where output from the model is fed back to itself, FFNs only pass information to forward layers[8][21]. A FFN is designed to approximate a function $\hat{y} = f^*(x;\delta)$. An input vector,

$$x = [x_1, x_2, \ldots, x_m]^T, \tag{3.1}$$

is mapped to a predicted output

$$\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{nl}]^T. \tag{3.2}$$

The network learns the optimal value of the parameters δ that result in the best function approximation. Learnable parameters of a FFN are the bias vectors,

$$b^i = [b^i_1, b^i_2, \ldots, b^i_{ni}]^T, \tag{3.3}$$

and the weight matrices,

$$W^i = \begin{bmatrix} w^i_{1,1} & \cdots & w^i_{1,n_{i-1}} \\ \vdots & \ddots & \vdots \\ w^i_{n_i,1} & \cdots & w^i_{n_i,n_{i-1}} \end{bmatrix}. \tag{3.4}$$

of all the neurons[8]. Figure 3.1 shows a *fully connected* FNN. This indicates that all nodes of a layer are connected to all nodes of the forward adjacent layer in the network. If there are missing connections the net is said to be *partially connected*[32].

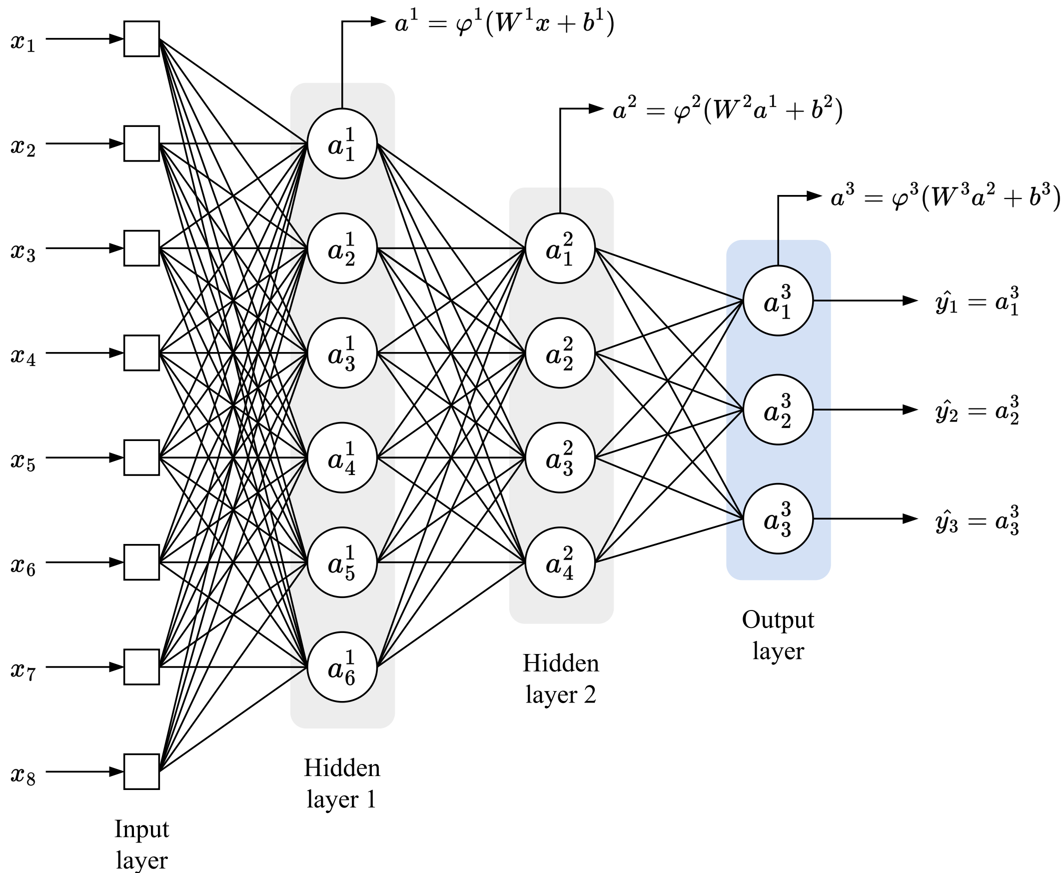


Fig. 3.1: An examplary graph of a fully connected feedforward neural network with two hidden layers and a width of six. The network processes an input vector x of dimension 8 and outputs a vector $\hat{y}$ of dimension 3. Adapted from[34].

The non-learnable parameters, the *hyperparameters*, define the structure and characteristics of a network and have to be defined before training. For example, the *depth* of the model is the number of layers of the FFN. A FFN has to have at least an input and an output layer. The layers inbetween are called *hidden layers*, because the training data does not show the desired outputs, the *hidden layer values*, for any of these layers. A neural network with at least one hidden layer is considered to be *deep*. Machine learning with deep networks is called *deep learning*. Finally, the *width* of the model is determined by the dimensionality of the hidden layers[8].

Training a FFN is similar to training a single perceptron. The biggest difference between the linear model of the perceptron and a FFN is that the loss-function becomes non-convex. This means that convergence is not guaranteed anymore[8].

3.1.2 Back-Propagation

We know that the input of hidden layer i is the output of hidden layer $i-1$. This results in the functions being chained together. We can evaluate the loss gradients of the individual weights and biases using the Leibniz chain rule. Given $f(x) = g(h(x))$, the derivative of $f(x)$ computes as:[30]

$$\frac{df(x)}{dx} = \frac{dh(x)}{dx}\frac{dg(z)}{dz}|_{z=g(x)} \tag{3.5}$$

It is easy to calculate the derivatives of the error for the final layers, and once they are estimated, the error derivatives for the previous layers can also be calculated. By going from end to beginning of the network, gradients are calculated layer by layer in a process called *back-propagation*[30]. When we use the network to predict an output y from an input x it is called forward propagation[8].

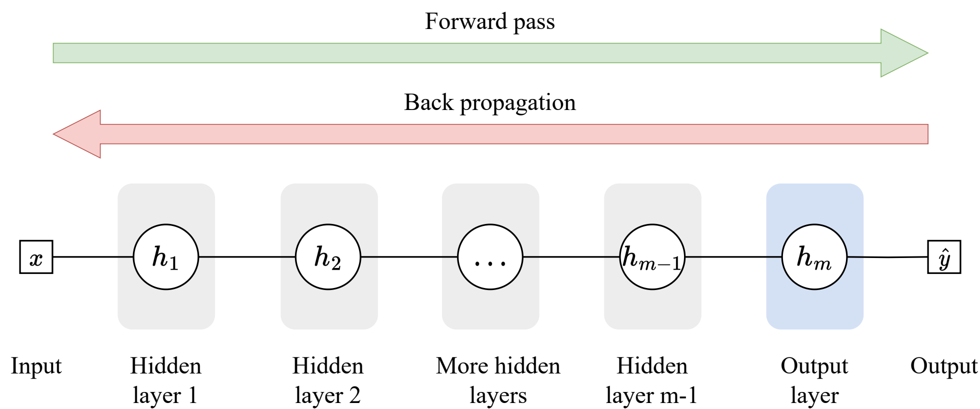


Fig. 3.2: A thin feedforward network with one neuron in each layer.

In order to better understand the back-propagation algorithm and its difficulties, consider the network depicted in Figure 3.2, a very deep FFN with a single neuron in each layer. Assume that there are m hidden layers and 1 non-computational input layer. The weights for the edges of the corresponding layers are represented by $w_1, w_2, \dots w_m$. Let x be the input; $h_1, h_2, \dots h_m$ be the hidden values and y be the final output. As an activation function ϕ we choose the sigmoid function σ to be applied in each hidden layer. We define $\phi'(h_t)$ to be the derivative of the activation function in hidden layer t. With a loss function L; $\frac{\partial L}{\partial h_t}$ becomes the derivative of the loss function with respect to the hidden activation h_t. We now see that the output of each hidden layer is defined from the previous layer as follows[35]:

$$h_{t+1} = \phi(w_{t+1}h_t) \tag{3.6}$$

Now we can formulate[35]:

$$\frac{\partial h_{t+1}}{\partial h_t} = \phi'(w_{t+1}h_t)w_{t+1} \tag{3.7}$$

Finally, applying the chain rule gives us[35]:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} = \phi'(w_{t+1}h_t)w_{t+1}\frac{\partial L}{\partial h_{t+1}} \tag{3.8}$$

From equation 2.25 and Figure 2.3d we can see that the derivative of a sigmoid takes on its maximum value 0.25 at $\sigma = 0.5$, resulting in $\phi'(w_{t+1}h_t)$ to be less or equal than 0.25. Assuming that the weights are initialized from a standard normal distribution, which is common practice, the expected average magnitude of each w_t is 1. Therefore, we can assume that typically the value of $\frac{\partial L}{\partial h_t}$ is less than 0.25 that of $\frac{\partial L}{\partial h_{t+1}}$. After propagating by r layers, this value will typically be smaller than 0.25^r. If we set $r = 8$, the gradient update magnitudes drop to 10^-5 of their original values. This means that the earlier layers will get very small weight updates compared to the later ones. This phenomena is called the *vanishing gradient problem*. When you use a function with a bigger derivative, the gradients can explode. [35]:

3.1.3 Recurrent Networks

As mentioned before, *Recurrent neural networks* (RNNs) differ from FFNs by allowing feedback loops. This has a far-reaching impact on the sequence learning capability of such neural networks. Instead of just mapping from input to output vector it can map from a history of previous inputs to each output. The quintessence is that the recurrent connections act as a kind of memory of previous inputs in the *internal states* of the network. This allows the handling of sequenced data like real-valued *time-series* or symbolic text.[36].

A time-series has an ordered value sequence. The information in the time series is completely distorted when the temporal order of values is changed. A crucial observation is that a time series' value at time t bears a strong correlation to its values from the previous window.[35]

Lets focus on the architecture. Figure 3.3 shows the simplest recurrent network possible, a network with a single self-recurrent unit performing self-feedback[32].

A sequence $\chi = \{x(1),\ldots,x(\tau)\}$ of input vectors $x(t)$, each associated with an indexing variable $t = \{1,\ldots,\tau\}$[8],

$$x(t) = [x_1(t), x_2(t), \ldots, x_m(t)]^T, \tag{3.9}$$

produces a corresponding sequence $\hat{\Upsilon} = \{\hat{y}(1),\ldots,\hat{y}(\lambda)\}$ of output vectors, indexed by $t = \{1,\ldots,\lambda\}$[8],

$$\hat{y}(t) = [\hat{y}_1(t), \hat{y}_2(t), \ldots, \hat{y}_n(t)]^T. \tag{3.10}$$

Note that the length of the input and the output vector don't have to be necessarily the same. The output can be smaller as well as greater in dimensionality. Furthermore, many recurrent networks are able to process varying sequence lengths. The neurons feed back its output to the input of itself. This changes the *hidden states*[8],

$$h^i(t) = [h_1^i(t), h_2^i(t), \ldots, h_k^i(t),]^T, \tag{3.11}$$

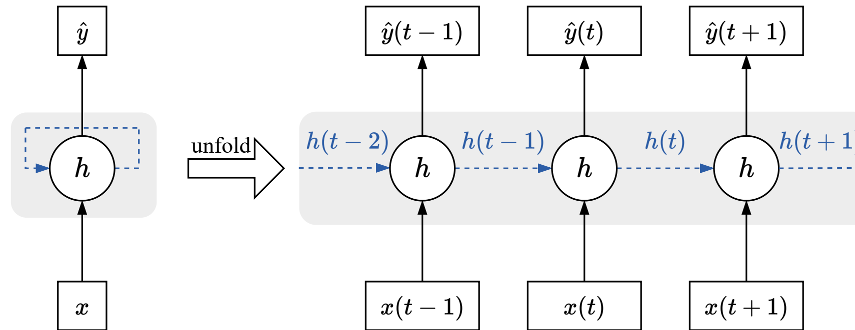


Fig. 3.3: Unfolding a recurrent network with one layer and a single recurrent unit. The recurrent unit stores information from the previous time step in a hidden state vector h. With every time step of the input vector the hidden state vector gets updated.

of the neurons of layer i after each item of a sequence is propagated through the network. This hidden state vector is a function of the current input $x^i(t)$ of layer i and the hidden state vector of the previous time step $h^i(t-1)$[8]:

$$h^i(t) = \phi_h^i(W_h^i x^i(t) + R_h^i h^i(t-1) + b_h^i). \tag{3.12}$$

Additionally to the bias vector b_h^i and a weight matrix W_h^i for the input layer, a recurrent weight matrix R_h^i for the hidden state $h^i(t-1)$ is introduced. Finally, to obtain the outputs an activation function, e.g. hyperbolic tangent, is applied to the weighted sum[8]:

$$\hat{y}(t) = \phi_y^i(R_y^i h^i(t) + b_y^i). \tag{3.13}$$

Training of RNNs is done with a process called back-propagation through time (BPTT). It is basically the same concept as in feed-forward networks. But some difficulties arise when dealing with complex recurrent network models. First, the temporal layering of the sequences is input dependent, which makes long sequences very hard to train. Second, the problems of exploding or vanishing gradients even gets worse with very deep neural networks, which limits their effectiveness of long-term relationships.[35]

3.1.3.1 Long short-term memory

A special type of recurrent network that tackles this problems is the long short-term memory (LSTM) network proposed by Sepp Hochreiter and Jürgen Schmidhuber in 1997 [37]. They introduced the cell state, that acts as a memory, storing information about past time steps. It is updated via gate units that decide which information is important to keep. Figure 3.4 shows a widely used improved architecture of a single LSTM-cell. It uses an input, a forget, a candidate and an output

gate to control the information flow into the and out of the cell. This allows to bridge long connections and model long term dependencies, while also minimizing the risk for vanishing or exploding gradients. There have been plenty of adaptions from the basic LSTM architecture developed. For

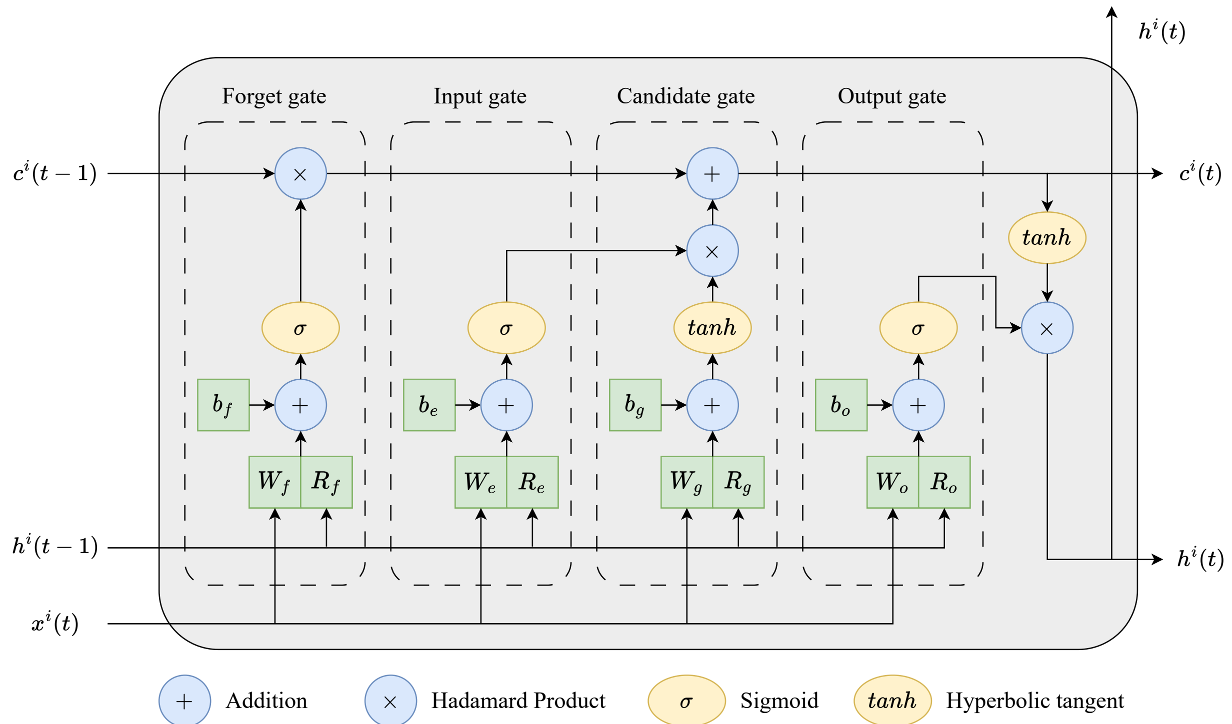


Fig. 3.4: A single LSTM-cell with a forget, input, candidate and output gate.

example, the bidirectional LSTM, or biLSTM, is a model that consist of two LSTMs. One working in the forward direction, and the other in the backward direction. This allows the model to improve its context of the sequence by knowing what items immediately follow and precede an item in a sequence.

3.2 Transformers

Before the introduction of the *transformer* in 2017 by Vaswani et al., RNNs used to be the state-of-the-art architecture for neural machine translation and language related machine learning tasks. Modern processing hardware such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) rely on parallel computing. The sequential computing nature of RNNs precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Transformer architectures inherently compute in parallel, because they take the whole sequence at once[4].

3.2.1 Vanilla Architecture

The original (vanilla) model shown in Figure 3.5 consists of an encoder and a decoder, each composed of a stack of 6 identical layers. The encoder maps an input sequence of symbol representations $(x_1,...,x_n =$ to a sequence of continuous representations $z = (z_1,...,z_n)$. The decoder receives z and generates an output sequence $(y_1,...,y_m)$ of symbols one element at a time. The model consumes the previously generated symbols as additional input when generating the next in an auto-regressive manner[4].

