Deep learning in medical document classification

Master's thesis
University of Turku
Department of Future Technologies
Computer Science
2020
Tuomas Jokioja

Supervisors:
Hans Moen
Filip Ginter

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Future Technologies

TUOMAS JOKIOJA: Deep learning in medical document classification

Master's thesis, 49 p.
Computer Science
May 2020

Tekstipohjaista tietoa tuotetaan vuosi vuodelta enemmän, mikä puolestaan on lisännyt tarvetta automaattiselle tekstinkäsittelylle. Täten myös automaattisia tekniikoita luonnollisen kielen käsittelyyn on enenevissä määrin tutkittu ja kehitetty, erityisesti viimeisen vuosikymmenen aikana. Tämä on johtanut huomattaviin parannuksiin erilaisissa luonnollisen kielen käsittelytehtävissä. Suuri läpimurto on ollut valtavilla tietomäärillä koulutettujen syvien hermoverkkojen käyttäminen. Tällaisten menetelmien käyttö alueilla, joilla aika on arvokasta, kuten lääketiede, voisi tarjota huomattavaa lisäarvoa.
Tässä tutkielmassa annetaan yleiskuva luonnollisesta kielenkäsittelystä syväoppimiseen ja tekstinluokitteluun liittyen. Lisäksi erilaisten syväoppivien menetelmien käytettävyys arvioitiin kouluttamalla tekstiluokittelijoita ennustamaan lääketieteellisten dokumenttien diagnoosikoodeja. Valittuihin menetelmiin kuuluvat syväoppimiseen perustuvat FinBERT, ULMFiT ja ELECTRA, sekä yksinkertaisempi lineaarinen luokittelija fastText.
Tulokset osoittavat, että rajallisella aineistolla lineaariset menetelmät, kuten fastText, toimivat yllättävän hyvin. Syväoppimiselle perustuvat menetelmät taasen vaikuttavat toimivan kohtuullisen hyvin, vaikkakin niiden aito potentiaali pitäisi todentaa käyttäen suurempia datajoukkoja. Täten jatkotutkimusta syväoppiviin menetelmiin liittyen tarvitaan.

Keywords: syväoppiminen, tekstinluokittelu, lääketieteellinen data

UNIVERSITY OF TURKU
Department of Future Technologies

TUOMAS JOKIOJA: Deep learning in medical document classification

Master's thesis, 49 p.
Computer Science
May 2020

Text-based data is produced at an ever growing rate each year which has in turn increased the need for automatic text processing. Thus, to keep up with the amount of data, automatic natural language processing techniques have also been increasingly researched and developed, especially in the last decade or so. This has lead to substantial improvements in various natural language processing tasks. A major breakthrough has been the utilization of deep neural networks and massive amounts of data to train them. Using such methods in areas where time is valuable, such as the medical field, could provide considerable value.
In this thesis, an overview is given of natural language processing w.r.t deep learning and text classification. Additionally, a dataset of medical reports was preprocessed and used to train and evaluate a number of text classifiers in order to define the feasibility of such methods for medical text classification. The chosen methods include deep learning -based FinBERT, ULMFiT and ELECTRA, and a simpler linear baseline classifier, fastText.
The results show that with a limited dataset, linear methods like fastText work surprisingly well. Deep learning -based methods, on the other hand, seem work reasonably well, and show a lot of potential especially in utilizing larger amounts of training data. In order to define the full potential of such methods, further investigation is required with larger datasets.

Keywords: deep learning, text classification, medical data

Contents

List Of Acronyms **1**

1 Introduction **2**

2 Text classification **4**

2.1 Preprocessing 5

2.1.1 Features of words 6

2.1.2 Features of text 7

2.1.3 Tokenization 7

2.2 Text classifiers 9

2.2.1 Naive Bayes 9

2.2.2 Nearest Neighbor Classifier 10

2.2.3 Support Vector Machine 11

2.2.4 Decision Trees 12

2.2.5 Artificial neural network 13

2.3 Evaluation 14

3 Deep learning in natural language processing **17**

3.1 Transfer learning 18

3.1.1 Language modeling 18

3.1.2 Embeddings 19

3.2 Training 21
3.2.1 Loss functions 22
3.2.2 Stochastic Gradient Descent 24
3.3 Architectures 25
3.3.1 Recurrent Neural Network 25
3.3.2 LSTM 27
3.3.3 Transformer 28
3.3.4 Other architectures 32
3.4 Modern models 34
3.4.1 ULMFiT 34
3.4.2 BERT 36
3.4.3 ELECTRA 37

4 Medical report document classification 39
4.1 Problem definition 39
4.2 Data 40
4.2.1 Medical reports 40
4.2.2 General Finnish 40
4.3 Compute resources 41
4.3.1 CSC 41
4.3.2 Turku University Hospital 42
4.4 Methods 43
4.5 Results 44
4.5.1 Tools 45
4.6 Discussion 46

5 Conclusion 48

References 50

List of Figures

2.1 A hyperplane defined by a SVM that separates samples of negative and positive classes with maximal margin 11

2.2 Example of a simple decision tree which predicts the sentiment of a person regarding esports 12

2.3 Example of a feedforward neural network 14

2.4 A confusion matrix. 15

3.1 Skip-gram model architecture. 20

3.2 An RNN-module which illustrates the recurrent nature of the model . . . 25

3.3 The insides of an RNN 26

3.4 Long short-term memory network 27

3.5 Gated Recurrent Unit 29

3.6 The Transformer, from Vaswani et al. 2017 [33] 30

3.7 The autoencoder 33

3.8 A convolutional neural network for classification 34

List Of Acronyms

NLP	Natural language processing
BoW	Bag of words
TF-IDF	Term frequency - inverse document frequency
NMT	Neural machine translation
kNN	k-nearest neighbor classifier
SVM	Support vector machine
SVC	Support vector clustering
GLoVe	Global Vectors for Word Representation
SGD	Stochastic gradient descent
RNN	Recurrent Neural Networks
bptt	Backpropagation through time
LSTM	Long short-term memory
GRU	Gated Recurrent Unit
CNN	Convolutional Neural Networks
ULMFiT	Universal Language Model fine-tuning
BERT	Bidirectional Encoder Representations from Transformers
MLM	Masked language model
NSP	Next sentence prediction

Chapter 1

Introduction

Manually processing and producing text-based data is a common task in everyday life. Given that an ever increasing amount of text is produced each year, the need for automatic computational processing of natural language is increasing as well. Due to major strides in machine learning — especially deep learning — during the last decade, the automatic processing of textual data has progressed immensely.

In hospitals, writing is a major time sink for doctors every day. As all of this is done in a hurry, it increases the probability for human error when entering critical patient data to a system. Thus, using computers to help in the process of data entering and validation provides meaningful value. For example, a model for classifying medical reports could be used to automatically define a proper diagnosis code for a new piece of text or to find older, mislabelled texts in an effort to reduce human-based errors.

The motivation for this thesis was to, firstly, define the feasibility and current state of deep learning for automatic text classification of medical text and secondly, to build a set of tools for quick and easy classifier training of a number of different models that the hospital can use.

This thesis gives an introduction to text classification, describes some of the most influential classifiers from the past and the present day, outlines the current state of machine learning -based text classification, and presents the most influential model architectures

and training methods for them. The medical report classification problem, available data, compute resources, chosen methods and results for the trained classifiers are presented and discussed at the latter part of the thesis, after which the final chapter concludes the thesis.

Chapter 2

Text classification

Natural language processing (NLP) is the field of designing methods and algorithms that take as input or produce as output unstructured, natural language data [1]. Text classification is a category of tasks in NLP which has many real-world applications such as document classification, spam, bot and fraud detection and web search [2], [3]. A text classification task requires a training set $D = (d_1, \ldots, d_n)$ of labelled documents with class $L \in \mathbb{L}$ (e.g. news, politics, sports). Then the task is to determine a *classification model*

$$f : D \to \mathbb{L} \qquad f(d) = L \tag{2.1}$$

which assigns the correct class to in domain document d [4].

The labels are assumed to be purely symbolic so that no additional knowledge of their meaning is available, and the data consists of only knowledge extracted from the documents. Thus metadata such as document type, publication date, etc. is not considered available to use. This ensures that all the methods that will be presented in the coming section are completely general and do not depend on some special-purpose resources. Given that classification is based on the semantics of documents, which is a subjective notion, the class of a document cannot be deterministically decided. This lead to the phenomenon called *inter-indexer inconsistency* which manifests itself when two human

experts decide if a document d_j should be classified as c_i and disagree on the matter, which happens surprisingly often [5].

The automated text classification task dates back to the early '60s but became a major subfield of information systems only in the early '90s due to increased applicative interest and availability of more powerful hardware. Until the late '80s the most common approach to text classification in real-world applications was a *knowledge engineering* one, which consisted of manually defining a set of rules on how to best classify documents to given categories. In the '90s this approach was passed in popularity by the *machine learning* paradigm, according to which an automatic text classifier is built by a general inductive process automatically, by learning the characteristics of categories from labelled data [5].

This chapter gives an overview on text classifiers and other tasks related to training a classifier. Section 2.1 describes common steps, such as feature extraction and tokenization, that are done before training a classifier, section 2.2 presents various text classification algorithms and models, and finally section 2.3 presents metrics for evaluating the performance of a classifier.

2.1 Preprocessing

Before a machine learning model such as a text classifier can be trained, the data for it has to be preprocessed and mapped to real valued vectors. Mapping textual data to vectors is called *feature extraction* or *feature representation*. For a machine learning project it is crucial that the right features for the problem are chosen. Even though feature engineering is not as important in deep learning, a good set of core features still needs to be defined which is especially true for language data where the data consists of a sequence of discrete symbols. Somehow this sequence has to be converted into a numerical vector, and in a non-obvious way [1].

Tokens are considered to be the atomic units of NLP. More often than not, words and tokens are used interchangeably in literature, but a token may consist of multiple words or multiple tokens can represent a single word. In the next sections, words and tokens are used quite interchangeably.

The following sections describe the features that can be extracted from words and text, and current methods for tokenization.

2.1.1 Features of words

The most obvious choice for tokens in NLP are individual words and it is common to do lemmatization or stemming on the words before they are turned into numerical form.

Lemmas are the "dictionary entries" of words, for example the lemma of *looking, looked, looks* is *look*. Determining the lemma of a word is usually done by using lemma lexicons or morphological analyzers. Adding context into a lemmatizer usually improves the accuracy of lemmatizing given that a lemma can be quite ambiguous without it. Lemmatization may not work well if words are misspelled or for forms that are not in the lemmatization lexicon. *Stemming* is a cruder way of determining common forms for words. It maps sequences of words into shorter sequences so that different inflections map to the same sequence. The results of stemming do not have to be valid words, e.g. picture, pictures and pictured could be stemmed to pictur [1].

