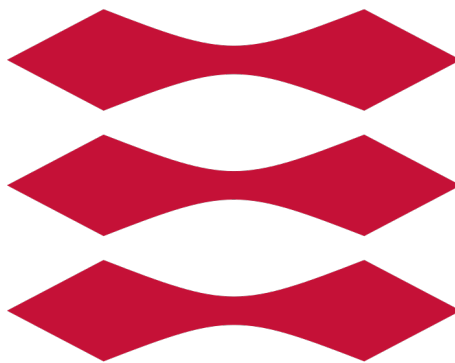


TECHNICAL UNIVERSITY OF DENMARK

**DTU Compute**

Department of Applied Mathematics and Computer Science

DTU



BACHELOR THESIS

---

**Segmentation of medical records with natural  
language processing tools**

Revised 11th of June 2021

---

**AUTHOR:**

Florian Philipp Stilz (s183193@student.dtu.dk)

**SUPERVISOR:**

Ole Winther (olwi@dtu.dk)

## **Abstract**

The correct diagnosis of rare diseases is a great challenge in clinical practice since individual cases are rare and medical personnel are often confronted for the first time with an unfamiliar symptom. Natural Language Processing enables a new approach to improve the clinical diagnosis with the startup "findzebra" being specialized for search engines in the area of rare diseases.

In this bachelor thesis a Named Entity Recognition is performed in order to improve the search results of "findzebra" by structuring medical articles/text data in a favourable way.

In the theoretical part the underlying theory and concepts concerning the model architecture and model optimization for the empirical part are displayed. The focus is set on the most recent state of the art models, namely BERT. Additionally, the labelling procedure and the desired properties of the dataset are explained.

In the experimental part of the project both a labelled dataset has been created as well as a model for the Named Entity Recognition has been designed. For the implementation of the model the transformer library has been used and the annotation of the data has been performed by Prodigy. As a key result it is shown in this thesis that the final BioBERT model delivers the best results, capable of providing relevant medical summaries. Predictions are accurate for articles specialized on Gaucher and Fabry disease abstracts as far as the training set vocabulary is concerned. Further improvements will be needed to extend the performance on a greater variety of different types of disease abstracts.

## PreFace

This thesis has been prepared at DTU Compute, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Bachelor of Science in Engineering (General Engineering).

Nærum, June 11th, 2021

A handwritten signature in black ink, appearing to be 'F. Stilz', written in a cursive style.

Florian Philipp Stilz, s183193

## Acknowledgments

First and foremost, I would like to thank my supervisor Ole Winther for giving me the opportunity to participate in an extremely exciting research project, for his major contributions in providing valuable labelled data, and for supporting me throughout the project. In addition, I would like to acknowledge Valentin Lievin for his support, major contributions regarding the data, and availability for answering any questions I had regarding the field of Natural Language Processing. Finally, I would like to thank the Technical University of Denmark (DTU) and in particular the Department of Compute for providing access to their servers and other relevant material.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Machine Learning</b>	<b>3</b>
2.1	General Machine Learning Concepts . . . . .	4
2.1.1	Learning Strategies . . . . .	4
2.1.2	Performance evaluation and Optimization . . . . .	4
2.1.3	Supervised Learning . . . . .	5
2.1.4	Generalization . . . . .	6
2.1.5	Regularization . . . . .	8
2.2	Neural Networks . . . . .	9
2.2.1	Activation Functions . . . . .	11
2.2.2	Optimization . . . . .	12
2.2.2.1	Gradient Descent . . . . .	12
2.2.2.2	Backpropagation . . . . .	13
2.2.2.3	Adam Optimizer . . . . .	14
2.2.3	Regularization for Neural Networks . . . . .	15
2.3	Convolutional Neural Network . . . . .	15
2.4	Attention . . . . .	16
2.4.1	Self-attention and Multi-Head Attention . . . . .	16
<b>3</b>	<b>Natural Language Processing with Deep Learning</b>	<b>18</b>
3.1	Tokenization and Input Embedding . . . . .	18
3.1.1	Tokenizer . . . . .	18
3.1.2	Special Tokens . . . . .	19
3.1.3	Input Embedding . . . . .	19
3.1.4	Padding . . . . .	20
3.1.5	Attention Mask and Segment Embeddings . . . . .	20
3.2	NLP Architectures . . . . .	21
3.2.1	Encoder-Decoder Architecture . . . . .	21
3.2.2	Transformer . . . . .	23
3.3	Named Entity Recognition: an NLP Framework . . . . .	25
3.4	Embeddings . . . . .	25

3.4.1	Word2Vec and GloVe Embeddings . . . . .	25
3.4.2	Contextualized Embeddings with ELMo . . . . .	26
3.5	BERT . . . . .	26
3.6	BERT Models . . . . .	28
3.6.1	Original BERT . . . . .	28
3.6.2	DistilBERT . . . . .	28
3.6.3	BioBERT . . . . .	29
3.6.4	Bio+ClinicalBERT . . . . .	29
<b>4</b>	<b>Data and Models</b>	<b>30</b>
4.1	Annotation . . . . .	32
4.2	Positional Tagging . . . . .	33
4.3	Models . . . . .	34
4.3.1	Basic Model Architecture for Named Entity Recognition . . . . .	34
4.3.2	Spacy CNN Baseline Model . . . . .	34
4.3.3	Final Model . . . . .	34
<b>5</b>	<b>Resources and Tools</b>	<b>36</b>
5.1	Libraries . . . . .	36
5.2	GPU Cluster . . . . .	36
5.3	Evaluation Metrics . . . . .	37
<b>6</b>	<b>Implementation</b>	<b>39</b>
6.1	Pre-Processing the Data . . . . .	39
6.2	Data Encoding . . . . .	41
6.3	Training . . . . .	42
6.4	Hyperparameter Optimization . . . . .	43
<b>7</b>	<b>Results</b>	<b>45</b>
<b>8</b>	<b>Conclusion and Outlook</b>	<b>48</b>
	<b>Bibliography</b>	<b>50</b>

## Abbreviations

<b>Adam</b>	Adaptive Moment Estimation
<b>BERT</b>	Bidirectional Encoder Representations from Transformers
<b>CNN</b>	Convolutional Neural Network
<b>ELMo</b>	Embeddings from Language Models
<b>LSTM</b>	Long Short Term Memory
<b>MLM</b>	Masked Language Modelling
<b>ML</b>	Machine Learning
<b>MSE</b>	Mean Squared Error
<b>NER</b>	Named Entity Recognition
<b>NLP</b>	Natural Language Processing
<b>NNL</b>	Negative Log-Likelihood
<b>NSP</b>	Next Sentence Prediction
<b>OOV</b>	Out of Vocabulary
<b>RNN</b>	Recurrent Neural Network
<b>Seq2Seq</b>	Sequence to Sequence
<b>SGD</b>	Stochastic Gradient Descent
<b>WPM</b>	WordPiece Model

# Chapter 1

## Introduction

This bachelor project is dealing with search routines that help facilitate medical doctors' difficulties matching patient cases with possible diagnoses for very rare diseases in many places of the world. The project is a cooperation with the startup company "findzebra", a provider of such a search routine. Specifically, this thesis is trying to improve search results by performing a segmentation task on medical articles. The segmentation task, also called Named Entity Recognition (NER), is extracting valuable information from medical articles corresponding to pre-defined categories.

A NER is a natural language processing (NLP) task which identifies entities and assigns these to predefined classes/labels. Entities in this case can range from single words up to a longer sequence of text within a single sentence. An example of an entity can be shown in the following sentence: "A 37-year old woman has high fever", where "37-year old" can be classified to the class *Age* and "woman" to the class *Gender*. The great benefit of being able to correctly categorize certain classes will help to structure large amounts of data and hereby pre-process the data in a way that facilitates their use afterwards for all sorts of other tasks.

The goal of this project is to bring some structure into medical articles, more precisely their abstracts, so that their valuable information can be extracted and further efficiently be used to better match certain documents with given patient cases. An example for this could be to know whether a certain article covers a patient case for a specific age group (e.g. child, adult) which helps with quickly filtering documents depending on the patient age. Classification will make search results much more flexible as clients can filter documents for their needs. Thus it will be possible to provide quick and easy overviews over documents by returning matches for given categories in a particular document.

The newest models that greatly improve performances in all sorts of NLP tasks are transformers. Hereby, specifically the BERT model developed by Google and several modifications of



BERT manage to break record after record also for the NER task. BERT provides contextualized word representations which is of great value for any task in NLP like the NER performed in this project. Therefore, BERT is the main module of this project's final architecture.

The final dataset used in the empirical part of the project consists of 186 abstracts for rare diseases (Gaucher and Fabry disease). The abstracts are partitioned into 11 different categories, like e.g. DIAGNOSIS which identifies diagnosed diseases (chapter 4). The labelling process is manually performed using the annotation tool "Prodigy". Currently, the dataset is suffering from ambiguities due to the similarity of some of the defined classes, which ultimately limits the learning process of a model. Additionally, the dataset is fairly small as it is individually annotated and specialized on only a small set of different types of disease abstracts.

The thesis is structured as follows: The first part presents some basic machine learning concepts to provide the reader with the core foundations of learning as well as more advanced machine learning like Neural Networks in order to get a better understanding on how the used models work. The second part dives into NLP in combination with the models in order to understand how models have to be designed for this particular field of application and hereby specifically for BERT. The third part will be a brief overview of utilized tools and resources for the project. The succeeding part is a discussion of the dataset and what is planned to achieve with it in more detail, followed by the implementation, and last but not least the result comparison of the different models.

## Chapter 2

# Machine Learning

This chapter serves as a guide to the most important concepts of **machine learning (ML)** and is designed to help understand how to create an algorithm for analyzing big amounts of data in order to extract important information like in this project. It starts with the most basic principles and ends with an overview of Neural Networks and even some Deep Learning, like e.g. the attention mechanism, which is needed to understand the transformer model BERT.

ML deals with the learning process of algorithms that use data and experience to learn and improve over time. The idea behind ML is to have an algorithm that, like humans, starts without knowing anything initially, but gradually learns by observations and patterns. In contrast to many other algorithms ML is able to improve without receiving any new instructions by the programmer and therefore is able to improve on a certain task by itself. In general a machine model represents a mathematical function  $f(w, x)$  parameterized by parameters  $w$ , where  $x$  is the input, the data observations. The model then uses the input and its parameters to make a prediction  $\hat{y}$ . The performance of a model is evaluated by a loss function. The goal of ML is to improve the performance of the model  $f(x)$  for a given task. This is achieved by training the model, also called "training phase". The model gets optimized during training by finding the parameters  $w$  for which the model achieves the best performance. This is obtained by optimizing the parameters with respect to the loss function.

## 2.1 General Machine Learning Concepts

This section leads through basic concepts like learning strategies, evaluation and optimization, generalization, and regularization.

### 2.1.1 Learning Strategies

In the following the two most used learning strategies, which are **Supervised Learning** and **Unsupervised Learning** are presented.

The most common way of learning in ML is through supervision. In Supervised Learning the data used for the model contains both its input  $x$  as well as the ground truth value  $y$  of the data for the corresponding problem. The ground truth value is used as the knowledge from which the model will learn. The goal in Supervised Learning is to predict  $y$  from  $x$  in the following way, where  $\epsilon$  represents a noise term also called bias. It follows a standard normal distribution [2].

$$y = f(x, w) + \epsilon \quad (2.1)$$

$y$  is the ground truth value,  $f$  is the mathematical function, where  $x$  is the input,  $w$  represents the parameters, and  $\epsilon$  is the noise term. The attempt in Supervised Learning is to approximate the mapping that maps  $x$  to  $y$ . The noise term shall take the fact into account that the predictions are usually not identical to the ground truth for observation  $x$ .