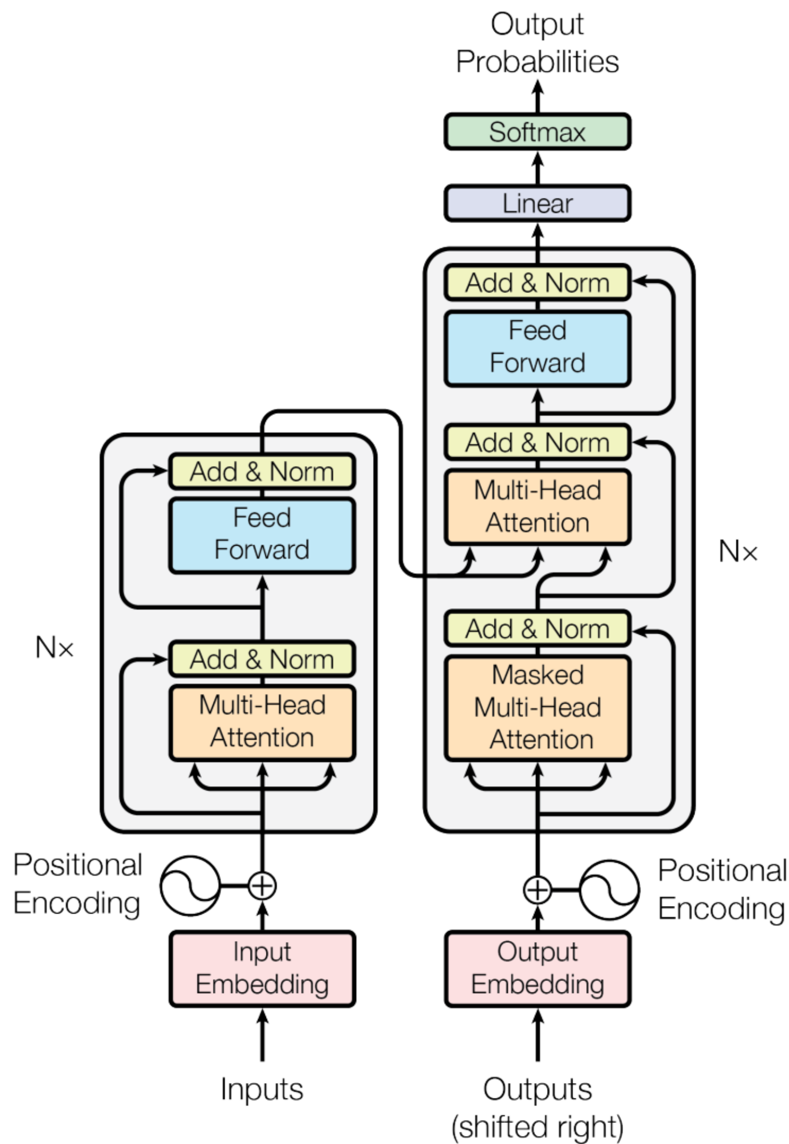


Fig. 3.5: The vanilla transformer model architecture[4].

Each layer in the encoder has two sub-layers. The first is a multi-head attention mechanism and the second a fully connected feed-forward network. Both have residual connections around each of

the two sub-layers, followed by a layer normalization. In the proposed architecture all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 512$. The decoder has three sub-layers, including a sub-layer that performs multi-head attention over the output of the encoder stack. The decoder sub-layers also have residual connections followed by a layer normalization. Additionally the self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions. This is done by masking and offsetting the output embeddings by one position[4].

3.2.2 Attention

The attention mechanism has become a key part of competitive natural language modeling tasks, allowing the modeling of dependencies regardless of their distance in the input or output sequence. In general it calculates weight matrices for each embedding in the context window. It can do so either in parallel in transformer architectures or sequentially in recurrent neural networks[38].

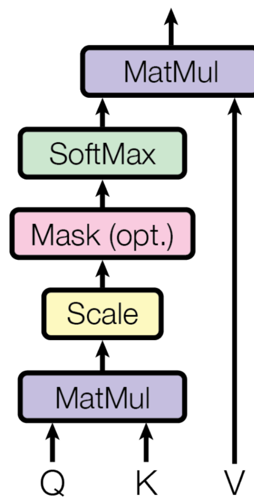


Fig. 3.6: Scaled Dot-Product Attention[4].

3.2.2.1 Scaled Dot-Product Attention

The transformer architecture uses scaled dot-product attention units as building blocks. The attention function takes three inputs: query Q, key K and value V. First, the dot-product of the queries and keys is calculated. The result is scaled by the square root of the number of key dimensions, d_k, producing the attention scores. They are then fed into a softmax function, obtaining a set of attention weights. Finally, the weights are used to scale the values in a weighted multiplication operation[4]:

$$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V. \tag{3.14}$$

3.2.2.2 Multi-Head Attention

The transformer repeats its attention computations multiple times in parallel according to the number of heads h. The independent attention outputs are then concatenated and linearly projected into the expected dimension[4]:

$$\begin{aligned} \text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,\ldots,\text{head}_h)W^O \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3.15}$$

And where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_K}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_V}$ and , $W^O \in \mathbb{R}^{hd_v \times d_{model}}$[4].

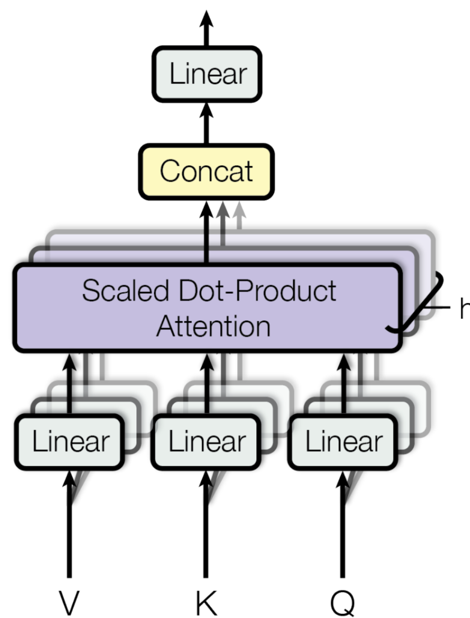


Fig. 3.7: Multi-Head Attention consists of several attention layers running in parallel[4].

3.2.2.3 Positional Encoding

Since the transformer model has no recurrence nor convolution, a positional information of the tokens in the sequence has to be injected. In the original paper *positional encodings* are added to the input embeddings at the bottoms of the encoder and decoder stacks. The dimensions of the input embedding and the positional encodings have to be equal in order to be sum up[4]:

$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}} \tag{3.16}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}} \tag{3.17}$$

Chapter 4
Natural Language Processing

Natural language processing (NLP) is an interdisciplinary sub-field of computer science and linguistics. It is primarily concerned with giving computers the ability to manipulate unstructured, natural language data. A classic approach when dealing with text or speech corpora is the use of bag-of-words representations and word n-gram models. These methods are based on counting frequencies of occurrences of short symbol sequences of length up to N. However, these approaches seem to be rather limited and not capable of dealing with state-of-the-art language tasks, such as document summarization or machine translation, the conversion of a word, text, document or corpus from one into another language. Another task could be classifying an article into different topics or analyse the sentiment of a document. We humans are good users of language in terms of producing and understanding language. However, it turns out that we have serious problems in describing the rules and formalities that govern language[39].

4.1 Brief Linguistics

Linguistics is the scientific study of language and its structure. It is divided into several subfields, each of which focuses on a specific aspect of language. These sub-fields include phonetics, phonology, morphology, syntax, semantics, and pragmatics. Phonetics is the study of the physical properties of speech sounds. It deals with the production, transmission, and perception of sounds in human language. Phonology, on the other hand, is concerned with the abstract representation of sounds in a language. It examines how sounds are organized and used in a particular language. Morphology is the study of the internal structure of words and how they are formed. It deals with the smallest units of meaning in a language, called morphemes. Syntax is the study of sentence structure and how words are combined to form phrases and sentences. Semantics is the study of meaning in language. It examines how words and sentences convey meaning and how meaning can be interpreted in different contexts. Pragmatics is concerned with how people use language in context to achieve their goals. It examines how context affects meaning and how people use language to communicate effectively[40].

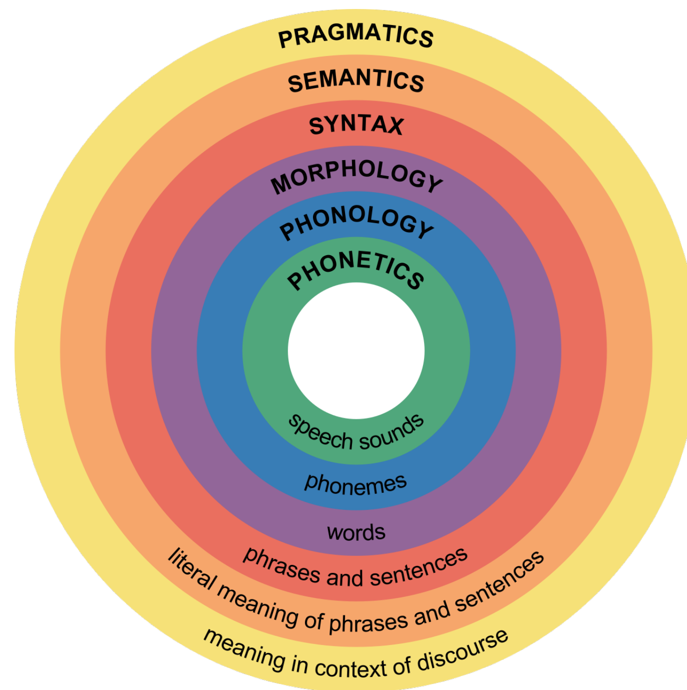


Fig. 4.1: Major levels of linguistic structure[41].

4.1.1 Challenges in Language Processing

English language is symbolic and discrete. The letters as well as the words are discrete symbols. Consider the two words ”elephant” and ”giraffe”. In our mind we know that there are certain relationships of these words. Both being animals for example. But this information can never be derived from the symbols themselves. Language is compositional. The building blocks of English language are characters that form words, and words form phrases and sentences. The meaning of a phrase can be bigger than the words that compromise it. Language is changing and evolving. There will be words in the future that don't exist today, for example, slang words, technical terms or particular names of persons or corporations. Most languages are also highly ambiguous and variable. Words can be synthesized in an enormous number of different ways as well as meaning can be conveyed in seemingly endless possible ways. Some of these properties lead to data sparseness in NLP tasks[39].

The nature of language makes natural language processing very challenging. NLP models often require features which can describe and generalize its textual data. The mapping from textual data to real valued vectors is called feature extraction and is done by a feature extraction function. Examining the linguistic structure can be beneficial in exploiting better features and improve model results[39]. Some selected textual features are presented in subsection 4.1.3.

4.1.2 Lexical Resources

To perform most NLP tasks usually huge textual data is needed for training to create reasonable and grammatically correct models. Although it is beneficial to use data from the same domain as working in, pre-training can be performed using a more general data set. Lexical resources in the form of machine-readable text, dictionaries or thesaurus provide a good starting point. They can contain linking to other words or provide additional information. Selected resources are mentioned in the following[1].

The Brown Corpus is one of the most widely known corpora in English language. It is an electronic collection of about one million words published in 1967 at Brown University. Every single word is labeled with a Part-of-Speech (POS) tag (article, adverb, preposition, determiner, etc.). It is balanced in the way that an attempt was made to create a representative corpus of American English[42]. The British pendant is the Lancaster-Oslo-Bergen corpus[1].

WordNet can be interpreted as a combination of a dictionary and a thesaurus, a ”synonym dictionary”. It links words into semantic relations including synonyms, meronyms and hyponyms. There is a hierarchy of organization of words. Each node consists of a synset of words with identical or similar meanings. WordNet is free and can be accessed or downloaded from the internet[43]. Googles NLP model, ”BERT”, was pretrained on the BooksCorpus with 800 million words and English Wikipedia, excluding lists, tables and headers with 2,500 million words[6].

4.1.3 Features for Textual Data

Extracting features from textual data is a key concept in NLP. The features can sometimes be used for machine learning models, but play a more significant role in statistical NLP models. A vast number of features has been explored and documented in literature. Finding suitable features mainly depends on the NLP task. Take for example a classification task of news articles. A very basic approach is finding the words with the highest frequencies in the article. These could be linked to topics, allowing classification without having to read the whole text. We can also just count the length of sentences, or count words with specific affixes. The ratio of short to long words in a document. The position of a word in a document or sentence, e.g. within the first or last n words. Or the context it often appears in, for example find words that often surround a specific word. Or the mean distance of a word pair and common words between them. Note that most features can both be applied to words or tokens[1].

Features can also require a lot of syntactic and semantic understanding. A good example for this is part-of-speech tagging. The ambiguity of language often requires to take the context of a word in a sentence or document into account. For example, the word ”book” can be used as a noun or a verb. But it can get even more ambiguous. The word ”well” can be used as an adjective, adverb, interjection and as noun[1]:

I was ill last week, but now I am *well* again.
She slept *well* in the comfortable bed.
Well, the strategy was less effective than we hoped.
The village sources its drinking water from a *well*.

POS is an important concept in NLP, because it allows a better understanding of the actual meaning of the words used. The main approaches are rule-based or stochastic[1].

4.2 Tokenization of Textual Data

One of the first processing steps of a NLP task is usually the tokenization of the textual data. That involves breaking down a piece of text into smaller units called tokens. These tokens are then usually assigned an unique ID, that is used for further handling[39]. There are different approaches of tokenization techniques available in NLP presented in the following.

4.2.1 Word-Based Tokenization

The most common technique used is *word-based tokenization*, splitting a text into words based on white-space or punctuation. This sounds trivial in English language, but in other languages, things can become much trickier. For example, in Arabic and Hebrew some words attach to the next one without white-spaces, and in Chinese there are no white-spaces at all. However, also in English language this is not straightforward, as there are some special cases we need to handle. Take abbreviations (I.B.M.) and titles (Dr.) for example. These probably doesn't need to be split. Also the word ”don't”, isn't it actually two words, ”do” and ”not”? More interestingly take words like ”North Africa”, ”New York” or ”ice cream”. It would probably make sense to create one token for those, but a white-space based algorithm will split them into two. We also have to define how to deal with upper and lower case. We could convert the whole text to lower-case beforehand probably on the cost of losing some information[39].

In NLP systems, there is a trade-off between the vocabulary size and the system's performance. Most systems only cover the most frequently used words and leave out rare ones. The words that don't make it into the vocabulary are usually called *out-of-vocabulary* (OOV) and are assigned a unique token, ”UNK”. This saves computational costs, but worsens the tasks performance, because the information for the word assigned an OOV token is completely lost[44].

4.2.2 Character-Based Tokenization

A different approach that doesn't really have to deal with these problems is *character-based tokenization*. The raw text is split into its individual characters. The logic behind it is that languages usually have way more words in their vocabulary than characters to represent them. There is no definite number of words in the english vocabulary. The number also highly depends on how to count and which words to include or not. Often only word families are counted, but there is no standard testing method for calculating the vocabulary size of a language. According to a study by Trefers-Daller and Milton, the vocabulary size of monolingual English speakers varies considerably, with an average of about 10,000 English word families for university entrants[45]. Don't foget that this number will significantly increase in a text corpus written from multiple people of different ages and cultures. Compare that to the size of 256 different characters (letters, numbers, special characters) in the English language. Reducing the vocabulary size by using characters has also a huge trade-off with the sequence length. Splitting the relatively short word ”neuron” into its characters will create six tokens instead of one[40].

4.2.3 Subword-Based Tokenization

A solution between word and character-based tokenization is *subword-based tokenization*. A common principle is to keep frequently used words, and split rare words into smaller meaningful subwords. We could split the word ”undo” into its prefix ”un” and its stem ”do”. Or the word ”unboxing” into its prefix ”un”, stem ”look” and suffix ”ing”. We can see that the vocabulary size will shrink in this way, because some affixes and stems will be used in the same syntactic situations. Another benefit of this method is that words that were not trained before can possibly be represented by sub-words. Usually sub-word algorithms add special symbols, e.g. ”##”, to the start of the tokens when words are split, to indicate which is the start (”un”) and which token is the completion of the word (”##do”)[40].

The large language model GPT-2 uses a subword tokenization algorithm called Byte-level BPE.[46] It is a modification of the byte pair encoding (BPE) algorithm first described in 1994. The algorithm compresses data by finding the most frequently used pairs of adjacent bytes and replaces all instances of the pair with a byte that was not in the original data. It does so till the defined token limit size is reached[47]. Here is a basic example of BPE applied to a string:

1. The input is ”aaabdaaabacaa”.
2. The byte pair ”aa” occurs most often and is replaced with a new byte ”Z”.
3. The converted string is now ”ZabdZabacZ”.
4. Then the process is repeated with byte pair ”ab”, replacing it with ”Y”.
5. We now get ”ZYdZYacZ”.

6. The process could continue with recursive byte pair encoding, replacing ”ZY” with ”X”.
7. This would result in ”XdXacZ”.

Byte-level BPE is a variation of the original BPE algorithm that uses bytes as the basic unit of segmentation instead of characters. GPT-2 has a vocabulary size of 50,257, which corresponds to the 256 bytes of base character tokens, a special end-of-text token and the symbols learned with 50,000 merges[46]. Googles NLP model, ”BERT”, has a vocabulary size of 30,000 tokens using the subword-based tokenization algorithm wordpiece. In contrast to BPE, it does not merge the most frequent symbol pair, but the one that maximizes the likelihood of the training data once added to the vocabulary[6][48].

4.3 N-Gram Language Models

N-gram language models are purely statistical and are based on assigning a probability to each possible next word. A n-gram in NLP is a series of n adjacent words or tokens. Other domains, for example biology uses n-grams for adjacent base pairs extracted from genomes. A n-gram of sequence length 1 is called a unigram. If $n = 2$ we refer to bigrams, if $n = 3$ trigrams and so on. Word n-gram language models are based on the idea that the probability of a word occuring in a text depends on the previous $n-1$ words. Obviously, if $n = 1$ then $n-1 = 0$, meaning there are no preceding words. However, we can still gain information on the texts from these models called bag-of-words models.

4.3.1 Bag-of-Words

A *bag-of-words* is a model of text represented as an unordered collection of words. It involves two things, a vocabulary of known words and a measure of the presence of known words. The measures are the counts of each word or the frequencies of each word out of all the words in the document. So basically it is a histogram of words within a text. Let us see how the bag-of-words looks like for an example sentence[1]:

The lion chased the deer.

It can be combined with a weighting scheme called term frequency-inverse document frequency to extract more information.

Sentence 1	#
the	2
deer	1
lion	1
chased	1

Table 4.1: Bag-of-words for the sentence, "The lion chased the deer."[49].

4.3.2 Term Frequency-Inverse Document frequency

The *term frequency-inverse document frequency* is a measure of importance of a word to a document in a set of documents. It actually consists of two measures multiplied, the term frequency TF and its inverse document frequency IDF. The term frequency TF of term t within document d is computed as[2]:

$$TF(t,d) = \frac{n_t(t,d)}{n(d)}, \tag{4.1}$$

whereas $n_t(t,d)$ is the number of times that term t occurs in document d. And n is the total number of terms in document d. It is a relative measure of how often a term occurs in a document. The inverse document frequency IDF is a measure of how significant a term t is to a document d, or more precise how often it is used among all documents D[2]:

$$IDF(t,D) = \log_{10} \frac{N(D)}{N_d(t,D)}, \tag{4.2}$$

with $N_d(t,D)$, the number of documents where the term t occurs, sometimes called document frequency DF, and $N(D)$, the total number of documents in the collection. There are different variants of scaling in literature. A common is applying a log-function to the IDF. Combining both measures we can calculate the term frequency-inverse document frequency for a term t in document d from the document collection D as[2]:

$$TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D). \tag{4.3}$$

Summarizing, if a TFIDF is high for a specific term in a specific document of a collection it is important and could represent the document. This can come in very handy in information retrieval or keyword extraction, as well as classification tasks. Let us see how this works in practice[49].