Distributional information of words can also be used, e.g. what words behave similarly in text. This distributional information is used in defining vectors for words so that words that behave similarly in text have vectors that are close to each other. Methods for deriving these vectors are discussed in more detail in chapter 3.1.2.

N-grams are consecutive word or letter sequences of a given length. For example, a word-bigram representation of "the dog is sleeping" would be ["the dog", "dog is", "is sleeping"]. Word-bigrams and trigrams - sequences of two and three items - are the most common of the n-grams. N-grams beyond trigrams are rarely used for words due to

sparsity issues although 4-grams and 5-grams are sometimes used for letters [1].

2.1.2 Features of text

Sequences of text, such as sentences, can be represented by a number of ways.

Bag of words (BOW) is a common feature extraction procedure used for sentences and documents. It looks at the histogram of words in a text and considers each word count as a feature. BOW can be generalized from words to any other word related feature, such as counting word bigrams instead of individual words.

Weighting is used to focus for example on words that appear frequently in a given document, but relatively few times in the whole corpus. Weighting can be used with the BOW approach, and a common way is to use TF-IDF (Term Frequency - Inverse Document Frequency) weighting which highlights words that are distinctive of the current text. N-grams can also be used for weighting instead of single words.

Windows focus on the immediate context of a word by considering the k surrounding words, and define features as identities of the words within the window. It is a version of BOW, but restricted only to the words within the defined window.

Position is an important part of textual data. A sentence that's words are shuffled will not be equivalent in meaning to the original sentence. Thus using the positional qualities of words — such as what the absolute position of a word is in a sentence or does it appear in the first 10 or so words — as features is also relevant [1].

2.1.3 Tokenization

The process of tokenization is in charge of splitting text into tokens. Tokenization is usually done based on white-space and punctuation in languages, such as English, that aren't as morphologically rich. This approach is called whole-word tokenization as each individual token represents a single word. In languages such as Hebrew and Arabic, some words attach to the next one without white-space, and in Chinese there is no white-space

at all. Thus, tokenization seems dependent on the language used [1].

In order to circumvent the problems of whole-word tokenization, one can use sub-word tokenization to divide words into multiple tokens. One such system to extract these tokens is called SentencePiece, which is a widely used tokenizer nowadays. SentencePiece will be explained in more detail in the next section.

SentencePiece

SentencePiece is a sub-word tokenizer and detokenizer that is language independent and designed for machine learning -based processing. It is comprised of four different components: **Normalizer**, **Trainer**, **Encoder** and **Decoder**. The Normalizer is used to transform semantically equivalent characters to a canonical form. The Trainer trains a sub-word segmentation model from the normalized corpus. The Encoder first normalizes the text with the Normalizer and encodes raw text into a sub-word sequence using the model generated by the Trainer. The Decoder can be used to transform the tokens into normalized text. [6]

SentencePiece builds a vocabulary of a predefined size of sub-word tokens. Depending on the given maximum size, the sub-words' length change. If, for example, the given maximum size is just 30 or so, the vocabulary could consist of all the letters of the English alphabet and not much else. On the other hand, if the vocabulary is excessively large, it would essentially work like a whole word tokenizer. Thus, maximum vocabulary size becomes a tunable hyperparameter for a model, which can have a considerable impact on it's performance.

SentencePiece's language-independent quality is quite important especially for tasks like Neural Machine Translation (NMT), which can perform automatic translation with a simple end-to-end system. Numerous NMT-systems rely on language dependent pre- and post-processors thus adding sentencepiece to those systems simplifies the processing pipeline and removes the need for custom processors for different languages.

Compared to whole-word tokenization, SentencePiece's sub-word tokenization achieves

a lossless representation of data. For example, a whole word tokenizer might tokenize "Hello world." as [Hello][World][.], thus losing the information of where there is white-space in the sentence. SentencePiece treats white-space as a normal symbol and replaces all occurrences of white-space with an underscore (U+2581) before tokenization. SentencePiece might tokenize the aforementioned example as [Hello][_wor][ld][.], thus preserving the white-space [6].

Compared to other sub-word segmentation systems, SentencePiece does not require that the input is pre-tokenized into word sequences. It works natively with raw sentences which makes a purely language independent and end-to-end system possible [6].

SentencePiece seems like a good choice for tokenization for morphologically rich languages such as Finnish. A version of sub-word tokenization similar to SentencePiece called WordPiece is used in modern models such as BERT (section 4.4) and ELECTRA (section 4.4).

2.2 Text classifiers

2.2.1 Naive Bayes

The Naive Bayes Classifier is a probabilistic classifier that assumes that a probabilistic mechanism has generated the words of a document. It is a simple classifier that estimates the joint probability of a class given a feature vector. It naively assumes that features are independent given class:

$$P(X|C) = \prod_{i=1}^{n} P(X_i|C) \tag{2.2}$$

Where $X = (X_1, \cdots, X_n)$ is a feature vector and C is a class. Although the assumption is unrealistic, the *Naive Bayes* classifier is surprisingly successful in practice [7].

Naive Bayes models are very efficient as they require minimal computational resources even for huge amounts of text and large vocabularies. However, there exists a

significant problem in this approach named the *never-seen-words* problem which manifests itself when a document containing a word that is not present in the training set is analyzed. The classifier estimates the statistics of a class by counting the occurrences of words in the training set, and a single out-of-vocabulary word will turn the probability of a document belonging to it's according class to 0. This could turn an otherwise clear-cut classed document into something else only due to a random word [8].

Naive Bayes models have shown good results in various classification tasks and have been used extensively due to their efficiency in training and classification. A huge setback for the method is its brittleness; to train a robust Naive Bayes Classifier one needs a dataset that covers the problem domain sufficiently, otherwise the model has a high variance. Thus a small dataset performs significantly worse when using a Naive Bayes Classifier than other document classification methods [8] [9].

2.2.2 Nearest Neighbor Classifier

Nearest neighbor classifiers select documents that are close to the target document instead of building an explicit model. The class of the document can then be inferred from the classes of the neighbouring documents. A classifier that selects the k closest documents is called a *k-nearest neighbor classifier* (kNN). There are a lot of usable measures for similarity. One example would be to count the number of common words in documents.

When deciding if a document belongs to a class, the document has to be compared to all the document in the training set. Then the k most similar documents are selected and their classes define the probability of whether the document belongs to a certain class or not. The class that has the largest proportion is then assigned to the document. Cross-validation can be used to estimate the optimal number for k from additional training data [4].

Nearest neighbor classifiers are computationally efficient in practice although they require some computation during classification since to determine the nearest neighbors

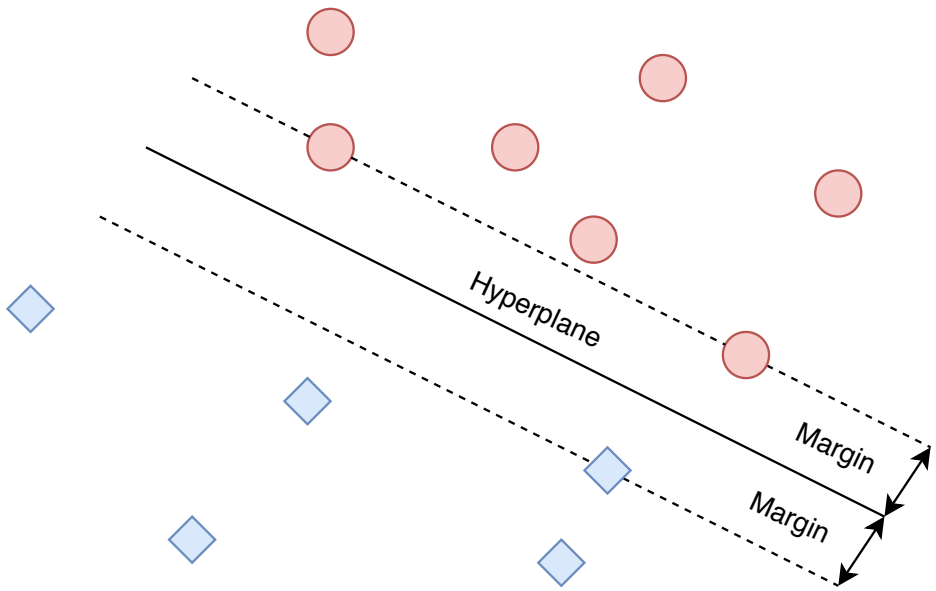

Figure 2.1: A hyperplane defined by a SVM that separates samples of negative and positive classes with maximal margin

the distance to all samples has to be calculated [4]. kNNs are more frequently used for unsupervised tasks, such as clustering, rather than supervised tasks [8].

2.2.3 Support Vector Machine

Support vector machines (SVM) are supervised machine learning algorithms that are used for classification and regression analysis. A single SVM algorithm separates data to two classes by defining a hyperplane that has a maximal distance (margin) to examples of opposing classes (Figure 2.1). The hyperplane is defined with labeled training data and prediction happens by defining the side in which the example is placed in. If a hyperplane which cuts the data perfectly so that each example is on it's own side is not possible, the algorithm tries to find a division so that as few an example are on the wrong side as possible [4].

In the case that the given classes can not be separated linearly, SVM transforms ("maps") the input space into a higher dimensional space in which regions can be linearly

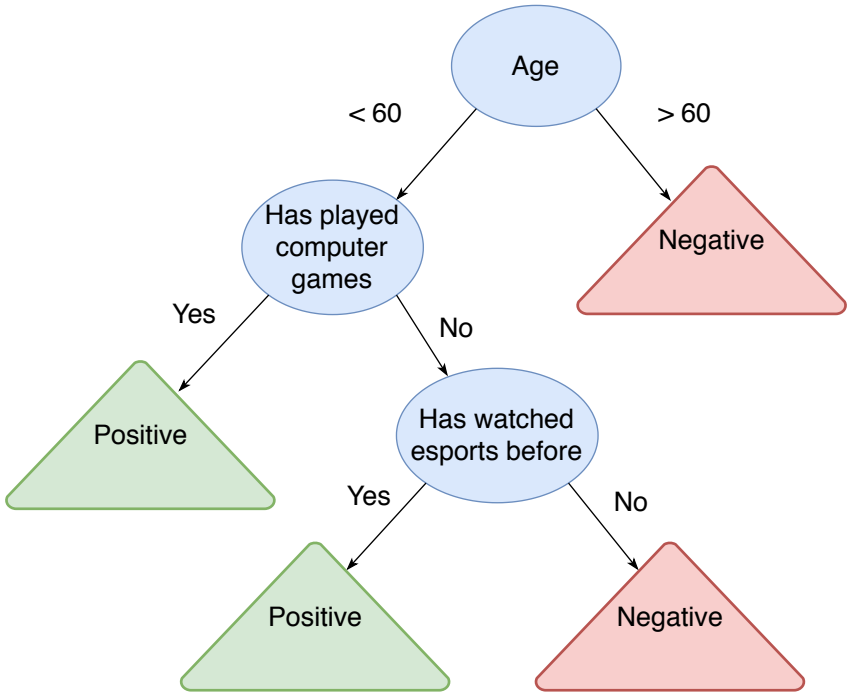

Figure 2.2: Example of a simple decision tree which predicts the sentiment of a person regarding esports

separated [8]. Support vector machines can also be used for unsupervised learning, when there is no labeled data, to find a natural grouping by using Support Vector Clustering (SVC) [10].

