Numerical and machine learning methods in automatic text summarization

Sokol, Domina

FACULTY OF SCIENCE
UNIVERSITY OF SPLIT

DOMINA SOKOL

NUMERICAL AND MACHINE LEARNING METHODS IN AUTOMATIC TEXT SUMMARIZATION

MASTER'S THESIS

Split, December 2022

FACULTY OF SCIENCE
UNIVERSITY OF SPLIT

DEPARTMENT OF MATHEMATICS

NUMERICAL AND MACHINE LEARNING METHODS IN AUTOMATIC TEXT SUMMARIZATION

MASTER'S THESIS

Student:
Domina Sokol

Mentor:
professor Saša Mladenović

Split, December 2022

Acknowledgements

I would like to thank the people who have made arriving at this thesis not only an opportunity for learning, but an important part of my life: my mentor for his insight and patience, the committee members for their help and interest in this thesis, and finally my family and friends for their endless support.

Introduction

In the recent years, natural language processing has become ever-more popular as a field of research, being introduced into formal education, and in the market, as a useful tool in creating products related to language such as chatbots. Historically, one of the first tasks of this sub-field of artificial intelligence is machine translation. This is a task which has had a large influence in the motivation of developing the field of computational linguistics, as well as natural language processing techniques.

From this task, a simpler one emerges - how to process the input of text, automatically, by the use of a computer? Text summarization is one such task, where a machine is expected to produce relevant output in human-readable, textual, format. This is no easy task, as many obstacles present themselves. Namely, the problem of text representation in a form suitable for management by the computer, and semantically relevant text production are two biggest ones.

Several approaches have been tried out to try to tackle the problem of creating meaningful summaries from given text; both numerical and, in modern times, deep learning. Neural networks have been shown to be the best models up to date for text generation problems, thanks to their immense power of generalization and ability to operate under the assumption of context.

Human language is one of the most complicated systems, and yet, it

is developed naturally, through evolution and intuition. The intricacies of natural language are extremely hard to capture by any machine or formal system which operates under strict and logical rules. To capture the true cultural and other, more local, contexts and meaning of some text, many complex models have been tried out. Perhaps truly "teaching" the machines how to speak is not possible, but attempts at mimicking are becoming more and more convincing.

This thesis covers the basics of text summarization, the historical development of techniques for this purpose, as well as an overview of the most popular and successful modern methods using deep learning. At the end, practical examples showcasing the methods in action are given.

Contents

Chapter 1

About natural language processing and automatic text summarization

1.1 About natural language processing

Natural language is any human language which has evolved through repetition and without conscious planning. Natural language is most easily defined by contrasting it with another kind of language, well known in mathematics - formal languages.

Natural language processing is a sub-discipline of artificial intelligence, linked to linguistics and computer science, concerned with the interactions between computer programs and human languages. Its primary concern is programming computers to process and analyze large quantities of natural language data.

The main goal is to achieve a computer program capable of understanding the natural language documents it is given, including context and nuances.

This enables extraction and analysis of information contained in the documents as well as organization and categorization of documents themselves.

The term *document* refers to a generally accepted definition in natural language processing field - a collection of meaningful natural language instances (words or sentences), treated as a single unit.

Natural language processing has many tasks. Some of them are machine translation, text summarization, and key-word extraction.

1.1.1 History of natural language processing

The history of natural language processing has its roots as early as in seventeenth century, with the first ideas of a system which translates human languages from one to another. Descartes and Leibniz both proposed ideas for codes which would do this task. However, none of these ideas became realized and further explored until the famous paper by Alan Turing - Computing Machinery and Intelligence, in which he proposed the procedure now called the Turing test. In 1957, Noam Chomsky took the ideas about language even further by constructing a universal grammar, a rule based system of syntactic structures.

The main task which was a primary goal of many of these ideas was a system, or a machine, which could automatically translate. Thus, machine translation sub-field of natural language processing was born. Many early successes occurred in the sub-field of machine translation, due to work at IBM Research, where successively more complicated statistical models were developed.

Recent research focuses on the use of machine learning models, which take the statistical logic even further, by implementing semi-supervised and unsupervised algorithms for various natural language processing tasks, not

just machine translation. One thing these algorithms need is a lot of data. This is available for the first time in history, in sufficiently large quantities, thanks to the World Wide Web and its enormous resources of text, in many different forms.

Today, there exist many models which utilize large amounts of textual data, called corpora, in order to train various algorithms. There are several virtual assistants on the market which utilize natural language processing techniques in order to appear realistic to users. The field of natural language processing continues to grow and expand, by introduction of more specific sub-fields and tasks, as well as research which is ever-more popular.

The history of natural language processing can be roughly divided into three phases:

- symbolic (1950s - early 1990s) - characterized by computer emulation of natural language by using the predefined set of rules,
- statistical (1990s - 2010s) - based on introduction of machine learning algorithms which discouraged the previous Chomskyan theories of modelling languages as formal grammars and encouraged the corpus linguistics principle of viewing language,
- neural (present) - brought on by the advances of machine learning which made deep neural network style algorithms more prevalent.

1.2 About automatic text summarization

Text summarization is a task in natural language processing with the goal of producing a shorter version of a given text by the use of a computer. The created output is called a summary, which should be representative of the full

text, that is, it should contain the most important information from the original text. There are other types of data which can be summarized, namely: images and video/audio content. The umbrella term for summarization of any kind of data here mentioned by the use of a computer is called automatic summarization.