A word like "the" is very common in English text and will likely have a high term frequency for any document. Because of that, its inverse document frequency is going to be very low compared to other words, resulting in a relative low TFIDF measure. Now lets say we examine a corpus of 37 Shakespeare plays. The term Romeo will only occur in one of those, whereas good and sweet

occurs in all 37 of them. So the term Romeo is gonna be very discriminative for the Romeo and Juliet play[49].

Word	DF	IDF
Romeo	1	1.57
salad	2	1.27
Falstaff	4	0.967
forest	12	0.489
battle	21	0.246
wit	34	0.037
fool	36	0.012
good	37	0
sweet	37	0

Table 4.2: Document frequency and inverse document frequency for selected words of a corpus of 37 Shakespeare plays[49].

4.3.3 Bag-of-n-grams

The *bag-of-n-grams* representation allows to save some information of the sequential ordering of words in a text. At least to a window size of n. This can help to gain deeper semantic insights and improve the model accuracy. Let's examine the two following sentences[35]:

1. The lion chased the deer.
2. The deer chased the lion.

The bag-of-words representation would look exactly the same for both sentences. However, both of them convey a completely different information and the second sentence is considered rather unusual. But if we allow to store n-grams with $n \geq 2$ we can see that the representations start to differ. The bag-of-bigrams are visualized in table 4.3. The order of the bigrams is messed up to show, that these model don't save any information of the original position in the text[35].

4.4 Neural Language Models

Till 2003 n-gram models were considered state-of-the-art for language modeling. This changed with the publication of a neural probabilistic language model from Y. Bengio et al. The shear advantage of neural model lies in the learning of distributed representations for words, which allow

Sentence 1	#	Sentence 2	#
the lion	1	the lion	1
the deer	1	the deer	1
chased the	1	chased the	1
lion chased	1	deer chased	1

Table 4.3: Bag-of-bigrams for the sentences, ”The lion chased the deer.”, and ”The deer chased the lion.”[49].

generalization. This means a sequence that has never been seen before can get a high probability if it is made of words that are similar to words forming an already seen sentence. The team used a RNN in their model[3]. Further development of RNN architectures lead to the invention of LSTMs (see subsection 3.1.3.1 for details), which are able to capture the dependencies of longer sequences compared to basic RNN models[37]. The combination of LSTMs with the word2vec word embedding algorithm published in 2013 came to dominate tasks like part-of-speech tagging, named entity recognition and a few more. In short, Word2vec can represent semantic similarities among words and map them to a vector space of typically several hundred dimensions with the use of a feed-forward network. It is described in detail in subsection 4.4.1.2[50]. The use of LSTMs as encoder-decoder model raised the bar again for many common NLP tasks[51]. Today, a mechanism called attention (subsection 3.2.2) is used for handling long-term dependencies. It was first developed by Graves in the context of handwriting recognition[52]. But soon later it was applied to machine translation with astonishing results[38]. Finally, by extending the idea of attention to self-attention and completely discarding recurrent connections the transformer was developed[4]. As of today it is considered to be the state-of-the-art architecture for several NLP tasks, such as machine translation, document summarization and question answering to name a few[49].

4.4.1 Word Embedding

In NLP word embedding is a technique for representing words as vectors. There are different approaches to get word embeddings and we will not cover all of them, but focus on one very popular called Word2Vec in subsection 4.4.1.2. But before applying Word2Vec one usually has to one-hot encode the tokens. This process is covered in the following subsection. The ultimate goal of word embeddings is to create vector representations that allow similar words to have similar representations[49].

4.4.1.1 One-Hot Encoding

Neural networks need numerical vectors as input. *One-hot encoding* is a technique used in ML to convert categorical data into numerical data. In NLP it is used to represent text data or tokens in a numerical form using a one-hot vector. The vector is binary and has size $1 \times N$, whereas N is the vocabulary size. Only one unique row is 1 for each token and the rest is 0. With large vocabularies, one-hot vectors tend to have a huge number of dimensions, which can take a lot of space. Also transfer learning with a different vocabulary is a bit difficult to implement. It is considered a sparse encoding.

4.4.1.2 Word2Vec

Word2Vec is among the most popular algorithms for word embedding in NLP. In contrast to one-hot it is a dense encoding. It is used to create a numerical representation from words or tokens. Word2Vec is able to capture semantics and relationships among words. We can demonstrate the relationship of words in a simple example. The relationship from Italy to Rome is similar to the one from France to Paris. A good representation can result in[49]:

$vec(\text{Italy}) - vec(\text{Rome}) + vec(\text{Paris}) \approx vec(\text{France})$

The original paper from T. Mikolov published in 2013 mentions two approaches for Word2Vec. The continuous bag-of-words (CBOW) and the skip-gram architecture. The CBOW method is based on predicting a center word from the surrounding context with context width m in both directions. This can be achieved by creating a very simple neural network with an input layer, a projection layer and an output layer. We first generate our one-hot encoded vectors for the input context window. These are then averaged in the projection layer. This means the position in the context does not make a difference. The projection layer has the size $N \times D$, whereas N is the vocabulary size and D is the embedding dimension. The projection layer has similarities to a lookup table and the dimension D is the number of features in the embedded vector. The model tries to predict the target words by outputting a vector of the size of our one-hot vectors. To create entries of the predicted vector in the range from 0 to 1 and set it's magnitude to 1. We calculate the probabilities of each word by a softmax function[50]:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4.4}$$

for $i = 1, \ldots, K$ and $z = (z_1, \ldots, z_K) \in \mathbb{R}^K$, a function often used as the last activation function for normalization of the output. It is a generalization to multiple dimensions of the logistic function defined in equation 2.24[39].

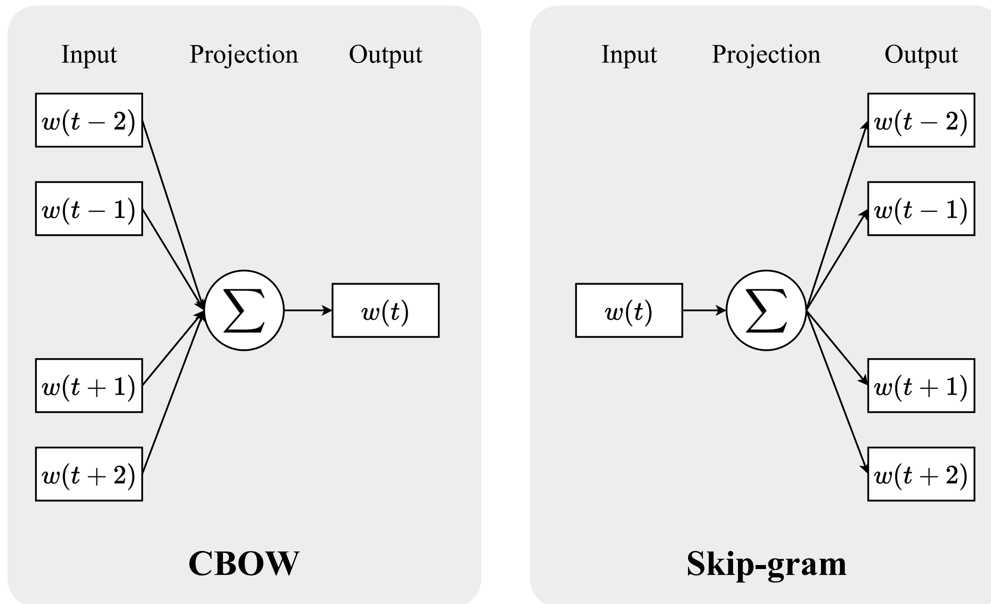


Fig. 4.2: Continuous-bag-of-words and Skip-gram models for Word2Vec learning[49].

The skip-gram method does kind of the inverted operation, given a word, try to predict the neighbouring words in the defined window size. It is slower in computation, but also performs well on words that are rare in the training corpus[50].

Due to the very long vector representation of one-hot encoding, the weights matrices tend to get very big and training tends to be very slow. To address these issues the authors of Word2Vec presented two innovations in their second paper. The first is subsampling frequent words to decrease the number of training examples. For example a word like ”the” is going to occur very often in any English corpus, therefore we can just delete a bunch of those from the corpus at random positions. The probability for a word w_i in the training set to be discarded is calculated as[53]:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}, \tag{4.5}$$

where $f(w_i)$ is the frequency of word w_i and t is a chosen threshold, typically 10^{-5}[53].

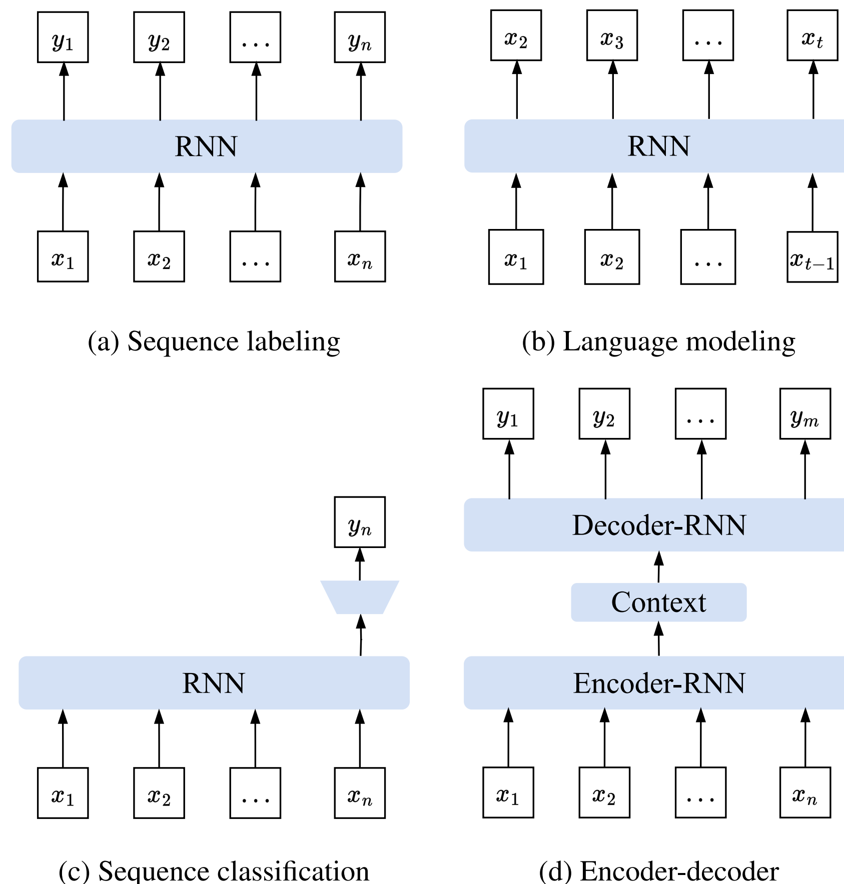
The second improvement presented is negative sampling. It aims to reduce the number of neuron weight updating operations to reduce training time and have a better prediction result. Instead of updating the whole corpus’ neuron weights, just the ones from the context window and the target are updated[53].

4.4.2 RNN Models

In simple RNNs sequences are processed one element at a time, with the output of each neural unit at a time t based both on the current input at t and the hidden layer from time $t-1$. The basic prin-

ciples were covered in subsection 3.1.3, we will only cover some applications and the architectures used to achieve these tasks. Figure 4.3 shows four RNN architectures used for different NLP tasks. Common language-based applications for RNNs include[49]:

1. Probabilistic language modeling: assigning a probability to a sequence, or to the next element of a sequence given the preceding words. The process is depicted in Figure 4.3b.
2. Auto-regressive generation using a trained language model.
3. Sequence labeling like part-of-speech tagging, where each element of a sequence is assigned a label. See Figure 4.3a.
4. Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification. Shown in Figure 4.3c.
5. Encoder-decoder architectures, where an input is mapped to an output of different length or alignment. Architecture depicted in Figure 4.3d.



(a) Sequence labeling (b) Language modeling

(c) Sequence classification (d) Encoder-decoder

Fig. 4.3: Four RNN architectures for NLP tasks[49].

4.4.3 Large Language Models

Large Language Models (LLMs) are language models that are pre-trained on a large amount of text with billions of learnable parameters to achieve a general-purpose language understanding and generation. They are mainly transformer architectures based on self-attention (Figure 4.4) and are trained using self-supervised and semi-supervised learning. To achieve a specific task they usually have to be fine-tuned especially for their intended purpose. However, models such as GPT-3 can be *prompt-engineered* to achieve similar results[49].

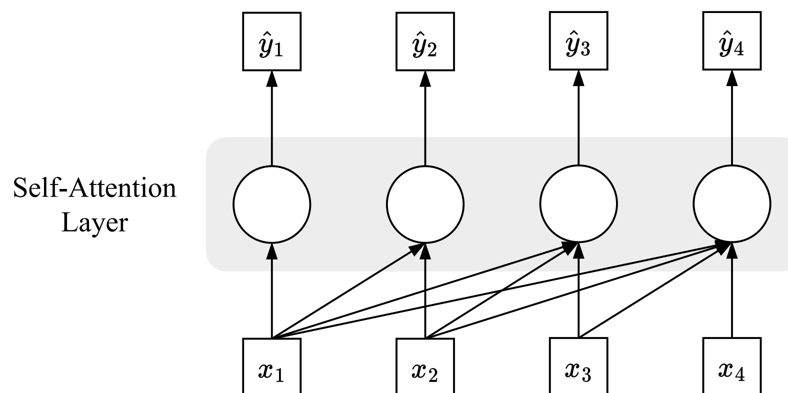


Fig. 4.4: Information flow in a self-attention model[49].

4.4.3.1 Causal Language Models

Causal Language Models (CLMs) are large language models that are pre-trained in an uni-directional autoregressive manner. They predict the next word or token in a sequence by using the previous words or tokens as a context. They can only attend to tokens to the left of the current token being generated. CLMs are useful for generating text and completing sentences.

The *Generative Pre-Trained Transformer* (GPT) series from the AI research and deployment company "OpenAI" is an example of a CLM. Unlike the vanilla transformer architecture the model only uses the unidirectional decoder portion. The name indicates the causal training mechanism of unlabeled data. To perform a specific task, the model uses *discriminative fine-tuning* on a task relevant data set to update its parameters. Version 3 has a parameter count of 175 billion. Their most recent model GPT-4 has an estimated number of over 1 trillion parameters, but the official count is undisclosed[5].

4.4.3.2 Masked Language Models

Masked Language Models (MLMs) are large language models that are pre-trained in an auto-encoded manner by masking the input sequence to recreate the unmasked sequence. They are better

suited for contextual understanding than causal language models. The ratio of masked tokens is defined with a masking rate. It is usually set to 15, but some researches suggest that especially large models with huge amounts of training data achieve better results with higher masking rates[54].

A notable example for a MLM is the *Bidirectional Encoder Representations form Transformers* (BERT) introduced in 2018 by researchers at Google. As the name implies, this architecture only uses the bidirectional encoder part of the vanilla transformer architecture. It is actually trained on two tasks simultaneously. Firstly, the one mentioned above. And secondly, with Next Sentence Prediction (NSP). NSP has the advantage of learning the relationship between two sentences. The data for NSP training can be trivially generated from any monolingual corpus. In 50% of the training data the sentences are actually consecutive and labeled with ”IsNext”, the other 50% consist of randomly merged sentences that are labeled ”NotNext”. BERT can be fine-tuned with one additional layer for specific tasks like question answering and language inference. [6].

Chapter 5
Application on Multi-Variate Time-Series Data

In this chapter two main approaches for the application of anomaly detection on multi-variate machine data from a ground improvement process are presented. The data set used consists of 272 time-series composed of 9 channels. The process is described in detail in section 5.1. The basic idea is to experiment with techniques from natural language processing. Therefore, the numerical time-series is converted into tokens to enable a word like processing of the data. That means the numerical time-series data is binned and labeled, resulting in a discrete set of symbols. The complete code for all tasks mentioned is written in MATLAB R2023b from Mathworks.

The first approach presented in section 5.3 is a statistical evaluation based on n-grams of the symbols. N-grams of different lengths are created and then weighted by their occurrence with term frequency-inverse document frequency measures. The higher the tf-idf score of an n-gram, the more unique it is among the different time-series, indicating an anomalous sub sequence in the data.

The second approach described in section 5.4 makes use of a state-of-the-art neural network, the transformer model. The data is split into a training and a test set. For training, the model receives a masked input sequence and the unmasked output sequence of tokens. After sufficient training the model is given a masked sequence it hasn't seen before and tries to reconstruct it. A bad reconstruction can potentially indicate unexpected behaviour in the time-series, therefore, indicating anomalous sequences.

The n-gram method is expected to work well on point anomalies, whereas the transformer approach should take the complete context of the time-series into account.

5.1 Underlying Process

The multi-variate time-series data used in this thesis was captured from a ground improvement process called vibro compaction performed by Keller UK Ltd. It is mainly used for[55]:

1. Reduction of foundation settlement.
2. Increase in bearing capacity, allowing reduction in foundation size or width.

3. Increase in stiffness.
4. Increase in shear strength.
5. Reduction of permeability.

As shown in Figure 5.1 the process can be split into three phases. The *initial phase* were gravel or sand is filled into a skip that loads into the inside of the cylindrical penetrator. The penetrator has an eccentric weight that is powered by an electric motor causing vibrations. In the *penetration phase* the vibrator is lowered till a hard rock layer is hit. The process continues with the *compaction phase* were the vibrator is raised in lifts and the gravel is unloaded into the pit to compensate for any decrease in soil volume. Lift by lift the energy of the vibrator compacts the back-fill and the surrounding soil till ground level is reached, resulting in a dense and compact column[55].