SVMs have shown good results in text categorization in the past, are quite computationally efficient and generalize well. Another strength of SVM is that it rarely requires feature selection given that it inherently picks support vectors (individual datapoints) needed for good classification [4].

2.2.4 Decision Trees

Decision trees are classifiers that apply a set of rules sequentially to reach a decision. They can be used for both regression and classification problems and have been among the most popular approaches used for text classification in the past [4] [11].

Figure 2.2 depicts a simple decision tree with three round internal nodes and four

triangular leaves. Nodes are labeled with the testable attribute and branches with the attribute's values.

The training process of a text classification decision tree is as follows: Given a training set M with labelled documents, find the word t_i that best predicts the class of the documents. Partition the training set into two subsets, M_i^+ and M_i^-, with M_i^+ containing examples with t_i and M_i^- containing examples without t_i. Apply this procedure recursively to M_i^+ and M_i^- until all the documents in a subset belong to the same class L_c. The generated tree of rules has an assignment to a class as its leaves [4].

As can be seen from figure 2.2, individual decision trees are quite simple to understand and to interpret, but are usually not competitive with other supervised learning approaches. They can, however, be dramatically improved by combining multiple trees together to get a consensus prediction with approaches such as *bagging, boosting and random forests* with the cost of losing some interpretability [12].

2.2.5 Artificial neural network

A neural network consists of layers of simple processing elements called neurons that are connected to each other. Each connection has an associated weight that is applied to input. The first layer is called the input layer after which comes any number of intermediate, or hidden, layers followed by a final output layer. Neurons are not interconnected within a layer but are only connected to the neurons in adjacent layers. In a text classifier neural network the prediction of the network can be determined from the values of the final layer's neurons. For example, a classifier with three different possible classes would have three neurons in the output layer, each corresponding to the probability of a single class [13].

Neural networks are usually trained using backpropagation which finetunes the parameters of the network by first feeding the network training data, checking the output and if it is misclassified, and then backtracking and updating the weights of the network

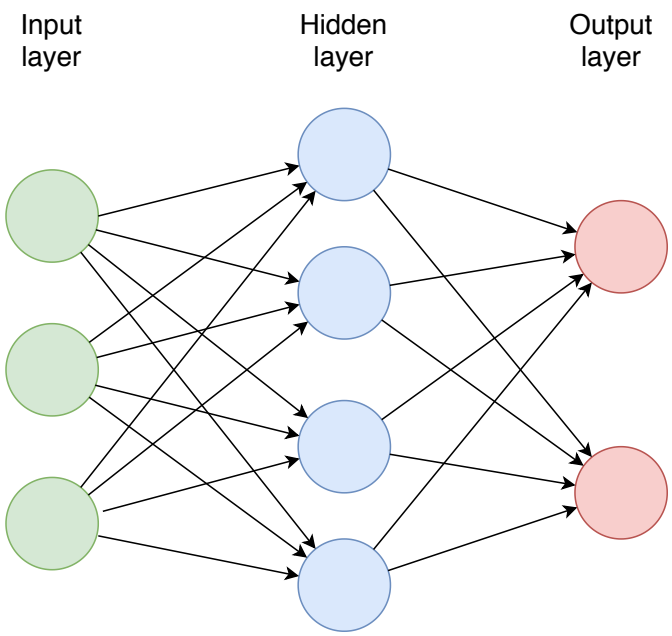

Figure 2.3: Example of a feedforward neural network

to eliminate or minimize the error [5].

The simplest neural network is the single-layer perceptron, introduced by Rosenblatt in 1958 [14], which consists of a single hidden layer between an input layer and an output layer.

Since neural networks consist of simple building blocks, layers of neurons or other functions, stacked on top of each other, it allows designing of deep, complex architectures with practically infinite ways of combination. The depth of a neural network is defined by the number of it's hidden layers. The latest state-of-the-art neural network models consist of very deep networks with millions of parameters.

2.3 Evaluation

Evaluating the performance of a classifier is an important step in order to define if the trained classifier actually learned something.

True positive, true negative, false positive and false negative (TP, TN, FP, FN,

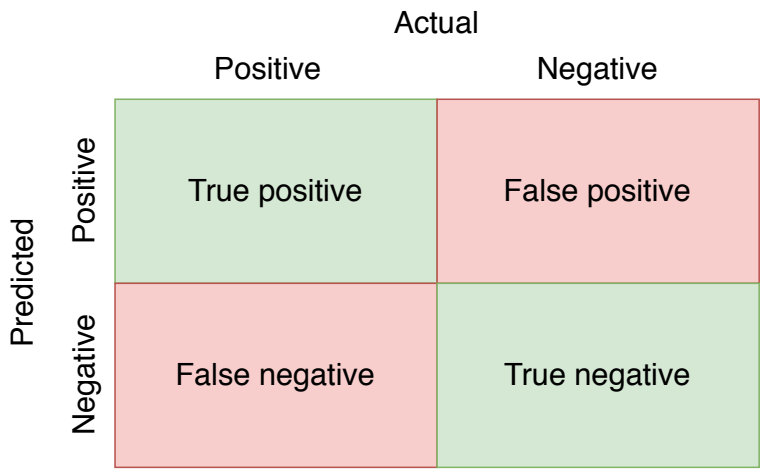

Figure 2.4: A confusion matrix.

respectively), are labels used to define the classification result of a single example in a classification task. If the classifier correctly predicts that the example is positive (e.g. a piece of text belongs to class 1), the example is considered a true positive. In the case when a negative example is correctly predicted, it is a true negative. If the classifier wrongly predicts that the example is positive when it is actually negative, or vice versa, the result is a false positive of a false negative. Depending on the case, a classifier can have requirements such as to minimize the false negative or false positive rate.

Figure 2.4 depicts a **confusion matrix** which is a common way to visualize classification results.

The simplest metric to use is **accuracy** (equation 2.3), which defines the fraction of correctly classified documents (true positives and true negatives) in relation to the total number of documents. It is a raw metric that doesn't give a whole lot of insight into the performance of the classifier in such cases where the training data is not evenly distributed.

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

Precision (equation 2.4) defines the accuracy for positive example predictions, e.g. how many of the texts that we classified as 1 were actually 1. It only takes into account

positive predictions that were either true or false.

$$\frac{TP}{TP + FP} \tag{2.4}$$

Recall (equation 2.5) defines the proportion of positive examples actually predicted as positive, e.g. how many of the positive examples did the classifier get right.

$$\frac{TP}{TP + FN} \tag{2.5}$$

Specificity (equation 2.6) defines the proportion of negative examples that were correctly predicted as negative, e.g. how many of the negative examples did the classifier get right. Specificity is the opposite of recall.

$$\frac{TN}{TN + FP} \tag{2.6}$$

F-score or F_1 score (equation 2.7) combines recall and precision in an effort to capture their properties in a single value. It is the harmonic mean of precision and recall.

$$F = \frac{2}{1/recall + 1/precision} \tag{2.7}$$

F-score is a commonly used metric to describe system performance in machine learning but it is criticized for lacking detail [15]. Two different models that have the same F-score on the same task are not necessarily successful in the same way.

Complementarity [16] is a measure of the difference in decisions made by two classifiers which attempts to capture the properties that F-score misses. It represents the amount of times when the other classifier was correct and the other wasn't [15].

Clearly, choosing and reporting only a single metric — such as accuracy — as the final performance score of a classifier doesn't truly represent the goodness of a model. A more full understanding of the shortcomings and strengths of a model is achieved by calculating a number of metrics, such as precision and recall.

Chapter 3

Deep learning in natural language processing

Deep learning is a branch of machine learning and a re-branded name for neural networks. All of machine learning can be defined as learning to predict the future based on past observations. In addition to learning to predict, deep learning also tries to correctly represent the data so that is is suitable for prediction. Given a large amount of desired input-output mappings, deep learning feeds the data into a network that successively transforms the data until a final transformation predicts the output. The transformations are learned from the mappings such that each successive transformation makes it easier to relate the data to the given label. After a human designer defines the architecture and training regime of the network, gathers and preprocesses a proper set of input-output examples and encodes the data, the network can automatically learn the how to best produce this mapping [1].

This chapter gives an overview of transfer learning in section 3.1, delves deeper into how a deep learning model is trained in section 3.2, provides an overview of different relevant architectures in section 3.3 and finally describes the methodology of three modern machine learning models: ULMFit, BERT and ELECTRA, in section 3.4.

3.1 Transfer learning

Transfer learning refers to the ability of a system to reuse knowledge learned in a previous task in a task of new or novel domain that shares some commonality with the previous task [17]. Often the motivation for using transfer learning comes from a lack of labelled training data for a specific task. In natural language processing it is widely used to first teach a model on a general language corpus, so that it learns the generalities of the given language (a *language model*), and then fine-tuning the model on some downstream task, such as classification.

All of the introduced NLP-models later in this chapter use transfer learning as each has a clear distinction between the pre-training and fine-tuning phases in model training where another model's pre-trained state can be transferred to another task and fine-tuned further for that task.

3.1.1 Language modeling

Language modeling is the task of assigning a probability to sentences in a language. In addition to assigning a probability to sequences of words, language models also define the probability that a word follows a sequence of words. Language modeling is an important part in several real-world applications such as speech recognition and machine-translation, where language models are used to score the transcriptions and translations that the systems output.

The formal definition of language modeling is to assign a probability to any sequence of words $P(w_{1:n})$, which can be rewritten as:

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{1:3})\dots P(w_n|w_{1:n-1}). \tag{3.1}$$

using the chain-rule of probability. What this means is that the probability of a sequence of words is defined by the probabilities of each word n following the previous

$n-1$ words. In the past, language models made use of the *markov-assumption* that states that the future is independent of the past given the present. Nowadays, with modern architectures such as recursive neural networks, this assumption can be abandoned and models can condition on entire sentences while taking word order into account, which has lead to impressive gains in language modeling [1].

Modern NLP networks are often pre-trained as language models before they are fine-tuned for a specific task. Pre-trained language models can thus be used as a starting point for a model, transferring knowledge from a previous task.

3.1.2 Embeddings

A common use case of transfer learning in NLP has been to use word embeddings that encode some information about a word in relation to other words in the feature space. Using embeddings is a general way of transferring knowledge as it doesn't depend on any specific model architecture but only on the language used. The most popular methods for generating vector representations for words have been Word2Vec [18], GLoVe [19], and fastText [20], all of which will be presented in the next sections.