This thesis focuses on the summarization of text. Two main types of text summarization are extractive and abstractive. The former does not include changing any words from the given text, but only rearranging them. In contrast, abstractive summarizing is based not only on extracting semantically valuable pieces of text, but employing various methods for building a semantic representation of the text which looks more like the product of a human expression. This includes paraphrasing.

One task of natural language processing which is related to summarization is key-word extraction, which refers to a practice of finding relevant words or phrases in the given text in order to identify the topic of the text. Summarization is similar to this in regards to the need of finding semantic connections in the text. The better the semantic representation of the text, the better the summary because it does not rely solely on detecting relevant whole sentences, but also on their core meaning, thus making it possible to combine them in order to achieve a more quality summary.

1.3 The goal of text summarization and established methods

The process of creating a summary has several steps:

1. text segmentation,

2. word encoding,

3. applying a chosen model to create a summary.

In order to present text to a computer, it needs to be transformed into a form which computers understand - numbers. The first thing that needs to be done in order to use any kind of model, be it numerical or machine learning based, is text encoding. This can be done in many ways.

Usually, text is split into sentences, and then into words. This procedure is called **text segmentation**. It can be done in simple ways, by using regular expressions, or in a more sophisticated way, by the use of machine learning models. The main point is dividing the initial text into smaller, more manageable units such as words.

The next step is mapping words into numbers or, more commonly, vectors which will represent them in the model. This is no trivial task, and is a sub-field of natural language processing itself. There are many ways to do this. Some of the most popular methods include one-hot encoding, tf-idf vectorization and word embeddings.

Definition 1.1 *Let* $K = \{1, \ldots, n\}$ *represent the set of words which occur in a text, where each number corresponds to a unique word. A one-hot vector is a vector* $(y_1, \ldots, y_n) \in \{0, 1\}^n$ *where* $y_i = 1$ *if* $(y_1, \ldots, y_n)$ *corresponds to the word represented by* i*,* $y_i = 0$ *otherwise.*

Remark 1.2 *One-hot vectors are called this way because only one coordinate is equal to* 1 *and all others are zero.*

This method of encoding words is limited, and there exist more advanced methods which encapsulate the frequency of word occurrence in text - an important and telling feature. One such method uses two scores in order

to produce a vector: number of word occurrences in a document (term frequency), and number of documents in which the word occurs (document frequency). Since the word is considered less meaningful if it appears in many documents, this score is higher for more specific words rich with meaning, and low for common words such as conjuction words, for example "a" and "the".

Remark 1.3 *It can be difficult to find the balance in scoring words based on their occurrence frequency since common words appear in most if not all documents, and many times at that, but words representative of the text topic occur many times as well. It is for this reason that universally common words, so-called stop words, are usually removed from the text.*

Tf-idf refers to term frequency - inverse document frequency, and is a score calculated for pairs of terms and documents (t, d) in the following way:

$$tf - idf(t, d) = tf(t, d) \times idf(t)$$

where $tf(t, d)$ is the number of occurrences of a term t in the document d, and

$$idf(t) = log\frac{1 + n}{1 + df(t)} + 1$$

where $df(t)$ is the number of documents in the document set which contain the term t.

A more sophisticated method of representing words with vectors is by word embeddings - typically real-valued vectors which represent words in such a way that words closer to each other in meaning are represented by vectors which are closer to each other in the embedding space. The resulting embedding space can be considered as a subset of $\mathbb{R}^n$, thus formalizing the notion of closeness.

The final step in producing a summary is applying the chosen model. This thesis presents established algorithms used for this purpose. As any task related to natural language is complicated, since natural language cannot be described by a finite set of rules, machine learning makes an excellent candidate as an improvement to the classical numerical models.

1.3.1 Document-term matrix

Document-term matrix is a matrix which represents the occurrences of terms in documents along a given set of documents.

Definition 1.4 *Let $D = \{d_1, \ldots, d_n\}$ be a set of documents and let $T = \{t_1, \ldots, t_m\}$ be a set of terms. A document-term matrix is a matrix $\left[\alpha_{ij}\right]_{nm}$ where α_{ij} denotes the number of occurrences of the term t_j in the document d_i.*

The rows represent the documents, and the columns represent the terms. A document-term matrix is one of the simpler methods of modelling text by counting words. It disregards the order of appearance in which the words occur, but it is still a powerful tool and an important concept.

Matrix factorization can be used in order to represent the initial matrix as a product of two smaller ones $V = WH$. In this case, W can be considered as a feature matrix, and H as a coefficient matrix. Now, each original document (row of V) can be considered to be built from a set of hidden features. The role of matrix factorization is to find useful features. This is how numerical methods are used for text summarization: in the form of matrix factorization techniques.

Chapter 2

Numerical methods

2.1 Singular value decomposition

Definition 2.1 (Singular value decomposition) *A singular value decomposition (SVD) of a matrix $A \in \mathbb{C}^{m\times n}$ is a factorization*

$$A = U\Sigma V^*$$

where

$$\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_p) \in \mathbb{R}^{m\times n},$$

$$p = min\{m, n\},$$

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$$

and both $U = [\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_m] \in \mathbb{C}^{m\times m}$ and $V = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n] \in \mathbb{C}^{n\times n}$ are unitary. The diagonal entries of Σ are called singular values of A. The columns of U are called left singular vectors of A. The columns of V are called right singular vectors of A.