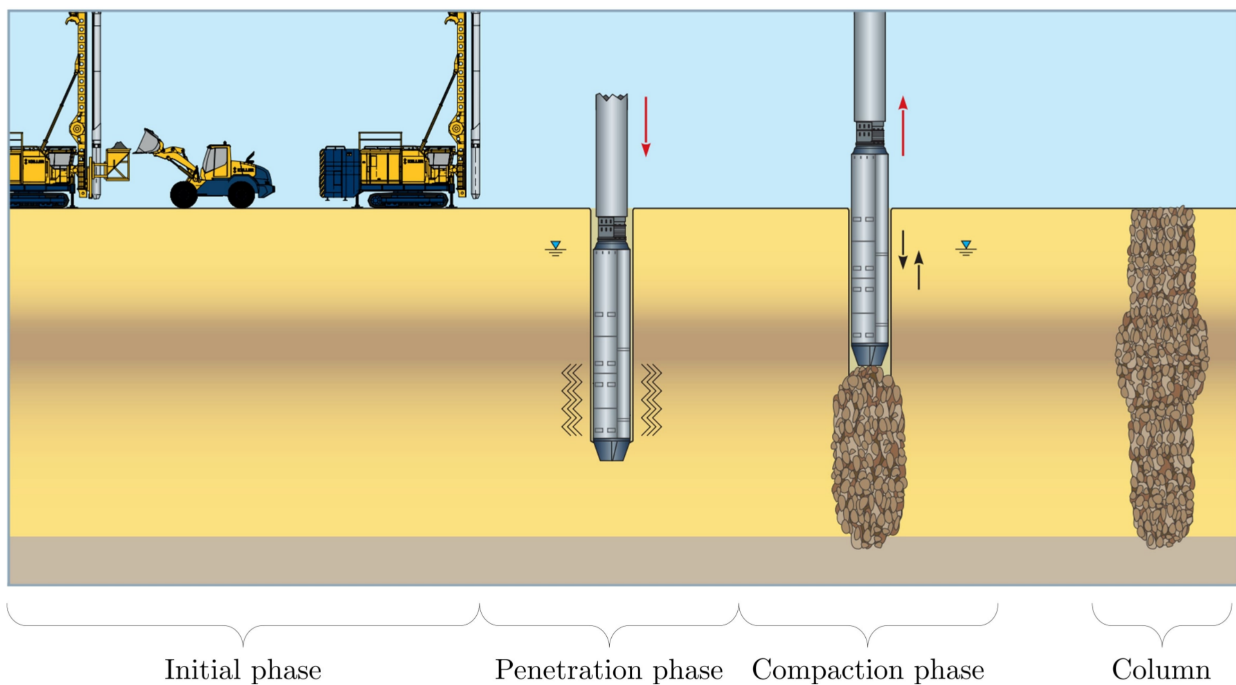


Fig. 5.1: Overview of the phases of the ground improvement process[34].

The columns often provide a stable foundation for the construction of buildings. A fault in the process can have fatal consequences. Imagine if the soil bearing capacity for a foundation is not as high as expected. Foundation settlements can not only cause an enormous financial damage, but also mean a serious safety hazard.

The machines are instrumented with $n = 9$ sensors that record a multi-variate time-series of each column at a sampling rate of $f_s = 1$Hz. An example is plotted in Figure 5.2. The 9 channels capture the following features:

1. *Depth* in metres
2. *Feedrate* in metres per minute
3. *Pulldown force* in kilo Newtons

4. *Amperage* in Ampere
5. *Frequency* in Hertz
6. *Vibrator temperature* in degree Celsius
7. *Weight loaded* in tons
8. *Inclination in X-direction* in degrees
9. *Inclination in Y-direction* in degrees

In the past the quality of the columns was ensured by Key-Performance-Indicators (KPI) derived from the data of the instrumented rigs. Usually, an installation report is created for every column, building the basis for quality control to this day. Geotechnical experts have to evaluate the safety of each column manually based on the reports. This can be tedious and also prone to errors[56]. The Chair of Automation in Leoben has done research over the last years to come up with improved methods to ensure the quality of the columns[57][58]. A Hybrid Machine Learning (HML) tool has been designed for the detection of anomalous multi-variate time-series. This novel approach combines the classic method based on approximately 50 physics-motivated KPIs and an unsupervised variational autoencoder with long short-term memory layers. The machine learning models were further improved by research and works in hyperparameter optimization based on different methods, for example genetic algorithms[34][59]. The idea of combining both methods enhances the classification safety by providing redundancy. A bad foreseen column could potentially be indicated by the machine learning model[60][7].

5.2 Tokenization of MVTS Data

We refer to tokenization as the conversion from a sequence of data into tokens. The sequence data can for example be of textual or numerical nature. The tokens have symbolic character. That means they can be represented in any symbolic way, may it be a character, numbers or emojis. The labels don't make a difference for the information represented by a token. But, it is important that they are unique to identify to ensure their separateness. At the beginning of this thesis, "American Standard Code for Information Interchange" (ASCII) characters were used to label different tokens, but it turned out that numbers work a lot better. The problem with characters is that they have a limited pool to choose from. The ASCII set contains 95 printable characters. We could solve this problem by using two symbols to describe one token, resulting in 9,025 possible combinations. But in the end all this symbol representations don't benefit the anomaly detection and make the data more confusing to handle, so it was discarded and we just enumerate the different tokens.

Fig. 5.2: Example time-series plot of all 9 channels of the vibro ground improvement process. The process is segmented into an initial phase (grey), a penetration phase (pink) and a compaction phase (yellow).

5.2.1 Time domain of MVTS Data

Additionally, the time domain can be preprocessed. Firstly, it had to be decided if the time domain gets re-sampled to create a sequence with less data points. This was tested with a method called piece-wise aggregate approximation (PAA). It is a simple dimension reduction method based on the mean values of intermediate time-steps. However, in order to not loose valuable information from the time domain no resampling was chosen.

A segmentation of the penetration and compaction phase was also taken into account. Segmenting those turned out in not really benefiting the process in most cases, but rather making it more confusing to handle the sequences. However, the initial phase was discarded in the final approach, because it is not of significant meaning for the quality of the process.

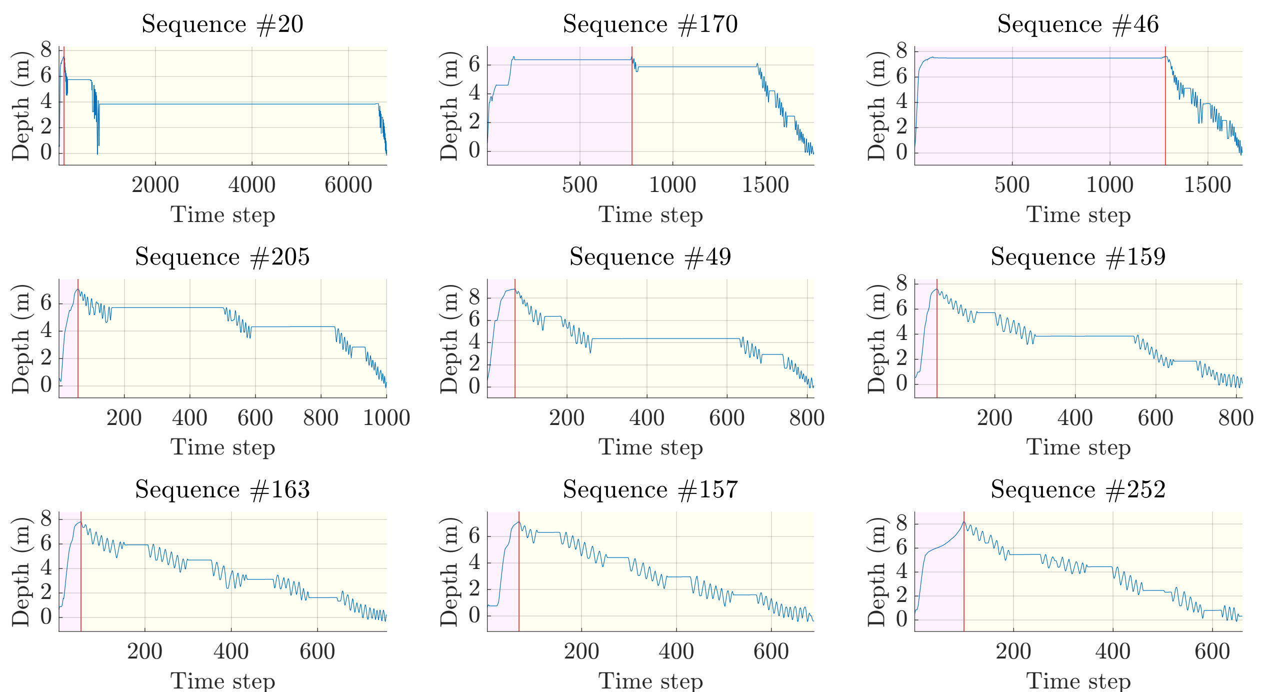
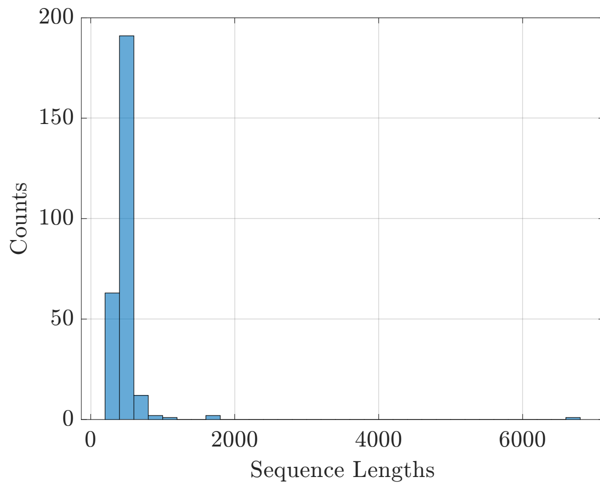
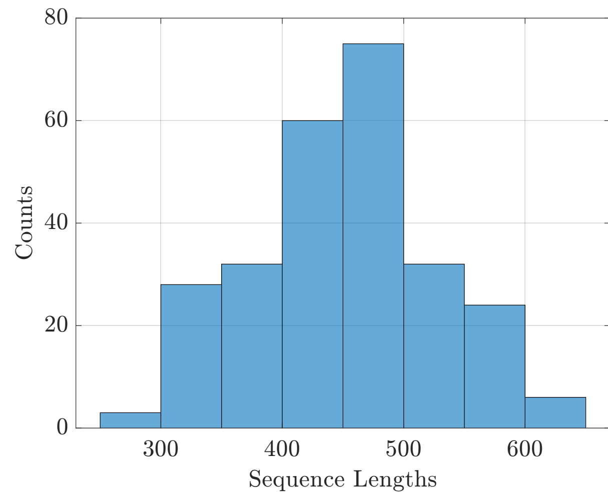


Fig. 5.3: The depth of the 9 longest sequences without the initial phase before reduction of the data set.

Lets have a look on the distribution of sequence lengths of the complete site data. We can see that most of the sequences have a length of around 500 time steps. But there are also a few outliers with a few thousand time steps. These could be straight away considered to be anomalous compared to the rest of the sequences. Mostly, they consist of very long idle times, which are not valuable to us in this analysis. They also make it different to handle the complete data set, especially in machine learning, where we have to build the model based on the longest sequence in the data set. So we exclude them from the data set by setting a maximum sequence length threshold of 650.



(a) Raw data set sequence lengths



(b) Cut data set sequence lengths

5.2.2 Discretization of MVTS Data

The discretization process is one of the key steps of this thesis, because the discrete sequences are the basis for both, the n-gram model and the transformer model. The discretizer takes a numeric input and assigns it to a number of discrete classes. The discretization process for each channel can be split into the following steps:

1. Define the cardinality, the number of bins.
2. Choose the distribution of bins: linear, exponential or equiprobable.
3. Normalize the data of the depth channel.
4. Evaluate the minimum and maximum values across all sequences in the data set.
5. Calculate the edges for all bins ranging from min to max.
6. Assign bin numbers according to data point values and edges.

The maximum depth reached in the process differ a bit from sequence to sequence. This is visualised in Figure 5.8. This is nothing anomalous, but rather due to the geology of the ground at the process site. The penetration phase is stopped when hard rock is hit, resulting in the maximum depth of the sequence. Naturally, the solid rock depth is not equal for different locations.

If we apply the discretization without normalizing of the depth channel, some sequences with a higher maximum depth get assigned bins, which others never reach. This results in higher anomaly scores for the n-gram model and might also do the same for the machine learning approach. That is why the depth channel has to be normalized before binning. Other channels don't have this characteristics.

There are some channels that range in both positive and negative direction. For example the feedrate or inclination. In general, it is likely that these follow a normal distribution with mean around zero. If we choose uniform distributed edges for the bins. We can see that there are at least

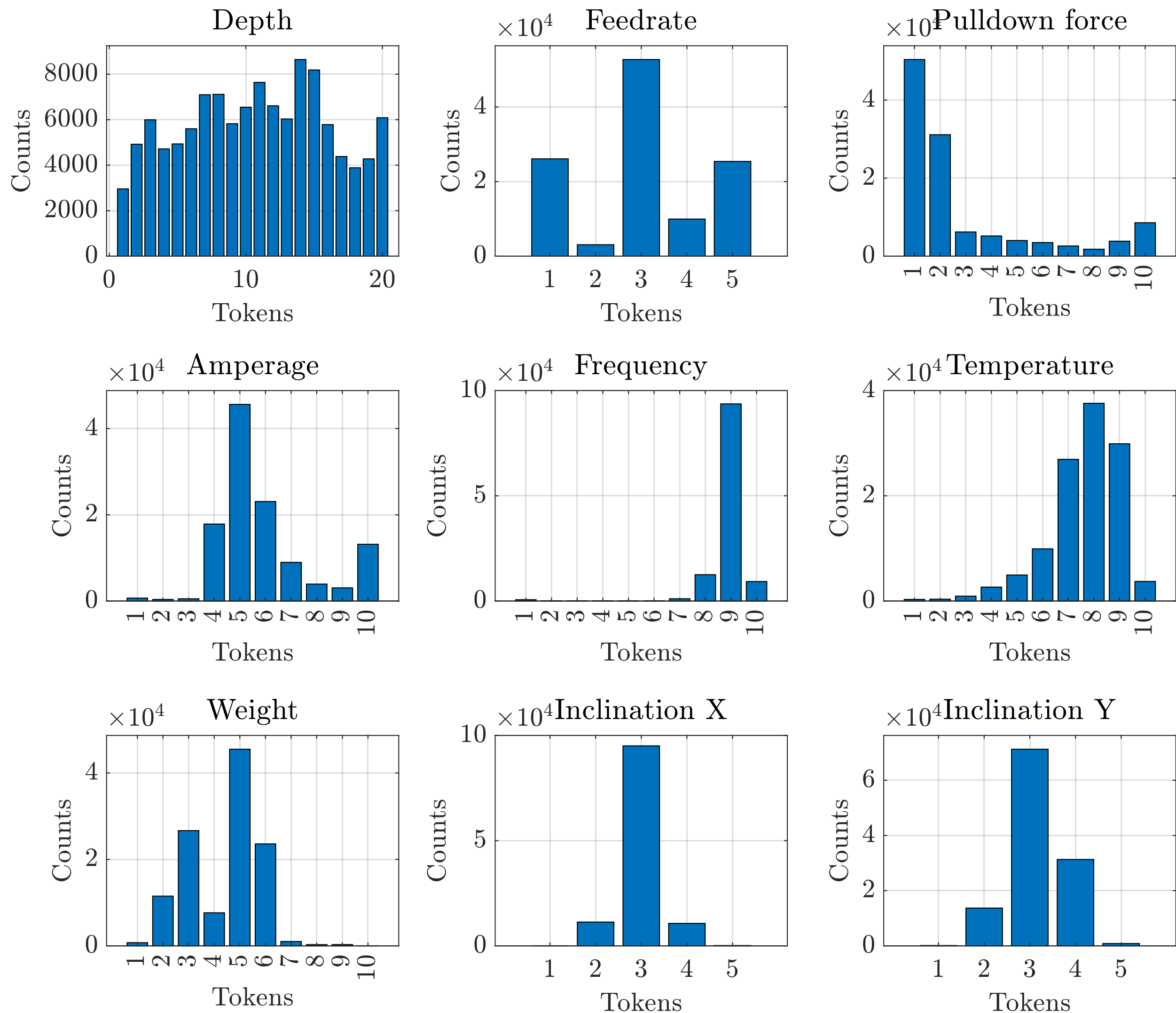


Fig. 5.5: Histograms of the token counts for all 9 channels.

10 times more values for the mid zero bin than for the others. To compensate a bit, the middle edges got shrink by a shrinking factor in an exponential manner.

5.2.3 Wording of MVTS Data

We want to capture inter-dimensional anomalies, therefore we have to somehow combine the symbols from the different channels into multi-variate tokens. One way of doing this is to just concatenate the symbols to create words of length of the number of channels. In our case, we have 9 channels, and therefore 9 sequences of symbols of equal length. However, this also has some drawbacks. When using neural network models for language processing there is the challenge of dealing with out-of-vocabulary (OOV) tokens. These are tokens that are fed to the model but haven't been seen during training. So the model has absolutely no information how to process them without

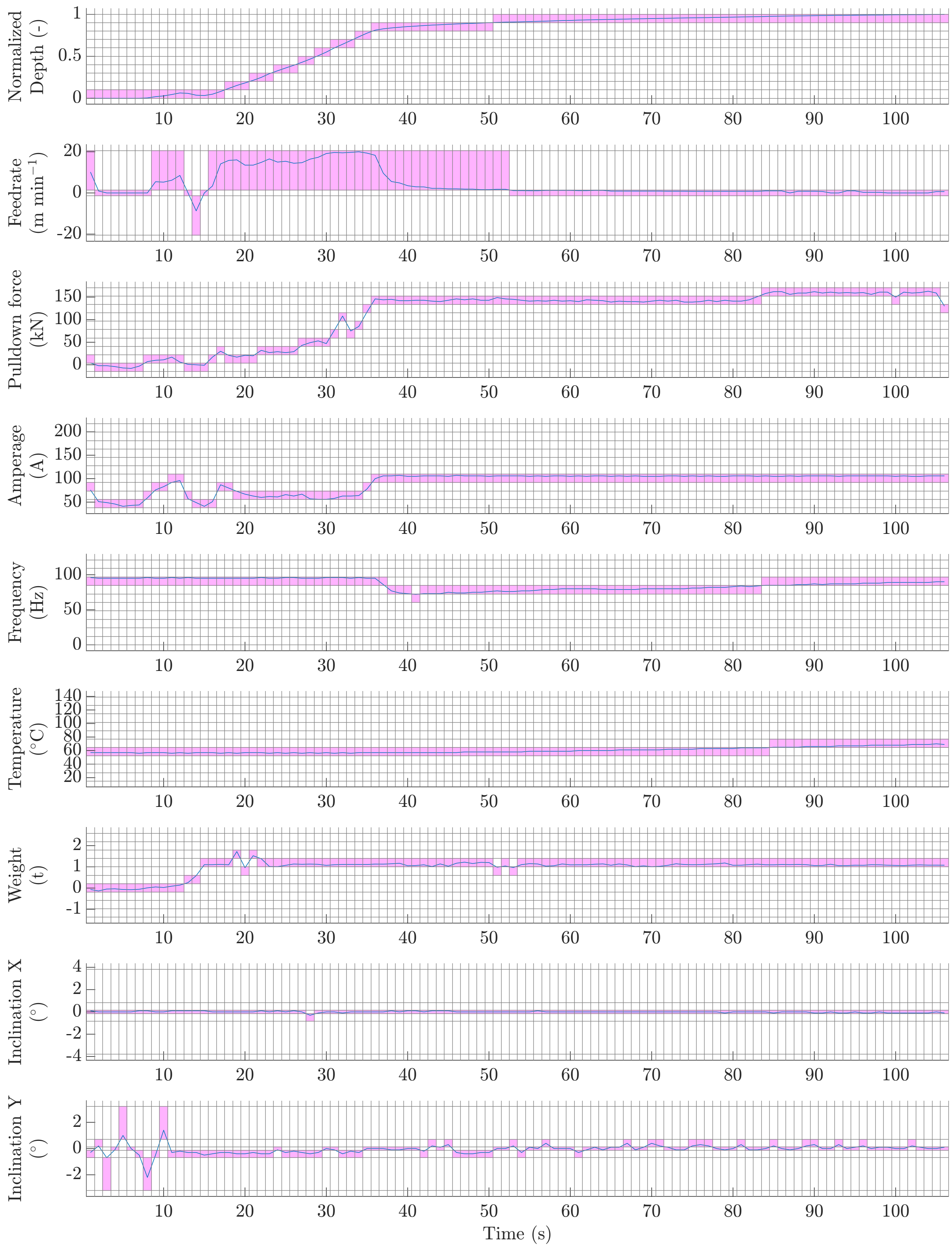


Fig. 5.6: Penetration phase of the raw sequence(blue) and its discrete representation(pink).