Word2Vec

Word2Vec is an open-source project based on the work by Mikolov et al. that can be used to train distributed representations of words and phrases [18].

It uses a skip-gram model (figure 3.1), proposed by Mikolov et al. in an earlier work [21], which is a prediction-based method. The training objective of the skip-gram model is to find useful word representations for predicting the surrounding words in a sentence or a document. It learns these representation by predicting the surrounding words for each word in a sentence within a defined max distance from the word. Mikolov et al. show, that the produced representations exhibit linear structure that makes precise analogical reasoning possible [18].

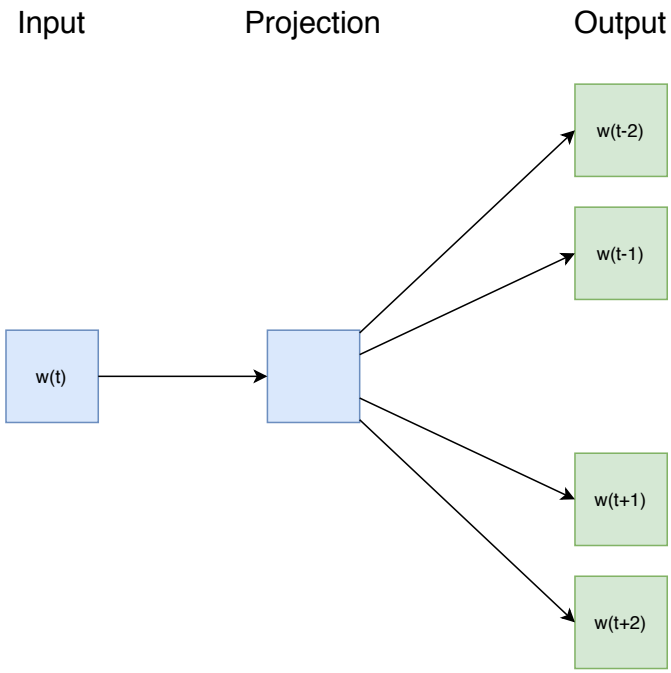

Figure 3.1: Skip-gram model architecture.

Given the computationally efficient model architecture of the skip-gram, the training times of Word2Vec are manageable even with huge amounts of data.

GLoVe

Global Vectors for Word Representation (GLoVe) is a model to construct word representations. It is a global log-bilinear regression model that combines the advantages from both Word2Vec-style local context window methods and global matrix factorization. Training of GLoVe is done on aggregated global word-word co-occurrence statistics [19].

Given the same corpus and equal compute, Pennington et al. show that it outperforms Word2Vec and achieves better results faster [19], although Levy et al. [22] came to the opposite conclusion after careful testing.

fastText

fastText is an open-source library for learning word embeddings and text classification. It builds on the work of Word2Vec and improves on the skip-gram model by incorporating character n-grams in it. Words are now represented as a sum of n-gram vectors instead of a single vector. This is especially important for morphologically rich languages, such as Finnish, that contain many word forms that occur rarely in the training corpus, which makes learning good word representations difficult [20].

Mikolov et al. show that fastText significantly outperforms GLoVe on a number of tasks [23].

3.2 Training

The goal of a neural network is to return a function $f()$ that accurately maps input examples to their labels. To make it more precise, a *loss function* is introduced to quantify the loss suffered when predicting examples in the training set. A loss function assigns a numerical score to predicted outputs given the expected, true outputs. The function is bounded from below such that the minimum value is only attainable in cases where the prediction is correct. The goal of the training algorithm is then to minimize the average loss over all training examples [1].

Attempting to minimize the loss at all costs may result in *overfitting* the training data, thus a soft restriction on the loss function is applied in the form of a *regularization term*, which tracks the "complexity" of parameters. Thus the objective of the optimization problem becomes keeping a balance between low loss and low complexity [1].

3.2.1 Loss functions

Binary cross entropy loss

Binary cross entropy loss, also referred to as *logistic loss* is a loss function used in binary classification with conditional probability outputs. A set of two target classes labeled 0 and 1 is assumed, with a correct label $y \in \{0, 1\}$. The output of the classifier, $\tilde{y}$, is transformed to the range $[0, 1]$ using the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$, and is interpreted as the conditional probability $\hat{y} = \sigma(\tilde{y}) = P(y = 1|x)$. The rule for prediction is:

$$prediction = \begin{cases} 0, & \hat{y} < 0.5 \\ 1, & \hat{y} \geq 0.5 \end{cases} \tag{3.2}$$

The network maximizes the log conditional probability $\log\ P(y = 1|x)$ for each training example (x, y). Logistic loss is defined as:

$$L_{logistic}\,(\hat{y}, y) = -y \log\ \hat{y} - (1 - y) \log\,(1 - \hat{y}). \tag{3.3}$$

The logistic loss is useful when we want the network to produce class conditional probability for a binary classification problem. It is assumed that the output layer is transformed using the sigmoid function when using the logistic loss [1].

Categorical cross-entropy loss

The categorical cross-entropy loss is a loss function that is used when a probabilistic interpretation of the scores is desired. Let $y = y_{[1]}, \ldots, y_{[n]}$ be a vector representing the true multinomial distribution over the labels $1, \ldots, n$, and let $\hat{y} = \hat{y}_{[1]}, \ldots, \hat{y}_{[n]}$ be the linear classifier's output transformed by the *softmax* function, and represent the class membership conditional distribution $\hat{y}_{[i]} = P(y = i|x)$. The softmax function forces the values of the output to be positive and sum to 1, thus making the output interpretable as a probability distribution.

The categorical cross entropy loss measures the dissimilarity between the true label distribution y and the predicted label distribution $\hat{y}$, and is defined as cross entropy:

$$L_{cross-entropy}\,(\hat{y}, y) = -\sum_i y_{[i]} \log(\hat{y}_{[i]}). \tag{3.4}$$

The case when each training example has only a single correct class assignment is called hard classification. In such cases y is a one-hot vector representing the true class, and the cross entropy can be simplified to:

$$L_{cross-entropy(hardclassification)}\,(\hat{y}, y) = -\log(\hat{y}_{[t]}), \tag{3.5}$$

where t is the correct class assignment. This equation attempts to set the probability mass assigned to the correct class t to 1. Increasing the mass assigned to the correct class means decreasing the mass assigned to all the other classes given that the scores $\hat{y}$ have been transformed using the softmax function to be non-negative and sum to one.

The cross-entropy loss is widely used in log-linear models and neural networks. It produces a multi-class classifier which predicts a distribution over all possible labels in addition to predicting the best class label. When using the cross-entropy loss, it is assumed that the classifier's output is transformed using the softmax function [1].

Ranking losses

Margin-based ranking loss can be used in cases where supervision is not given as labels but as pairs of correct and incorrect samples x and x', and where the goal is to give a higher score to correct items. Such a situations may occur when the training set consists of only positive examples and the generation of negative examples is done by corrupting positive ones. Margin-based ranking loss is defined as follows for a pair of correct and incorrect samples:

$$L_{ranking(margin)}(x, x') = \max(0, 1 - (f(x) - f(x'))), \tag{3.6}$$

where $f(x)$ is the score assigned by the classifier for input vector x. The objective of the function is to rank correct inputs above incorrect ones with a margin of at least 1.

Ranking loss is used in language tasks such as deriving pre-trained word embeddings (section 3.1.2) given a correct and corrupted word sequence, and the goal being to rank the correct sentence over the corrupt one [1].

3.2.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a general optimization algorithm which powers nearly everything in deep learning. The goal of the algorithm is to minimize the total loss over the training set by repeatedly sampling a training example and computing the gradient of the error on the example [1].

Sampling each individual example and calculating a gradient for them quickly becomes infeasible when training set sizes are large, thus SGD uses **minibatching** to draw a uniform set of examples at a time from the training set to compute a gradient for. The gradient is then propagated though the network and a new gradient is calculated from the next minibatch [24].

Learning rate is a parameter that defines the amount that the weights of the network are updated with each gradient. It is a small positive value, usually in the range $[0, 1]$.

Gradient descent has been regarded as slow or unreliable in the past. Nowadays we know that it works great with neural networks even though it is not guaranteed that the algorithm will arrive even at a local minimum in a reasonable amount of time. It however finds a very low value for the cost function quickly enough [24].

In addition to SGD, there exists other optimization algorithms which are used nowadays such as Adam [25], which is designed to define the learning rate on a minibatch basis. Algorithms such as SGD+momentum [26] are variants of SGD where the accumulated previous gradients affect the current update.

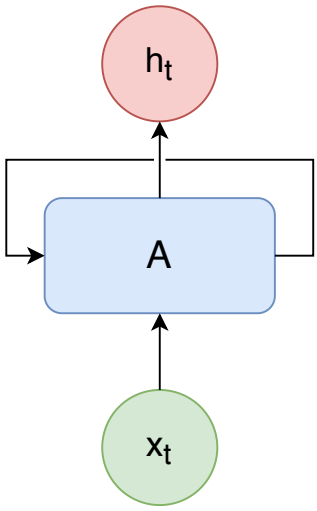

Figure 3.2: An RNN-module which illustrates the recurrent nature of the model

3.3 Architectures

3.3.1 Recurrent Neural Network

Recurrent Neural Networks (RNN) are networks that process an input sequence one token at a time and maintain a state in its hidden units that contains information about the past elements in the sequence. This approach has been proven to work well with tasks that contain sequential input such as speech, language and music [27]. Figure 3.2 shows the basic methodology behind an RNN: a chunk of neural network, A, looks at the input x_t and outputs a value h_t. The output value is looped back as a second input value which allows for information to be passed from one step to the other.

Figure 3.3 shows a visualization of the insides of an RNN module which is quite simple in practice. x_t represents an input at timestep t, $tanh$ is a function that returns the hyperbolic tangent of the input and h_t is the hidden state of the network at timestep t. First the hidden state from the previous timestep and the current input value are added to each other after which the sum is fed into the $tanh$ function. $tanh$ essentially squishes the input value between -1 and 1 to keep the values from exploding due to repeated multiplication. The output of tanh is the new hidden state, the memory of the network, which is then fed

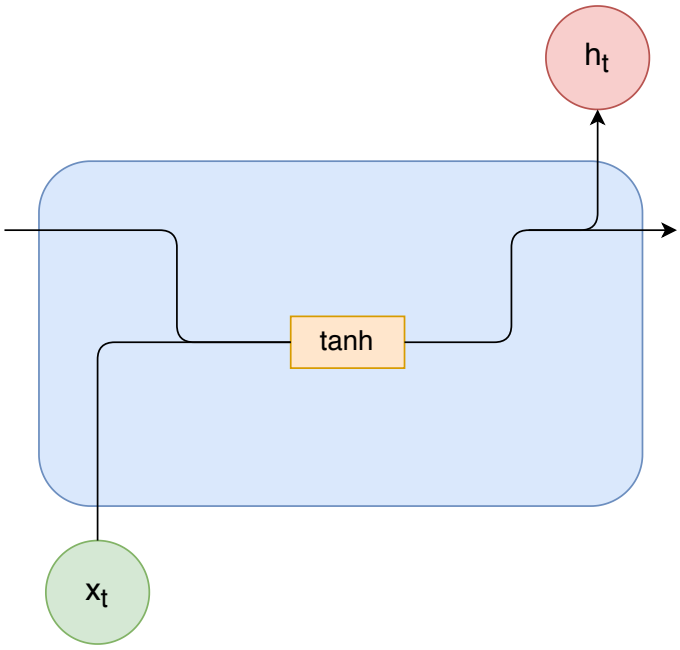

Figure 3.3: The insides of an RNN

to the next timestep.

