Transfer learning in medical report classification..

Master of Science Thesis
University of Turku
Department of Future Technologies
Computer Science
2020
Tuomas Jokioja

UNIVERSITY OF TURKU
Department of Future Technologies

TUOMAS JOKIOJA: Transfer learning in medical report classification..

Master of Science Thesis, 30 p.
Computer Science
April 2020

Tarkempia ohjeita tiivistelmäsivun laadintaan läytyy opiskelijan yleisoppaasta, josta alla lyhyt katkelma.
Bibliografisten tietojen jälkeen kirjoitetaan varsinainen tiivistelmä. Sen on oletettava, että lukijalla on yleiset tiedot aiheesta. Tiivistelmän tulee olla ymmärrettävissä ilman tarvetta perehtyä koko tutkielmaan. Se on kirjoitettava täydellisinä virkkeinä, väliotsakeluettelona. On käytettävä vakiintuneita termejä. Viittauksia ja lainauksia tiivistelmään ei saa sisällyttää, eikä myäskään tietoja tai väitteitä, jotka eivät sisälly itse tutkimukseen. Tiivistelmän on oltava mahdollisimman ytimekäs n. 120–250 sanan pituinen itsenäinen kokonaisuus, joka mahtuu ykkäsvälillä kirjoitettuna vaivatta tiivistelmäsivulle. Tiivistelmässä tulisi ilmetä mm. tutkielman aihe tutkimuksen kohde, populaatio, alue ja tarkoitus käytetyt tutkimusmenetelmät (mikäli tutkimus on luonteeltaan teoreettinen ja tiettyyn kirjalliseen materiaaliin, on mainittava tärkeimmät lähdeteokset; mikäli on luonteeltaan empiirinen, on mainittava käytetyt metodit) keskeiset tutkimustulokset tulosten perusteella tehdyt päätelmät ja toimenpidesuositukset asiasanat

Keywords: tähän, lista, avainsanoista

UNIVERSITY OF TURKU
Department of Future Technologies

TUOMAS JOKIOJA: Transfer learning in medical report classification..

Master of Science Thesis, 30 p.
Computer Science
April 2020

Second abstract in english (in case the document main language is not english)

Keywords: tähän, lista, avainsanoista

Contents

1 Introduction

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 from a labelled dataset the characteristics of the categories [1].

Nowadays, huge deep neural networks dominate the field of automatic text classification. Even though they post the best results, other methods exist as well and have been used for a long time.

Chapter 2 gives a definition of text classification, insight on some of the most influential classifiers in the field of text classification in the past and a quick overview of neural networks as well. Chapter 3 outlines the current state of machine learning -based natural language processing by explaining the different parts of a machine learning pipeline such as preprocessing, training and finetuning, and relevant model architectures. Chapter 4 describes the medical report classification problem, the available data and compute resources and chosen methods, and discusses the results gained from utilizing said methods. Chapter 5 is a concluding chapter which gives an overview of the thesis and its results.

2 Text classification

2.1 A definition of text classification

Text classification is a category of tasks in Natural Language Processing (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. The phenomenon called *inter-indexer inconsistency* is a result of this undeterministicity 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 [1].

2.2 Text classifiers

2.2.1 Naive Bayes

The Naive Bayes Classifier is a probabilistic classifier that assumes, like all probabilistic classifiers, 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. Even though this assumption is unrealistic, the *naive Bayes* classifier is surprisingly successful in practice [5]. Naive Bayes models are very efficient in that they require minimal computational resources even for huge amounts of text and large vocabularies. There exists a significant problem in this approach, however, 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 class to 0. This could turn an otherwise clear-cut classed document into something else only due to a random word. [6]

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. [6] [7]

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). [4]

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 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. [6]