Remark 2.2 *Notation $diag(\sigma_1, \sigma_2, \ldots, \sigma_p)$ denotes the following diagonal*

matrix:

$$\begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_p \end{bmatrix}.$$

Singular value decomposition is, most notably, used in latent semantic analysis. This is a technique in distributional semantics, a natural language processing area which studies theories and methods of quantifying and categorizing semantic similarities between linguistic instances based on their distributional properties in large samples of natural language data.

2.1.1 Latent semantic analysis

The goal of latent semantic analysis is producing a set of concepts related to documents and terms, based on analyzing the relationships between them. The main assumption is that words which are close in meaning occur in similar pieces of text. This is referred to as distributional hypothesis.

Latent semantic analysis is based on the vector space model idea which uses linear algebra knowledge. The vector space model was initially developed for information retrieval. More precisely, it was developed to handle text retrieval from large databases with heterogeneous text. One of the first systems which used this model, in its traditional version, was System for the Mechanical Analysis and Retrieval of Text (SMART) [9].

The main premise of this model style is that documents can be derived from their components. The underlying mathematical model of vector space models defines unique vectors for each document and uses these vectors for querying by comparing each one with the given query in order to find the right one. Query-document similarity is based on similar semantic content.

Latent semantic analysis is considered a truncated vector space model which represents documents in a high-dimensional document-term vector space. It uncovers the hidden underlying ("latent") semantic structure in text to define documents in a collection. A document is defined as the sum of the meaning of its terms. By using the truncated singular value decomposition, latent semantic analysis exploits the meaning of terms by removing "noise", which is evidenced by polysemy and synonymy found in documents [10]. As a results, document similarity is dependent on their semantic content, and not on the terms which they contain. This is especially emphasized by the fact that documents relevant to queries do not need to necessarily contain any terms from the query itself [11].

The algorithm

A document-term matrix is constructed from large quantities of text and then singular value decomposition is applied to it in order to reduce the number of rows (which represent documents) while preserving the similarity structure along columns (which represent terms). Finally, a comparison of documents is performed, by calculating cosine similarity of vectors or the dot product of normalized vectors formed by any two columns. Values close to 1 represent similar documents, and those close to 0 very dissimilar ones.

The document-term matrix undergoes a procedure of weighting, in the sense that each nonzero element α_{ij} gets a weight associated to it. A weighting function should give low weight to high-frequency terms which occur in many documents and high weight to terms that only occur in some documents. Latent semantic analysis applied both a local and a global weighting function to each nonzero element in order to increase or decrease the importance of a term within documents (local) and across the entire document

collection (global). The local and global weighting functions are directly related to frequency of term occurrence within documents and inversely related to frequency of term occurrence across the whole document collection.

Local weighting functions include using term frequency and binary frequency (0 if term is not in a document, 1 if it is) and logarithms of term frequency plus 1. Global weighting functions include normal, tf-idf, and entropy. A common local and global weighting function is log-entropy which decreases the effect of large differences in frequencies.

Entropy in the context of term-document frequency is defined as

$$1 + \sum_j \frac{p_{ij} log_2(p_{ij})}{log_2 n}$$

where

$$p_{ij} = \frac{tf_{ij}}{gf_i},$$

tf_{ij} being the number of times a term i appears in the document j, gf_i being the number of times a term i appears in the entire document collection, and n being the number of documents in the collection.

After constructing the document-term matrix, a low-rank approximation method is applied to it. This is done for multiple reasons: original document-term matrix is too large for computing resources, too noisy (contains too many anectodal instances of terms), or too sparse relative to the possible document-term matrix which would also contain words related to each document (synonyms) and not only precisely words which do appear.

Rank lowering is expected to merge the instances associated with words which have similar meaning. This mitigates the problem of synonymy.

Singular value decomposition is applied to the low-rank approximation of the original document-term matrix. When selecting the k largest singular values $\{\sigma_1, \ldots, \sigma_k\}$ and their corresponding singular vectors, one gets the

rank-k approximation of the original document-term matrix with the smallest error (regarding the Frobenius norm).

Definition 2.3 *Frobenius norm of a matrix A is defined as*

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}.$$

Now, it is possible to treat the term and document vectors as a semantic space. Each term and document vector then has k entries which represent its mapping to a lower-dimensional space.