The main difference between Supervised Learning and Unsupervised Learning is that there are no ground truth values in Unsupervised Learning. The data as such only consists of the input/observations  $x$ . Unsupervised Learning helps in finding hidden patterns within the data and is used in cases where there is no labelled data since it is usually too time consuming to produce labelled data. It basically tries to discover the labelling from the data itself. An example of an unsupervised task is **Clustering**, where the data gets split into different clusters based on the similarities of the data points.

### 2.1.2 Performance evaluation and Optimization

The performance of a model is evaluated using a loss function. The loss function measures the difference between the predicted result and the expected result. The general aim in ML is to minimize the loss and thus try to let the predictions converge towards the expected results. This is done by finding the parameters that yield the least possible loss. For this the partial derivatives of the parameters with respect to the loss function are used to gradually find the global minimum of the loss function (more details in section 2.2.2.1).

### 2.1.3 Supervised Learning

This part goes more in depth with Supervised Learning as it is actively used in the empirical part.

In Supervised Learning there are two sub tasks namely **Classification** and **Regression**. Classification attempts to classify data points into predefined classes and thus returns discrete values ( $y$  in equation 2.1) where each value stands for a certain class. Regression predicts numerical values (imagine  $y$  in equation 2.1) and thus has a continuous output. An example for classification is the Categorization of the temperate seasons, while an example for regression is the prediction of weather temperatures.

In order to understand how the model learns from the knowledge contained in the data it is vital to look at the concept of the loss function. The general pipeline of the training phase is to feed the input into the model and let the model make its prediction. This prediction will then be compared to the ground truth value of the input. The loss function in Supervised Learning computes the discrepancy between the ground truth and the prediction. For regression a commonly used loss function is the mean squared error (MSE) which calculates the error in the following way:

$$L_{MSE}(w) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i, w))^2 = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (2.2)$$

The MSE is defined as the average of the squared difference between the prediction and the ground truth value with  $N$  being the length of the dataset. The benefit of squaring the error before summing it up is to take the different error signs into account. MSE is simply preventing a deflation of the error where positive signs and negative signs could otherwise offset each other

Another common loss function which is used for multilabel classification is the Categorical Cross-Entropy Loss. This function is defined as follows:

$$L_{Cross-Entropy}(w) = - \sum_i y_i \log(f(x_i, w)) = - \sum_i y_i \log(\hat{y}_i) \quad (2.3)$$

It is suited for multilabel Classifications in which the aim is to model the likelihood  $p(y|x)$ . The goal is to find the parameters that maximize the Log-likelihood or minimize the negative Log-likelihood (NNL). The output of the model will then produce a probability distribution over a certain set of classes using a softmax function (more detail see section 2.2.1). The Categorical Cross-Entropy loss compares the predicted class probabilities with the ground truth class probabilities and thus is comparing the difference in probability distributions. In most cases the ground truth value will be a one-hot vector since only one class will be correct and thus it has the value "1" and all other classes contain the value "0". Cross-Entropy uses the NNL in order

to find the difference in distributions. For each class  $i$  of the class set the formula computes the product between the log of the predicted class with the ground truth probability and sums up all the results for each class per data point. When understanding how the log function behaves it quickly becomes clear that the loss of this function will be greater the smaller the predicted probability compared to the true probability. For example the ground truth probability for class 1 is 100% and the probability of the predicted class is fairly low like e.g. 5% then the loss will be greater than in the case where the predicted probability for class 1 is 80%.

## 2.1.4 Generalization

In general, in ML the data set is not entirely used for training the model. The dataset is split into 2 or 3 different sets. The entire dataset consists of the training data and the test data. The test data is used for evaluating the final model's result. It has not been used for training and thus is unknown to the model itself which is essential because it otherwise will not evaluate on how well the model works when seeing new data and, if used for training, is biased towards it. Why this is a problem can be shown easily by imagining a very complex model with many parameters which will be able to better fit the data it is trained upon than a comparatively simple model. However, this might eventually lead to not being able to work well for the complex model on data points outside of the training set. Instead of having a model that is able to fit well to the training set it is desired to get a model which is able to recognize the general trend of the data out of the sample training data set and thus also generalizes better to unseen data. Additionally, there might be a third set of data which is called validation data and that is used for optimizing the hyperparameters of the learning process. Hyperparameters will be explained later.

Using a loss function these two data sets will produce two different errors called  $E_{train}$  training error and  $E_{test}$  test error, respectively. In general, the consideration of both the training and the test error is to take both the training of the model as well as the noise term into account. For example only considering the training error is causing the issue that the most complex model will always perform best since it is most able to adapt to the data. However, it might not be able to follow the general trend of the data which is important since each observation contains a bias (see above). Overcomplicated models might overfit due to the bias in data. Therefore the test error is taken into consideration and it shows how well the model generalizes to new/unseen data. The following two equations show both errors using MSE.

$$E_{train}(w) = \frac{1}{N_{train}} \sum_{i \in D_{train}} (y_i - f(x_i, w))^2 \quad (2.4)$$

$$E_{test}(w) = \frac{1}{N_{test}} \sum_{j \in D_{test}} (y_j - f(x_j, w))^2 \quad (2.5)$$

$D_{train}$  is the training data and  $D_{test}$  is the test data, respectively.

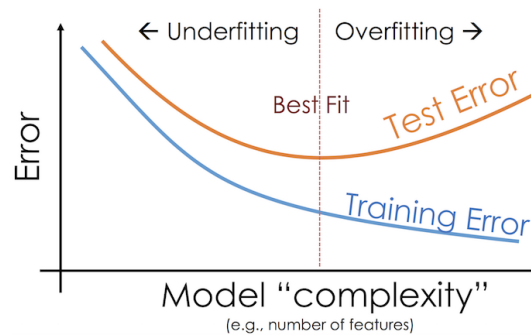


Figure 2.1: Relation between training and test error to model complexity <sup>1</sup>

To round it up a third error is introduced called the generalization error. The problem with the test error is that it only displays a random selection of data used for testing the model and therefore the error will vary depending on how the particular test data looks like. The generalization error is the average of several test errors needed and can be seen as an average test error.

At this point cross-validation comes into play by trying to quantify the generalization error. There are three different algorithms for cross-validation. The first and most straightforward one is the **Hold-out method** where the data set is split in one training set and one test set. Thus the generalization error is set to be more or less equal to the test error since the test error is only computed once using the single test set.

The second approach is the **K-fold cross-validation** which splits the entire dataset into  $K$  parts. The goal of this algorithm is to use every data point for testing exactly once. This leads to  $K$ -iterations in which  $\frac{1}{K}$  of the data is used for testing and the rest for training. The generalization error is then computed as the average of the different test errors where each error is weighted by the amount of data points in proportion to the total amount of data points in the data.

The last algorithm is an extreme version of the  $K$ -fold cross-validation. It covers the case where  $K$  is equal to  $N$ , the size of the data set. This in return means that the size of the test set per iteration is 1. The generalization error is then computed in the same manner as in  $K$ -fold cross-validation. The name of this special algorithm is **Leave-one-out**.

<sup>1</sup>Adapted from [https://www.textbook.ds100.org/ch/20/bias\\_cv.html](https://www.textbook.ds100.org/ch/20/bias_cv.html)

Each algorithm has advantages and disadvantages. On the one hand the estimates of the generalization error improve from approach to approach in the same order as listed above, while on the other hand the computational complexity increases since more and more models have to be trained. The selection of which algorithm to use is therefore very dependent on the needs and tools available like data set size and computational power.

### 2.1.5 Regularization

**Regularization** is a technique for controlling model complexity and aims at improving generalization. There are many different ways of regularizing during training and the chosen method, among others, depends on the specific model type. The following part introduces a few examples out of the big toolbox of regularization strategies. Additionally, Dropout will be mentioned in section 2.2.3 but for a better understanding this requires some knowledge about neural networks (see section 2.2).

Some of the different techniques to regularize a model are **L2-regularization** or also called **weight decay**, **L1-regularization**, and having more training data. In equation (2.6) the loss function is shown with an addition of the L2-regularization term which uses the square of the sum of all weights.

$$L(w) = L_0(w) + \frac{\lambda}{2N} \sum_w w^2 \quad (2.6)$$

In this equation  $N$  represents the size of the training data,  $L_0$  represents a normal loss function (see section 2.1.3) without a regularization term, and  $\lambda$  is the regularization parameter. It eventually leads to preferring smaller weights in order to reduce the total loss. This regularization is therefore a trade off between having small weights on one side and minimizing the overall loss on the other side as bigger weights increase the final loss via the regularization term. An example of how the regularization parameter impacts the training and test error see Figure 2.2.

L1-regularization sums up the absolute value of the weights in comparison to the sum of squares in L2-regularization. The advantage of this is that many weights will become 0 and thus reduce the amount of features. The disadvantage is that the difference in loss for bigger weight changes is much smaller than for L2.

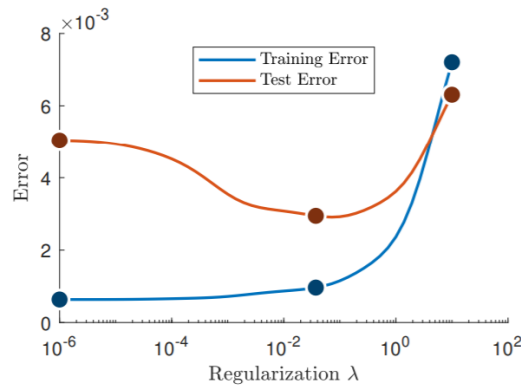


Figure 2.2: Example for different regularization parameters concerning the test and training error <sup>2</sup>

A simple strategy to improve generalization is to use bigger sets of training data as it will naturally cover a greater proportion of the entire data. However, this depends on the amount of data available and is often not possible as all datapoints are already used for both training and testing.

The regularization parameter used for L1- and L2-regularization is a typical hyperparameter. A hyperparameter is a parameter which does not have a direct impact on the model's performance but on the learning process itself. Therefore the parameter is needed for training the model but is not actually part of the model and thus does not get tuned together with the model parameters. It is not immediately clear what the value  $\lambda$  will be since it is task and model dependent. The way to find the best regularization parameter  $\lambda$  is, like with other hyperparameters, by using several values for it and by using cross-validation to perform a model selection.

## 2.2 Neural Networks

This section presents an introduction to neural networks and looks at the **feed-forward neural network (FFNN)** which is the most basic neural network. This section serves as an introduction to better understand the workflow of neural networks which will be the basis of the following chapters. Specific focus will be placed on how the models work from a mathematical point of view and how they learn.

Neural networks are a special type of machine learning models. They are inspired by the biological neural networks. In general, a neural network consists of many different neurons organised in layers where each neuron is related to other neurons via weighted connections.

<sup>2</sup>Adapted from [2]



The information flow proceeds from one layer into the next, starting from the input layer over to one or several hidden layers and finally all the way up to the output layer. In FFNN neurons pass on their information solely to the succeeding layer. In addition each neuron in the hidden layers has a non-linear activation function. This is essential since otherwise the entire network could be shrunk down to a single linear equation. Therefore, these activation functions are exactly what makes neural networks so powerful when deployed correctly because they make it possible to stack layer after layer and thus make it possible to represent complex patterns within the data. The **forward pass** through the neural network works in the following way:

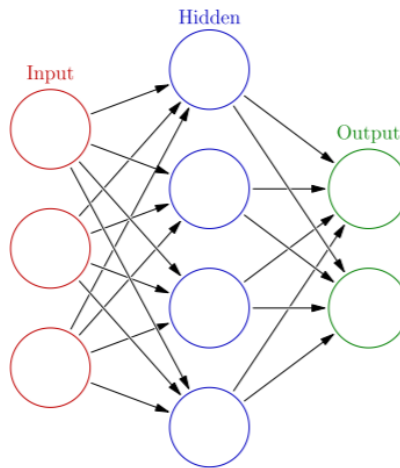


Figure 2.3: FFNN with one hidden layer, two input neurons, 4 hidden neurons, and 2 output neurons.<sup>3</sup>

First, the observation  $x$  is inserted into the input layer where each neuron in the input layer stands for one feature of the observation. For example if the used dataset consists of 3 features, temperature, humidity, and atmospheric pressure, then the input layer will consist of three neurons (see Figure 2.3) where the temperature of observation  $x_1$  is inserted in neuron 1, the humidity of observation  $x_1$  in neuron 2, and atmospheric pressure of observation  $x_1$  in neuron 3. These inputs are then transferred via the connections to the next layer where the weights assigned to the connections are multiplied with the output value of the neurons. Finally, before being further transferred from layer two to the next layer (either hidden layer 2 or output layer) an activation function is performed on the neurons. This procedure continues from layer to layer until the last/output layer has been computed. The general output of a neuron  $j$  in layer  $l$  can be described like the following:

$$a_j^l = act(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \quad (2.7)$$

---

<sup>3</sup>Adapted from [2]

Worth mentioning is that  $w_{jk}^l$  is the weight between neuron  $k$  in layer  $l - 1$  and neuron  $j$  in layer  $l$ . The activation output from neuron  $k$  in layer  $l - 1$  is described by  $a_k^{l-1}$  and  $b_j^l$  is simply the noise term from neuron  $j$  in layer  $l$ .  $act$  describes the activation function for the particular neuron.