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

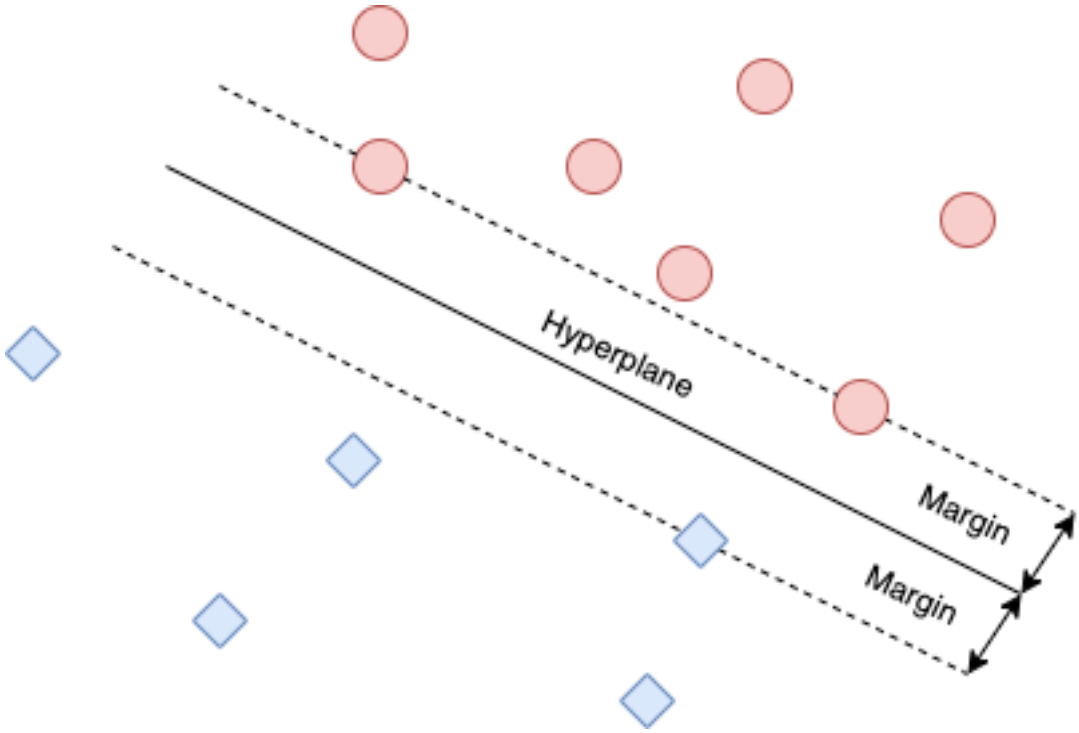

Figure 2.1: A hyperplane that separates examples of negative and positive classes with maximal margin

transforms ("maps") the input space into a higher dimensional space in which regions can be linearly separated. [6] 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)[8].

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 are among the most

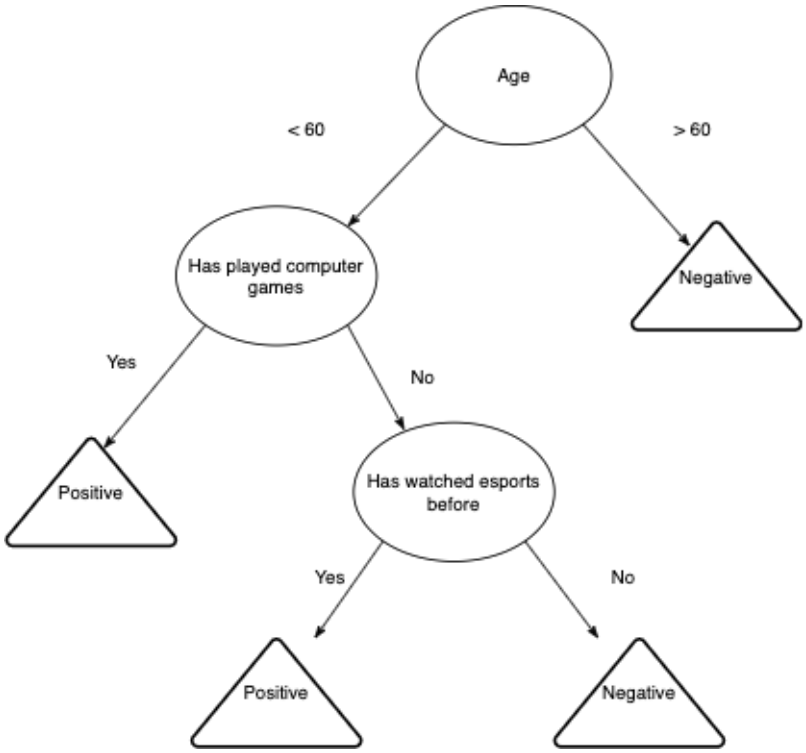

Figure 2.2: Example of a simple decision tree which predicts the sentiment of a person regarding esports

popular approaches used for text classification. [4] [9]