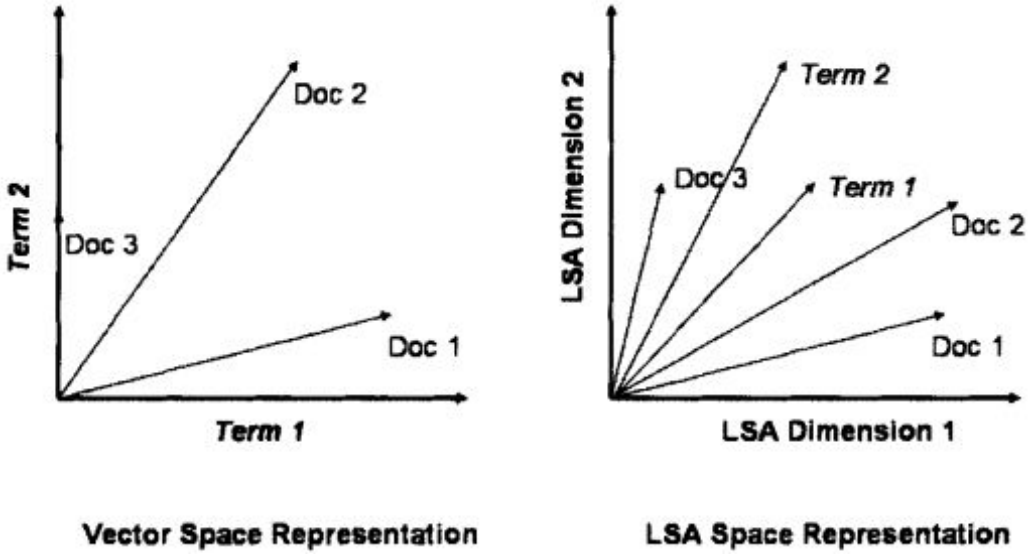

Figure 2.1: Vector space and latent semantic analysis space representations comparison (figure from [5]).

The new lower-dimensional space can be used for different purposes:

- comparison of documents (data clustering, document classification),
- finding similar documents across languages (cross-language information retrieval),

- discovering relations between terms (synonymy and polysemy),
- translation of given query into low-dimensional space and finding matching documents (information retrieval),
- expansion of the feature space of text mining systems,
- analysis of word association in text.

Singular value decomposition is the chosen orthogonal matrix decomposition for multiple reasons. Firstly, it decomposes the original document-term matrix into orthogonal factors that represent both terms and documents. These vector representations are achieved at the same time. Secondly, it sufficiently captures the latent semantic structure of the document collection and it allows for adjustment of term and document representation in the vector space by choosing the number of dimensions (appropriate number of singular values k). Finally, its computation is manageable for large datasets.

By dimension reduction to k, variability in term usage referred to as "noise" is removed. Terms similar in meaning are near each other in the k-dimensional space even if they never occur next to each other in a document. Documents similar in meaning are near each other even if they share no terms. In practice, choosing k depends on the given data. For large document collections, this value usually falls between 100 and 300 [12].

2.2 Non-negative matrix factorization

2.2.1 Basic idea and use in text summarization

Non-negative matrix factorization is a family of algorithms in multivariate analysis and linear algebra. The goal of non-negative matrix factorization

is interpreting the initial matrix as a product of two matrices with non-negative entries. The non-negativity condition insures easier manipulation with matrices, in the context of matrix calculations. Also, for certain types of data non-negativity is a given instance - sometimes negative entries make no sense (audio spectrograms, muscle activity...).

Such a factorization is not always possible to achieve analytically. Hence, it is solved by numeric methods.

2.2.2 Definition and formalization

Let

$$V = W \cdot H$$

where

$$V \in M_{m,n}(\mathbb{R}), W \in M_{m,p}(\mathbb{R}), H \in M_{p,n}(\mathbb{R}).$$

The crucial property of matrix multiplication which non-negative matrix factorization uses is arbitrarity of the chain dimension p. Dimensions m and n are determined by the type of V, but p can be any natural number. This means that the number of entries in W and H is a lot smaller than the number of entries of V and it can be chosen.

The goal of factorization is to approximate the starting matrix as well as possible. This translates to minimizing the error function. Thus, the task of finding a suitable matrix factorization becomes an optimization problem. The error function is defined as

$$\|V - WH\|_F$$

where $\|\cdot\|_F$ denotes the Frobenius norm.

Intuitively, it is desirable to shrink the distance between the approximation WH and the original V.

The reason why a numerical approach is used instead of an analytical one is because non-negative matrix factorization belongs to NP (nondeterministic polynomial) class of problems, regarding algorithmic complexity. Simply put, the result of the factorization is easy to verify by multiplying W and H, but not easy to find. More specifically, it has not been proven that the result can be found in polynomial time.

So, the previous expression $V \approx WH$ can be replaced with

$$V = WH + U$$

where U is the residual, which does not necessarily need to be non-negative.

The Hadamard multiplication algorithm

The simplest algorithm for finding W and H is the one by the name of Hadamard multiplication algorithm, which got its name by the main operation it uses. The use of Hadamard multiplication is in the so-called multiplicative update rule.

To define the update rule of the iterations in the algorithm, it is necessary to first fix one of the factors W, H and then minimize the cost function regarding the other, variable factor.

The cost function can be written in the form of:

$$\|V - WH\|_F^2 = \sum_{i=1}^{n} \|V_{:i} - WH_{:i}\|_2^2$$

More specifically, the cost function problem can be reduced to a series of n independent smaller problems where each column of H is minimized separately. This means that the original problem has been transformed to a series of quadratic minimization problems:

$$\underset{h \geq 0}{min} F(h) = \underset{h \geq 0}{min} \|v - Wh\|_2^2$$

where v, h are the columns of V, H.