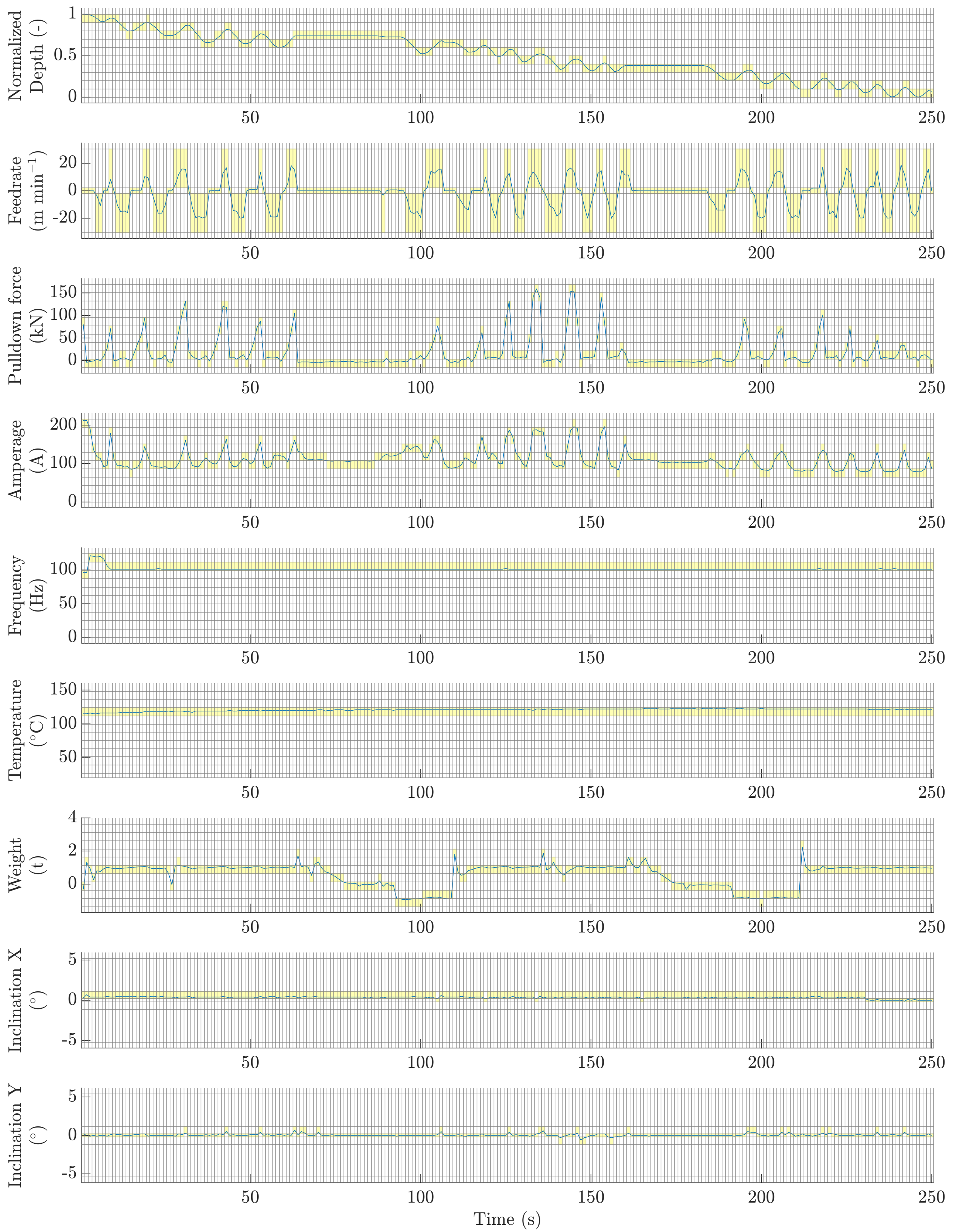


Fig. 5.7: Compaction phase of the raw sequence(blue) and its discrete representation(yellow).

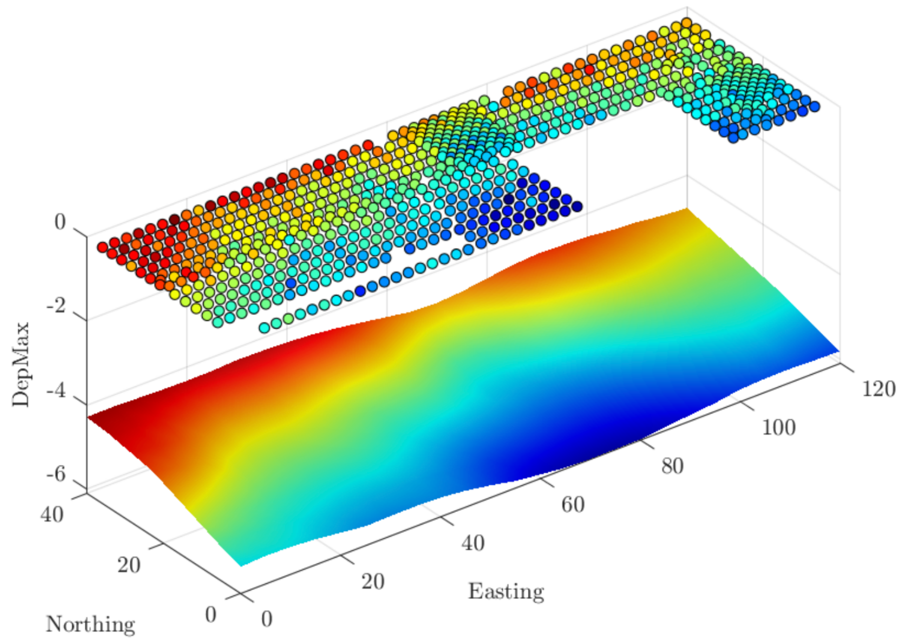


Fig. 5.8: Maximum reached depth during vibro ground improvement at different GPS coordinates of a site and its inferred surface below. Reprinted from [60].

some work-arounds. In conclusion, if we create inter-dimensional tokens the chances of OOV tokens occuring during testing rise significantly, because the possible combinations of tokens and therefore the vocabulary size increases exponential with the number of channels combined. There is no problem with OOV tokens in the n-gram-based anomaly detection, but we also don't necessarily improve the models performance by combining different channels. It should though be beneficial if we would have a huge amount (>1000) of sequences to analyse.

5.3 N-Gram-based Anomaly Detection

An anomaly detection based on n-grams and term frequency-inverse document frequency is presented. A n-gram is a sequence of n adjacent symbols, words or tokens. A thorough explanation to n-grams is given in subsection 4.3.3. Term frequency - inverse document frequency (TF-IDF) is a measure of importance of a word to a document in a collection. It is used in information retrieval, for example in text-based recommender systems in digital libraries. For details see subsection 4.3.2. The analogy to this data is as follows. A document represents a time series and the terms are the tokens created. A high term frequency indicates that a term can be found often in a particular time series. A high inverse document frequency means a token occurs only in a few documents. This signalizes a unique token in the corpus and therefore anomalous behaviour. If we combine both measures by multiplication, we get a TF-IDF score for each token in each time-series.

5.3.1 Process Description

The n-gram-based anomaly detection consists of the following steps:

1. Create the token sequences as desribed in subsection 5.2.2.
2. Create a bag-of-n-grams for each channel and n-gram length from 1 to the maximum n-gram length defined.
3. Compute the term frequency-inverse document frequency matrices for each bag-of-n-grams.
4. Sum up the frequencies of each token sequence to get a sequence anomaly score.
5. The sequences with an anomaly scores above a defined threshold are classified as anomalous.

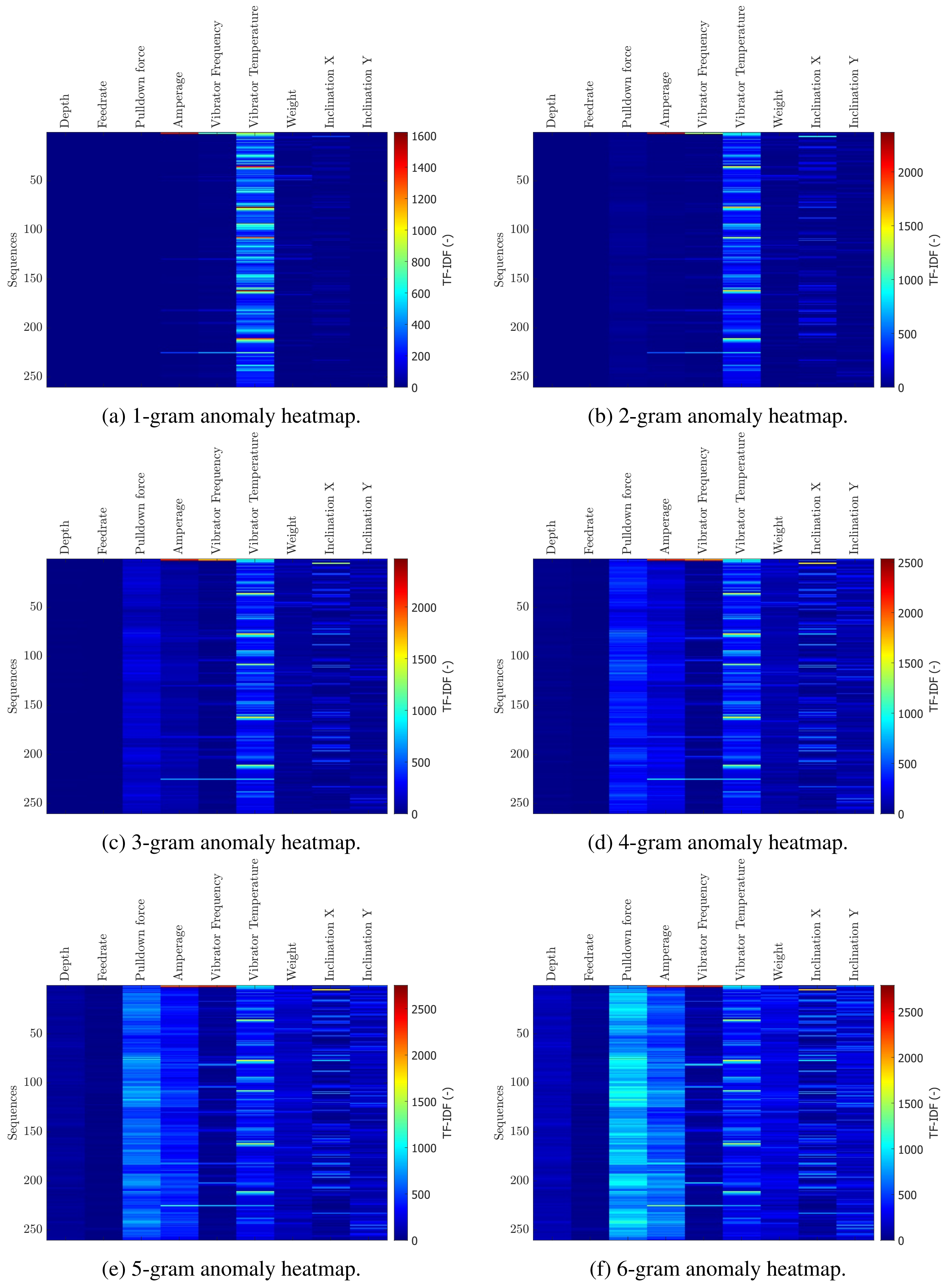
In the application a maximum n-gram length of 12 was chosen. Longer n-gram lengths resulted in almost every sequence being classified as anomalous. The problem here is that if the number is set to high, too many unique tokens are created. Consider the extreme case where the n-gram length is the sequence length, every sequence would have a unique n-gram and therefore, every sequence would get the same anomaly score, unless some tokenized sequences are exactly the same. This is not very likely in a complex real world data set. However, a larger data set would probably benefit longer n-gram lengths.

The 9 (number of channels) TF-IDF score matrices for each n-gram length have dimensions that are defined by the vocabulary size of the bag-of-n-grams and the number of sequences. The vocabulary size of a bag-of-n-grams results from the number of unique n-grams in the bag.

5.3.2 TF-IDF Scores

Figure 5.9 visualizes the anomaly measures for each channel and sequence for n-gram lengths from 1 to 6. High anomaly scores for short n-gram lengths were mostly found for values that are out of the usual range. For example, some sequences have a higher maximum amperage than the average. These get assigned tokens, that are not occurring in other sequences. The TF-IDF measure assigns high IDF scores to those, resulting in a high sequence anomaly score. The same applies to the inclinations. It can also be seen that the vibrator temperatures get significantly higher anomaly scores compared to other channels in short n-gram lengths. This phenomena was also detected for the depth channel before normalizing. More details on the normalization of the depth channels were covered in subsection 5.2.2.

Longer n-gram lengths from 7 to 12 are depicted in Figure 5.10. It can be seen that the longer the n-gram lengths, the higher are the anomaly scores for the pulldown force and the amperage. A deeper insight is given in Figure 5.11 where anomalous tokens of different n-gram lengths are highlighted. In the unigram graphic 5.11a, we can see that the very low and the very high values are highlighted. This indicates, that they can not be found in every sequence.



(a) 1-gram anomaly heatmap.

(b) 2-gram anomaly heatmap.

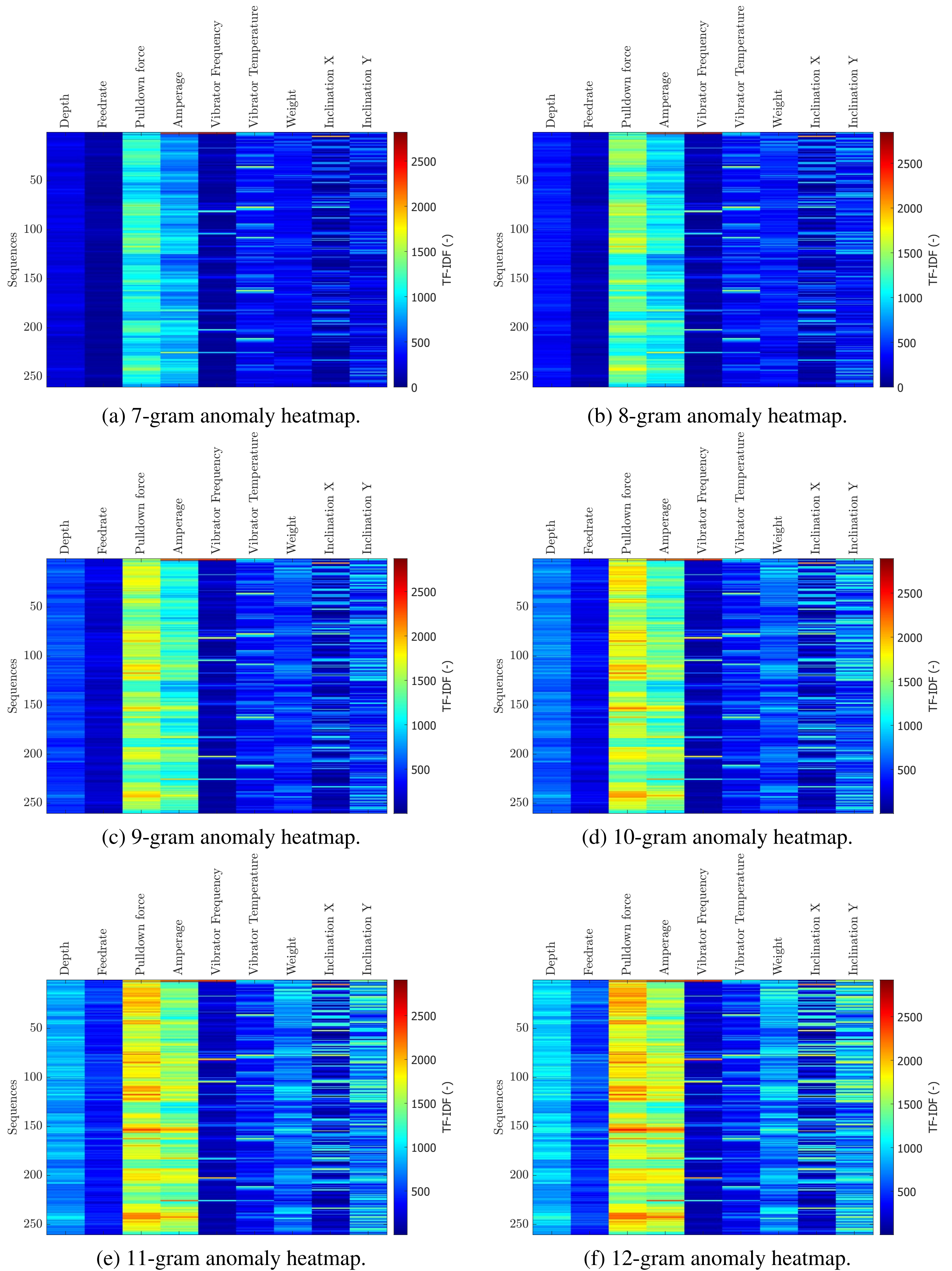
(c) 3-gram anomaly heatmap.

(d) 4-gram anomaly heatmap.

(e) 5-gram anomaly heatmap.

(f) 6-gram anomaly heatmap.

Fig. 5.9: Anomaly heatmaps for short n-gram lengths of 1 to 6.



(a) 7-gram anomaly heatmap.

(b) 8-gram anomaly heatmap.