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 ap-

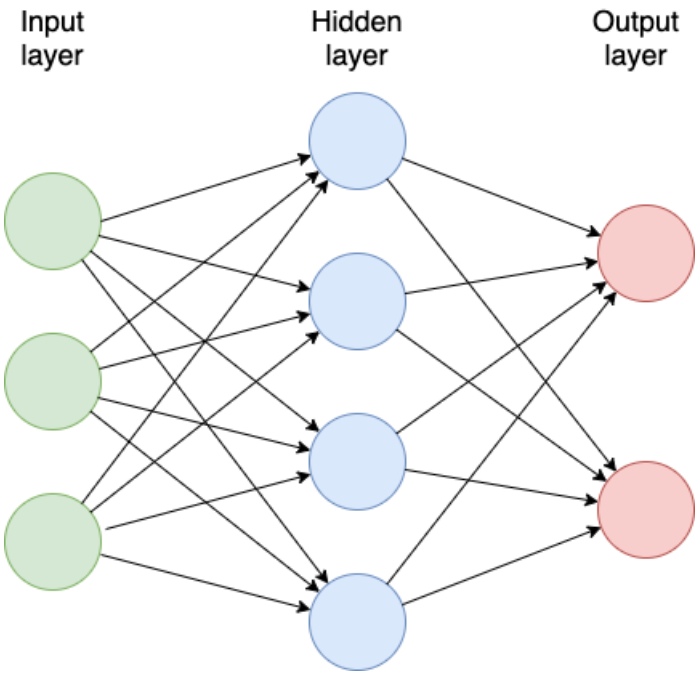

Figure 2.3: Example of a simple feedforward neural network

proaches. 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 [10].

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 [11].

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 to eliminate or minimize the error [1].

The simplest neural network is the single-layer perceptron, introduced by Rosenblatt in 1958 , which consists of a single hidden layer between an input layer and an output layer [12].

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.

3 Machine learning in natural language processing

- General pipeline 5p - Ulmfit, bert, electra 10p

What to write about: - Simple preprocessing tasks such as manual data cleaning, data quality improvements - Tokenization and data representation with topics such as n-grams, language modeling, embeddings (word2vec) and word/sentencepiece tokenization - Modern (BERT/ULMFit/ELECTRA) way of training models: pretraining, finetuning with a notion to transfer learning - Evaluation metrics

- 3.1

This chapter first gives an overview of the parts of a general machine learning pipeline and then describes the methodology of three modern machine learning models: ULMFit, BERT and ELECTRA.

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 [13]. 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, 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 finetuning phases in model training where another model's pre-trained state can be transferred to another task and finetuned further for that task.

3.1.1 Embeddings

The tranferral of knowledge can be achieved by many means. A common case 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 [14], GLoVe [15], and fastText [16], 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 [14].

It uses a skip-gram model (figure 3.1), proposed by Mikolov et al. in an earlier work [17], that 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 [14].

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

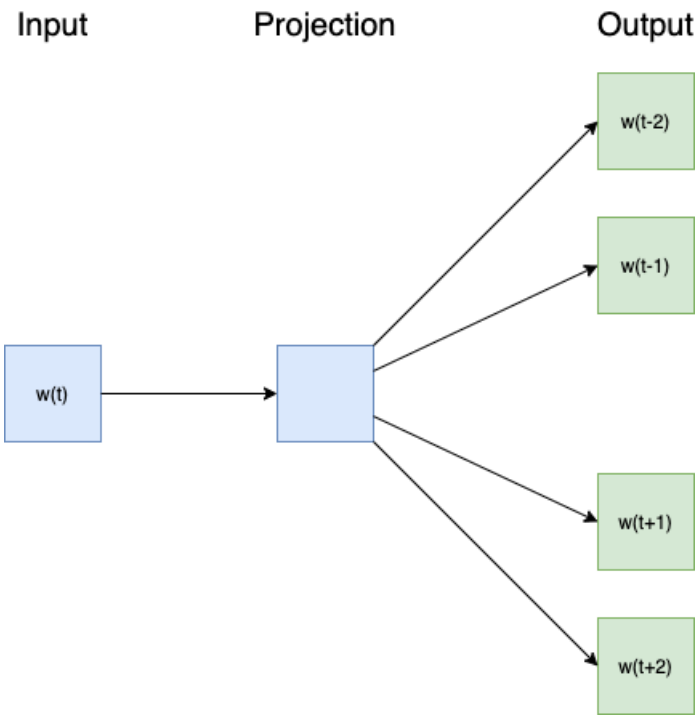

Figure 3.1: Skip-gram model architecture.