Definition 2.4 *Quadratic minimization problem is defined as a quadratic programming problem with n variables and m constraints: given a vector $c \in \mathbb{R}^n$, a symmetric positive definite matrix $Q \in M_n(\mathbb{R})$, a matrix $A \in M_{mn}(\mathbb{R})$, and a vector $b \in \mathbb{R}^m$, the objective of quadratic programming is to find a vector $x \in \mathbb{R}^n$ such that*

$$\frac{1}{2}x^T Q x + c^T x$$

is minimized subject to

$$Ax \preceq b,$$

where $\preceq$ denotes component-wise inequality.

Furthermore, for the fixed approximation $\tilde{h} \geq 0$ it can be shown that the gradient of the cost function F is:

$$\nabla_h F = W^T W h - W^T v + V_{\tilde{h}}(h - \tilde{h}).$$

Equating the gradient with zero, one gets the following expression, where h^* is the minimum:

$$(W^T W + V_{\tilde{h}})h^* = W^T v - V_{\tilde{h}}\tilde{h}.$$

Given that $\tilde{h}$ is the global minimum of the cost function in the fixed iteration $\bar{F}$, it follows that:

$$F(h^*) \leq \bar{F}(h^*) \leq \bar{F}(\tilde{h}) = F(\tilde{h}).$$

This means that there exists a negative slope in the cost function.

The update rule for the whole matrix H is obtained by repeating the aforementioned procedure for each row of H. Similarly, the update rule is applied to the matrix W.

1. Initialize W i H.

2. Calculate new W i H:

$$H_{i,j}^{n+1} = H_{i,j}^{n} \frac{((W^n)^T V)_{i,j}}{((W^n)^T W^n H^n)_{i,j}}$$

$$W_{i,j}^{n+1} = W_{i,j}^{n} \frac{(V(H^{n+1})^T)_{i,j}}{(W^n H^{n+1}(H^{n+1})^T)_{i,j}}$$

   until W and H are stabilized.

In this process, Hadamard multiplication is used instead of the standard matrix multiplication. That is why this algorithm is sometimes also called Hadamard product algorithm. This is the most popular and the oldest algorithm of non-negative matrix factorization [3].

Remark 2.5 *Note that the factors*

$$\frac{W^T V}{W^T W H}, \frac{V H^T}{W H H^T}$$

are equal to the neutral element for Hadamard multiplication

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

when $V = WH$ (exactly equal).

It is important to note that this factorization is not unique. Factors W and H can be transformed into some different factors (which still work) by using an appropriate invertible matrix and its inverse

$$WH = WBB^{-1}H = \tilde{W}\tilde{H}$$

where B is a non-negative generalized permutation matrix.

These conditions are neccessary in order to guarantee non-negativity of the new matrices $\tilde{W}$ i $\tilde{H}$. Generalized permutation matrices are permutation matrices with non-zero elements which are any real numbers (and not multiplicative neutral element of the given field - 1 in $(\mathbb{R}, +, \cdot)$ with standard addition and multiplication). Here, it is additionaly required that B is non-negative, so one example of such a matrix could be:

$$\begin{bmatrix} 0 & 2 & 0 \\ 3 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix}.$$

What is left is ensuring non-negativity of the new matrices W^{k+1}, H^{k+1}, with the condition that W^k, H^k are non-negative. Actually, this is an attempt to avoid a situation where the algorithm gets "stuck" in some extreme which is not stationary, that is Karush-Kuhn-Tucker condition is not satisfied but the following holds: $\nabla_{H_{ij}} F < 0$.

If a stronger assumption is made that the starting matrices W, H are positive (instead of non-negative), the following holds:

$$\nabla_{H_{ij}^k} F < 0,$$

that is

$$\left[(W^k)^T W^k H^k\right]_{ij} - \left[(W^k)^T V\right]_{ij} < 0.$$

From this, by using the Hadamard product iterations it follows:

$$H_{ij}^{k+1} = H_{ij}^k \frac{\left[(W^k)^T V\right]_{ij}}{\left[(W^k)^T W^k H^k\right]_{ij}} > H_{ij}^k > 0.$$

Alternating least squares algorithm

Alternating least squares algorithm (ANLS) have first been introduced in 1994 [20]. The motivation of this algorithm lies in the convexity of the minimization function in the case where one of the factors V or W is fixed. Fixing one factor the factorization problem becomes a non-negative least squares problem (NNLS).

In every step of updating the matrices H and W, two least squares sub-problems are solved. A basic idea behind this process is given here:

1. Initialize $W^0 \geq 0, H^0 \geq 0$.

2. Repeat until the stopping criterion is satisfied:

$$\text{Solve } W_{k+1} = \underset{W \geq 0}{arg\,min} \|V - WH^k\|_F^2.$$
$$\text{Solve } H_{k+1} = \underset{H \geq 0}{arg\,min} \|V - W^{k+1}H\|_F^2.$$

Since the least squares method gives an optimal solution to the current sub-problem iteration in every step of the algorithm, every algorithm iteration reduces the error more than the previous algorithm of Hadamard multiplication. The downside of this effect is that this algorithm is slower to run and harder to implement.

As an improvement of this idea, another alternating least squares algorithm has been introduced - an inexact version of the standard non-negative alternating least squares algorithm. The difference is that non-negativity is not required in solving the least squares sub-problems. Since the exact solution is replaced with a projection of the solution of the unrestricted least squares problem to later achieve non-negativity, there is a loss of the convergence property. However, this process speeds up the runtime of the algorithm.