The training of an RNN happens by using a variant of backpropagation called *backpropagation through time* (bptt) which is a generalization of backpropagation for networks which store the activations of units while going forward in time [28]. The backward gradient update pass is thus also backward in time and recursively computes the required gradients with the saved activations. It is easy to see how this works when the different timesteps of an RNN are unrolled and displayed as if they combine to make a single neural network with multiple layers (fig 3.3).

RNNs have difficulties maintaining long-term dependencies when processing lengthy input sequences that originates from the *vanishing gradients* problem [29]: During backpropagation gradients, with which the weights of the network are updated, change as they are applied backwards through time. A small change to a layer before means an even smaller change in the current layer. On the contrary, gradients with big changes tend to "blow up". This means that the earlier layers in the network either stop learning since they only receive small gradient updates or their weights oscillate due to big changes [30]. In

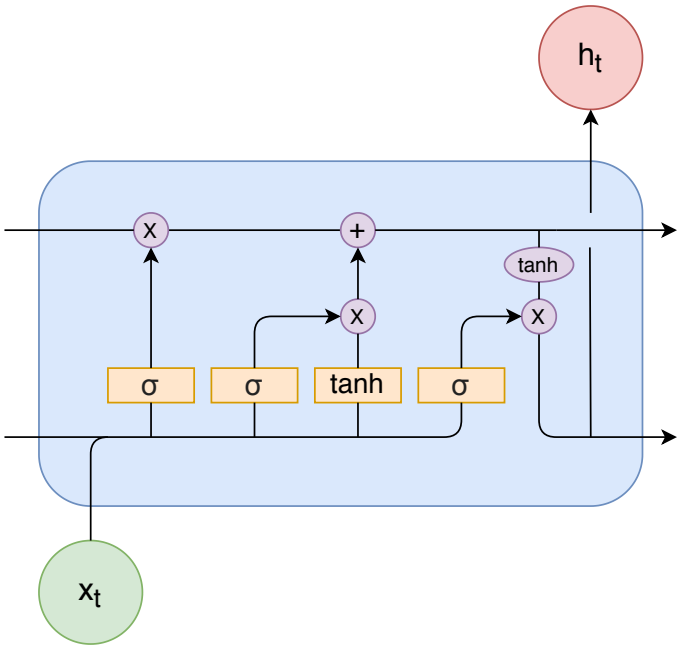

Figure 3.4: Long short-term memory network

an RNN this problem is magnified due to backpropagation being applied to each time step.

3.3.2 LSTM

Long short-term memory (LSTM) is a recurrent network architecture proposed by Hochreiter and Schmidhuber in 1997 [30]. LSTM was designed to combat the vanishing gradients problem that is especially prevalent in RNNs. LSTMs work exceptionally well on a large variety of problems and are widely used nowadays.

Figure 3.4 illustrates an lstm module. Each line carries a vector, merging lines denote concatenation, forking lines denote copying of the vector and each copy going to different directions, orange boxes are learned neural network layers and purple circles represent pointwise operations such as multiplication, addition and hyperbolic tangent. In addition to keeping track of the hidden state of the network, LSTM adds another state called cell state that is denoted by the horizontal line running through the top of the figure. Information is added and removed to the cell state by gates that are made up of sigmoid

(σ) neural net layers and pointwise multiplication operations. A sigmoid neural net layer takes as input the concatenation of the previous hidden state h_{t-1} and the current input x_t and output a vector of values between 0 and 1 to describe how much of each value is to be let through. A value of 1 lets everything through while a 0 lets nothing through. An LSTM has three of these gates. From left to right, the first gate forms the forget gate layer which decides what data to keep and what to discard from the previous cell state. Next, the data to add to the cell state is decided with a combination of a $sigmoid$ layer and a $tanh$ layer. The $tanh$ layer outputs new candidate values and the $sigmoid$ layer decides which of them to add to the cell state. Finally, the last layer determines the output (hidden state) of the cell by taking the current cell state's values and applying a $tanh$ function to push the values between -1 and 1 and multiplying them with the output of the $sigmoid$ gate. The new cell state and hidden state are then passed on to the next time step.

The LSTM architecture described above is considered a standard version of LSTM, but other variants exist too. One popular variant of LSTM introduced by Gers & Schmidhuber [31] adds peephole-connections that allow the gate layers to look at the cell state. Another variant called the **Gated Recurrent Unit** (GRU, figure 3.5), introduced by Cho, et al. [32], simplifies the model by combining the forget and input gates into a single update gate.

3.3.3 Transformer

RNNs, as presented before, are inherently sequential in nature. They take an input at timestep t and compute a hidden state h_t with knowledge from the previous hidden state h_{t-1}. This sequentiality prohibits efficient parallelization within training examples since one has to come before the other in training. Parallelization across examples is also critically constrained by memory at longer sequence lengths. In addition to this, RNNs suffer from the so-called *vanishing gradient problem* which is exacerbated at longer sequence lengths [33].

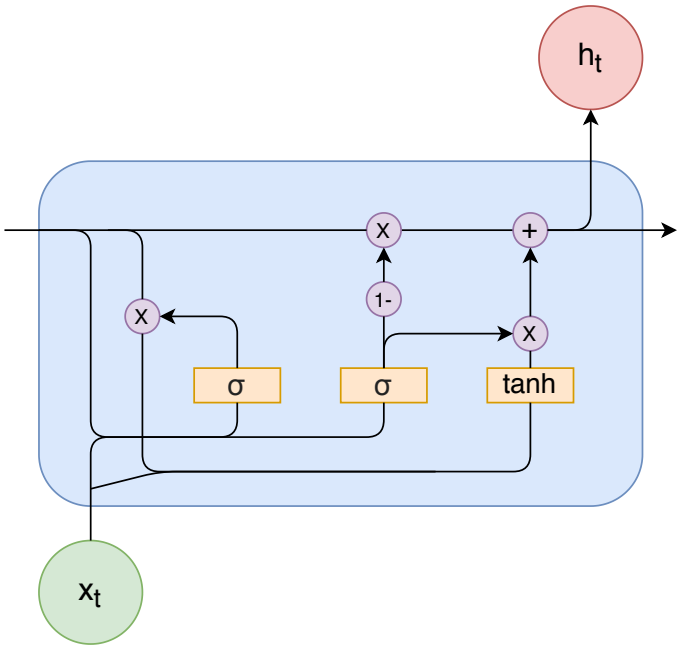

Figure 3.5: Gated Recurrent Unit

Attention is a mechanism that allows the modeling of dependencies without regard for distance in input or output sequences. It has been used in conjunction with recurrent neural networks to achieve good results, but using it with RNNs somewhat limits the power of attention since the model is still constrained by the aforementioned problems of RNNs. Thus Vaswani et al. proposed a novel architecture in 2017, the Transformer, to combat these limitations. The Transformer has been the foundation of neural networks that have achieved state-of-the-art results in various language-related tasks in the last couple of years [33].

The Transformer consists of an encoder, which maps an input sequence of tokens to a sequence of continuous representations, and a decoder, which takes a continuous representation and generates an output sequence. The output sequence is generated one token at a time while taking the previous generated tokens as additional input.

The overall architecture of the Transformer is shown in Figure 3.6, with the left side of the figure representing the encoder and the right side the decoder. The encoder is composed of six identical layers stacked on top of each other. Each layer consists of two

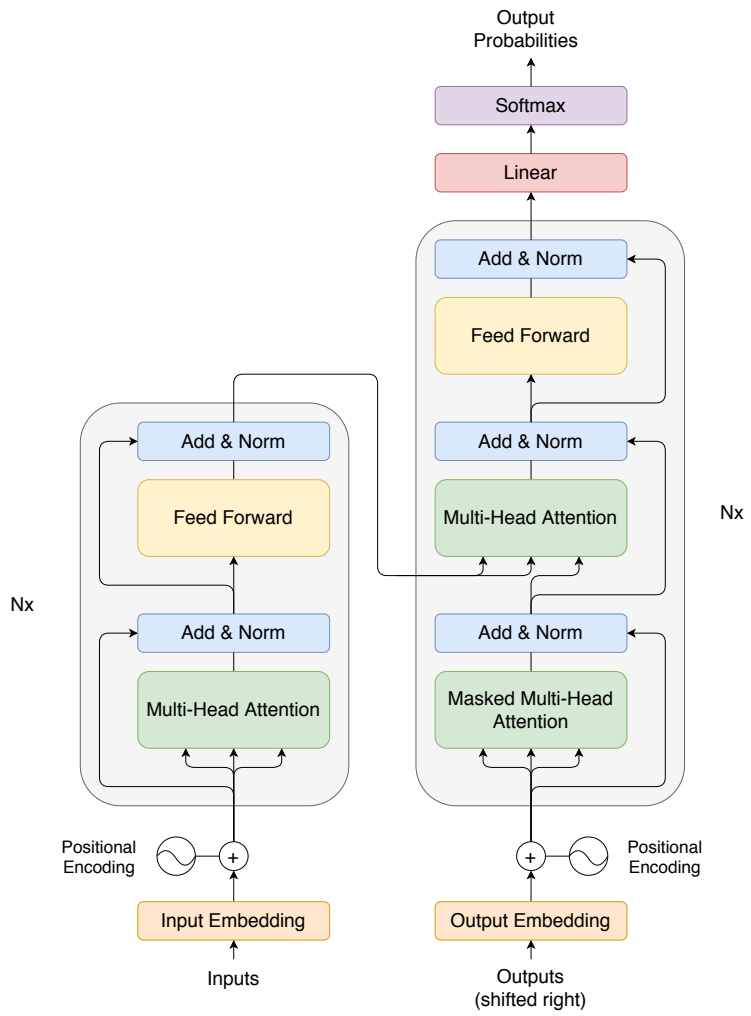

Figure 3.6: The Transformer, from Vaswani et al. 2017 [33]

sub-layers; a multi-head self-attention mechanism and a fully connected feed-forward network. Residual connections [34] are used around each sub-layer to shortcut the sub-layers while training. This leads to faster training times and a more robust model as the connections are gradually restored during training. Finally, layer normalization is applied to the output of the sub-layer as $LayerNorm(x + Sublayer(x))$, where $Sublayer(x)$ refers to the function implemented by the sub-layer. Due to the residual connections, all sub-layers and embedding layers have to produce outputs of the same dimension. Thus in the Transformer the dimension is defined as $d_{model} = 512$.