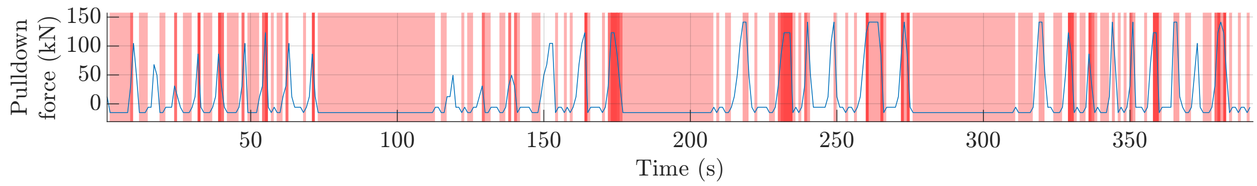
(c) 9-gram anomaly heatmap.

(d) 10-gram anomaly heatmap.

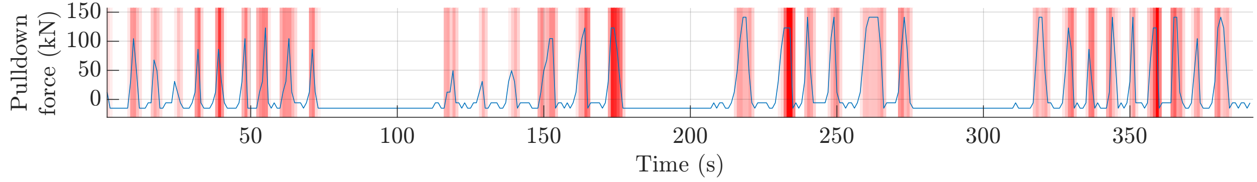
(e) 11-gram anomaly heatmap.

(f) 12-gram anomaly heatmap.

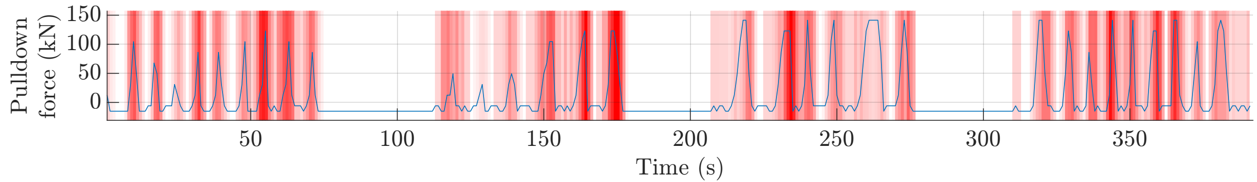
Fig. 5.10: Anomaly heatmaps for long n-gram lengths of 7 to 12.



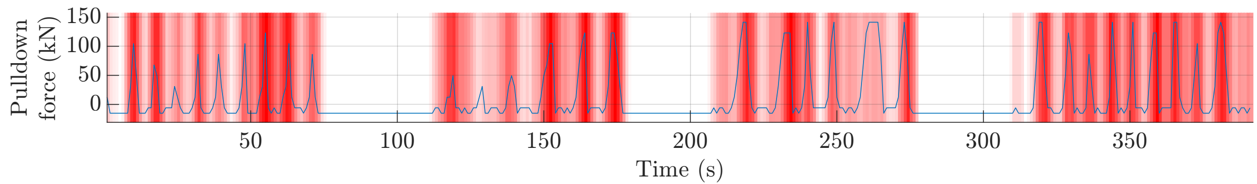
(a) Highlighted unigram anomalies for the pulldown force channel of the compaction phase.



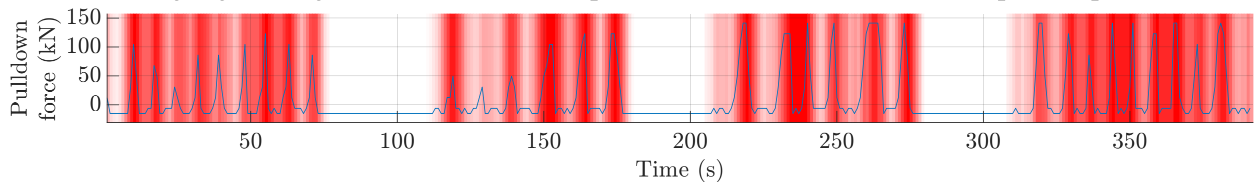
(b) Highlighted 2-gram anomalies for the pulldown force channel of the compaction phase.



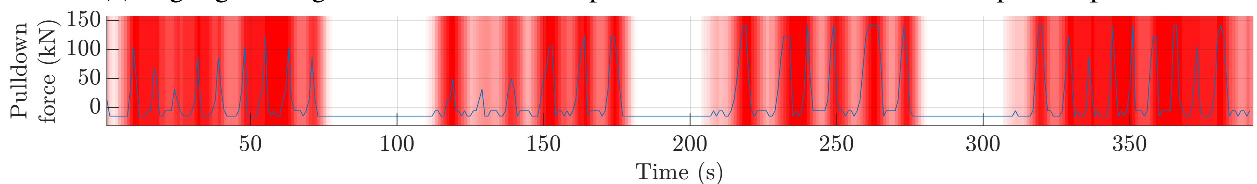
(c) Highlighted 3-gram anomalies for the pulldown force channel of the compaction phase.



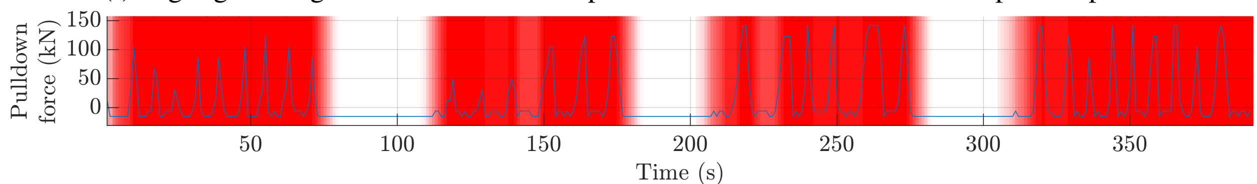
(d) Highlighted 4-gram anomalies for the pulldown force channel of the compaction phase.



(e) Highlighted 5-gram anomalies for the pulldown force channel of the compaction phase.



(f) Highlighted 6-gram anomalies for the pulldown force channel of the compaction phase.



(g) Highlighted 8-gram anomalies for the pulldown force channel of the compaction phase.

Fig. 5.11: Highlighted anomalies for the pulldown force channel of a compaction sequence. The colors are normalized and do not represent the absolute value of the anomalies.

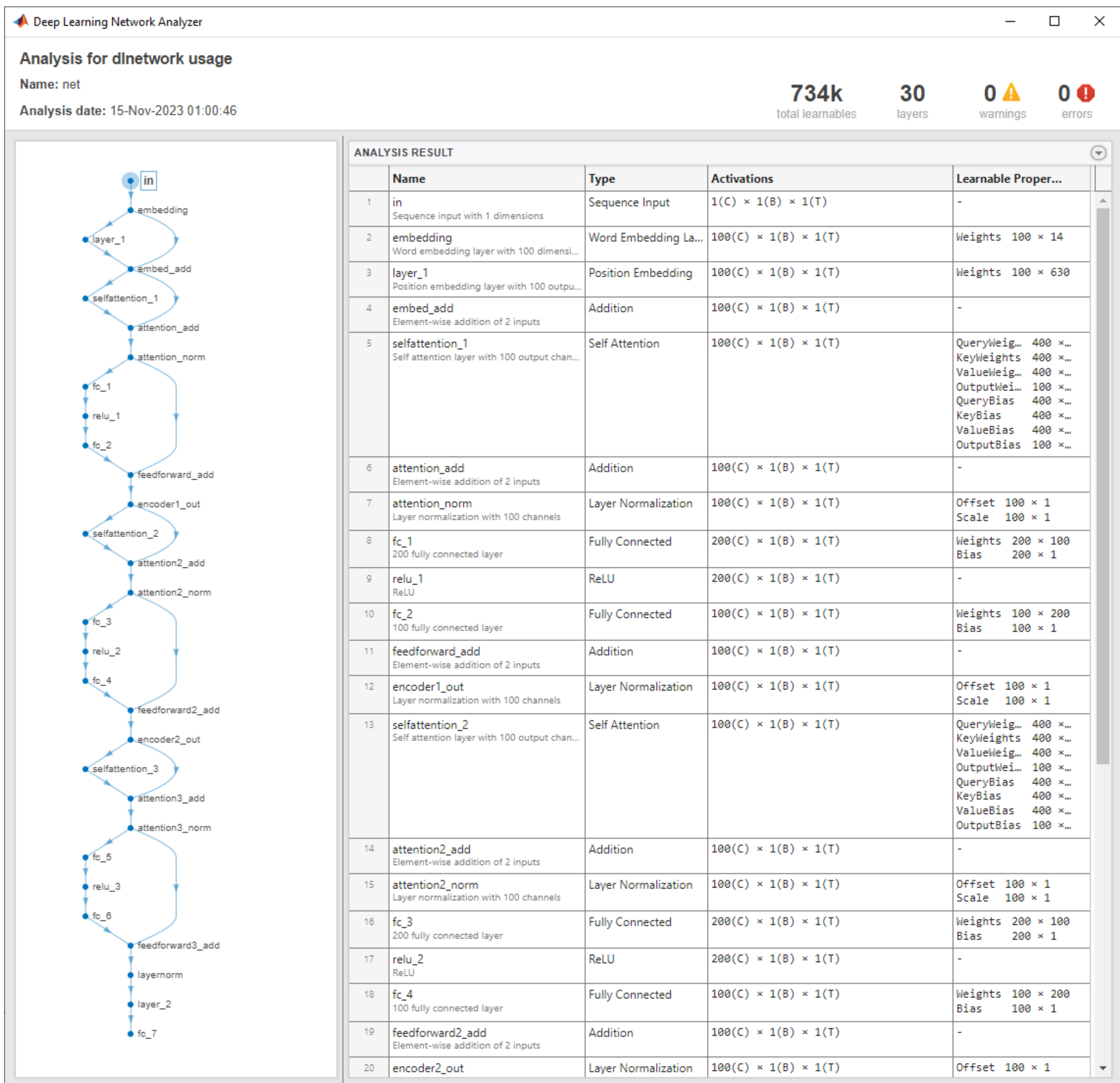
5.4 Transformer-based Anomaly Detection

In the second method for anomaly detection a transformer model is used. More specific, the encoder block of the vanilla transformer architecture described in subsection 3.2.1. The inspiration for the use of this model came from the Google NLP model BERT, the Bidirectional Encoder Representations from Transformers, and its clever unsupervised learning method. BERT learns from a large corpus of text inter alia by feeding the input a sequence with randomly masked tokens and the output the unmasked sequence. In this way the model can learn the structure of natural language without the need of any labeled data. Normally the model is fine-tuned for specific tasks[6].

Because the data used for this application has no labels to indicate where the anomalies are located, this approach is very promising. Furthermore, the transformer architectures are well known for handling complex long term dependencies. In general, neural networks are very good in generalizing. This sentence may sound odd, but it has a profound meaning. Generalization is a crucial aspect of machine learning models and it refers to the performance of handling unseen data. This implies that a model can be trained on a large data set to learn its structure and meanings or in the context of NLP its syntax and semantics. In the context of anomaly detection this is a huge benefit, because the model can potentially detect anomalies it has not been trained on.

5.4.1 Transformer Model

An encoder block consist of a self-attention, an addition, a layer normalization, a fully connected another addition and another layer normalization layer. These can be stacked multiple times. Configurations ranging from 2 to 8 blocks were tested. Figure 5.12 depicts a tested network with 3 encoder blocks and a total of 734.000 learnable parameters.



	Name	Type	Activations	Learnable Proper...
1	in Sequence input with 1 dimensions	Sequence Input	1(C) × 1(B) × 1(T)	-
2	embedding Word embedding layer with 100 dimensi...	Word Embedding La...	100(C) × 1(B) × 1(T)	Weights 100 × 14
3	layer_1 Position embedding layer with 100 outpu...	Position Embedding	100(C) × 1(B) × 1(T)	Weights 100 × 630
4	embed_add Element-wise addition of 2 inputs	Addition	100(C) × 1(B) × 1(T)	-
5	selfattention_1 Self attention layer with 100 output chan...	Self Attention	100(C) × 1(B) × 1(T)	QueryWeig... 400 ×... KeyWeights 400 ×... ValueWeig... 400 ×... OutputWei... 100 ×... QueryBias 400 ×... KeyBias 400 ×... ValueBias 400 ×... OutputBias 100 ×...
6	attention_add Element-wise addition of 2 inputs	Addition	100(C) × 1(B) × 1(T)	-
7	attention_norm Layer normalization with 100 channels	Layer Normalization	100(C) × 1(B) × 1(T)	Offset 100 × 1 Scale 100 × 1
8	fc_1 200 fully connected layer	Fully Connected	200(C) × 1(B) × 1(T)	Weights 200 × 100 Bias 200 × 1
9	relu_1 ReLU	ReLU	200(C) × 1(B) × 1(T)	-
10	fc_2 100 fully connected layer	Fully Connected	100(C) × 1(B) × 1(T)	Weights 100 × 200 Bias 100 × 1
11	feedforward_add Element-wise addition of 2 inputs	Addition	100(C) × 1(B) × 1(T)	-
12	encoder1_out Layer normalization with 100 channels	Layer Normalization	100(C) × 1(B) × 1(T)	Offset 100 × 1 Scale 100 × 1
13	selfattention_2 Self attention layer with 100 output chan...	Self Attention	100(C) × 1(B) × 1(T)	QueryWeig... 400 ×... KeyWeights 400 ×... ValueWeig... 400 ×... OutputWei... 100 ×... QueryBias 400 ×... KeyBias 400 ×... ValueBias 400 ×... OutputBias 100 ×...
14	attention2_add Element-wise addition of 2 inputs	Addition	100(C) × 1(B) × 1(T)	-
15	attention2_norm Layer normalization with 100 channels	Layer Normalization	100(C) × 1(B) × 1(T)	Offset 100 × 1 Scale 100 × 1
16	fc_3 200 fully connected layer	Fully Connected	200(C) × 1(B) × 1(T)	Weights 200 × 100 Bias 200 × 1
17	relu_2 ReLU	ReLU	200(C) × 1(B) × 1(T)	-
18	fc_4 100 fully connected layer	Fully Connected	100(C) × 1(B) × 1(T)	Weights 100 × 200 Bias 100 × 1
19	feedforward2_add Element-wise addition of 2 inputs	Addition	100(C) × 1(B) × 1(T)	-
20	encoder2_out	Layer Normalization	100(C) × 1(B) × 1(T)	Offset 100 × 1

Fig. 5.12: A summary of the transformer encoder applied for anomaly detection printed from the Matlab network analyser. It has 3 encoder blocks totaling 30 layers and 734.000 learnable parameters.

5.4.2 Data Preprocessing for Machine Learning

Before the data is fed to the transformer some preparations of the data have to be done. The same tokenized sequences used for the n-gram model are also used for the transformer. The exclusion of very long sequences is also essential, because the model structure is based on the longest sequence in the data. And the longer the longest sequence the harder the model is to train due to increasing complexity. The model can only take input vectors of equal size, therefore shorter sequences have

to be padded. Padding is usually realized by inserting a padding token. Some models can make use of padding masks to have information which elements in the sequence got padded. In this way the model doesn't pay attention to those while optimizing. Unfortunately, because of lack of detailed documentation in the Matlab self-attention layer no padding masks were used.

Different padding directions were tested. If padding tokens were added at the beginning of the sequence strong oscillations were found in the predicted sequence. This could be solved by changing the padding direction to the right. The model was mainly tested on the depth channel of the data set. The characteristics of this time-series with padding is way smoother if padding is added at the end, because normally the penetration process has a steep curve, whereas compaction has a rather flat descent. This oscillatory characteristics is similar to the gibbs phenomena occuring in fourier transformations. For the masked training approach random tokens have to be masked. Therefore, a masking ratio has to be defined. Initially this was set to a value of 0.15, like in the BERT model[6]. But a recent paper states, that higher values do benefit the model[54].

Because random tokens are masked and the number of time-series in the data set is rather small for a machine learning model, a little trick was tested to enlargen the data set. A multiplier integer m was defined that duplicates each sequence m times before masking random positions.

5.4.3 Training

The model was run on a dedicated graphical processing unit with 8 gigabyte of dynamic random access memory. Training time heavily depends on the hyperparameters, but ranged from 15 minutes to eight hours for one run. Some hyperparameters of the model and are listed below:

1. Number of encoder blocks
2. Number of attention heads
3. Number of hidden units in the feedforward layers
4. Dimension of the word embedding layer
5. Training ratio
6. Number of epochs
7. Mini-batch size
8. Loss function
9. Solver
10. Initial learn rate
11. Learn rate drop factor
12. Learn rate drop period

The encoder blocks are stacked multiple times. Configurations ranging from 2 to 8 blocks were tested. The number of attention heads was set to 2, but up to 10 heads were tested. The number of hidden units in the feedforward layer was set between 20 and 200. the dimension of the word

embedding layer from 10 to 200. The training ratio defines the portion of the data used for training. It was set between 0.4 and 0.8. The total passes of the complete data set was set from 30 to 500. A mean square error loss function was used and a stochastic gradient descent with momentum as well as an an adaptive moment estimation optimizer was mainly used as a solver. Both solvers are available out of the box in Matlab. And an adaptive learning rate with an initial learn rate around 0.1 and a learn rate drop factor of around 0.9 was used.

The targets for the predictions were tested with the actual values as well as the mean values of the discrete bins. There could not be found a significant difference in the results of both.

5.4.4 Validation

Because of the unlabeled nature of the data set, a numeric validation proved to be very difficult. The goal of this thesis was not to solve the anomaly detection problem for this data set, rather it should be an exploratory task to see if NLP approaches can be applied to anomaly detection. However, we can evaluate the quality of the reconstructed sequences.