1. Initialize $W^0 \geq 0, H^0 \geq 0$.

2. Repeat until the stopping criterion is satisfied:

$$\text{Solve } W_{k+1} = \underset{W \geq 0}{arg\,min} \|V - WH^k\|_F^2.$$
$$W^{k+1} = [W^{k+1}]_+$$
$$\text{Solve } H_{k+1} = \underset{H \geq 0}{arg\,min} \|V - W^{k+1}H\|_F^2.$$
$$H^{k+1} = [H^{k+1}]_+$$

Non-negativity is achieved in the simplest possible way here - by replacing the negative elements with zero. This produces another desirable property of matrices - sparsity. Though, unlike the Hadamard product algorithm, a zero in the matrix can later be replaced by a non-zero element in later algorithm iterations.

The difference between ALS and ANLS is that ANLS always produced a descent in the minimization function and has a better approximational error, but it needs a significantly greater amount of time to run. This is why ANLS is rarely used in practice. Usually a hybrid approach is used where one starts with ALS because it is faster and induced matrix sparsity and finishes with ANLS because it always converges towards a solution.

There is also a third option: hierarchical alternating least squares (HALS) algorithm which solves the non-negative least squares sub-problems by using the exact coordinate descent method, described in detail in [21]. Each time, one column of W or one row of H is updated. $r - 1$ columns (rows) is fixed and j-th column (row) is minimized:

$$\min J_j(W_{:j}, H_{j:}) = \|R^{(j)} - W_{:j}H_{j:}\|_F^2$$

where $R^{(j)}$ is the j-th residual

$$R^{(j)} = V - \sum_{i \neq j}^{r} W_{:i}H_{i:}.$$

Now, stationary points are found by calculating the gradient in $W_{:j}$ and $H_{j:}$

$$0 = \frac{\partial J_j}{\partial W_{:j}} = W_{:j} H_{j:} H_{j:}^T - R^{(j)} H_{j:}^T$$

$$0 = \frac{\partial J_j}{\partial H_{j:}} = H_{j:}^T W_{:j}^T W_{:j} - (R^{(j)})^T W_{:j}$$

It follows that the updating rules for the j-th components W and H are going to be

$$W_{:j} \leftarrow \left[\frac{R^{(j)} H_{j:}^T}{H_{j:} H_{j:}^T} \right]_+$$

$$H_{j:} \leftarrow \left[\frac{R^{(j)} W_{:j}}{W_{:j}^T W_{:j}} \right]_+$$

The residual can be written in the following form:

$$R^{(j)} = V - \sum_{i \neq j}^{r} W_{:i} H_{i:} = V - WH + W_{:j} H_{:j}$$

which can be substituted into the previous expressions for W and H, thus producing the final updating rule for W and H:

1. Initialize $W^0 \geq 0, H^0 \geq 0$.

2. Repeat until the stopping criterion is satisfied:

for $j = 1, \ldots, r$ do:

$$W_{j:}^{(k+1)} = \left[W_{:j} + \frac{[XH]_{:j}^T - W\left[HH^T\right]_{:j}}{[HH^T]_{jj}} \right]_+$$

$$H_{j:}^{(k+1)} = \left[H_{j:} + \frac{\left[X^T W\right]_{:j}^T - H^T\left[W^T W\right]_{:j}}{[W^T W]_{jj}} \right]_+$$

$k = k + 1$

The HALS algorithm converges much faster than the Hadamard product algorithm while having a similar complexity. HALS is the best option for both sparse and dense matrices.

Gradient descent method

Projected gradient descent (PGD) method can be used thanks to the achieved non-negativity of the matrices by replacing negative entries with zero. The process consists of finding the gradient $\nabla F(h^k)$, choosing a step size of the descent a^k and projecting the updated solution onto $\mathbb{R}^n_+$:

$$h^{k+1} = \left[h^k - a_k \nabla F(h^k)\right]_+ .$$

In order to improve the convergence speed of gradient descent, some method of choosing the step size which ensures variability if often used, instead of a fixed step size. One such method of choosing the step size a^k is the so-called Armijo rule:

1. Initialize $W^0 \geq 0, H^0 \geq 0, 0 < \beta < 1, 0 < \sigma < 1, k = 1$
2. Repeat until the stopping criterion is satisfied:

   $W^{k+1} = \left[W^k - a_k \nabla_W F(W^k, H^k)\right]_+$

   $a_k = \beta^{t_k}$ where t_k is the smallest $t \in \mathbb{N}$ such that **(*)** is satisfied.

   $H^{k+1} = \left[H^k - a_k \nabla_H F(W^{k+1}, H^k)\right]_+$

   $a_k = \beta^{t_k}$ where t_k is the smallest $t \in \mathbb{N}$ such that **(**)** is satisfied.

(*):

$$(1-\sigma)\left\langle \nabla_W F(W^k, H^k), W^{k+1} - W^k \right\rangle + \frac{1}{2}\left\langle W^{k+1} - W^k, (W^{k+1} - W^k)(H^k(H^k)^T) \right\rangle \leq 0$$

(**):

$$(1-\sigma)\left\langle \nabla_H F(W^{k+1}, H^k), H^{k+1} - H^k \right\rangle + \frac{1}{2}\left\langle H^{k+1} - H^k, (W^{k+1})^T W^{k+1}(H^{k+1} - H^k) \right\rangle \leq 0$$

The Armijo rule is used in order to ensure a steep enough descent in each algorithm iteration. This gives faster convergence compared to using a fixed step size a.