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 [15].

Given the same corpus and equal compute, Pennington et al. show that it outperforms Word2Vec and achieves better results faster [15], although Levy et al. [18] 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 [16].

Mikolov et al. show that fastText significantly outperforms GLoVe on a number of tasks [19].

3.1.2 Language modeling

– TALK ABOUT HOW LM IS CURRENTLY THE WAY TRANSFER LEARNING IS GENERALLY USED IN DEEP LEARNING NLP –

3.2 Preprocessing

Before a machine learning model can be trained, the data for it has to be gathered and preprocessed. Preprocessing in NLP usually consists of a number of steps such as lemmatization and tokenization.

This section describes common steps used in preprocessing in NLP.

3.2.1 Tokenization

[20]

SentencePiece

SentencePiece is a subword tokenizer and detokenizer that is language independent and designed for machine learning -based processing. [21] Compared to other subword segmentation systems, SentencePiece does not require that the input is pre-tokenized into word sequences. It works natively with raw sentences thus allowing a purely language independent and end-to-end system.

SentencePiece's language-independent quality is quite important especially for 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. Adding sentencepiece to those systems simplifies the processing pipeline and removes the need for custom processors for different languages.

SentencePiece is comprised of four different components: Normalizer, Trainer, Encoder and Decoder. [21] Normalizer is used to transform semantically equivalent characters to a canonical form. The Trainer trains a subword segmentation model from the normalized corpus. The Encoder first normalizes the text with the Normalizer and encodes raw text into a subword sequence using the model generated by the Trainer. The Decoder can be used to transform the tokens into normalized text. [21]

Compared to whole word tokenization, SentencePiece's subword tokenization achieves a lossless representation of data. For example, a traditional tokenizer might tokenize "Hello world." as [Hello][World][.], thus losing the information of where there are spaces in the sentence. SentencePiece treats whitespace as a normal symbol and replaces all occurrences of whitespace with an underscore (U+2581) before tokenization. SentencePiece might tokenize the aforementioned example as [Hello][_wor][ld][.] thus preserving the whitespace.[21]

SentencePiece builds a vocabulary of a predefined size of subword tokens. Depending on the given maximum size, the subwords' length changes. 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 normal whole word tokenizer. Thus, the maximum vocab size becomes another tunable hyperparameter, which has a considerable impact on model performance.

Given the abundance of inflections in the Finnish language, one would think that SentencePiece could work better for preprocessing it than other alternatives such as spaCy,

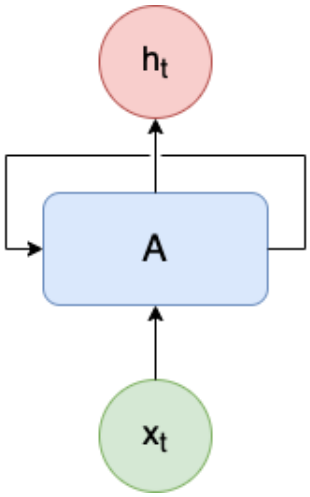

Figure 3.2: An RNN-module which illustrates the recurrent nature of the model

which tokenizes and numericalizes whole words instead of subwords.

Other tokenizers

3.2.2 Lemmatisation

3.3 Training

3.3.1 Pretraining and finetuning

3.3.2 Methods

SGD

3.4 Architectures

3.4.1 RNN

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 [22]. Figure 3.2 shows the basic methodology behind an RNN: a chunk of neural network, A, looks at the input x_t and ouputs 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.