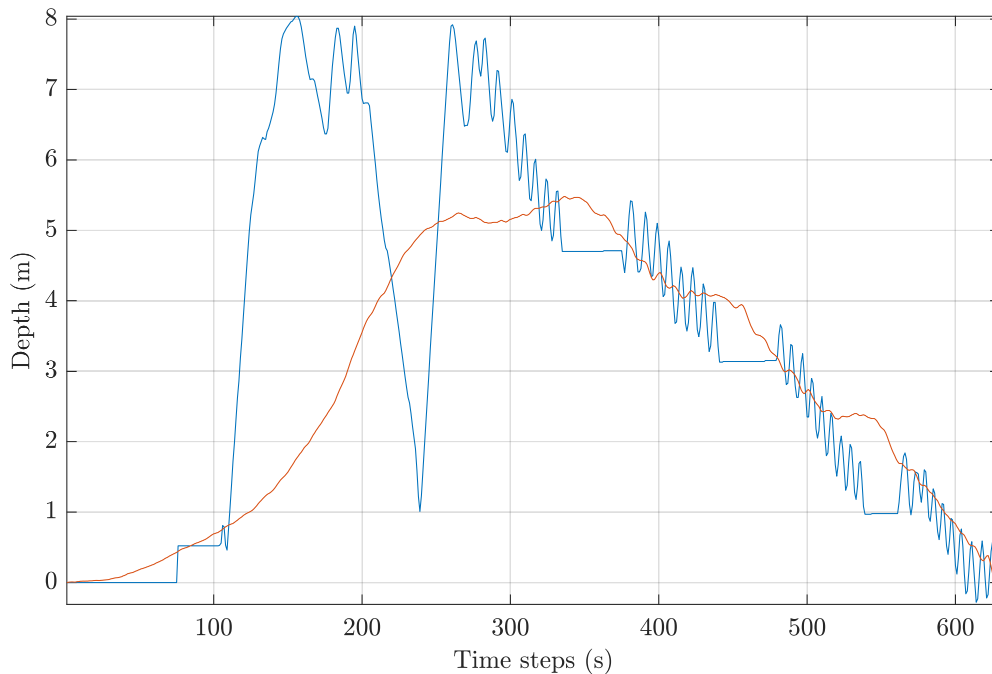


Fig. 5.13: The original depth sequence(blue) and its reconstruction(orange) from a left padded tokenized sequence. The model was trained on 100 epochs with the adam solver and a mean squared error loss function. There are 3 encoder blocks and 4 attention heads used in this particular model.

Figure 5.13 and Figure 5.14 show the reconstructions of a depth channel sequence from two different models. Figure 5.13 uses a left padded approach indicated by the zero values at the start

of the sequence. The original input sequence in blue is a rather anomalous one. The operator has lifted the penetrator completely out of the column during the compaction phase. The reconstruction somehow tried to compensate, but there are big deviations to the original sequence. Figure 5.14 uses padding tokens in the end of the sequence. This graph of the normalized depth could be considered rather normal compared to the other one. This model creates a closer reconstruction than the one mentioned before. There is definitely a lot of room for improvement here. Machine

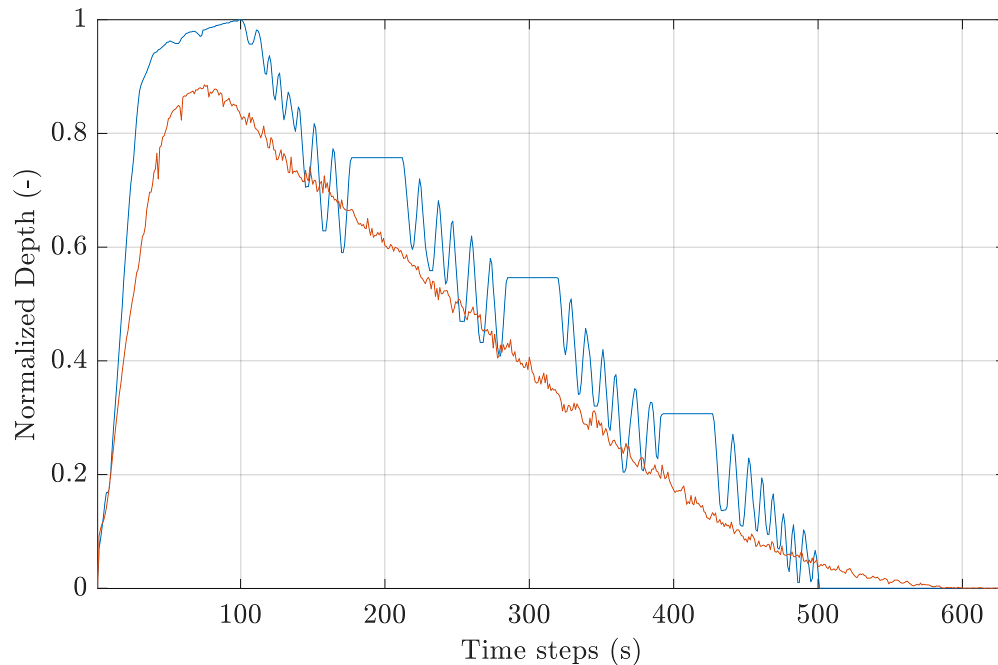


Fig. 5.14: The normalized depth sequence(blue) and its reconstruction(orange) from right padded tokenized sequence. Training was performed over 150 epochs with the adam solver and a mean squared error loss function. There are 3 encoder blocks and 4 attention heads used in this particular model.

learning models are very complex and setting up a new architecture isn't a straightforward task if you don't have a lot of experience in this field. At some point a line had to be drawn where a conclusion had to be made.

Chapter 6
Summary and Conclusion

This thesis investigated the use of natural language processing methods to identify quasi-linguistic patterns in multi-variate machine data for unsupervised anomaly detection. The basis for this task was laid by converting numerical data into tokenized data in a discretization process. The tokens are utilized to mimic the symbolic character of natural language.

Further, the tokens were concatenated to a length of n to create so called n-grams. These n-grams were then collected in an unordered bag-of-n-grams. The counts of the n-grams were weighed with a term frequency - inverse document frequency measure. This allowed statistical reasoning of the distributions of small sub-sequences across the complete data set.

The evaluation of this method on data for a ground improvement gave an insight of its possibilities and limits. Short n-gram lengths are useful for detecting values that are out of the usual range. In general, short-term anomalies can be detected with this approach. To ensure a reliable detection of longer anomalies the data set inspected would have to consist of a lot of very similar sequences. Improvements could be made by also taking other features into account, e.g., the n-gram occurs k times in the first half of the sequence, or specific pairs often have another n-gram in between. Another possible approach is to exchange n-grams to skip-grams.

The token sequences derived from the machine data was also used in a machine learning model. Training methods originating from natural language processing were applied. Random tokens in the input sequences got masked and the transformer was trained to recreate the numeric sequences. The idea behind it is to make use of the generalization abilities of machine learning models by comparing the reconstruction with the original sequence. The greater the difference between them, the more likely an anomalous sequence has been found. The results from the application of this implementation indicated that this method is principally functional. However, the model has to be further optimized to exploit its full potential and allow reliable anomaly detection. Future improvements of the model could be made by optimizing it with hyperparameter optimization and the generation of interdimensional tokens.

List of Figures

2.1 The receiver operating characteristics (ROC) curve from a multiclass iris plant classifier. The area under the curve values for the classes can be seen in the legend. The model operating points are also plotted. Note the AUC score for the setosa class, indicating a perfect true positive and false positive-rate[22]. 9
2.2 The architecture of the perceptron. 11
2.3 Different activation functions and its derivatives used in neural networks. 14

3.1 An examplary graph of a fully connected feedforward neural network with two hidden layers and a width of six. The network processes an input vector x of dimension 8 and outputs a vector $\hat{y}$ of dimension 3. Adapted from[34]. 16
3.2 A thin feedforward network with one neuron in each layer. 17
3.3 Unfolding a recurrent network with one layer and a single recurrent unit. The recurrent unit stores information from the previous time step in a hidden state vector h. With every time step of the input vector the hidden state vector gets updated. 19
3.4 A single LSTM-cell with a forget, input, candidate and output gate. 20
3.5 The vanilla transformer model architecture[4]. 21
3.6 Scaled Dot-Product Attention[4]. 22
3.7 Multi-Head Attention consists of several attention layers running in parallel[4]. . . . 23

4.1 Major levels of linguistic structure[41]. 26
4.2 Continuous-bag-of-words and Skip-gram models for Word2Vec learning[49]. 35
4.3 Four RNN architectures for NLP tasks[49]. 36
4.4 Information flow in a self-attention model[49]. 37

5.1 Overview of the phases of the ground improvement process[34]. 40
5.2 Example time-series plot of all 9 channels of the vibro ground improvement process. The process is segmented into an initial phase (grey), a penetration phase (pink) and a compaction phase (yellow). 42

5.3 The depth of the 9 longest sequences without the initial phase before reduction of the data set. . . . 43
5.5 Histograms of the token counts for all 9 channels. . . . 45
5.6 Penetration phase of the raw sequence(blue) and its discrete representation(pink). . . 46
5.7 Compaction phase of the raw sequence(blue) and its discrete representation(yellow). 47
5.8 Maximum reached depth during vibro ground improvement at different GPS coordinates of a site and its inferred surface below. Reprinted from [60]. . . . 48
5.9 Anomaly heatmaps for short n-gram lengths of 1 to 6. . . . 50
5.10 Anomaly heatmaps for long n-gram lengths of 7 to 12. . . . 51
5.11 Highlighted anomalies for the pulldown force channel of a compaction sequence. The colors are normalized and do not represent the absolute value of the anomalies. 52
5.12 A summary of the transformer encoder applied for anomaly detection printed from the Matlab network analyser. It has 3 encoder blocks totaling 30 layers and 734.000 learnable parameters. . . . 54
5.13 The original depth sequence(blue) and its reconstruction(orange) from a left padded tokenized sequence. The model was trained on 100 epochs with the adam solver and a mean squared error loss function. There are 3 encoder blocks and 4 attention heads used in this particular model. . . . 56
5.14 The normalized depth sequence(blue) and its reconstruction(orange) from right padded tokenized sequence. Training was performed over 150 epochs with the adam solver and a mean squared error loss function. There are 3 encoder blocks and 4 attention heads used in this particular model. . . . 57

List of Tables

2.1 A binary confusion matrix. 7
2.2 A mutli-class confusion matrix with 5 different classes. 7

4.1 Bag-of-words for the sentence, ”The lion chased the deer.”[49]. 31
4.2 Document frequency and inverse document frequency for selected words of a corpus of 37 Shakespeare plays[49]. 32
4.3 Bag-of-bigrams for the sentences, ”The lion chased the deer.”, and ”The deer chased the lion.”[49]. 33

Appendix A

Appendix A

A.1 Matlab Tokenizer Code

Tokenize Time-Series Data

Author: Philip Nuser

History: \change{1.0}{06-Oct-2023}{Original}

Source: Chair of Automation, University of Leoben, Austria

email: automation@unileoben.ac.at *url:* automation.unileoben.ac.at

(c) 2023, Philip Nuser

Abstract

This is a script to load the site data and tokenize all channels. The data is then stored for later statistics and evaluation.

```
close all;
clear;
```

Select File Path

Set a default directory for the data.

```
dataPath = 'C:\Users\phili\Documents\MATLAB\Master Thesis\Data\Fehring\Timeseries';
```

Uncomment for ui selection of folder:

```
% dataPath = uigetdir( defaultDir, 'Select Site MAT dir');
```

Define a search string to look for .mat files.

```
searchStr = fullfile(dataPath, '*.mat');
```

Set the site name.

```
SiteName = 'Fehring';
```

Get a list of files.

```
listOfFiles = dir(searchStr);
```

Set true for saving the tokens to a file.

```
saveTokensBool = false;
```

Define Plot Parameters

Define the labels, fonts, colors, etc.

```
channelLabels = [{"Depth";"Depth (-)"}; {"Feedrate";"(m min$^{-1}$)"}; {"Pulldown force";"(kN)"}; ...
    {"Amperage";"(A)"}; {"Frequency";"(Hz)"}; {"Temperature";"($^{\circ}$C)"}; {"Weight";"(t)"}; ...
    {"Inclination X";"($^{\circ}$)"}; {"Inclination Y";"($^{\circ}$)"} ];
```

Set true for creating and exporting the graphics to a .pdf file.

```
exportGraphicsBool = false;
```

Set Segment Parameters

Define the segments to be included for the tokenization. [true; false; true] is not allowed! Otherwise, we create discontinuinities.

```
initialBool = false;
penetrationBool = true;
compactionBool = true;
segmentsBool = [initialBool; penetrationBool; compactionBool];
segmentNames = ["Initial"; "Penetration"; "Compaction"];
```

Load the Data into Structure Object

Define if the depth gets normalized.

```
normalizeDepth = true;
```

Create two empty structs, one for the data and one for the statistics of the data.

```
data = struct;
statistics = struct;
```

Get the channel names from the first timetable in the list.

```
dataPath = listOfFiles(1).folder;
```

```
fileName = listOfFiles(1).name;
fullFileRef = fullfile( dataPath, fileName );
load(fullFileRef);
channelNames = dataTT.Properties.VariableNames;
nrChannels = length(channelNames);
statistics.("ChannelNames") = channelNames;
```

Get the number of all .mat files:

```
nrFiles = numel(listOfFiles);
```

Iterate through all .mat files to:

- get the timetable object
- exclude the initial phase
- save the channel data
- find and save minimum and maximum of each channel
- save the sequence length for easier handling
- extract and save the indices of the penetration and compaction events for later plotting

```
for k=1:nrFiles
    fileName = listOfFiles(k).name;
    fullFileRef = fullfile(dataPath, fileName);
    load(fullFileRef);
    data(k).fileName = fileName;
    dataTT = segmentSeq(dataTT, segmentsBool);
    for name = channelNames
        channelName = string(name{1});
        channelData = dataTT.(channelName);
        if channelName == "Depth"
            channelData = normalize(channelData,"range");
        end
        data(k).(channelName) = channelData;
        data(k).("Max"+channelName) = max(channelData);
        data(k).("Min"+channelName) = min(channelData);
    end
    data(k).("seqLength") = length(channelData);
    TTevents = dataTT.Properties.CustomProperties.Events;
    eventsTable = eventtable(TTevents);
    if penetrationBool
        data(k).("PenetrationStart") = find(eventsTable.Time(1) == dataTT.Properties.RowTimes);
        data(k).("PenetrationEnd") = find(eventsTable.Time(2) == dataTT.Properties.RowTimes);
    end
    if compactionBool
        data(k).("CompactionStart") = find(eventsTable.Time(3) == dataTT.Properties.RowTimes);
        data(k).("CompactionEnd") = find(eventsTable.Time(4) == dataTT.Properties.RowTimes);
    end
end
```

Exclude Extremely Long Sequences

There are some sequences in the data, that are extremely long just because of downtime. These would distort the statistics and are not representative for the process. Also, we need to define a maximum length for the machine learning model. All the sequences shorter than this will get padded at the end. If we have a few sequences that are multiple times longer than the mean, the majority of sequences will only be composed of padding tokens. We don't want that.

```
seqLengths = extractfield(data, "seqLength");
tempLengths = seqLengths;
if exportGraphicsBool
    uncutSequences = figureGen(15,20,20);
    h = histogram(tempLengths);
    grid on;
    ylabel("Counts")
    xlabel("Sequence Lengths")
    exportgraphics(uncutSequences,'uncutseqs.pdf');
end
```

Sort the sequences in ascending and descending order.

```
[ascLengths, ascInd] = sort(seqLengths);
descLengths = flip(ascLengths);
descInd = flip(ascInd);
```

Don't exclude any short sequences, because they are closely distributed. But we have to get rid of the really long ones. Lets plot the N longest sequences:

```
N = 9;
cols = 3;
```

```
rows = ceil(N/cols);
if exportGraphicsBool
    longestSeqsFig = figureGen(20,40,20);
    tLayout = tiledlayout(rows,cols);
    for i=1:N
        ind = descInd(i);
        leng = descLengths(i);
        nexttile;
        p = plotSegments(data, statistics, "Depth", ind, segmentsBool, false);
        grid on;
        title("Sequence \#"+ind);
        xlabel("Time step")
        ylabel("Depth (m)")
    end
    exportgraphics(longestSeqsFig,'longestSeqs.pdf');
end
```

Remove all sequences that are longer than 650.

```
seqLengthThresh = 650;
mask = seqLengths(:) >seqLengthThresh;
data(mask) = [];
nrIncludedFiles = length(data);
```

See how the histogram of sequence lengths looks like now.

```
seqLengths = extractfield(data, "seqLength");
if exportGraphicsBool
    cutSequences = figureGen(15,20,20);
    histogram(seqLengths);
    grid on;
    ylabel("Counts")
    xlabel("Sequence Lengths")
    exportgraphics(cutSequences,'cutseqs.pdf')
end
```

Examine the Range of the Values before Binning

Look for extremely low or high values to find unplausible signals.

```
for name = channelNames
    channelName = string(name{1});
    maxChannelName = "Max"+name;
    minChannelName = "Min"+name;
    maxPointValues = extractfield(data, maxChannelName);
    minPointValues = extractfield(data, minChannelName);
    maxChannelValue = max(maxPointValues);
    minChannelValue = min(minPointValues);
    statistics.(channelName+"Max") = maxChannelValue;
    statistics.(channelName+"Min") = minChannelValue;
end
```

Discretize the Data

It is important to have the same bin edges across all points in a channel. Otherwise the same symbol can refer to a different range of values and distort the complete dataset. Also differ between the charachteristics of the channel. The values of a channel like depth always range from a value around zero to the maximum depth. Therefore, the edges will be linearly spaced across those values for channels like this, including pulldown force, amperage, frequency, temperature and weights. A different approach is used for the feed rate and both inclinations, because the values of these channels are ranging from a negative to a positive value. Therefore, the symbols should be uniformly spaced in both directions from the mid zero point.