Algorithm initialization and stopping criterion

Since non-negative matrix factorization problem is not convex, it is to be expected that there are local minima. This is why it is important to initialize the starting matrices well in order to reduce the chance of the algorithm becoming "stuck" in a local minimum instead of finding the global one.

Starting far away from the stationary point of the minimizing function is likely when using random initialization which results in prolonging the runtime of the algorithm, i.e. increasing the number of iterations. This is why some improved methods of initialization have been designed:

- improved random initialization - an extra step is used in order to improve the initial point by finding an optimal factor for scaling the step,
- clustering methods - the columns of W are initialized by using centroids found by clustering algorithms and then H is initialized in response to the found W,
- singular value decomposition - each factor of range 1 of the best approximation of range r of V can contain negative and positive elements. Each factor of range 1 is replaced by a corresponding non-negative factor of range 1, with the condition of maximum possible norm.

There are also several kinds of stopping criteria. The difference is in what they are based on - cost function minimization, optimality conditions, or difference between cost function values in two consecutive iterations. These criteria are usually combined with an additional condition such as maximum number of iterations or runtime limitation in order to ensure that the program stops, whether the algorithm converges or not.

Chapter 3

Machine learning methods

Machine learning, and especially deep learning, is immensely popular as a new tool for solving all kinds of problems in science. The main advantage of using deep neural networks, essentially composites of special functions, is the generalization power of the model. In recent years, this has become a very popular approach to solving many classical problems because of the advance in computing power.

3.1 Text summarization overview

Text summarization tasks can be split into the following classes:

- extractive summarization - intermediate representation has two approaches: topic representation and indicator representation,
- topic extraction - frequency based approaches, latent semantic analysis, bayesian approaches,
- knowledge bases and automatic summarization - graph methods, machine learning (labeled and semi-supervised),

- abstractive summarization - usage of sequence-to-sequence deep learning models, further described in the last chapter.

In the context of classification, each sentence in the text gets a sentence score - probability that it is a summary sentence. The final summary is created from highly ranked sentences. Machine learning algorithms and deep neural networks are used as classifiers of sentences.

As text summarization is a task in natural language processing, it makes sense to introduce neural network architectures which are used for multiple tasks of this field. Because language is inherently a temporal phenomenon, model architectures need to reflect this fact in their functionality. This means that an approach is needed where simultaneous access to all input is not assumed. Another thing which is crucial in sequential network architectures that deal with language input is the notion of context. Neural network architectures which satisfy these two demands are recurrent neural networks and transformers.

The recurrent neural network offers a new way of representation of prior context, allowing the model's decision to be influenced by information on a large number of words from the past. The transformer offers new mechanisms that help represent time and focus on relation between words which are distantly positioned in the text - self-attention and positional encodings.

In order to explain the mentioned models, a probabilistic language model approach is going to be used. Probabilistic language models predict the next word in a given sequence by looking at the preceding context. They give an option of assigning conditional probabilities to each possible word, where conditions come in the form of previously seen words. This yields a distribution over the entire vocabulary[1]. It is also possible to assign probabilities

[1]A vocabulary is the set of all words in a given text. Words are thought of as sequences

to whole sequences of words by using the chain rule 3.6 (defined on the page 31).

Definition 3.1 (Conditional probability) *Let (Ω, F, P) be a probability space, and $A \in F$ such that $P(A) > 0$. A function $P_A : F \longrightarrow [0, 1]$ defined with $P_A(B) = P(B|A) = \frac{P(A \cap B)}{P(A)}$ is a probability function and it is called conditional probability (conditioned by the event A).*

The term conditional probability is used in the language model context to denote the value of the conditional probability function when talking about occurrences of words as events. For example, let

$$B = \{\text{word } x \text{ appears in the sequence}\}$$

$$A = \{\text{words } y_1, y_2, \ldots, y_n \text{ appear in the sequence}\}$$

be events. Then the probability of word x appearing after the words $y_1, \ldots, y_n$ in the sequence is equal to $P(B|A)$, sequence usually being a sentence, but not necessarily.

3.1.1 Neural networks

In order to explain the special types of neural networks in the following text, the basic concepts all neural networks share are going to be briefly presented. Neural network architecture is most simply explained by using the mathematical term of a directed graph.

Definition 3.2 *Directed graph is an ordered triple $G = (V, E, \phi)$, where V is a non-empty set whose elements are called vertices, E is a set whose elements are called arcs, and $\phi : E \longrightarrow V \times V$ a function which maps arcs to vertices pairs $e \mapsto (u, v)$. Vertex u is called the beginning, and v the ending of an arc.*

of letters from a natural language alphabet.

Definition 3.3 *Weighted directed graph is an ordered pair* $W = (G, w)$ *where* G *is a directed graph and* w *is a weighting function* $w : E \longrightarrow \mathbb{R}^+$ *which maps non-negative numbers, called weights, to each arc of the graph* G.