### 2.2.1 Activation Functions

There are 4 main non-linear activation functions in neural networks. The first one is the **sigmoid function**, which quickly saturates to the extreme values with the range between 0 and 1:

$$act(x) = \sigma(x) = \frac{1}{1 + \exp x} \quad (2.8)$$

The second activation function is the **tanh function**, which is simply a linear transformation of the sigmoid function and squashes in between -1 and 1.

$$act(x) = \tanh(x) = \frac{\exp x - \exp -x}{\exp x + \exp -x} \quad (2.9)$$

The third activation function is an extremely simple but effective one. Therefore, it is the most frequently used activation function at the moment by mapping every positive value as it is and every negative input value as 0. Its name is **ReLU** which means **rectified linear unit**.

$$act(x) = \max(0, x) \quad (2.10)$$

However, ReLU has its own problem which is the so called dying ReLU problem. The dying ReLU problem occurs when outputs of the neural network get stuck at 0 and are not able to recover. A solution for this particular problem is the Parametric ReLU (PReLU). In PReLU a leak coefficient  $a$  is used which is a learned parameter that is able to adjust the ReLU function in the following manner:

$$act(x) = \max(x, ax) \quad (2.11)$$

The last function is the **softmax function** which like the sigmoid function maps the input into the range between 0 and 1. Additionally, it divides each output by the sum of all the other outputs and thus is well suited for classification tasks since it creates a discrete probability distribution [3].

$$act(x) = softmax(x_i) = \frac{\exp x_i}{\sum_{j=1}^k \exp x_j} \quad (2.12)$$

## 2.2.2 Optimization

Optimization of a neural network covers the part of how a neural network actually studies and learns patterns from the data. This subsection focuses on the mathematics behind the learning of neural networks. The first part will be to look at an optimization algorithm named **Gradient descent**. Next the method of **Backpropagation** will be explained which is required for the gradient descent to work. Last but not least the **Adam Optimizer** which is an extension of SGD will be presented.

### 2.2.2.1 Gradient Descent

In order to make a neural network learn and improve it has to be trained. Following the supervised learning paradigm we consider the case where the data set contains the observations  $x$  and the ground truth values  $y$ . As described earlier the model is evaluated on its own performance with the help of a loss function. The goal again is to minimize the loss. Minimizing the loss can be a quite difficult task considering the potential complexity of a neural network with many parameters left to tune. The strategy to find the loss function's minimum is to use gradient descent. Gradient descent achieves this goal by tweaking the function's parameters in a stepwise manner. In gradient descent the change of the overall loss is related to the change of the function's parameters which can be depicted with the gradient vector as can be seen in equation 2.13. The parameters are the weights and the biases of the neural network and the gradient vector consists of the partial derivatives with respect to the weights and biases (2.13) [1]:

$$\nabla L \equiv \left( \frac{\partial L}{\partial w_1}, \dots, \frac{\partial L}{\partial w_k}, \dots, \frac{\partial L}{\partial b_1}, \dots, \frac{\partial L}{\partial b_l} \right)^T \quad (2.13)$$

This is intuitive since a change in the parameters will naturally lead to a change in the function itself. To make the change of the loss function ( $\nabla L$ /gradient vector) negative and thus reduce the loss a new parameter  $\eta$  is introduced which represents the learning rate. The learning rate defines the strength in which the parameters are reduced per time step:

$$\Delta b_l = b_l - \eta \nabla L \quad (2.14)$$

$$\Delta w_k = w_k - \eta \nabla L \quad (2.15)$$

Together with the gradient the learning rate defines the step size per iteration. By updating the parameters step by step the loss will be reduced by moving in the opposite direction of its gradient in an attempt to find the global minimum. Choosing the ideal learning rate can be complicated since having the learning rate too high might cause the change in  $L$  to be greater than 0 and thus might fail to reach the global minimum. Alternatively, if the learning rate is too small it will take a long time to reach the minimum since more computations than needed for a stronger learning rate will be required.

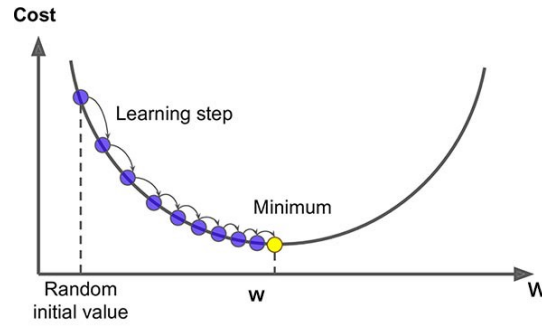


Figure 2.4: Gradient descent with loss function on y-axis and parameters on x-axis <sup>4</sup>

### Stochastic Gradient Descent:

Although gradient descent provides a solution in terms of how to optimize our model it is highly computationally demanding. A way to improve the running time of gradient descent is to use the stochastic gradient descent (SGD). SGD randomly picks one data point out of the entire training set in order to compute the gradient instead of using the entire training set as is done in the normal gradient descent algorithm. This greatly improves the running time since it only performs a fraction of the computations needed otherwise when using the entire dataset. However, it is rather an estimation than the precise gradient.

#### 2.2.2.2 Backpropagation

Backpropagation is an important part of the learning procedure as it computes the gradient of the loss function with respect to its parameters, which is needed for the gradient descent algorithm and thus for optimizing the model.

The first step of the backpropagation is to compute a quantity, called error  $\delta$ , which can in the later phase be related to the partial derivatives of the loss function with respect to the weights and the biases. Equation (2.16) shows the error for neuron  $j$  in the output layer. The error of a particular neuron is simply the partial derivative of the loss function with respect to the output of the neuron.

$$\delta_j^l = \frac{\partial L}{\partial a_j^l} \text{act}'(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \quad (2.16)$$

In this equation the left part of the right hand side describes the change of the loss function with respect to the output of the neuron and the right part is the change of the neuron's output itself. A more compact way is to use the matrix notation for each layer instead for each neuron within a certain layer (2.17). In equation (2.17)  $\nabla_a C$  represents a vector which contains all the

<sup>4</sup>Adapted from <https://morioh.com/p/15c995420be6>

partial derivatives of the loss function with respect to the activations for each neuron in layer  $l$ . On the basis of this equation for the output layer equation (2.18) will be used to propagate backwards by taking the error from layer  $l$  to find the error for layer  $l - 1$ . This means when layer  $l$  is the output/last layer in the neural network then the error for the second to last layer  $l - 1$  can be found using the last layer and then continue using layer  $l - 1$  for finding the error for  $l - 2$  and so on.

$$\delta^l = \nabla_{\mathbf{a}} L \odot \text{act}'(w^l a^{l-1} + b^l) \quad (2.17)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1} \odot \text{act}'(w^l a^{l-1} + b^l)) \quad (2.18)$$

The interesting part is now how to find the partial derivatives of the loss function with respect to the bias and the weights in order to compute gradient descent. For this the following two formulas are used:

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l \quad (2.19)$$

$$\frac{\partial L}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \quad (2.20)$$

These two equations make it possible to compute the change of the loss function for any weight and any bias term in the entire neural network. Together with equations (2.17) and (2.18) all the necessary ingredients are now available to compute the partial derivatives needed for the optimization algorithm to work.

### 2.2.2.3 Adam Optimizer

There are many different implementations and adaptations of the stochastic gradient descent algorithm. The following part briefly introduces the **Adam optimizer** as it will be used later on in the empirical part of the project.

Although SGD is a very typical choice it, unfortunately, has its flaw when it comes to choosing an optimizer for the neural network, too. The flaw is that for difficult optimization problems SGD has convergence issues. Therefore, other optimizers became popular, e.g. the Adam optimizer, which stands for Adaptive Moment Estimation [4]. In comparison to SGD Adam not just uses one learning rate  $\eta$  but instead uses a learning rate for each weight in the network and these learning rates adapt independently from each other during the training phase. This adaptation avoids convergence issues.

### 2.2.3 Regularization for Neural Networks

The named regularization strategies in section 2.1.5 like L2-regularization are commonly used in Neural Networks. In addition to these techniques another one is used called **Dropout**. The idea of Dropout is to temporarily deactivate a portion of the neurons during the training process. The amount of how many neurons are deactivated depends on the initial setting. A very common choice is 50%. When training is complete and all weights have been trained then the weights will be halved when having used Dropout of 50% [1]. What this does is simply try and average the results of several different neural networks. This prevents the different neurons from depending too much on other neurons and rather forces the neuron to identify certain trends in the data itself.

## 2.3 Convolutional Neural Network

This section briefly introduces the **Convolutional Neural Network (CNN)**. The intention is to get a rough idea of the baseline model's mechanism. The CNN consists out of several different types of layers with the convolutional layer being introduced first. In comparison to the common FFNN layer the convolutional layer does not connect every neuron in the previous layer with every neuron in the next layer. For a CNN the input has to be imagined as a matrix where e.g. in the context of NLP a row represents a word within a text sequence. So in case the sequence consists of 10 words and each word is represented by a 10 dimensional vector the input is a 10 X 10 matrix. The convolutional layer computes the output of a neuron in the hidden layer using a weight matrix called kernel, sometimes also called local receptive field. The neurons in the same hidden layer share the same weights and bias. The output of a neuron can be depicted in the following equation for neuron  $j, k$  in a hidden layer:

$$a_{j,k} = act(b + \sum_x \sum_y w_{xy} a_{j+x,k+y}) \quad (2.21)$$

The parameter  $b$  represents the shared bias,  $w_{x,y}$  is the kernel weight matrix,  $act$  is again an activation function, and  $a_{j+x,k+y}$  is the input matrix containing the outputs from the previous layer for the neuron in the hidden layer at position  $j, k$  in the matrix in the previous layer. The result of performing these operations for each hidden neuron is a matrix called the feature map. The kernel weights are free parameters of the model. Additionally, there can be several channels in such a layer that make it possible that another input matrix with another kernel can be assigned to. For NLP this may imply that two different word embedding representations can be used for the same text sequence. Furthermore, another layer called pooling layer can also be used for a CNN model with the input matrix being partitioned into several regions and then for a certain criterion one of the input values within that region is selected like e.g. the maximum value (Max Pooling). The final layer of a CNN is a Linear layer with each neuron within the layer being connected to each neuron in the previous layer like for the hidden layers

and output layers in the FFNN (see section 2.2).

## 2.4 Attention

This section introduces the idea of attention which derives from cognitive attention where the focus is placed on a fraction of the incoming information. The attention mechanism is another neural network layer with its first real breakthrough within a Recurrent Neural Network encoder-decoder architecture [17]. For example it combines the hidden states of the encoder, the so called key vectors, and the last hidden state of the decoder, called query vector, to an output vector (more on this example in section 3.2.1). Attention layers like other layers in neural networks are optimized via training so that the model can learn on what to focus in the input information.