Define a shrinking factor to smallen mid segments for uniform distributions around the mid zero point.

```
midSegmentShrinkFactor = 0.8;
```

Define the characteristics of the channels, 1 for "minmax" distribution, 0 for a uniform distribution around the mid zero point.

```
depthType = 1;
feedRateType = 0;
pullDownForceType = 1;
vibrAmpType = 1;
vibrFreqType = 1;
vibrTempType = 1;
weightNetType = 1;
inclXType = 0;
inclYType = 0;
```

Combine in single array for better handling:

```
allTypes = [depthType; feedRateType; pullDownForceType; vibrAmpType;vibrFreqType;...
```

```
    vibrTempType; weightNetType; inclXType; inclYType];
```

Define the cardinality (number of bins) for each channel.

```
depthCard = 20;
feedRateCard = 5;
pullDownForceCard = 10;
vibrAmpCard = 10;
vibrFreqCard = 10;
vibrTempCard = 10;
weightNetCard = 10;
inclXCard = 5;
inclYCard = 5;
```

Combine in single array for better handling of values:

```
allChannelCards = [depthCard, feedRateCard, pullDownForceCard, vibrAmpCard, vibrFreqCard, vibrTempCard,...
    weightNetCard, inclXCard, inclYCard];
```

Find the global channel min and max values over all points in the data to define binning edges. A uniform linear spaced approach is generally used, except for a zero mean distributed channel. Then we scale the inner edges with the shrinking factor and and the next outer edges with the shrinking factor squared.

```
for i=1:nrChannels
    name = channelNames(i);
    channelName = string(name{1});
    channelCard = allChannelCards(i);
    maxChannelValue = statistics.(channelName+"Max");
    minChannelValue = statistics.(channelName+"Min");
    if allTypes(i) == 0 % distributed around zero
        absoluteMaxChannelValue = max(abs(minChannelValue),abs(maxChannelValue));
        edges = linspace(-absoluteMaxChannelValue,absoluteMaxChannelValue,channelCard+1);
        channelCard = allChannelCards(i);
        shrinkingSegs = floor(channelCard/2);
        shrinkingExponents = linspace(1,shrinkingSegs,shrinkingSegs);
        shrinkingVals = midSegmentShrinkFactor.^(shrinkingExponents);
        shrinkingRels = 1-(shrinkingVals);
        if rem(channelCard, 2) == 0 % even cardinality
            edgeShrinkingFactors = [1,flip(shrinkingRels),1,shrinkingRels,1];
        else % odd cardinality
            edgeShrinkingFactors = [1,flip(shrinkingRels),shrinkingRels,1];
        end
        edges = edges .* edgeShrinkingFactors;
    else % mainly positive distributed
        edges = linspace(minChannelValue, maxChannelValue, channelCard+1);
    end
    means = movmean(edges,2);
    bins = means(2:end);
    for k=1:nrIncludedFiles
        [data(k).(channelName+"HistCounts"), ~, data(k).(channelName+"Tkns")] = ...
        histcounts(data(k).(channelName), edges);
    end
    concatenatedChannelHistCounts = reshape(extractfield(data, channelName+"HistCounts"),channelCard,nrIncludedFiles);
    statistics.(channelName+"HistCounts") = sum(concatenatedChannelHistCounts,2)';
    statistics.(channelName+"Edges") = edges;
    statistics.(channelName+"Means") = means;
    statistics.(channelName+"Bins") = bins;
end
```

Visualize the Token Bins

Draw the lines for the binning edges and a selected discretized sequence.

```
selSequenceNr = 194;
selSequenceLength = data(selSequenceNr).("seqLength");
```

Create a vector with the vertical segment borders.

```
seqSpace = linspace(1,selSequenceLength,selSequenceLength);
seqMeanSpace = movmean(seqSpace,2);
seqMeanSpace(1) = [];
```

Iterate through all channels of the selected sequence.

```
if exportGraphicsBool
    B = figureGen(42,32,17);
    runner = 1;
    for i=1:length(channelNames)
        name = channelNames(i);
```

```
        channelName = string(name{1});
        subplot(length(channelNames),1,i);
        plotSegments(data, statistics, channelName, selSequenceNr, [false;false;false], true);
        channelYLabel = ylabel(channelLabels(runner:runner+1),"FontWeight",'normal');
        channelYLabel.Position(1) = -9;
        runner = runner + 2;
    end
    xlabel("Time (s)")
    saveFileName = strjoin(["HighlightedTokenBins",(segmentNames(segmentsBool))',".pdf'],"");
    exportgraphics(B,saveFileName)
end
```

Visualize the Token Statistics

Initialize a figure with its properties:

```
figWidth = 12;
figHeight = 9;
labelFontSize = 8;
```

Create a tiled layout:

```
if exportGraphicsBool
    TokenStatfigure = figureGen(20,25,15);
    tiledlayout(3,3);
end
```

Plot the counts into a bar chart:

```
if exportGraphicsBool
    for i=1:nrChannels
        nexttile;
        name = channelNames(i);
        channelName = string(name{1});
        channelCard = allChannelCards(i);
        channelCounts = statistics.(channelName+"HistCounts");
        bar(1:channelCard,channelCounts);
        grid on;
        ylim padded;
        title(channelLabels(i*2-1),"FontWeight",'normal');
        ylabel("Counts")
        xlabel("Tokens")
    end
    exportgraphics(TokenStatfigure,'TokenStats.pdf');
end
```

Save the Tokens

Include the defined number of tokens in the name. And save the tokens to the curerent directory.

```
if saveTokensBool
    tkncnt =  num2str(allChannelCards);
    tkncnt = strrep(tkncnt,"    ","  ");
    tkncnt_frmt = strrep(tkncnt,"  ","-");
    saveFileName = strjoin([SiteName,(segmentNames(segmentsBool))',string(seqLengthThresh), ...
        "AllChannelsTokenized",tkncnt_frmt,'.mat'],"");
    save( fullfile( cd, saveFileName), 'data', 'statistics');
end
```

Define Functions

Functions used in this script are implemented below.

Segment Timetable

Use this function to segment the timetable into the defined phases.

```
function segmentTT = segmentSeq(dataTT, segmentsBool)
    segments = dataTT.Properties.CustomProperties.Segments;
    if segmentsBool(1) && segmentsBool(2) && segmentsBool(3) == 1
        segmentTT = dataTT;
    else
        if segmentsBool(1) && segmentsBool(2) == 1
            startTime = dataTT.Time(1);
            endIndex = contains(segments.Name, "Penetration");
            endTime = segments{endIndex, 'EndTime'};
        elseif segmentsBool(2) && segmentsBool(3) == 1
            startIndex = contains(segments.Name, "Penetration");
            endIndex = contains(segments.Name, "Compaction");
            startTime = segments{startIndex, 'StartTime'};
            endTime = segments{endIndex, 'EndTime'};
```

```
        elseif segmentsBool(2) == 1
            startIndex = contains(segments.Name, "Penetration");
            endIndex = contains(segments.Name, "Penetration");
            startTime = segments{startIndex, 'StartTime'};
            endTime = segments{endIndex, 'EndTime'};
        elseif segmentsBool(3) == 1
            startIndex = contains(segments.Name, "Compaction");
            endIndex = contains(segments.Name, "Compaction");
            startTime = segments{startIndex, 'StartTime'};
            endTime = segments{endIndex, 'EndTime'};
        else
            startTime = dataTT.Time(1);
            endIndex = contains(segments.Name, "Penetration");
            endTime = segments{endIndex, 'EndTime'};
        end
    TimeRange = timerange(startTime, endTime, 'closed');
    segmentTT = dataTT(TimeRange,:);
    end
end
```

Custom Plot Function

Create a plot with highlighted segments. The segments included are defined in the "segmentsBool" variable. If "disreteBool" is set true, we convert the numeric sequence into the tokens and plot the bins of the sequence. Use p to define plot properties.

```
function p = plotSegments(data, statistics, channelName, sequenceInd, segmentsBool, discreteBool)
    k = sequenceInd;
    channelData = data(sequenceInd).(channelName);
    channelMin = data(k).("Min"+channelName);
    channelMax = data(k).("Max"+channelName);
    channelRange = channelMax-channelMin;
    padding = 0.1*channelRange;
    seqLength = data(k).seqLength;

    if discreteBool
        channelEdges = statistics.(channelName+"Edges");
        yline(channelEdges, Color=[0.5,0.5,0.5]);
        xline((0:seqLength)+0.5, Color=[0.5,0.5,0.5])
        channelTokens = data(sequenceInd).(channelName+"Tkns");
        for i=1:seqLength
            currentToken = channelTokens(i);
            rectangle(Position=[i-0.5,channelEdges(currentToken),1, ...
                channelEdges(currentToken+1)-channelEdges(currentToken)], FaceColor=[1 1 0.7], EdgeColor=[1 1 0.7]);
        end
        ylim padded;
        if segmentsBool(1) && segmentsBool(2) == 1
            PenetrationStartIndex = data(k).("PenetrationStart");
            xline(PenetrationStartIndex, Color="r");
        end
        if segmentsBool(2) && segmentsBool(3) == 1
            CompactionStartIndex = data(k).("CompactionStart");
            xline(CompactionStartIndex, Color="r");
        end
    else
        if segmentsBool(1) && segmentsBool(2) && segmentsBool(3) == 1
            PenetrationStartIndex = data(k).("PenetrationStart");
            CompactionStartIndex = data(k).("CompactionStart");
            rectangle(Position=[1,channelMin-padding,PenetrationStartIndex, ...
                channelMax-channelMin+2*padding], FaceColor=[0.95 0.95 0.95], EdgeColor=[1 1 1]);
            rectangle(Position=[PenetrationStartIndex,channelMin-padding,CompactionStartIndex-PenetrationStartIndex, ...
                channelMax-channelMin+2*padding], FaceColor=[1 0.95 1],EdgeColor=[1 1 1]);
            rectangle(Position=[CompactionStartIndex,channelMin-padding,seqLength-CompactionStartIndex, ...
                channelMax-channelMin+2*padding], FaceColor=[1 1 0.95], EdgeColor=[1 1 1]);
        elseif segmentsBool(1) && segmentsBool(2) == 1
            PenetrationStartIndex = data(k).("PenetrationStart");
            xline(PenetrationStartIndex, Color="r");
            rectangle(Position=[1,channelMin-padding,PenetrationStartIndex,channelMax-channelMin+2*padding], ...
                FaceColor=[0.95 0.95 0.95], EdgeColor=[1 1 1]);
            rectangle(Position=[PenetrationStartIndex,channelMin-padding,seqLength-PenetrationStartIndex, ...
                channelMax-channelMin+2*padding], FaceColor=[1 0.95 1],EdgeColor=[1 1 1]);
        elseif segmentsBool(2) && segmentsBool(3) == 1
            PenetrationEndIndex = data(k).("PenetrationEnd");
            CompactionStartIndex = data(k).("CompactionStart");
            xline(CompactionStartIndex, Color="r");
            rectangle(Position=[1,channelMin-padding,PenetrationEndIndex,channelMax-channelMin+2*padding], ...
                FaceColor=[1 0.95 1],EdgeColor=[1 1 1]);
            rectangle(Position=[CompactionStartIndex,channelMin-padding,seqLength-CompactionStartIndex, ...
                channelMax-channelMin+2*padding], FaceColor=[1 1 0.95], EdgeColor=[1 1 1]);
        elseif segmentsBool(1) == 1
```

```
            rectangle(Position=[1,channelMin-padding,seqLength,channelMax-channelMin+2*padding], ...
                FaceColor=[0.95 0.95 0.95], EdgeColor=[1 1 1]);
        elseif segmentsBool(2) == 1
            rectangle(Position=[1,channelMin-padding,seqLength,channelMax-channelMin+2*padding], ...
                FaceColor=[1 0.95 1],EdgeColor=[1 1 1]);
        elseif segmentsBool(3) == 1
            rectangle(Position=[1,channelMin-padding,seqLength,channelMax-channelMin+2*padding], ...
                FaceColor=[1 1 0.95], EdgeColor=[1 1 1]);
        end
        ylim tight;
    end
    hold on;
    p = plot(channelData);
    xlim tight;
    set(gca, "Layer", "top");
    hold off;
end
```

References

[1] C. Manning and H. Schutze. *Foundations of Statistical Natural Language Processing*. Foundations of Statistical Natural Language Processing. MIT Press, 1999. ISBN: 9780262133609.

[2] Karen Sparck Jones. “A statistical interpretation of term specificity and its application in retrieval”. en. In: *Journal of documentation* 28.1 (1972), pp. 11–21.

[3] Yoshua Bengio et al. “A Neural Probabilistic Language Model”. In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[4] Ashish Vaswani et al. “Attention is All You Need”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[5] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).

[6] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: `1810.04805 [cs.CL]`.

[7] Anika Terbuch et al. “Detecting Anomalous Multivariate Time-Series via Hybrid Machine Learning”. English. In: *IEEE transactions on instrumentation and measurement* 72.2023 (Jan. 2023). Publisher Copyright: © 1963-2012 IEEE. ISSN: 0018-9456. DOI: `10.1109/TIM.2023.3236354`.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[9] Alan M. Turing. “Computing Machinery and Intelligence”. In: *Mind* LIX.236 (1950), pp. 433–460.

[10] Tom M. Mitchell. *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997. ISBN: 0070428077.

[11] Feng-hsiung Hsu and Jon Kleinberg. 2022.

[12] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. DOI: `10.1017/CBO9781107298019`.

[13] Yves Grandvalet and Y. Bengio. “Semi-supervised Learning by Entropy Minimization”. In: vol. 17. Jan. 2004.

[14] Yuxi Li. *Reinforcement Learning in Practice: Opportunities and Challenges*. 2022. arXiv: `2202.11296 [cs.LG]`.

[15] R. A. Fisher. “The Use of Multiple Measurements in Taxonomic Problems”. In: *Annals of Eugenics* 7.7 (1936), pp. 179–188.

[16] Kevin P Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.

[17] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011. ISBN: 9780123814807.

[18] M. Cord and P. Cunningham. *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*. Cognitive Technologies. Springer Berlin Heidelberg, 2008. ISBN: 9783540751717.

[19] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Comput. Surv.* 41.3 (July 2009). ISSN: 0360-0300. DOI: `10.1145/1541880.1541882`.

[20] Lorenzo Ciampiconi et al. *A survey and taxonomy of loss functions in machine learning*. 2023. arXiv: `2301.05579 [cs.LG]`.

[21] E. Alpaydin. *Introduction to Machine Learning, fourth edition*. Adaptive Computation and Machine Learning series. MIT Press, 2020. ISBN: 9780262043793.

[22] The MathWorks Inc. *MATLAB Documentation on Plot receiver operating characteristic (ROC) curves and other performance curves: Plot ROC Curves for Multiclass Classifier Example*. 2021. URL: `https://www.mathworks.com/help/stats/rocmetrics.plot.html` (visited on 11/01/2023).

[23] C.J. Van Rijsbergen. *Information Retrieval*. Butterworths, 1979. ISBN: 9780408709293.

[24] Nancy Chinchor. "MUC-4 Evaluation Metrics". In: *Fourth Message Understanding Conference (MUC-4): Proceedings of a Conference Held in McLean, Virginia, June 16-18, 1992*. 1992.

[25] Yutaka Sasaki. "The truth of the F-measure". In: *Teach Tutor Mater* (Jan. 2007).

[26] Aryan Jadon, Avinash Patil, and Shruti Jadon. *A Comprehensive Survey of Regression Based Loss Functions for Time Series Forecasting*. Nov. 2022. DOI: `10.48550/arXiv.2211.02989`.

[27] Michael K. Tippett et al. "Sources of Bias in the Monthly CFSv2 Forecast Climatology". In: *Journal of Applied Meteorology and Climatology* 57.5 (2018), pp. 1111–1122.

[28] WarrenS McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biology* 5.4 (Dec. 1943), pp. 115–133. ISSN: 0007-4985. DOI: `10.1007/bf02478259`.

[29] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408.

[30] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[31] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press, 2014, pp. I–XVI, 1–397. ISBN: 978-1-10-705713-5.

[32] Simon S. Haykin. *Neural networks and learning machines*. Third. Upper Saddle River, NJ: Pearson Education, 2009.

[33] B. Mehlig. *Machine Learning with Neural Networks: An Introduction for Scientists and Engineers*. Cambridge University Press, 2021. ISBN: 9781108849562.

[34] Gernot Steiner. "Optimization of Autoencoders for Anomaly Detection in Multivariate Real-Time Measurement Data Acquired From Production Machinery". English. Montanuniversitaet Leoben, 2022. DOI: 10.34901/mul.pub.2023.70.

[35] Charu C. Aggarwal. "Neural Networks and Deep Learning". In: 2023.

[36] Alex Graves. "Neural Networks". In: *Supervised Sequence Labelling with Recurrent Neural Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 15–35. ISBN: 978-3-642-24797-2. DOI: 10.1007/978-3-642-24797-2_3.

[37] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[38] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[39] Y. Goldberg and G. Hirst. *Neural Network Methods for Natural Language Processing*. Synthesis digital library of engineering and computer science. Morgan & Claypool, 2017. ISBN: 9781627052986.

[40] E.M. Bender. *Linguistic Fundamentals for Natural Language Processing: 100 Essentials from Morphology and Syntax*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2013. ISBN: 9781627050128.

[41] Brian Fisher. "Illuminating the Path: An R&D Agenda for Visual Analytics". In: Jan. 2005, pp. 69–104. ISBN: 0769523234.

[42] H. Kučera and W.N. Francis. *Computational Analysis of Present-day American English*. Brown University Press, 1967. ISBN: 9780870571053.

[43] C. Fellbaum. *WordNet: An Electronic Lexical Database*. Language, speech, and communication. MIT Press, 1998. ISBN: 9780262061971.

[44] M. Garg, S. Kumar, and Khader Jilani Saudagar. "Natural Language Processing and Information Retrieval: Principles and Applications". In: 1 (2023).

[45] James Milton and Jeanine Treffers-Daller. "Vocabulary size revisited: the link between vocabulary size and academic achievement". In: *Applied Linguistics Review* 4.1 (2013), pp. 151–172. DOI: doi:10.1515/applirev-2013-0007.

[46] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2016. arXiv: 1508.07909 [cs.CL].

[47] Philip Gage. "A new algorithm for data compression". In: *The C Users Journal archive* 12 (1994), pp. 23–38.

[48] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. arXiv: 1609.08144 [cs.CL].

[49] D. Jurafsky and J.H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall series in artificial intelligence. Pearson Prentice Hall, 2009. ISBN: 9780131873216.

[50] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[51] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014.

[52] Alex Graves. *Generating Sequences With Recurrent Neural Networks*. 2014. arXiv: 1308.0850 [cs.NE].

[53] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: 1310.4546 [cs.CL].

[54] Alexander Wettig et al. *Should You Mask 15 percent in Masked Language Modeling?* 2023. arXiv: 2202.08005 [cs.CL].

[55] Keller UK ltd. *Vibro compaction*. 2023. URL: https://www.keller.co.uk/expertise/techniques/vibro-compaction (visited on 11/13/2023).

[56] Anika Terbuch, Paul O'Leary, and Peter Auer. "Hybrid Machine Learning for Anomaly Detection in Industrial Time-Series Measurement Data". English. In: *I2MTC 2022 - IEEE International Instrumentation and Measurement Technology Conference*. Conference Record - IEEE Instrumentation and Measurement Technology Conference. Publisher Copyright: © 2022 IEEE.; 2022 IEEE International Instrumentation and Measurement Technology Conference, I2MTC 2022 ; Conference date: 16-05-2022 Through 19-05-2022. United States: Institute of Electrical and Electronics Engineers, 2022. DOI: 10.1109/I2MTC48687.2022.9806663.

[57] Stefan Herdy. "Machine Learning in the Context of Time Series". English. Montanuniversitaet Leoben, 2020.

[58] Maria Haider. "Machine Learning and KPI Analysis applied to Time-Series Data in Physical Systems: Comparison and Combination". English. Montanuniversitaet Leoben, 2021.

[59] "Anika Terbuch". ""LSTM Hyperparameter Optimization: Impact of the Selection of Hyperparameters on Machine Learning Performance when applied to Time Series in Physical Systems"". "English". "Montanuniversitaet Leoben", 2021.

[60] Anika Terbuch et al. "Quality monitoring in vibro ground improvement: A hybrid machine learning approach". English. In: *Geomechanics and tunnelling* 15.2022.5 (Oct. 2022). Publisher Copyright: © 2022, Ernst und Sohn. All rights reserved., pp. 658–664. ISSN: 1865-7362. DOI: 10.1002/geot.202200028.