The decoder is also composed of six identical layers but additionally includes a third

sub-layer, which applies multi-head attention over the output of the encoder stack. The self-attention sub-layer in the decoder stack is also modified to prevent positions from attending to subsequent positions. Thus predictions for position i can only depend on the outputs before i.

Attention can be described as a function of mapping a query and a set of key-value pairs to an output. The query, keys, values and output are all vectors. The output is a weighted sum of the values, where the weight of each value is defined by a compatibility function of the query with the corresponding key. The particular version of attention in the Transformer is called *Scaled Dot-Product Attention* in which the input consists of queries and keys of dimension d_k and values of dimension d_v. The dot products of queries with all keys are computed first, denoted by QK^T in equation 3.7. The results are divided by $\sqrt{d_k}$ and then a softmax function is applied to obtain the weights on the values [33].

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.7}$$

Instead of using a single attention function over all the keys, values and queries with dimension d_{model}, the Transformer uses so-called *Multi-Head Attention* to linearly project the keys, values and queries h times to d_k, d_k and d_v dimensions. The attention function is then applied in parallel to all these projected versions, yielding d_v-dimensional outputs. These outputs are then concatenated and projected to achieve the final result. This allows the model to jointly attend to information from different representation subspaces at different positions. In the Transformer, 8 parallel attention heads are used [33].

Attention is used in three different ways in the Transformer. In encoder-decoder attention, where the output of the encoder is used in the decoder, the queries come from the previous decoder layer and the keys and values come from the output of the encoder, and in encoder and decoder self-attention. In encoder self-attention all the queries, keys and values come from the output of the previous encoder layer, thus each position in the encoder can attend to all positions in the previous encoder layer. Decoder self-attention

similarly receives its input from the previous decoder layer's output, but doesn't allow attention over all the positions but only up to and including the current position. This is achieved by masking out all the input values of the softmax corresponding to illegal connections.

The decision to use self-attention was made based on three requirements: to minimize the total computational complexity of each layer, to maximize the amount of parallelizable computation and to minimize the path length between long-range dependencies. A side benefit of self-attention is more interpretable models as attention distributions can be inspected and tested with different examples to gain insight into the behaviour of individual attention heads [33].

3.3.4 Other architectures

Autoencoder

Autoencoders are neural networks that are trained to attempt to copy its input to its output. They compress the input into a lower-dimensional representation after which they attempt to reconstruct the original input from this representation. An autoencoder that succeeds at a perfect copy is not very useful, thus they are designed to be unable to learn to copy perfectly. Because the model has to prioritize which aspects of the input should be copied, it often learns useful properties of the data [24].

An autoencoder consists of three components: an encoder, code and a decoder. The encoder compresses the input and produces the code, the decoder reconstructs the input from the code. Figure 3.7 illustrates the simple architecture of the autoencoder. Both the encoder and the decoder are fully connected feedforward networks and are essentially mirror images of each other. The only requirement of the autoencoder is that the input and output dimensions are the same.

Autoencoders have traditionally been used for dimensionality reduction or feature learning [24]. For natural language processing, Variational Autoencoders [35] have been

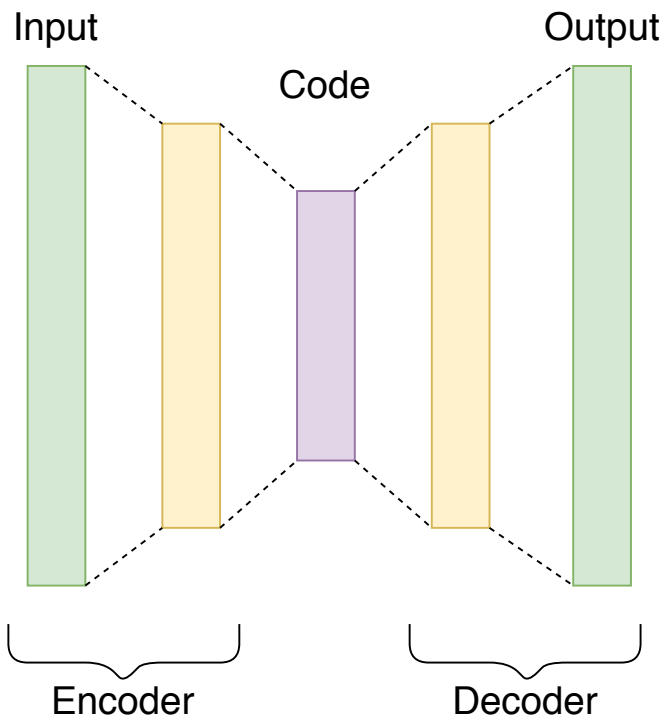

Figure 3.7: The autoencoder

used for generative document modelling and supervised question answering [36].

Convolutional Neural Network

Convolutional Neural Networks (CNN) are networks that incorporate convolutional and pooling layers into a neural network. CNN's have four key ideas that take advantage of the properties of signals: shared weights, pooling, local connections and the use of many layers. A typical CNN (Figure 3.8) is structured as a series of stages. The first stages are composed of convolutional layers and pooling layers. Convolutional layers try to detect local conjunctions of features from the previous layer. A convolutional layer's units are organized in feature maps, within which each unit is connected to local patches in feature maps of the previous layer. These connections are weighted and the weights are contained in a filter bank, that is shared within all the units of a feature map. Every feature map thus has it's own filter bank. The role of the pooling layers is to merge semantically similar features into one. They reduce the dimension of the representation in an effort to

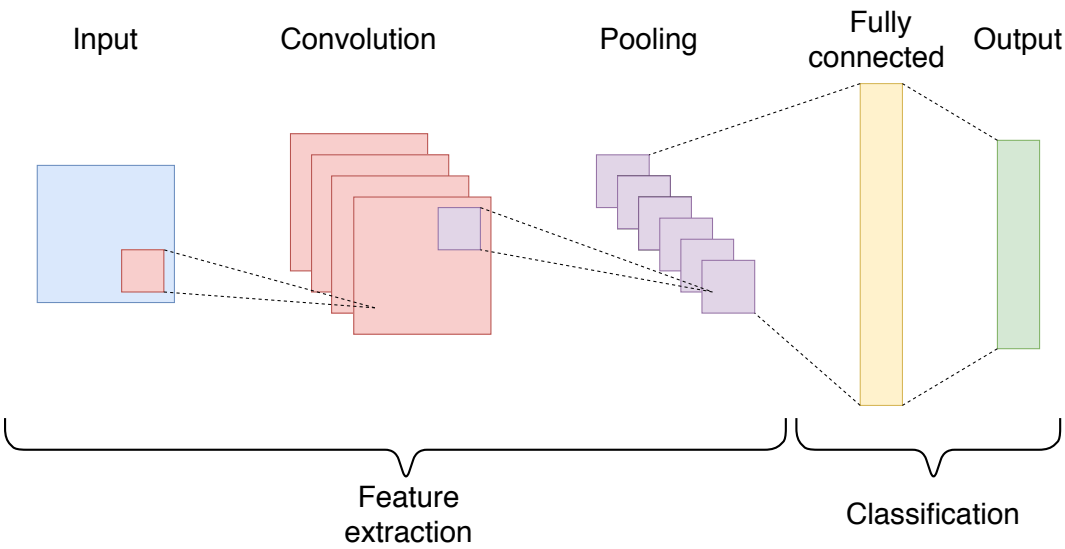

Figure 3.8: A convolutional neural network for classification

make the network more robust against small shifts and distortions in the data. Stacks of convolutions and pooling are finally followed by fully-connected layers that are trained to output the result [27].

Convolutional neural networks have been used for a multitude of tasks in natural language processing in the past such as sentiment classification and sentence modeling [37].

3.4 Modern models

3.4.1 ULMFiT

Universal Language Model fine-tuning (ULMFiT) is a transfer learning based methodology for text classification which posted state-of-the-art results when it was published in 2018. It consists of firstly pre-training a language model on a general-domain corpus and then fine-tuning it on a classification task. This idea of first pre-training on a large general corpus and then fine-tuning it has been tried before, but has proven to be a challenging task due to it requiring millions of in-domain documents to achieve good performance [38]. With ULMFiT Howard et al. proposed novel training techniques to make the train-

ing feasible even with a small corpus [2].

ULMFiT is a *universal* method in that it uses a single architecture and training process, requires no custom feature engineering, works across tasks with variable document sizes and label types, and doesn't require additional in-domain documents or labels [2].

ULMFiT uses a 3-layer weight-dropped long shot-term memory (AWD-LSTM) network, proposed by Merity et al. [39], which is a recurrent neural network (RNN). AWD-LSTM is a vanilla LSTM with various regularization and optimization strategies such as DropConnect [40], which prevents over-fitting by randomly dropping connections between the recurrent hidden to hidden weight matrices, and averaged gradient descent as its optimization algorithm.

The training begins with pre-training a general-domain language model with unlabeled text data to capture the general properties of language. This initial training step is the most expensive in the whole method but only needs to be done once.

After the general language model is trained, it is fine-tuned with the target task data. This fine-tuning converges faster than the initial pre-training since the model needs to only adapt to the idiosyncrasies of the fine-tuning data. This allows the training of robust language models even with small datasets. ULMFiT uses *discriminative fine-tuning* and *slanted triangular learning rates* for the fine-tuning step of the language model. With discriminative fine-tuning, instead of using a single learning rate for all the layers in the model, each layer is fine-tuned with a learning rate of its own. The motivation for this comes from the fact that different layers capture different types of information, thus each layer should be fine-tuned for different amounts. With slanted triangular learning rates, the learning rate is first linearly increased in order to get the model to quickly converge to a suitable region, and then linearly decreased to refine its parameters.

For the final step, two additional linear blocks are added to the end of the network, and the final classifier is fine-tuned with *gradual unfreezing*. Gradual unfreezing is used to prevent *catastrophic forgetting* by unfreezing the layers of the model one by one, starting

from the last. Unfrozen layers are fine-tuned for one epoch after which the next layer is unfrozen until all the layers of the model have been unfrozen. The whole model is then fine-tuned until convergence [2].

Although outshined by BERT and other huge transformer-based models, ULMFiT is much fast to train and doesn't require a lot of data to get good results.

3.4.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a language representation model based on the Transformer [33] architecture (subsection 3.3.3). It uses a *fine-tuning* based approach, in which all the pre-trained parameters are fine-tuned when applying pre-trained language representations to down-stream tasks, as opposed to a *feature-based* approach, where the pre-trained representations are only used as additional features in a task-specific architecture.