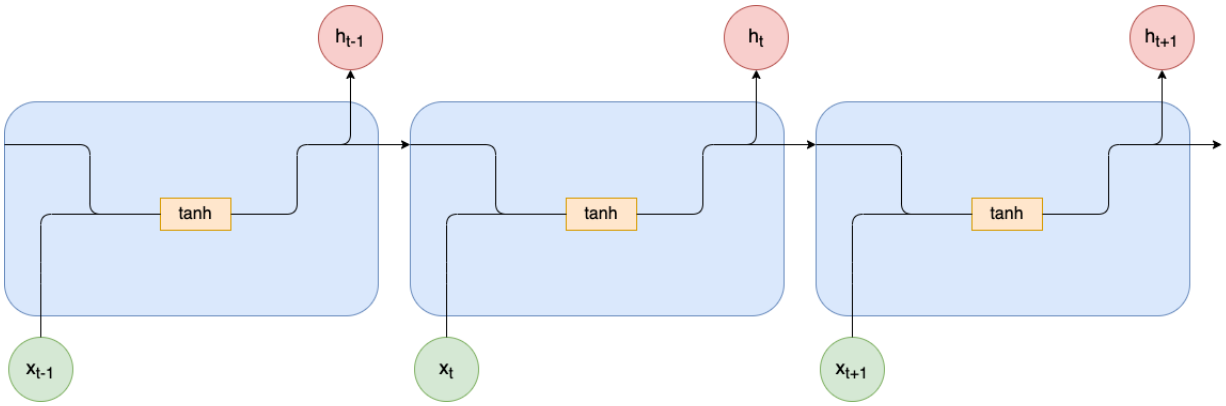

Figure 3.3: An unrolled RNN

The output of tanh is the new hidden state, the memory of the network, which is then fed 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 [23]. 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 [24]: 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 [25]. In an RNN this problem is magnified due to backpropagation being applied to each time step.

3.4.2 LSTM

Long short-term memory (LSTM) is a recurrent network architecture proposed by Hochreiter and Schmidhuber in 1997 [25]. 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 outpus 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 [26] adds peephole-connections that allow the gate layers to look at the cell state.

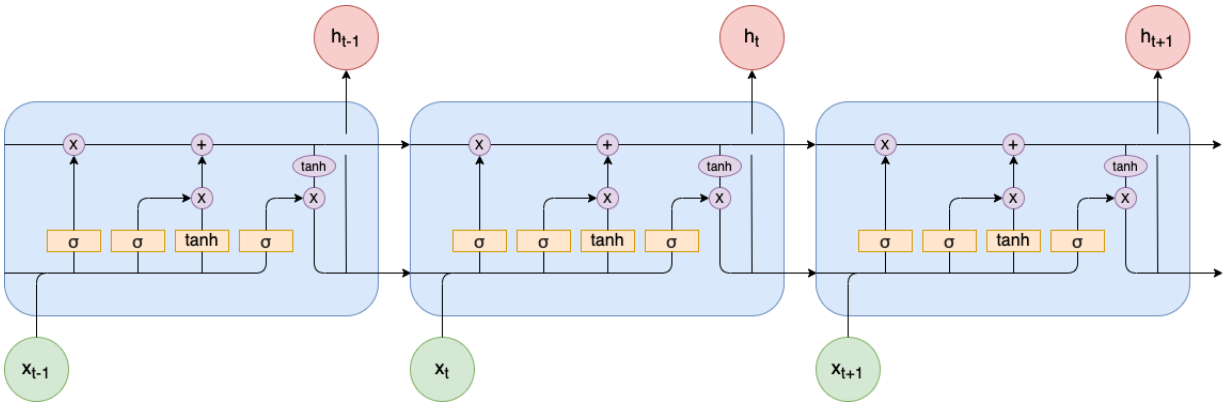

Figure 3.4: Long short-term memory network

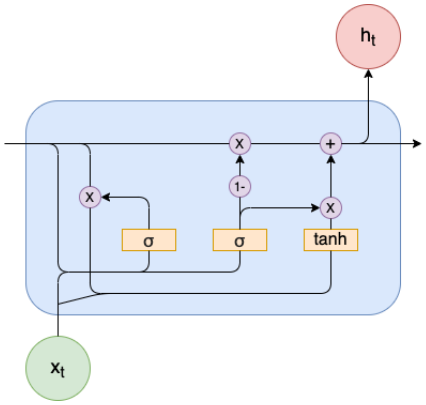

Figure 3.5: Gated Recurrent Unit

Another variant called the **Gated Recurrent Unit** (GRU, figure 3.5), introduced by Cho, et al. [27], simplifies the model by combining the forget and input gates into a single update gate.

3.4.3 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. [28]

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, in the paper *Attention is all you need*, 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 [28].