In NLP attention is used to tell the model for each input word within an input text sequence on which other words it has to put special emphasis on. This comes very close to what we do when reading a sentence like e.g. *"John wants to go to the movies since he enjoys it a lot."* Then *he* is strongly connected to the word *John* and *it* to both *the* and *movies*.

### 2.4.1 Self-attention and Multi-Head Attention

Self-attention is a special way of computing attention and will be relevant later for the transformer model in section 3.2.2 [8]. It is performed by first multiplying the input vectors with three different weight matrices to get the query matrix Q, the key matrix K, and the value matrix V. The three weight matrices contain free parameters of the model and thus will be tuned during the training procedure. The attention is then found by the following formula using the Q, K, and V matrices.

$$Attention(Q, K, V) = softmax(\frac{Q \cdot K^T}{\sqrt{d_k}}) \cdot V \quad (2.22)$$

The formula computes the dot product of the query and the key and divides it by the square root of the dimension of the key matrix  $d_k$  in order to stabilize the gradients and thus obtain the attention scores. The product then scales it between 0 and 1 using softmax and multiplies it with the value matrix V to get the attention. The benefit of scaling the attention scores between 0 and 1 results in making the non relevant words less influential on the final outcome.

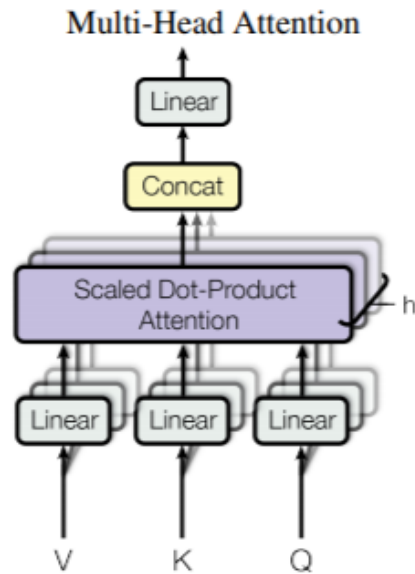


Figure 2.5: Multi-headed attention mechanism, V=value, K=key, Q=query, and  $h$  different heads <sup>5</sup>

In addition to the definition above Transformer models use a multi-headed attention mechanism which has the benefit of running the attention computation in parallel and thus compute the attention for several words at a time [8]. The model uses several, let's call them  $h$ , query, key, and value matrices instead of just one. The  $h$  query, key, and value matrices will then be computed using the input and basically perform the procedure explained above  $h$  times. The advantage of having several so called attention-heads is to have an even greater representation power since each head can be trained to represent another feature of the data which makes it possible to cover more complex structures.

---

<sup>5</sup>Adapted from [6]



## Chapter 3

# Natural Language Processing with Deep Learning

The following sections introduce NLP in combination with deep neural networks. It starts with providing an overview of how to encode text into numerical representations for neural networks (section 3.1). Afterwards, the development of recent models that lead to transformers is presented (sections 3.2). Subsequently, the idea of embeddings is studied (section 3.4) and then the BERT architecture is looked at in more detail (section 3.5). Finally, the chapter concludes with a description of different BERT models used in this project (section 3.6). Throughout the chapter the focus lies on BERT in order for the reader to understand how the model works since it is the model used for the empirical part of the project.

### 3.1 Tokenization and Input Embedding

Before being able to perform NLP with deep learning models it is required to represent the text numerically in a way that a Neural Network can actually work with it. This section takes a closer look at input in general in order to understand how the data is transformed before it will be inserted into models. This is a general overview combined with a more detailed description for how it is done for BERT in 5 steps.

#### 3.1.1 Tokenizer

The first step of transforming the data is to tokenize them meaning to split the text sequence into smaller chunks. The set of all chunks that the tokenizer recognizes is called vocabulary and is a pre-trained instance. There are three different types of **Tokenization** called **Word Tokenization**, **Subword Tokenization**, and **Character Tokenization**. The Word Tokenization chops the sequence into words depending on the pre-trained vocabulary of the tokenizer. However, if a word is not part of the vocabulary or also called out of vocabulary (OOV) then it cannot be recognized which of course is problematic. A typical word tokenization solves this problem

by replacing the word with a **UNK token** which stands for "unknown". However, this would mean that the model will simply not take the word into account at all and thus will lose valuable information, especially, when the dataset contains a lot of special terms like e.g. medical ones. Then a big chunk of the sentences might be unrepresented. **Character Tokenization** partitions text sequences into a sequence of characters. This prevents the problem of OOV words and keeps the vocabulary size at 26. Unfortunately, it expands the size of the text representation drastically compared to Word Tokenization and thus makes it difficult to capture the relationship between the characters in terms of representing words. Another way to solve this problem is by using **Subword Tokenization**. A typical approach for this is the WordPiece Model (WPM) designed by Google. This data-driven approach ensures a deterministic split for any order of characters and is the model of choice when it comes to BERT.

The size of the BERT vocabulary is fixed around 30000 words/tokens containing the most frequent words/subwords of the trained corpus.

An example from the project's dataset using WPM is: "Retinoblastoma develops from cells that have cancer." Since the medical term "Retinoblastoma" is not within the vocabulary the word is split into several subtokens and thus the sentence looks as follows: "Re ##tino ##blast ##oma develops from cells that have cancer." The double hashtags symbolize a follow up sub-token.

### 3.1.2 Special Tokens

In addition to the input sequence special tokens like e.g. the UNK token are often added. These special tokens have different purposes and are contained in the vocabulary of the tokenizer. For a BERT model two additional tokens are added. The first one shall represent the entire sequence. This is especially useful for classification tasks where the entire input sequence shall be categorized for a given set of labels. This particular token is called the **CLS token** and is always the first vector in the input sequence. The second token is needed in order to indicate the end of the first sentence and should be placed at the start of the next one. It is supposed to separate the sentences within the input sequence by applying a so called **SEP token**. SEP is placed between two sentences if the sequence consists of two. Otherwise, if there is only one sentence, it is placed as the last token of the sequence.

Continuing with the given example above, after adding the special tokens, the sequence is:

"CLS Re ##tino ##blast ##oma develops from cells that have cancer. SEP"

### 3.1.3 Input Embedding

The next step requires each token to be represented numerically. This is done by the input embeddings stored in the vocabulary of BERT. These input embeddings are feature vector

representations of a word. There exists an Input Embedding Lookup Table for the whole vocabulary. The size of the embedding vector for BERT depends on the size of the model, e.g.  $BERT_{BASE}$  has an embedding vector with 768 elements. Each entry into the lookup table stands for one token. So there are around 30k entries and to connect the words in the input sequence with the corresponding input embeddings each word is converted by the tokenizer to the index of its entry into the lookup table.

The example sentence is now converted to the following sequence of entries:

character seq.	CLS	Re	##tino	##blast	##oma	develops	from	cells	that	have	cancer	.	SEP
entry index	102	11336	20064	27184	7903	11926	1121	3652	1115	1138	4182	119	101

### 3.1.4 Padding

The models need the same input sequence length for each input sentence of a given dataset. So a maximum length needs to be defined which ideally should match the length of the longest sentence/sequence in terms of tokens. For all shorter sentences the length has to be increased by appending empty tokens to the end of the sentence. These empty tokens are called **PAD tokens** and take the index value 0 in the lookup table. If there is e.g. another sentence in the dataset that has 2 tokens more than the example sentence then it is necessary to append another 2 PAD tokens at the end of the example sentence. This process is called Padding.

### 3.1.5 Attention Mask and Segment Embeddings

The last step is to create a vector indicating which tokens to pay attention to in the input sequence and to which ones not (PAD tokens). This is done by a simple vector consisting of 1's and 0's where 1's indicate that the token should be attended to and 0's the opposite. This procedure is called **attention mask** and is important for the self-attention mechanism within the encoder in BERT.

Additionally, for BERT there is a vector containing the segment ID's that categorize each token to a segment of the sequence. This is another way of splitting sentences within a sequence. The segment ID's are then mapped to the segment embeddings.

## 3.2 NLP Architectures

### 3.2.1 Encoder-Decoder Architecture

The **Encoder-Decoder Model** is a special architecture based on two elements the encoder and the decoder. In the general structure the encoder deciphers the input into a context vector which is a summary of the input. The decoder then uses the context vector to produce a task dependent output. An example of an Encoder-Decoder Model is the **Sequence to Sequence (Seq2Seq)** model which was originally invented in order to perform sequence transformation with the help of neural networks (see Figure 3.1) [16]. This is extremely relevant for NLP as text is simply a sequence. In that sense the encoder enciphers a text sequence into a context vector, whereas the decoder deciphers the context back and returns a sequence of probabilities for certain output words using a softmax function where the size of the sequence is task dependent. Both modules originally consist of **Recurrent Neural Network's (RNN)** and hereby preferably **Long Short Term Memory (LSTM)** models. Important to notice is that the inputs for the Encoder are the word embeddings instead of the plain text (see both section 3.1 for the encoding pipeline and section 3.4.1 for word embeddings).

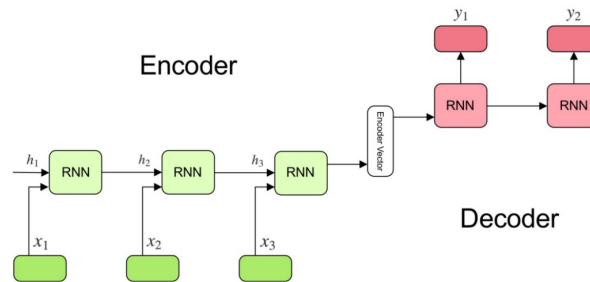


Figure 3.1: Basic encoder decoder architecture <sup>6</sup>

#### RNN:

A RNN is a neural network that consists of neurons in comparison to the FFNN with both the initial input and the output of its activation from the previous time step. Basically, RNN's add a temporal aspect and thus takes historical information into account. The output of a hidden layer in a traditional RNN can be computed in the following way:

$$a^t = \text{act}(W_{aa}a^{t-1} + W_{ax}x + b) \quad (3.1)$$

In this function  $a^t$  represents the output of the hidden layer,  $x$  is the input,  $b$  the bias,  $W_{ax}$  the weight matrix for the input,  $W_{aa}$  the weight matrix for the output of the previous time step, and finally  $a^{t-1}$  is the output of the hidden layer at the previous time step.

---

<sup>6</sup>Adapted from

<https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346>

## **LSTM:**

A LSTM is a modification of the traditional RNN and tries to reduce the problem of unstable gradients within RNN's where earlier layers are hard to train as the gradients get backpropagated from layer to layer (see section 2.2.2.2). The output of the LSTM cell depends on three elements which are the input, the output of the hidden state at the previous time step, and finally the current cell state representing the long term memory of the cell. Furthermore, the LSTM consists of three gates called the forget gate, input gate, and output gate which decide on which information to keep, update the cell state, and perform the output of the cell, respectively.

Unfortunately, these Seq2Seq models were struggling with long sequences and the fixed length of the context vector proved to be a bottleneck. A new Seq2Seq model was designed which adds two main aspects [17]. The first one is to use a bidirectional RNN (BiRNN) and an attention mechanism. The idea for the BiRNN hereby is to use a common RNN and perform a forward pass which sequentially inserts the input in its initial order and computes a vector much like before and a backward pass which computes a vector for the reversed order. The two vectors are then concatenated together to build a sentential representation of the word. This gives the model the additional strength of including the knowledge of previously occurring words within the same sequence and adds the knowledge about words occurring after a certain word. The attention mechanism uses the hidden states for all words returned by the encoder as its values and the hidden state of the decoder at the previous time step as the query vector to compute the context vector. As in the previous model the context vector then is the input to the decoder which uses the hidden state of the previous time step and the context vector to produce a final output.