Both the fine-tuning and feature-based approach share the same objective function during pre-training of learning general language representations by using unidirectional language models. BERT uses a *masked language model* (MLM) pre-training objective to achieve bi-directionality in context, thus allowing the model to see both left and right of the input token when training. MLM randomly masks tokens in the input, and the objective is to predict the vocabulary id of the masked token based on its context. BERT also uses a *next sentence prediction* (NSP) task that jointly pre-trains text-pair representations [41].

As with ULMFiT, BERT also has a unified architecture across different tasks with a minimal difference between the pre-trained architecture and the final downstream one. BERT's architecture is almost identical with the Transformer described in section 3.3.3, the difference comes mainly in the number of layers (Transformer blocks), denoted as L, hidden size, denoted as H and the number of attention heads, denoted as A. Results for BERT performance was primarily reported for two sizes: $BERT_{BASE}$ (L=12, H=768,

A=12, total parameters=110M) and $BERT_{LARGE}$ (L=24, H=1024, A=16, total parameters=340M).

BERT uses WordPiece embeddings [42], essentially similar to SentencePiece embeddings (section 2.1.3). Pre-training happens by the two aforementioned unsupervised tasks: Masked language modeling and next sentence prediction.

In MLM, 15% of all WordPiece tokens are randomly chosen and each chosen token has an 80% chance to actually be masked, a 10% chance to be replaced with a random token, and a 10% chance to stay unchanged. The reason for not always masking the chosen tokens is because the [MASK] token does not appear during fine-tuning, causing a mismatch between the two steps otherwise.

In NSP, training examples consist of two sentences A and B. 50% of the time B is actually the next sentence that follows A (label $IsNext$) and 50% of the time its a random sentence from the corpus (label $NotNext$).

With the use of the Transformer-architecture, WordPiece embeddings and two pre-training objectives that allow bidirectional context, BERT was able to exceed the state-of-the-art on multiple downstream tasks [41].

3.4.3 ELECTRA

Efficiently Learning an Encoder that Classifies Token Replacements Accurately (ELECTRA) is a further elaboration on the BERT model by Clark, et al. [43] The primary motivation for ELECTRA is making pre-training more efficient given that training a full-sized BERT or any derivation of it (ALBERT, RoBERTa etc.) requires a considerable amount of compute and training data. As described in the BERT chapter (chapter 4.4), BERT uses masked language modeling as a pre-training objective. This MLM is inherently quite inefficient in it's usage of the training data; only the masked tokens, approximately 15% of the data, needs to be predicted. As an alternative, ELECTRA proposes *replaces token detection*, a pre-training task in which the model has to predict whether or not a token

is the original token in the corpus or if it has been swapped for a token generated by a small masked language model. This also solves the mismatch in BERT where the model sees [MASK] tokens during pre-training but not during fine-tuning. ELECTRA trains the network as a discriminator rather than a generator, since it is predicting whether an input is corrupted or not. This way ELECTRA learns from all the input tokens, rather than a small subset of them.

ELECTRA substantially outperforms MLM-based methods given the same amount of data and compute and works well even with relatively small amounts of compute [43].

Chapter 4

Medical report document classification

4.1 Problem definition

Automatically tagging a medical document with a proper diagnosis code was deemed valuable from the hospital's side. To find out if this would be possible, a couple of deep learning -based approaches were chosen to be trained on medical data. Since compute resources were a major constraint, only approaches with a reasonable single-gpu training time were considered. Thus, a pretrained and finetuned ULMFiT and ELECTRA, and a finetuned FinBERT (pretrained by Virtanen et al. [44]), were chosen for comparison. In addition to comparing the different models with each other, each model — excluding FinBERT, which provided a vocabulary of it's own from pretraining — was trained multiple times with different sizes of a vocabulary which was extracted from the medical texts to see if vocabulary size meaningfully impacted the results. Given the success of BERT and it's WordPiece tokenization in various NLP tasks, SentencePiece was chosen as a tokenizer for ULMFiT as well.

4.2 Data

4.2.1 Medical reports

The medical data for the work was provided by Auria services at the Turku university hospital. It consists of doctor's statements with corresponding diagnosis codes and other metadata, such as date of visit and hospital ward, of patients in the Turku-region. No metadata was utilized in the training data, only the written text and the corresponding diagnosis code. Since the actual diagnosis code for the text appeared often in it, all diagnosis codes in text were masked with a [CODE] token in the training data to prevent the model from learning from them. HTML-tags such as <br> were removed in preprocessing and texts that were deemed too short (less than 80 characters) were also removed from the training set.

After extracting the documents that had one of two knee-related diagnosis codes as positive samples, the class balance for the dataset was 85% negative and 15% positive samples. The dataset was eventually balanced by upsampling the positive samples. Overall, the data consisted of 175000 documents with varying lengths. Token amounts were dependent on the size of the vocabulary used.

4.2.2 General Finnish

Since receiving access to the actual medical data took some time and due to the fact that it was not allowed to be taken out of the hospital, the chosen models and code were additionally trained and tested on a general Finnish corpus that Virtanen et al. compiled for FinBERT [44]. It is composed from three different sources: news articles, online discussions, and documents crawled from the Finnish internet, and consists of 3.3 billion tokens from 234 million sentences. The total size of the corpus is roughly 30 times the size of the Finnish Wikipedia. The corpus was extensively preprocessed by filtering out documents that had too high a ratio of digits, uppercase or non-Finnish alphabetic characters, or low

average sentence length. Additionally, documents that featured 25% or more duplication were removed as well as heuristically defined undesirables [44].

4.3 Compute resources

Compute resources for the project were provided by Auria services for the medical models, and CSC for the general Finnish models.

4.3.1 CSC

CSC (IT Center for Science Ltd.) is a non-profit state enterprise owned by the Finnish state and higher education institutions in Finland. It offers compute resources for scientific purposes to universities and upkeeps the FUNET network, which is the Finnish national research and education network. CSC operates two supercomputers, namely Taito and Puhti, and is working on a new supercomputer, Mahti, that is scheduled to open for use some time in 2020. For this project, Puhti was chosen since it provides an "artificial intelligence partition" with access to GPU nodes with multiple Nvidia V100 graphics cards.

Puhti

Puhti was launched on September 2, 2019. It is an Atos cluster system and has a variety of different node types.

Puhti has 682 CPU nodes, with a theoretical peak performance of 1,8 petaflops, and an AI partition of 80 GPU nodes with a peak performance of 2,7 petaflops. Each node is equipped with two Intel Xeon processors, code name Cascade Lake, with 20 cores each running at 2,1 GHz. Each GPU node also has four Nvidia Volta V100 GPUs with 32 GB of memory each. The nodes are equipped with 384 GB of main memory and 3,6 TB of fast local storage. The AI partition is engineered to allow GPU-intensive workloads to

scale well across multiple nodes [45].

When working with Puhti on this project, the workflow consisted of coding and testing the neural networks locally first, and then using Slurm to run batch jobs on Puhti. This lead to some additional overhead in time for the project since working simultaneously on two environments proved quite arduous. Additionally, keeping tabs on the versions of code was very important since it could be altered in both locations, thus git [46] was used for this version control.

Slurm

Puhti uses the Slurm workload manager [47] to handle scheduling jobs for compute clusters. It is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters. First, it manages the allocation of exclusive and/or non-exclusive access to compute nodes to users for some duration of time during which they can perform work. Second, it provides a framework for starting, executing, and monitoring work on the set of allocated nodes. Finally, it manages a queue of pending work to arbitrate the contention for resources [48].

4.3.2 Turku University Hospital

For training models on the clinical data, access was granted to **Blackbird**, a computer for artificial intelligence owned by Auria services, with four Nvidia V100 graphics cards allowing simultaneous training of multiple models. All training and handling of clinical data happened on this computer which is located in the hospital and behind the hospital firewall in order to ensure that the sensitive data and resulting models did not accidentally or otherwise leak into the outer world.

Although the architecture could have managed training a medium-sized BERT, it was considered too long of a task to reserve the compute resources for. Additionally, there wasn't enough domain-specific data for such a task.

The workflow on Blackbird consisted of connecting to the linux-based computer using secure shell (SSH), using screen [49] to multiplex the connection to multiple shells, and running a training process on each shell. Jupyter notebooks [50] were also used on the machine for data visualization and prototyping, and were accessed locally by using ssh tunneling [51].

4.4 Methods

ULMFiT

The fastai v1 -library was used for implementing ULMFiT[1]. As the library's support for SentencePiece[2] was at the time quite limited, a considerable amount of custom code had to be written to incorporate the sub-word tokenizer in the training process. The scripts used for training ULMFiT with SentencePiece are open-sourced and can be found from[3].

ELECTRA

For training ELECTRA, the pretraining and finetuning scripts were used from the official github repository[4]. The code was forked in order to make some changes to it regarding finetuning and evaluation parameters[5].

BERT

For finetuning FinBERT, HuggingFace's transformers library was ultimately used [52]. It provides pretraining and finetuning scripts for a multitude of transformer-based models, such as BERT, XLNet and RoBERTa.

[1] https://github.com/fastai/fastai

[2] https://github.com/google/sentencepiece

[3] https://github.com/invisiblesheep/ulmfit-sentencepiece

[4] https://github.com/google-research/electra

[5] https://github.com/invisiblesheep/electra

fastText

fastText[6] was trained as a baseline model. In addition to providing word embeddings, fastText can be used as a classifier as well. fastText obtains document vectors by averaging word embeddings after which it uses multinomial logistic regression for classification. As with most neural network classifiers, a probability distribution over classes is gained as a result after applying the softmax function to the results. It uses a bunch of tricks, such as hierarchical softmax, to up the speed of training the model. Thus it's an order of magnitude faster to train than a deep learning model but it still is somewhat competitive with one.

4.5 Results

A binary classifier was built to identify texts that had one of two knee-related diagnosis codes from the training data. The data was divided into training, validation and test sets with a 80–10–10 split, and stratified so that each set had an equal percentage wise representation of the classes.

fastText's results were achieved by first running it's hyperparameter optimization command to find well-performing parameters after which the final parameters were forked manually. The final hyperparameters were a learning rate of 0.05, 25 epochs of training and the usage of word bigrams.

Mainly default hyperparameters were used while training ULMFiT, except the learning rate was found with fast.ai's learning rate finder. The underlying AWD-LSTM - architecture was not changed in any way. ULMFiT was pretrained and finetuned for 5 and 12 epochs, respectively. The finetuning results stopped improving after 8 finetuning epochs.

For finetuning ELECTRA, the best and final results were obtained with a maximum

[6]https://github.com/facebookresearch/fastText

	Precision	Recall	F1
FinBERT	68.02	74.16	70.96
ELECTRA-30K	66.06	73.72	69.68
ELECTRA-50K	69.29	76.19	72.58
ELECTRA-100K	69.64	76.14	72.74
ULMFiT-30K	84.28	72.54	77.97
ULMFiT-50K	82.91	75.41	78.98
ULMFiT-100K	83.95	75.17	**79.32**
fastText	84.34	70.07	76.54

Table 4.1: Classification results of models with different vocabulary sizes on evaluation set.

sequence length of 256, a learning rate of 10^{-4} and 12 epochs of training. For FinBERT, final hyperparameters were a maximum sequence length of 256, learning rate of $2e^{-5}$ and 10 epochs.