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 ouput 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 sub-layers; a multi-head self-attention mechanism and a fully connected feed-forward network. Residual connections [29] 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

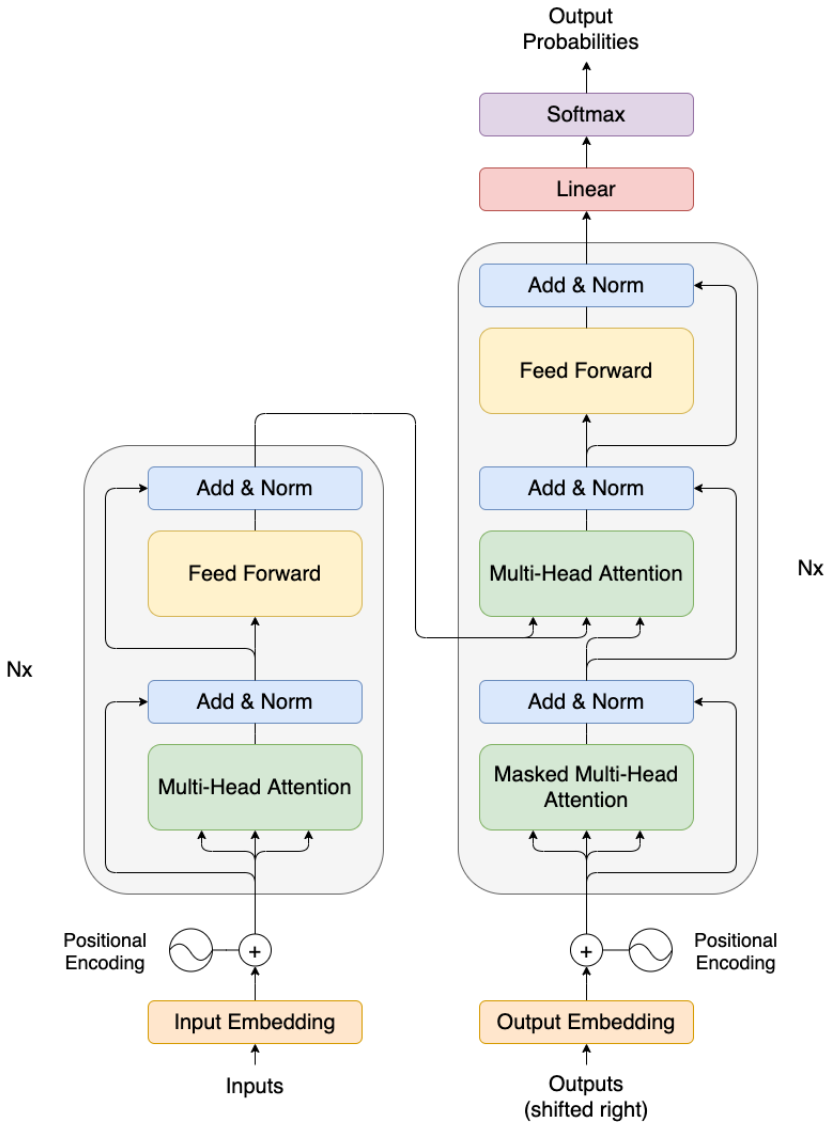

Figure 3.6: The Transformer, from the paper *Attention is all you need* [28]

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 [28]. 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 3.1. The results are divided by $\sqrt{d_k}$ and then a softmax function is applied to obtain the weights on the values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.1}$$

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 [28]. In the Transformer, 8 parallel attention heads are used.

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 [28].

3.4.4 Other architectures

– AUTOENCODER – – CNN –

3.5 Evaluation

3.5.1 Tasks

3.5.2 Measures

Accuracy

Accuracy is the fraction of correctly classified documents in relation to the total number of documents.

Precision

$$\frac{\#\{relevant \cap retrieved\}}{\#retrieved} \tag{3.2}$$

Recall

$$\frac{\#\{relevant \cap retrieved\}}{\#relevant} \tag{3.3}$$

F-score

$$F = \frac{2}{1/recall + 1/precision} \tag{3.4}$$

3.6 Modern models

3.6.1 ULMFiT

Universal Language Model Finetuning (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 pretraining a language model on a general-domain corpus and then finetuning it on a classification task. This idea of first pretraining on a large general corpus and then finetuning 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 [30]. With ULMFiT Howard et al. proposed novel training techniques to make the training 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. [31], which is a recurrent neural network (RNN). AWD-LSTM is a vanilla LSTM with various regularization and optimization strategies such as DropConnect [32], which prevents overfitting by randomly dropping connections between the recurrent hidden to hidden weight matrices, and averaged gradient descent as its optimization algorithm.