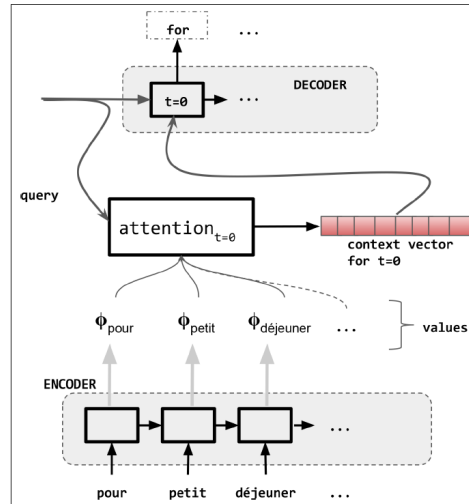


Figure 3.2: Seq2Seq model using attention at time step  $t=0$  of the decoder. The predicted output is “for” and the attention mechanism takes all the hidden states of the encoder (values) and the hidden state of the decoder at the previous time step (query) into account. <sup>7</sup>

### 3.2.2 Transformer

The **transformer** is a new type of model, which is an encoder-decoder architecture type. The transformer models, like the Seq2Seq models, are mainly used for NLP tasks due to their ability to operate on sequential data. One elementary feature of transformers is the fact that they can handle entire text sequences at a time which makes it possible to parallelize the computation and thus makes the model much faster than the Seq2Seq models. Transformer models pick up the idea of using attention from the Seq2Seq models but evolve it further by being built mostly out of attention [6]. This makes the transformer model capable of learning context of words better as it trains from both directions simultaneously in comparison to training it independently like a BiRNN.

The transformer models typically consist of a stack of encoders followed by a stack of decoders which are all similar in their structure.

---

<sup>7</sup>Adapted from [3]

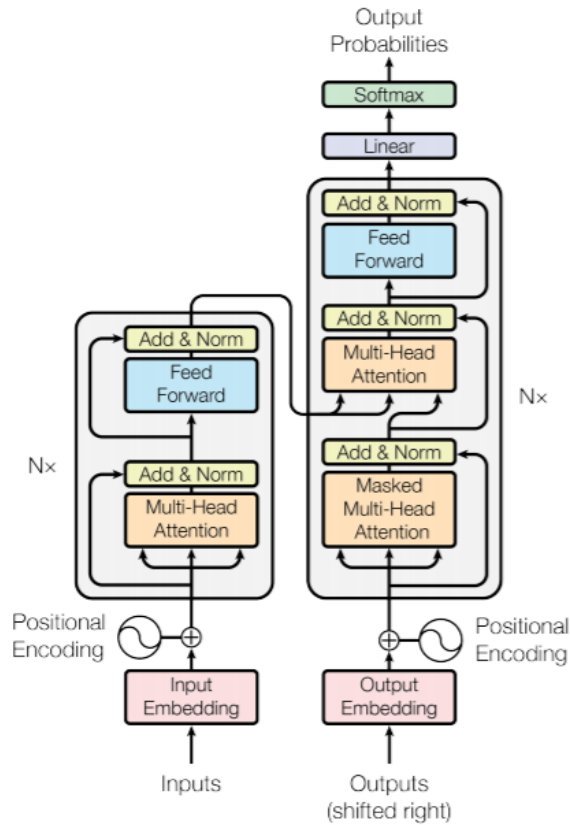


Figure 3.3: Basic Transformer Architecture consisting of one encoder and one decoder (encoder module left and decoder module right) <sup>8</sup>

The encoder consists of a self-attention layer followed by a FFNN as can be seen in Figure 3.3. After each sub-module in the encoder the output is layer normalized which means for each feature of the activations the mean is set to 0 and the variance to 1.

The decoder also contains a self-attention layer followed by an encoder-decoder-attention and a FFNN at the end. The encoder-decoder attention makes it possible for the decoder to put a special focus on certain parts of the input sentence. Like for the encoder the output of each sub-module of the decoder is layer normalized. In addition, the input into the self-attention layer of the first decoder is the previous output of the decoder stack. Important to mention is hereby that this particular attention layer does not cover the entire sentence but only the already returned words of the sequence. The future upcoming words are masked out to ensure that the model cannot foresee the future. This implies the model cannot get input of words from the sentence that have not yet been predicted by the model itself.

The last step of the transformer is to take another FFNN after the decoder stack which cre-

<sup>8</sup>Adapted from [6]

ates task relevant logits. Logits are a vector of non-normalized predictions of the model. For a classification task the logits have a size equal to the amount of possible classes. The last step is another softmax layer which converts the logits into probabilities by normalizing them representing the likelihood for each class as the final output.

The embeddings into the transformer like for other NLP models are word vectors as explained in section 3.4. However, that representation does not include any exact information on where the word is positioned along the sentence. Furthermore, since transformer models do not parse one word at a time but instead operate simultaneously on each word in a sentence there is nothing to represent the word order. Yet, this is a crucial information since it can have quite an effect on the meaning of the word. Therefore another embedding is needed for transformers called the positional embedding.

The **positional embedding** is another vector which uses sinusoidal functions to represent the position of a word within a sentence. It is afterwards added to the input embedding itself.

### 3.3 Named Entity Recognition: an NLP Framework

Among other tasks like translation, which is the conversion from one sequence of words to another sequence of words, NER is a popular task to be performed. The task has a many-to-many relationship where the input has the same dimension as the output. This is the case as each element in the input sequence needs to be classified. In the case of NER it is to assign a label to each word/token within a text sequence. More details follow on how the precise NER task for this project is designed in chapter 4.

### 3.4 Embeddings

In general, it is necessary for NLP tasks to represent words in a way that a neural network can work with them. This is done by embedding words into value vectors. Initially, this was solved by using one-hot encoding where the vector took the length of the entire vocabulary. Unfortunately, this does not describe the context of words well as words with similar meaning like e.g. "good" and "great" are equally different as e.g. "good" and "home".

#### 3.4.1 Word2Vec and GloVe Embeddings

A way to improve the embeddings was the idea of representing words within a feature space in order to compare semantic and linguistic similarities between words. These features could be anything ranging from topics like winter e.g. where words like "ice" and "snow" would share a similarity or opposites where "good" and "bad" would share a connection. Obtaining such word representations can be done by using a technique called **Word2Vec**. Word2Vec can hereby



be applied by different algorithms which use neural networks to either predict the context of a word given a one-hot encoding of the word or basically the other way around[15].

A different approach of creating word embeddings is by using a method called **Global Vectors (GloVe)** which also computes the feature vectors for words. These vectors are acquired by using global statistics of the trained data to count for how many times two words occur next to each other.

### 3.4.2 Contextualized Embeddings with ELMo

The big problem with word embeddings obtained via Word2Vec or GloVe is that they do not contain information about the context of the words within a given sentence as they are fixed vectors. For example the meaning of the word "case" is quite different in "In this case I prefer going to the beach" in comparison to "John's case is black". This is where **Embeddings from Language Models (ELMo)** comes into play as this model uses the entire sentence to build an embedding for a word [18]. The model consists of BiRNN's (bidirectional LSTM's) which combine forward representation and backward representations as described in section 3.2.1. The way ELMo gains these embeddings is by performing a task called **Language Modeling (LM)**. The goal of LM is to predict the next word given a text sequence. The model will then output a set of probabilities for a set of different words indicating the probability for the next word.

## 3.5 BERT

BERT, which stands for Bidirectional Encoder Representations from Transformers, is the state of the art when it comes to all sorts of NLP problems. As the name already suggests BERT is a transformer model with the big difference that it only consists of a big encoder stack. The encoders themselves are built in the same fashion like the ones mentioned for transformers in section 3.2.2 by containing the same sub-structure. One of the great advantages of BERT is the idea of transfer learning which means first pre-training BERT and then fine-tuning it on so called downstream tasks and hereby transferring the knowledge from the pre-training to achieve better results on the desired final task. Essentially, BERT combines the idea of ELMo having contextualized embeddings together with a transformer architecture. Furthermore, in comparison to other attempts of building language models via transformers like OpenAI Transformer [19] BERT is trained bidirectionally.

The initial embeddings of BERT are built of three parts namely the token embeddings (see section 3.1.1), segment embeddings (see section 3.1.5), and positional embeddings (see section 3.2.2). Each encoder will then produce a new embedding for each word.

BERT is pre-trained on a large amount of text gathered from the BooksCorpus and the English Wikipedia which combined contain more than 3.3 billion words. The pre-training of BERT is then done via two unsupervised tasks.

The first one is called **Masked Language Modelling** (MLM) in the original paper ([5]). The great advantage of using MLM in comparison to traditional LM is that the transformer model is able to train bidirectionally and thus greatly improve the ability of capturing the context. In MLM the model masks out 15% of the words in each input sequence. The model's task is then to predict the masked out word. However, that particular procedure creates a discrepancy between pre-training and fine-tuning due to the MASK token which is used for the masked out word in order to perform MLM. Furthermore, to decrease this discrepancy not every masked word is replaced with a MASK token. Instead only 80% of the tokens are replaced by the MASK token, 10% with a random word, and the rest is kept as it was. The other task is called **Next Sentence Prediction** (NSP) which is a binary classification task given an input sequence which contains two sentences A and B. The model then has to decide whether sentence B is the follow up sentence for A or a randomly selected one. [5]

To fine-tune the BERT model another layer has to be placed on top of BERT also called head in order to give the desired task-specific output. Afterwards, the model is fed with the relevant inputs and outputs to the model and during training the free parameters of the model are optimized from end-to-end.

The pre-training of a BERT model is an expensive task computational-wise when considering the large amount of data it is trained upon. The fine-tuning, however, is much less expensive since it usually does not require such a huge dataset in comparison to pre-training. The pre-trained BERT models are made available by Google via their Github and are currently accessible via all sorts of libraries. For this project the focus is therefore solely placed on fine-tuning these pre-trained models for NER. The structure of how to fine-tune BERT for NER is displayed in Figure 3.4.

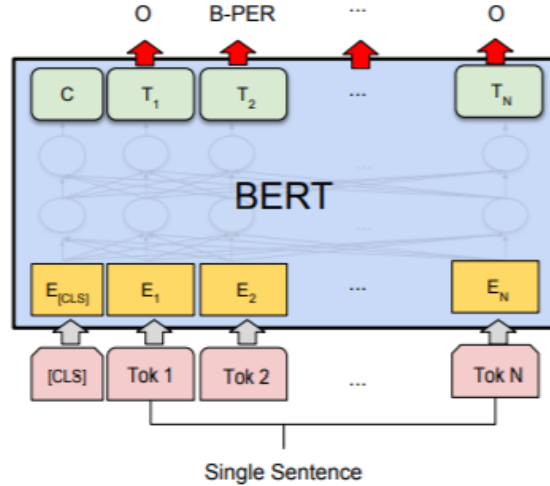


Figure 3.4: Fine-tuning of BERT for NER <sup>9</sup>

## 3.6 BERT Models

The following section provides a quick overview of the implemented BERT models. It starts with the original two BERT models followed by several modified versions, specifically, domain specific models like e.g. **BioBERT**. It is worth mentioning that the models used are all case sensitive as the vast variety of technical terms has an impact on performance in the medical domain.