This sequence length parameter is particular to the transformer-model. It defines the maximum amount of tokens that can be input into the network at one time. Given that the input documents for classification were oftentimes quite a bit longer than this imposed restriction, documents were chopped to multiple chunks with this maximum token amount to get around it.

4.5.1 Tools

For training these models, a number of scripts were written and documented for the hospital to use. The scripts comprise of a combination of shell and python scripts. A general preprocessing script processes and splits the data of a specific given format of medical text data into a training and evaluation set. Separate scripts are provided for pretraining and finetuning the various models.

4.6 Discussion

As seen from the results, the transformer-based models — ELECTRA and FinBERT — performed the worst of all the models. These results should be taken with a grain of salt since the dataset wasn't quite large enough to fully utilize the power of these models. It is suspected that a major factor for this lack of performance is due to the lack of a proper general medical text dataset that could be utilized in the pretraining of such models. Additionally, since these transformers have a maximum sequence length for input tokens which prohibits the utilization of all the tokens in longer texts, the lengthier documents — which there were a lot of — in the training set could not be fully utilized. Thus for FinBERT and ELECTRA, a major improvement came from increasing the maximum sequence length.

Given that the relevant text regarding the diagnosis of the document seemed to usually reside at the end of a document, encapsulating this information in the training samples for ELECTRA and BERT perhaps needs a different approach than chunking to maximum size in which the relevant information only appears in the later chunks of a document. This means that the previous chunks in a document can possibly negatively impact the performance of the classifier as well by making it focus on unimportant details.

The superior results of ULMFiT can also be due to the compatibility of LSTM-networks with long texts. Contrary to the transformers which get as input the whole document, LSTM's are recurrently fed tokens one after the other until the whole document has gone through the network. Thus, LSTM's have no limit in how long of a text it can process which proved to be an important feature with this dataset. Additionally, as the important features of text were usually at the end of a document, the LSTM saw these features last and thus they had a relatively bigger impact on the network's parameters.

Class imbalance in the training data was suspected to be a factor for model performance, thus positive examples were upsampled by random duplication to balance the classes which overall somewhat improved the performance of all the models. Compar-

ing the different vocabulary sizes, the results clearly show that a bigger vocabulary size worked better than a smaller one on this dataset, which could be due to the fact that the data — being medical text — included a lot of medical terms rarely found in everyday language.

For future work, pretraining ELECTRA or BERT on a larger, general medical text corpus before finetuning on domain data could yield overall improvements to diagnosis code classifiers. Pretraining on a larger dataset would give the models a good starting point for finetuning due to the fact that the models would then have a proper representation of the medical textual data domain, thus increasing the likelihood that the models generalize better. Additionally, comparing the performance of the models with different amounts of training data would give insight into the limits of ULMFiT and ELECTRA.

Chapter 5

Conclusion

In this thesis, an overview is given of natural language processing w.r.t deep learning and text classification. Additionally, a dataset of medical reports was preprocessed and used to train a number of text classifiers in order to define the feasibility of such methods for medical text classification.

It was found that the LSTM-based ULMFiT performed the best out of the chosen models, others being ELECTRA, FinBERT and fastText. The performance of a simpler method proved to be surprisingly good when compared to the more complex and training intensive deep neural networks. This is a promising finding due to the fact that smaller hospitals don't have the resources to train and use the weightier deep neural networks in analyzing their documents. Utilizing simpler and lighter models such as fastText levels the playing field and brings the benefits of AI to more people.

The lacklustre results of the transformer-based models are advised to be taken with a grain of salt as these approaches are deemed to require further investigation in this domain.

As a results of this thesis, the scripts for training the aforementioned models were compiled and provided to the hospital as tools for preprocessing medical data and pre-training and finetuning various classifiers for different diagnosis codes.

For future work, it is suggested that a larger general corpus of medical text is gathered

and used to pre-train a deep neural network first in order to improve performance of the various introduced methods such as transformer-based BERT and ELECTRA.

References

[1] Y. Goldberg, "Neural network methods for natural language processing", *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017.

[2] J. Howard and S. Ruder, "Universal Language Model Fine-tuning for Text Classification", *arXiv:1801.06146 [cs, stat]*, May 2018. arXiv: `1801.06146 [cs, stat]`.

[3] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of Tricks for Efficient Text Classification", *arXiv:1607.01759 [cs]*, Aug. 2016. arXiv: `1607.01759 [cs]`.

[4] A. Hotho, A. Nurnberger, G. Paaß, and S. Augustin, "A Brief Survey of Text Mining", en, p. 37,

[5] F. Sebastiani, "Machine learning in automated text categorization", *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[6] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing", *arXiv:1808.06226 [cs]*, Aug. 2018. arXiv: `1808.06226 [cs]`.

[7] I. Rish, "An empirical study of the naive Bayes classifier", en, p. 6,

[8] L. Rigutini and M. Maggini, "Automatic text processing: Machine learning techniques", PhD thesis, Citeseer, 2004.

[9] D. D. Lewis, "Naive (Bayes) at forty: The independence assumption in information retrieval", en, in *Machine Learning: ECML-98*, J. G. Carbonell, J. Siekmann, G. Goos, J. Hartmanis, J. van Leeuwen, C. Nédellec, and C. Rouveirol, Eds., vol. 1398, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 4–15, ISBN: 978-3-540-64417-0 978-3-540-69781-7. DOI: 10.1007/BFb0026666.

[10] A. Ben-Hur, D. Horn, H. T. Siegelmann, and V. Vapnik, "Support vector clustering", *Journal of machine learning research*, vol. 2, no. Dec, pp. 125–137, 2001.

[11] L. Rokach and O. Maimon, "Top-Down Induction of Decision Trees Classifiers—A Survey", en, *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 35, no. 4, pp. 476–487, Nov. 2005, ISSN: 1094-6977. DOI: 10.1109/TSMCC.2004.843247.

[12] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, en, ser. Springer Texts in Statistics. New York, NY: Springer New York, 2013, vol. 103, ISBN: 978-1-4614-7137-0 978-1-4614-7138-7. DOI: 10.1007/978-1-4614-7138-7.

[13] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, classifiaction", 1992.

[14] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[15] L. Derczynski, "Complementarity, F-score, and NLP Evaluation", in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, Portorož, Slovenia: European Language Resources Association (ELRA), May 2016, pp. 261–266.

[16] E. Brill and J. Wu, "Classifier combination for improved lexical disambiguation", in *Proceedings of the 17th International Conference on Computational Linguistics-Volume 1*, Association for Computational Linguistics, 1998, pp. 191–195.

[17] Q. Yang, Y. Zhang, W. Dai, and S. J. Pan, *Transfer Learning*. Cambridge University Press, 2020, ISBN: 1-107-01690-8.

[18] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality", in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2013, pp. 3111–3119.

[19] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation", in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[20] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information", *arXiv:1607.04606 [cs]*, Jun. 2017. arXiv: 1607.04606 [cs].

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space", *arXiv:1301.3781 [cs]*, Sep. 2013. arXiv: 1301.3781 [cs].

[22] O. Levy, Y. Goldberg, and I. Dagan, "Improving Distributional Similarity with Lessons Learned from Word Embeddings", *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, Dec. 2015. DOI: 10.1162/tacl_a_00134.

[23] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin, "Advances in Pre-Training Distributed Word Representations", *arXiv:1712.09405 [cs]*, Dec. 2017. arXiv: 1712.09405 [cs].

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016, ISBN: 0-262-33737-1.

[25] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *arXiv:1412.6980 [cs]*, Jan. 2017. arXiv: 1412.6980 [cs].

[26] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods", en, *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, Jan. 1964, ISSN: 0041-5553. DOI: 10.1016/0041-5553(64)90137-5.

[27] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation", California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[29] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] F. A. Gers and J. Schmidhuber, "Recurrent nets that time and count", in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. 3, IEEE, 2000, pp. 189–194, ISBN: 0-7695-0619-4.

[32] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation", *arXiv:1406.1078 [cs, stat]*, Sep. 2014. arXiv: 1406.1078 [cs, stat].

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need", *arXiv:1706.03762 [cs]*, Dec. 2017. arXiv: 1706.03762 [cs].

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[35] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes", *arXiv:1312.6114 [cs, stat]*, May 2014. arXiv: 1312.6114 [cs, stat].

[36] Y. Miao, L. Yu, and P. Blunsom, "Neural Variational Inference for Text Processing", *arXiv:1511.06038 [cs, stat]*, Jun. 2016. arXiv: 1511.06038 [cs, stat].

[37] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A Convolutional Neural Network for Modelling Sentences", *arXiv:1404.2188 [cs]*, Apr. 2014. arXiv: 1404.2188 [cs].

[38] A. M. Dai and Q. V. Le, "Semi-supervised Sequence Learning", *arXiv:1511.01432 [cs]*, Nov. 2015. arXiv: 1511.01432 [cs].

[39] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and Optimizing LSTM Language Models", *arXiv:1708.02182 [cs]*, Aug. 2017. arXiv: 1708.02182 [cs].

[40] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of Neural Networks using DropConnect", en, p. 9,

[41] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", *arXiv:1810.04805 [cs]*, May 2019. arXiv: 1810.04805 [cs].

[42] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, and K. Macherey, "Google's neural machine translation system: Bridging the gap between human and machine translation", *arXiv preprint arXiv:1609.08144*, 2016.

[43] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators", *arXiv preprint arXiv:2003.10555*, 2020.

[44] A. Virtanen, J. Kanerva, R. Ilo, J. Luoma, J. Luotolahti, T. Salakoski, F. Ginter, and S. Pyysalo, "Multilingual is not enough: BERT for Finnish", *arXiv:1912.07076 [cs]*, Dec. 2019. arXiv: 1912.07076 [cs].

[45] *Systems - Docs CSC*, https://docs.csc.fi/computing/system/#puhti.

[46] *Git*, https://git-scm.com/.

[47] *Slurm Workload Manager - Documentation*, https://slurm.schedmd.com/documentation.html.

[48] *Slurm Workload Manager - Quick Start User Guide*, https://slurm.schedmd.com/quickstart.html

[49] *Gnu.org*, en, https://www.gnu.org/software/screen/, Library Catalog: www.gnu.org.

[50] *The Jupyter Notebook — Jupyter Notebook 6.0.3 documentation*, https://jupyter-notebook.readthedocs.io/en/stable/.

[51] *SSH Tunnel*, https://www.ssh.com/ssh/tunneling/.

[52] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "HuggingFace's Transformers: State-of-the-art Natural Language Processing", *arXiv:1910.03771 [cs]*, Feb. 2020. arXiv: 1910.03771 [cs].