The training begins with pretraining 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 finetuned with the target task data. This finetuning converges faster than the initial pretraining since the model needs to only adapt to the idiosyncrasies of the finetuning 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 finetuning 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.

3.6.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a language representation model based on the Transformer [28] architecture (subsection 3.4.3). It uses a *fine-tuning* based approach, in which all the pretrained parameters are finetuned 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 bidirectionality 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 [33].

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 3.4.3, 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 performace 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 [34], essentially similar to SentencePiece embeddings (section 3.2.1). 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% chacne 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.

3.6.3 ELECTRA

Efficiently Learning an Encoder that Classifies Token Replacements Accurately (ELECTRA) is a further elaboration on the BERT model by Clark, et al. [35] The primary motivation for ELECTRA is making pretraining 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 3.6.2), BERT uses masked language modeling as a pretraining 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 pretraining 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 pretraining but not during finetuning. 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 [35].

4 Medical report document classification

4.1 Problem

The challenge was to classify medical reports based on their diagnosis codes. The motivation for this was to first to define the feasability and current state of deep learning for automatic text classification of medical text. Second, to build a easy to use framework of sorts for the hospital to train more classifiers for different diagnosis codes.

A binary classifier was built to identify two knee-related diagnosis codes from the training data. Since compute resources was a major constraint, only approaches with a reasonable single-gpu training time were considered. Thus, ULMFiT, a small ELECTRA model, and a finetuned FinBERT, pretrained by Virtanen et al. [36], were chosen for comparison.

4.2 Data

The data for the work was provided by (VSSHP/Auria/TYKS). It consisted of doctor's statements with corresponding diagnosis codes and other metadata such as date of visit and (toimipiste). Additionally, since receiving access to the actual medical data took some time, the chosen models and code were trained and tested on some general finnish data from various news agencies and open online forums compiled by Virtanen et al. for

FinBERT [36].

4.2.1 General Finnish

The general finnish corpus is composed from three different sources: news articles, online discussions, and documents crawled from the Finnish internet. It 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. – TELL ABOUT PREPROCESSING –

4.2.2 Clinical data

– GET DATA FOR CLINICAL, TOKENS AND SENTENCES – – TELL ABOUT PREPROCESSING –

4.3 Compute resources

The models were trained on Compute resources for the project were provided by (VSSHP/Auria/Tyks) for the medical models and CSC for the general finnish models.

4.3.1 CSC Puhti

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.

4.3.2 VSSHP Blackbird

For this project access was granted to Blackbird, a computer for artificial intelligence owned by (AURIA/TYKS/VSSHP?) that has four Nvidia V100 graphics cards allowing the simultaneous training of multiple models. 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.

– TABLE OF TRAINING TIMES FOR DIFFERENT MODELS, PRETRAINING AND FINETUNING, OMIT FINBERT PRETRAINING TIME SINCE ALREADY PRETRAINED –

4.4 Results

– RESULT TABLE WITH: – – Acc, precision, f1 – – FOR:– – ULMFIT 30K, 50K, 100K – – ELECTRA 30K, 50K, 100K – – FinBERT with original vocab of 50K –

4.5 Discussion

– BLAH BLAH – – SURPRISING RESULTS FROM ULMFIT WRT THE PERFORMANCE OF ELECTRA – – FINBERT IS OKAY THOUGH MODELS TRAINED ON THE ACTUAL DATA PERFORM BETTER – – ELECTRA SEEMS PROMISING – – FUTURE WORK COULD INCLUDE PRETRAINING ON A LARGER CORPUS OF GENERAL MEDICAL TEXT FIRST, THEN FINETUNE TO TASK –

5 Conclusion

References

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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].

[17] 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].

[18] 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.

[19] 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].

[20] J. J. Webster and C. Kit, "Tokenization as the Initial Phase in NLP", *COLING 1992 Volume 4: The 15th International Conference on Computational Linguistics*, 1992.

[21] 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].

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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].

[28] 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].

[29] 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.

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

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

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

[33] 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].

[34] 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.

[35] 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.

[36] 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].