### 3.6.1 Original BERT

Google originally designed two BERT models. The first and smaller one called  $BERT_{BASE}$  consists of 12 Encoders, 12 self-attention heads, and a hidden size of 768 with a total of 110 million parameters and the larger one  $BERT_{LARGE}$  runs with 24 Encoders, 16 self-attention heads, and a hidden size of 1024 with a total of 340 million parameters. Although  $BERT_{LARGE}$  yields better performances on a wide range of tasks in comparison to  $BERT_{BASE}$  it is due to its massive size much more time consuming to train and due to limited hardware capacities not always used. For this project both variants will be used for testing. These two models will be tested against more domain specific models. [5]

### 3.6.2 DistilBERT

DistilBERT is a reduced version of BERT and therefore has much better running times which at the beginning of the project was of great value as everything was computed via a CPU setup. **DistilBERT** uses 40% less parameters than  $BERT_{BASE}$  and kept 97% of BERT's language understanding according to its original paper. The big upside of DistilBERT is that it

<sup>9</sup>Adapted from [5]

is around 60% faster than  $BERT_{BASE}$ . The way the model got extracted from BERT was by using Knowledge Distillation which is a compression technique that transfers knowledge from a larger model/teacher (BERT) to a smaller model/student (DistilBERT). [12]

### 3.6.3 BioBERT

The original BERT models are trained upon the general domain corpora BooksCorpus and English Wikipedia. This causes an issue since this particular way of pre-training BERT is not domain specific. A possible way to improve performances of the model is by making it more aware of the technical terms in the respective field of interest. Here the model BioBERT (Bidirectional Encoder Representations from Transformers for Biomedical Text Mining) comes into play. BioBERT is initiated as a normal BERT model pre-trained on the above mentioned corpora and then further pre-trained on a medical corpus. In this particular case for BioBERT-v1.1 it is pre-trained on PubMed abstracts <sup>10</sup>. The initial BERT model hereby is  $BERT_{BASE}$ . The results in the BioBERT paper clearly indicate that for the biomedical domain BioBERT has a significant effect on performance for all sorts of tasks like NER among others (see [10]). Since the domain of the project is medical it is expected to outperform the original BERT models.

### 3.6.4 Bio+ClinicalBERT

ClinicalBERT is also a domain specific model like BioBERT as it tries to be trained on clinical data. The precise clinical domain data used is the MIMIC-III v1.4 database <sup>11</sup>. In its original paper ClinicalBERT comes in two different versions. The first version uses the original BERT model like BioBERT does which is trained on the general knowledge corpora before further pre-training it on the MIMIC dataset (ClinicalBERT). The other version uses the initial weights of BioBERT before further pre-training it on the MIMIC dataset (Bio+ClinicalBERT). As the Bio+ClinicalBERT outperformed the ClinicalBERT in the vast majority of tasks (see [11]) it has also been chosen for this project.

---

<sup>10</sup><https://pubmed.ncbi.nlm.nih.gov/>

<sup>11</sup><https://mimic.physionet.org/>

## Chapter 4

# Data and Models

The following sections of this chapter provide an overview of the data used by the author and how they got collected. Furthermore, the task of classifying entities within medical abstracts is discussed. In the succeeding sections the final model architecture is presented.

The goal of this project is to categorize certain entities in medical articles into pre-defined classes. These pre-defined classes are selected due to their ability to split the articles into the most relevant categories for the medical domain. It is hereby desired to use the final model for providing an abstract summary for health personnel in order to quickly scan through the most important information of an article and thus evaluate whether it fits to the query or not. This summary will simply list all the entities within the abstract to each corresponding class. Therefore, the entities should be comprehensive and self-explanatory like e.g. notes rather than too short. The classes used are described in Table 4.1:

<b>Classes</b>	<b>Explanation</b>
Age	patient's age
Gender	patient's gender
Ethnicity	patient's ethnical background
Diagnosis	diagnosed disease of the patient
Symptoms	patient's symptoms which are not in connection to any biochemical term or paraclinical finding
Biochemical	the biochemical state of the patient like e.g. blood tests
Genetics	information about the genetics of the patient
Paraclinical	paraclinical tests/studies and their results
Negative Finding	negative test results (normal health condition in medical terms)
Family	family history of the patient
Medication	treatments, therapies, or medication of the patient

Table 4.1: Explanation for each class

To better understand how each class' entities look like check Table 4.2. For simplicity Table 4.2 will only highlight the example entities. However, many sentences contain two or more entities belonging to different classes.

Classes	Example
Age	We present a rare case of <b>neonatal</b> cholestasis in a female <b>infant</b> with Gaucher Disease (GD).
Gender	<b>She</b> was initially diagnosed of Gaucher disease.
Ethnicity	A 21-year-old <b>Caucasian</b> patient was diagnosed with Gaucher disease.
Diagnosis	She was initially diagnosed of <b>Gaucher disease</b> .
Symptoms	We present a rare case of neonatal <b>cholestasis</b> in a female infant with Gaucher Disease (GD).
Biochemical	Her <b>altered lipid profile</b> led to a clinical suspicion of NPD.
Genetics	A unique compound mutation in <b>SMPD1 gene</b> is described.
Paraclinical	Clinical and <b>bone marrow examination</b> is the mainstay of diagnosis.
Negative Finding	Furthermore, the course of the disease is usually occult with <b>no spectacular symptoms</b> .
Family	Patient 1 was diagnosed after <b>familial screening</b> .
Medication	Combined <b>high doses of atorvastatin</b> and <b>ezetimibe</b> were given to treat the severe hypercholesterolemia.

Table 4.2: Simplified example sentence per class with the corresponding entity highlighted. It is simplified in the sense that there are more entities which are not highlighted in each sentence.

Another goal of the segmentation is to perform more specific matches between patient cases and abstracts by using the specific entity classes. One issue with **Question Answering** (QA) tasks in general but also specifically for the medical domain is the problem of matching negations with searches. Often search engines connect a search query like e.g. "Corona" with an article that contains "negative corona test". This is problematic since if a patient has Corona then the article containing the negation of this disease is in most cases a bad match. To solve this issue the "Negative Finding" class is designed to detect instances of tests that are negative or conditions of a person that are considered to be normal. When the model is capable of differentiating between the case of an instance being true/the case and false/not the case then it will be extremely valuable for search results. A possible way to solve such an issue is then to specifically match patient cases with abstracts class wise by e.g. matching symptoms of the patient case with symptoms in all sorts of abstracts and depending on that filter out only those articles that actually contain similar or exact symptoms. Additionally, this approach also works better with unstructured patient cases as the search engine is e.g. only filtering for symptoms rather than trying to match the entire patient case with abstracts. This method could potentially lead to a better fit for certain abstracts since the order of the information does not matter so much anymore.

## 4.1 Annotation

One major problem of this project was the lack of labelled data for the selected NER task. Therefore, it was necessary to annotate data. Thus the dataset got bigger and bigger over the course of the project and is still in progress. Initially, the dataset purely consisted of "Gaucher disease" abstracts. Gaucher is a rare disease and thus covered by findzebra. It is an inherited metabolic disorder due to deficiency of the lysosomal enzyme glucocerebrosidase. Later on several "Fabry disease" abstracts got added to the dataset. Fabry is a storage disorder which is also a rare genetic disease. It is likely that the dataset will be further expanded in its size by covering more types of diseases. For the final results the dataset contains 186 abstracts with 109 Gaucher abstracts and 76 Fabry abstracts, respectively.

For the annotation Prodigy was of great importance. The makers of Spacy designed Prodigy and therefore it is built on top of the Spacy Python library. Prodigy offers an API which makes it possible to annotate data easily and even makes use of the learning state of the currently trained model by letting it support the annotation process. That is to say the API simply displays the model's predictions on the currently annotated abstract and then gives the annotator himself the possibility to correct the predictions. Thus, a much faster labelling process is ensured and the annotator gets provided with a good understanding of how well the model performs on unseen data.

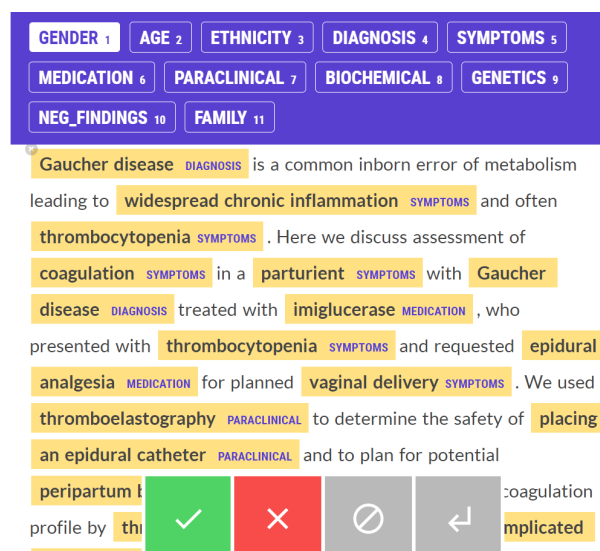


Figure 4.1: Prodigy API with annotation example of a gaucher abstract (entities are highlighted in yellow)

## 4.2 Positional Tagging

As can be seen from previous annotated examples like in Figure 4.1 an entity often contains several words. This creates the following issue in terms of how to match the labelled entities with the corresponding words in a logical sense. One way could be to just assign the class name to each word within an entity for the ground truth value  $y$ . However, this would ignore the positional order within an entity and could lead to problems where the data contains an entity like "the infant" labelled as AGE. When now labelling "the" and "infant" as belonging to AGE then the model might not correctly learn the connection between "the" and "infant" but rather incorrectly learn to predict each "the" as AGE. To solve this issue positional tagging is used when working with NER. The system used here is the **BILUO** tagging system. It splits up each class into sub-categories by adding a prefix to the class: the prefix "B" stands for the first token of a multi-token entity. "I" represents tokens neither at the beginning nor at the end of multi-token entities and "L" the last token. "U" represents single token entities. Finally, "O" gets assigned to each word/token within the data that is not assigned to any class. An example for such a possible tagging system is the following sentence from Figure 4.1 [9]:

Word	Tag
Gaucher	B-Diagnosis
disease	L-Diagnosis
is	O
a	O
common	O
inborn	O
error	O
of	O
metabolism	O
leading	O
to	O
widespread	B-Symptoms
chronic	I-Symptoms
inflammation	L-Symptoms
and	O
often	O
thrombocytopenia	U-Symptoms
.	O

Table 4.3: Example sentence with BILUO tagging

As learned earlier the vocabulary of the model might not contain certain words and thus it will



break words into smaller sub-tokens throughout the course of this project. Therefore, the exact tagging of the particular sentence in Table 4.3 depends on the specific model but it essentially follows the same structure demonstrated in the example sentence.

## 4.3 Models

This section provides an overview over the architectural structure of the models and includes a guideline model mainly used for the annotation. It concludes with the exact design of the final model.

### 4.3.1 Basic Model Architecture for Named Entity Recognition

The general structure of the models is partitioned into two main parts. The first one is to convert the given text into numerical representations using a certain embedding strategy (see section 3.4 for more detail). The second part is the head or final layer that makes use of the embedded sentences/words in order to make predictions. For the NER in this project each token within the input sequence is assigned to a class which means the output is a set of probabilities for each class. The model's predictions are then given by the class with the highest probability.

### 4.3.2 Spacy CNN Baseline Model

The last model is only briefly presented due to the lack of documentation. It is a CNN type model designed by Spacy called *en\_vectors\_web\_lg*. Since the model belongs to the Spacy library it is mainly utilized within Prodigy for the annotation of the data. As this model is not a BERT type model it serves purely as a baseline for this project. The model consists of approximately 1.1 million 300-dimensional vectors which are the initial word embeddings. These embeddings were trained on the Common Crawl corpus with **GloVe** [20]. In addition, a compression technique called Bloom Embeddings is applied which essentially helps storing the vectors in an efficient way [20]. Spacy uses 4 different hash keys to build together unique word vectors. In the model the initial word embeddings are encoded to a matrix which Spacy calls "sentence matrix" where each row represents a word/token in the context of the 4 previous words/tokens and the following 4 words/tokens within a text sequence. For this encoding step a residual CNN model is used (for CNN see section 2.3)[20]. Afterwards a type of attention mechanism is performed on the context embeddings to extract the most relevant features. The final layer of this model is a basic Linear layer (see section 2.2). [13].

### 4.3.3 Final Model

The precise architecture of the final model uses BERT as an embedder in order to utilize the contextualization within the word embeddings. These embeddings are used by the final layer that are added on top of BERT. This final layer is also a basic Linear layer. In the empirical part of the project different types of BERT models, as described in section 3.6, are substituted

for the embedding's part. The input size of the final layer (word embedding) changes between 768 and 1024 depending on the size of the hidden states of the different BERT models.

## Chapter 5

# Resources and Tools

This chapter briefly names the most important resources and tools used in this project. The following tools greatly improved the implementation of the final model as well as the creation of the labelled data in order to perform a supervised learning algorithm. Before the results are compared the evaluation of the model is explained.

### 5.1 Libraries

For the implementation Python is the way to go as it combines a simple syntax with great text processing tools and thus is ideal for NLP. It hereby contains many great libraries that are extremely helpful when it comes to working with NLP. The most important Python library required for obtaining pre-trained BERT models and for fine-tuning the models on the labelled data is the transformers library from "HuggingFace". It contains more than 8000 models for all sorts of different tasks like e.g. models for pre-training, NER, Question Answering and so on. Another library which is specialised on NLP problems is "Spacy". It contains great text processing tools e.g. sentence splitting and also different models designed for NLP problems. Prodigy is built on top of Spacy and as already mentioned played an important role in labelling data. There are many more libraries used for this project listed in its source file.

### 5.2 GPU Cluster

Although the project's data is rather small at this point of time the complexity of the BERT models makes the training computationally quite expensive. Therefore, the running time for fine-tuning a pre-trained BERT model with the current data will take several hours on a normal CPU setup. In order to be able to try out new ideas and optimize the hyperparameters for one particular learning procedure a faster training would be needed in order to be able to quickly adopt. The way to go in this project was to use a GPU cluster which was made available by DTU and contained 10 servers. For the training 1-3 GPU's were used and thus greatly improved the running time by shrinking the training time down to less than an hour.

## 5.3 Evaluation Metrics

In order to evaluate results after the training procedure and to be able to compare the models with each other certain metrics are needed. The most common metric is the F1-score or also called balanced F-score. The F1-score defines the accuracy of the model and is a value between 0 and 1 being stated as percentage. It is formed out of precision and recall in the following manner:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (5.1)$$

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

Figure 5.1: Confusion matrix <sup>12</sup>

- 1 TP = True Positive
- 2 FP = False Positive
- 3 FN = False Negative
- 4 TN = True Negative

The confusion matrix (Figure 5.1) is used in order to define precision and recall. Each point in the confusion matrix stands for an entire entity span and only a fully correctly classified entity is counted as a True Positive. Essentially, it means precision is the proportion of correctly predicted labels out of all predicted labels. Recall is the proportion of correctly predicted labels out of all the correct labels. The formal definitions for both elements are the following:

$$precision = \frac{TP}{TP + FP} \quad (5.2)$$

$$recall = \frac{TP}{TP + FN} \quad (5.3)$$

<sup>12</sup>Adapted from

<https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826>

For the evaluation the F1-Score is calculated for each label. Consequently, the overall F1-Score used to compare the models is the average of all the F1-Scores for each label. However, the weighted F1-Score is used which also takes the number of occurrences for each class into account. Additionally, for the final evaluation all classes are considered except the "O" class whose task it is to categorize all words that do not belong to any pre-defined class.

## Chapter 6

# Implementation

In this chapter a more in-depth look is offered at the actual implementation starting with the preprocessing of the data, encoding of the tags, and ending with the hyperparameter settings of the final model. The source code of the project can be found on Github <sup>13</sup>.

### 6.1 Pre-Processing the Data

For training the model it is necessary to put the annotated data into the desired format. The annotated data can be extracted from Prodigy as a so-called "jsonl" file. The format in which each abstract is stored is a dictionary of 9 elements. The most important elements are the text of the actual abstract, the entity spans within the abstract, and finally the tokens in the order they occur within the abstract. Especially the tokens and entity spans are of greater interest for the next step. The main idea is to feed the model sentence-wise with the abstracts. Therefore, the abstracts need to be split into sentences first. What follows is the matching of the sentences with the entity spans within the sentences. It is worth noticing here that the labelling is performed such that a single entity will always be placed within one sentence. The entities are stored together with the index of their first character within the sentence string and the index of their last character. The following function performs the sentence splitting and sentence entity matching:

---

<sup>13</sup>[https://github.com/s183193/NER\\_BT](https://github.com/s183193/NER_BT)

```

def preprocess_data():
    raw_data = []
    for line in open('data.jsonl', 'r'):
        raw_data.append(json.loads(line))

    data = []
    for abstract in raw_data:
        token_s = []
        token_e = []
        abstract_size = 0
        label = []
        sentence = []
        sentences = nltk.tokenize.sent_tokenize(abstract['text'])

        for span in abstract['spans']:
            token_s.append(span['start'])
            token_e.append(span['end'])
            label.append(span['label'])

        label_pos = 0
        s_count = 0
        for sentence in sentences:
            tags = []
            l = abstract_size + len(sentence)+s_count

            while len(label)>label_pos and token_e[label_pos]<=l:
                tags.append((token_s[label_pos]-abstract_size-s_count,
                             token_e[label_pos]-abstract_size-s_count,
                             label[label_pos]))
                label_pos+=1

            data.append((sentence, {'entities': tags}))
            abstract_size = abstract_size + len(sentence)
            s_count+=1
    return data

```

Figure 6.1: Data pre-processing function

Now it is possible to represent a sentence as a list of its words and create another list containing all the labels together with the correct positional tags. Since Spacy also uses the BILUO tagging system it provides a function called "biluo\_tags\_from\_offsets" which converts the entity spans into the BILUO tagging system. The exact function can be seen in Figure 6.2:

```

data = preprocess_data()
nlp = spacy.blank('en')
texts = []
tags = []
for text, annotation in data:
    doc = nlp(text)
    entities = annotation['entities']
    tags.append(spacy.gold.biluo_tags_from_offsets(doc, entities))
    subtext = []
    for token in doc:
        subtext.append(str(token))
    texts.append(subtext)

with open('tags.json', 'w') as outfile:
    json.dump(tags, outfile)
with open('texts.json', 'w') as outfile:
    json.dump(texts, outfile)

```

Figure 6.2: Data conversion to BILUO tagging

In this function the sentence is stored as a list containing each word as an element. All the sentences are then combined by appending them in another list. The corresponding tags are then stored in a similar manner so that the same entries of the sentence data structure yield the corresponding tag in the "tags" data structure. For further use both data structures are stored in so called "json" files.

## 6.2 Data Encoding

In the next step the data is encoded according to the required format of BERT. For this the Tokenizer class from transformers is used. Since the positions of the subtokens matter for the tags it is necessary to adapt the tags to the new encodings in many cases. Several ways of solving the problem are experimented with but the most successful approach is to label every subtoken and by that adjust the positional tagging depending on the amount of newly created tokens. The following figure displays the function for it:



```

def encode_tags2(tags, encodings):
    labels = [[tag2id[tag] for tag in doc] for doc in tags]
    encoded_labels = []
    for doc_labels, doc_offset, doc_input_ids in zip(labels, encodings.offset_mapping, encodings.input_ids):
        # create an empty array of -100
        doc_enc_labels = np.ones(len(doc_offset), dtype=int) * -100
        arr_offset = np.array(doc_offset)

        # set labels whose first offset position is 0 and the second is not 0
        doc_enc_labels[(arr_offset[:,0] == 0) & (arr_offset[:,1] != 0)] = doc_labels

        #labelling the special tokens CLS and SEP:
        doc_enc_labels[0] = tag2id['0']
        last_index = doc_input_ids.index(102)
        doc_enc_labels[last_index] = tag2id['0']

        # adjusting label positions to the created subtokens
        for i in range(last_index):
            if doc_enc_labels[i] == -100:
                if 'B-' in id2tag[doc_enc_labels[i-1]]:
                    doc_enc_labels[i] = tag2id[id2tag[doc_enc_labels[i-1]].replace('B-', 'I-')]
                elif 'L-' in id2tag[doc_enc_labels[i-1]]:
                    doc_enc_labels[i-1] = tag2id[id2tag[doc_enc_labels[i-1]].replace('L-', 'I-')]
                    doc_enc_labels[i] = tag2id[id2tag[doc_enc_labels[i-1]].replace('I-', 'L-')]
                elif 'U-' in id2tag[doc_enc_labels[i-1]]:
                    doc_enc_labels[i-1] = tag2id[id2tag[doc_enc_labels[i-1]].replace('U-', 'B-')]
                    doc_enc_labels[i] = tag2id[id2tag[doc_enc_labels[i-1]].replace('B-', 'L-')]
                else:
                    doc_enc_labels[i] = tag2id[id2tag[doc_enc_labels[i-1]]]
            encoded_labels.append(doc_enc_labels.tolist())

    return encoded_labels

```

Figure 6.3: Tag encoding function

The function uses the offset mapping from the tokenizer. The offset mapping consists of a pair to represent each token in the encoded version. The first element of each pair contains the starting character of the word where the token belongs to and the second element states the last character. This is extremely helpful as it marks the beginning of a new word every time the first element is a 0. The idea is then to match the original tag of the word to the first token of the word which is marked by a 0 in the offset mapping. The succeeding subtokens are initially marked by a label of -100 to indicate that they are not yet correctly tagged. In the later part of the function the tags for the subtokens are created by taking the prior tags into consideration. The positional prefixes of the labels are then changed accordingly. For example, as learned earlier in Section 3.1.1, the word "Retinoblastoma" which gets encoded to "Re ##tino ##blast ##oma" got initially labelled as "U-Diagnosis" meaning it is a single token entity. According to the adjustment the new labelling looks as follows:

Re	##tino	##blast	##oma
B-Diagnosis	I-Diagnosis	I-Diagnosis	L-Diagnosis

## 6.3 Training

The structure of the code can be applied to each BERT model with the variations of exchanging the model type and the corresponding tokenizer.

With the data ready to be inserted into the model it is time for the training. Thanks to the great tools of "HuggingFace" (see section 5.1) this part is rather straightforward by using the Trainer class provided by transformers. The test and training data has to be assigned to the Trainer class as well as the loaded pre-trained BERT model. The Trainer class uses the Adam optimizer (see section 2.2.2.3) if not otherwise specified. The only thing left to do now is to specify the hyperparameters. A way to find the best possible hyperparameters is by using "optuna" as a support tool for automating the hyperparameter search, which will be explained in the next section.

The final setting for the training of the BioBERT model v1.1 is displayed in the succeeding figure:

```
training_args = TrainingArguments(
    num_train_epochs=10,          # total number of training epochs
    per_device_train_batch_size=16, # batch size per device during training
    per_device_eval_batch_size=64, # batch size for evaluation
    warmup_steps=169,             # number of warmup steps for learning rate scheduler
    weight_decay=0.0042,
    learning_rate=6.917e-05,
)
```

Figure 6.4: Training arguments for Trainer containing the hyperparameters for the final BioBERT model

The very last thing is to run the *trainer.train()* method to execute the training procedure of the model.

## 6.4 Hyperparameter Optimization

In order to achieve the best performance the learning procedure requires to be optimized. So the best hyperparameters for a specific model have to be found measured on the basis of the F1-Score. Hyperparameter optimization can be quite time consuming since for each trial the model needs to be trained and afterwards evaluated. The set of hyperparameters that yield the best result is then selected as the final setting for the model. In order to be most efficient the automatic hyperparameter optimization algorithm should try to learn as much as possible from the past trials.

First of all, the hyperparameter space has to be defined (e.g. a range for the learning rate, number of epochs, and so on). Then a **Tree-structured Parzen Estimator (TPE) Algorithm** is performed to find the best setting of hyperparameters while minimizing the amount of actually performed model trainings. In short the TPE uses two distributions for the set of hyperparameters in order to form an estimation of the real objective function. One distribution  $d_1(h)$  (where  $h$  is the set of hyperparameters) tries to describe the sets of hyperparameters that perform well

on the F1-Score and the other one  $d_2(h)$  represents the low attempts with comparatively low F1-Scores. The distributions are hereby based on the past evaluations of hyperparameters which are categorized by being above or below an automatically chosen threshold [14]. The algorithm now uses Bayes rule to select the best set of hyperparameters. It does so by selecting the set of hyperparameters that maximizes the ratio  $\frac{d_1(h)}{d_2(h)}$ . This means that the set of hyperparameters is chosen that maximizes the likelihood of belonging to the better performing distribution in comparison to the likelihood of belonging to the weaker performing distribution. Essentially, the distributions are updated after every iteration of the algorithm.

As an example the set of hyperparameters for the final BioBERT model that result from this algorithm is shown in Figure 6.4 (see above).

## Chapter 7

# Results

The following chapter presents the final results of the different models used for comparison. Subsequently, the output of the best model is presented. Strengths and weaknesses of the model are then discussed.

For comparing the different models the Hold-out method has been chosen. Important to notice hereby is that each model is given exactly the same train-val-test-split, meaning the same training data, validation data, and test data, in order to ensure a statistically meaningful comparison. The final evaluation using the test data was repeated 5 times to average for the randomness in neural networks. Table 7.1 displays the employed models with their corresponding F1-Score and associated standard deviations.

Model	F1-Score
BioBERT	$66.90 \pm 0.74\%$
BERT <sub>LARGE</sub>	$66.37 \pm 0.34\%$
Bio+ClinicalBERT	$66.27 \pm 0.29\%$
BERT <sub>BASE</sub>	$65.67 \pm 0.41\%$
en_vectors_web_lg (CNN)	$65.37 \pm 0.12\%$
DistilBERT	$64.25 \pm 0.39\%$

Table 7.1: Final results determined via the Hold-out method

According to the F1-Score BioBERT performs best. In general it can be observed that both domain specific models (BioBERT and Bio+ClinicalBERT) seem to be among the models that capture the context of the data best. This is in line with previous research [10] [11]. As to be expected BERT<sub>LARGE</sub> outperforms both its smaller version (BERT<sub>BASE</sub>) and the distilled BERT version (DistilBERT). In addition, it can be seen that Spacy’s CNN model performs less well as far as F1-Score is concerned and thus hints at the strength of the BERT models in comparison to other types of models.

The impact of the final result on the class level is demonstrated in Table 7.2 displaying the final

BioBERT model.

Class	Precision	Recall	F1-Score	Support
AGE	0.66	0.86	0.75	36
BIOCHEMICAL	0.55	0.36	0.43	87
DIAGNOSIS	0.84	0.80	0.82	207
ETHNICITY	0.88	0.88	0.88	8
FAMILY	0.00	0.00	0.00	5
GENDER	0.91	0.91	0.91	32
GENETICS	0.43	0.38	0.40	50
MEDICATION	0.68	0.84	0.75	51
NEG_FINDINGS	0.64	0.39	0.49	23
PARACLINICAL	0.46	0.37	0.41	59
SYMPTOMS	0.68	0.76	0.71	164
micro avg	0.69	0.67	0.68	722
macro avg	0.61	0.59	0.60	722
weighted avg	0.68	0.67	0.67	722

Table 7.2: Final BioBERT model results per class showing the precision, recall, F1-Score, and number of predictions per class (support)

From Table 7.2 it can be inferred that BioBERT performs quite well on classes that have many occurrences within the dataset. The dataset contains especially many entities belonging to DIAGNOSIS and SYMPTOMS. There are, however, comparatively few occurrences of the GENETICS and PARACLINICAL class which results in lower F1-Scores. Furthermore, considering the complexity of each class it becomes clear that the model has difficulties detecting entities of GENETICS correctly as compared to classes like AGE. This results even when considering the same amount of occurrences within the data as entities of AGE are usually simpler in their structure.

Let's take a closer look at the BioBERT model's predictions on unseen Gaucher disease abstracts not yet annotated (abstract 1 and 2).






Label	Color
DIAGNOSIS	
SYMPTOMS	
AGE	
GENDER	
BIOCHEMICAL	

Table 7.3: Color code for the following model predictions

### Abstract 1 with predictions:

Oculomotor apraxia may be idiopathic or a symptom of a variety of diseases. In Gaucher disease, oculomotor deficit is characterized by a failure of volitional horizontal gaze with preservation of vertical movements. We present 2 sisters, 6 1/2 and 5 1/2 years of age, in whom the presenting sign was oculomotor apraxia. Oculomotor apraxia has not been previously reported as the presenting manifestation of Gaucher disease.

The first abstract shows nicely that the model is able to capture several different types of entities correctly. It is even capable of capturing that "oculomotor deficit" is a symptom despite the fact that the model has never seen the combination of those two words before. This clearly showcases that the model detects certain patterns within the language like e.g. the combination of a medical term which was previously part of a symptom like "oculomotor" and the word "deficit" likely being a symptom. Furthermore, it is also quite remarkable to see that the model is able to detect "6 1/2" as an individual entity of AGE and the subsequent part "5 1/2 years of age" as another entity of AGE. It must be able to connect the "6 1/2" to the "years of age" later on in the sentence because the number alone does not lead to such a classification. This shows that the model does not simply highlight the same words again but is actually capable of detecting context within the sentences.

### Abstract 2 with predictions:

Gaucher's disease has been associated with plasma cell dyscrasias. A patient had Gaucher's disease, nephrotic syndrome, and systemic amyloidosis DIAGNOSIS. Plasmacytosis in the bone marrow, the presence of light chains in the urine and renal glomeruli, and the finding of low circulating immunoglobulin levels suggest that the amyloid in this patient is related to a plasma cell dyscrasia.

Abstract 2 shows that the model is struggling with previously unseen words like e.g. "renal glomeruli". It starts alternating between the BIOCHEMICAL and SYMPTOMS class around the name of renal glomeruli (see alternating colors in boxes). The expression "renal glomeruli" should be connected to the BIOCHEMICAL entity beforehand. However, it tends in the right direction by partly predicting it as BIOCHEMICAL.

In general, it can be observed that the model performs fairly well on texts containing words it has seen in the past. Even if the word order changes it is often able to classify correctly and thus is able to extract valuable information from texts. However, the model is not well equipped to correctly predict unseen words so far, especially medical terms, as they are often split into subtokens. Then the model starts classifying in an unstructured order.

## Chapter 8

# Conclusion and Outlook

The purpose of this project is to improve search results for findzebra, a platform that provides diagnoses for rare diseases, by means of **Named Entity Recognition** for medical articles.

As a foundation of the project it was essential to create labelled data with the help of Prodigy as an annotation tool for the Supervised Learning algorithm.

In the main part, the thesis focuses on implementing a working model for NER using **BERT**. Experiments are conducted with a vast variety of different BERT models to achieve the best possible outcome. Initially, smaller and faster models are used, like e.g. DistilBERT, due to hardware limitations. Later the focus is shifted to domain specific models like **BioBERT** influenced by recent research results using domain specific models particular for the medical domain [10]. As the dataset is manually labelled the project starts with a small set of data which grows gradually and improves the models over time. In addition, a lot of experiments are conducted around finding optimized hyperparameters for the learning procedure of the models.

Last but not least, the models in their final configurations are trained and afterwards tested with the F1-Score as the final metric of choice in order to compare their performance and to select the best among them. **BioBERT** achieves the best score with 66.90% on average. Considering the current size of the dataset, which contains 186 medical abstracts and some inconsistencies throughout the data, this is a good result. The final model is hereby capable of recognizing several common patterns within medical articles and can provide good summaries on common articles regarding Gaucher and Fabry diseases. However, there is still room for improvements in terms of labelling the data as well as setting up the model.

The future goals of the project are split into two categories. The first one is to further improve the performance of the model and the second is to incorporate the final version into the "findzebra" API and by that make it production ready. When looking at the first goal it becomes

inevitable that the ambiguities within the dataset need to be resolved in order to achieve significantly better results. Additionally, the training data needs to be expanded in terms of covering different types of diseases and by that different types of technical terms as the model is too specialized on Gaucher and Fabry diseases at the moment. The Unified Medical Language System (UMLS) Metathesaurus could be used for further improvements as it provides synonyms for medical terms as well as relationships between the terms [21]. This could be highly relevant to reduce the issue of problems with unseen words/terms as seen in section 7. One way to utilize it could be to perform data augmentation which has proven to be extremely effective in image classification by reproducing sentences where several words are replaced with synonyms.

The final future goal is to insert the final model into the "findzebra" API in order to make it accessible for medical personnel. Both the article summaries and the specialized search have to be implemented in order to fully utilize the advantages that the segmentation task provides.



# Bibliography

- [1] Michael Nielsen, *Neural Networks and Deep Learning*, Dec 2019, accessed at: <http://neuralnetworksanddeeplearning.com/>
- [2] Tue Herlau, Mikkel N. Schmidt and Morten Mørup, *Introduction to Machine Learning and Data Mining*, Aug 2020
- [3] Delip Rao and Ryan McMahan, *Natural Language Processing with Pytorch*, Feb 2019
- [4] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, Jan 2017, arXiv:1412.6980 [cs.LG]
- [5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, May 2019, arXiv:1810.04805v2 [cs.CL]
- [6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, *Attention Is All You Need*, Dec 2017, arXiv:1706.03762v5 [cs.CL]
- [7] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, Jeffrey Dean, *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, Oct 2016, arXiv:1609.08144v2 [cs.CL]
- [8] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, Alexander M. Rush, *Transformers: State-of-the-Art Natural Language Processing*, Oct 2020, accessed at: <https://www.aclweb.org/anthology/2020.emnlp-demos.6/>
- [9] Lev Ratinov and Dan Roth, *Design Challenges and Misconceptions in Named Entity Recognition*, Jun 2009, accessed at: <http://cogcomp.org/papers/RatinovRo09.pdf>

- [10] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, Jaewoo Kang, *BioBERT: a pre-trained biomedical language representation model for biomedical text mining*, Oct 2019, arXiv:1901.08746v4[cs.CL]
- [11] Emily Alsentzer, John R. Murphy, Willie Boag, Wei-Hung Weng, Di Jin, Tristan Naumann, Matthew B. A. McDermott, *Publicly Available Clinical BERT Embeddings*, Jun 2019, arXiv:1904.03323v3 [cs.CL]
- [12] Victor Sanh, Lysandre Debut, Julien Chaumond, Thomas Wolf, *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*, Mar 2020, arXiv:1910.01108v4 [cs.CL]
- [13] Matthew Honnibal, *Embed, encode, attend, predict: The new deep learning formula for state-of-the-art NLP models*, Nov 2016, accessed at: <https://explosion.ai/blog/deep-learning-formula-nlp>
- [14] James Bergstra, Remi Bardenet, Yoshua Bengio and Balazs Kegl, *Algorithms for Hyper-Parameter Optimization*, Dec 2011, accessed at: <https://papers.nips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf>
- [15] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed Representations of Words and Phrases and their Compositionality*, Oct 2013, arXiv:1310.4546 [cs.CL]
- [16] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le, *Sequence to Sequence Learning with Neural Networks*, Dec 2014, arXiv:1409.3215v3 [cs.CL]
- [17] Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, May 2016, arXiv:1409.0473v7 [cs.CL]
- [18] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer, *Deep contextualized word representations*, Mar 2018, arXiv:1802.05365v2 [cs.CL]
- [19] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever, *Improving Language Understanding by Generative Pre-Training*, 2018, accessed at: [https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language\\_understanding\\_paper.pdf](https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf)
- [20] Explosion, *SPACY'S ENTITY RECOGNITION MODEL: incremental parsing with Bloom embeddings residual CNNs*, Nov 2017, accessed at: <https://www.youtube.com/watch?v=sqDHBH9IjRU>
- [21] Gary H. Merrill, *Concepts and Synonymy in the UMLS Metathesaurus*, Sep 2009, accessed at: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2850250/>