Definition 3.4 *Let* $G = (V, E, \phi)$ *be a directed graph,* $v_0, \ldots, v_n \in V$ *such that* $v_n = v_0$, *and* $e_j = (v_{j-1}, v_j) \in E$ *for* $j = 1, \ldots, n$. *Ordered tuple* $(e_1, \ldots, e_n)$ *is called a cycle.*

A neural network can be described as a weighted directed graph $G = (V, E, \phi)$ with a weighting function $w : E \longrightarrow \mathbb{R}$. Many types of neural networks do not contain cycles, but recurrent neural networks do, and this is precisely where their power lies. This is why acyclicity is excluded from the definition of a neural network here, even though it is usually included in the standard definition.

The structure is organized in the form of **layers**. The set of vertices V can be considered as a union of non-empty disjoint finite sets V_l, $V = \bigcup_{l=0}^{m} V_l$ such that for the set of edges E the following holds: $E \subseteq \bigcup_{l=0}^{m} (V_{l-1} \times V_l)$, that is, each edge connects one vertex from the layer V_{l-1} with one vertex from the next layer V_l.

Number m is called the **depth** of a network. The first layer V_0 is called the **input layer**, the last layer V_m is called the **output layer**, and the layers between them $V_1, \ldots, V_{m-1}$ are called the **hidden layers**. A layer V_l is said to be **fully connected** if each vertex from the previous layer V_{l-1} is connected with each vertex from V_l.

Each vertex contains a numeric value which it propagates forward, except if it is a part of the output layer. In neural networks which have cycles, these values can also be sent back to previous layers. The values of nodes from

the input layer are called input values and are usually denoted as $x_i, i = 1, \ldots, |V_0|$. The vector $(x_1, \ldots, x_{|V_0|})$ is called an **input** to the network. The vertices $v \in V \setminus V_0$ are called **neurons**. Neurons are modelled by scalar functions $g : \mathbb{R} \longrightarrow \mathbb{R}$ which are called **activation functions**. The value of a neuron is equal to the value of the activation function on a sum of multiples of activation function outputs from the neurons in the previous layer and weights of the edges which connect them to the given neuron. The vector of activation function results on the neurons in the output layer $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_{|V_m|})$ is called the **output** of the network. Each layer which is not the output layer contains one vertex with the constant value 1. The edge which connects this neuron with some other neuron is called the **bias**. The vector of all parameters (weights and biases) of a network is denoted by a Greek letter θ.

Remark 3.5 *Commonly, the terms defined above are written in matrix notation form:*

- $n_0 = |V_0|$ *the length of the input* x
- $n_l = |V_l|$ *the number of neurons in the* l*-th layer*
- $a^{[l]} \in \mathbb{R}^{n_l \times 1}$ *the vector of activation function outputs of the* l*-th layer*
- $W^{[l]} = (w_{ij}^{[l]}) \in \mathbb{R}^{n_l \times n_{l-1}}$ *the matrix of edge weights, where* $w_{ij}^{[l]}$ *denotes the weight of an edge which connects the* j*-th neuron in the layer* V_{l-1} *with the* i*-th neuron in the layer* V_l
- $x = a^{[0]}$ *the input vector*
- $\hat{y} = a^{[m]}$ *the output vector*

- $b^{[l]} \in \mathbb{R}^{n_l \times 1}$ *the bias of the layer* V_l *(which can be included in the weights matrix* $W^{[l]}$*)*
- $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
- $g^{[l]}$ *the activation function of the* l*-th layer (most commonly all neurons in the layer use the same activation function)*

Neural networks operate on data, in a specific way, such that a portion of the given data is used to determine the best possible values for weights, and another portion of the data is used for evaluation of the calculated weights. Let X, Y be sets of possible inputs and outputs of the network, respectively. The most common task of deep learning is so-called supervised learning, where data has a form of feature vectors along with specified labels, which denote the class to which the feature vector belongs. The goal of such tasks is to learn a map $f : X \longrightarrow Y$ such that $\hat{y} := f(x)$ is a good prediction of the exact value y for the input x. The process of learning this function is called **training** the network, and the data portion which is used for this purpose

$$D_{train} = \{(x_i, y_i) : i = 1, \ldots, n\} \subseteq X \times Y$$

is called a **train set**, where y_i denotes the label of the input x_i.

The term "model" is commonly used to denote the prediction function f. If $Y = \{1, \ldots, K\}$, then the task of learning the prediction function is called **classification**, and if $Y = \mathbb{R}$, then the task is called **regression**. A neural network can be understood as a function $f_\theta : x \mapsto \hat{y}$ completely determined by its parameters θ, the weights and biases, which are calculated during the training process. The parameters which also determine the network architecture, but are not directly included in calculations of the training process, such as network depth, are called **hyperparameters**.

The portion of the data which is used for evaluation of the network can be split into two parts, one of which is used for hyperparameter tuning $D_{val} \subseteq X \times Y$ (**validation set**), and another for the final evaluation of the network capabilities $D_{test} \subseteq X \times Y$ (**test set**). This can be summed up in the following way: the whole data set D is partitioned into three disjoint sets $D_{train}, D_{val}, D_{test}$, or into two disjoint sets D_{train}, D_{test}, where D_{train} can then, optionally, be used for validation too. The most important condition related to the partition of data which must be met is that testing data must not ever be seen during training. If it is, it can skew results of the evaluation and give incorrectly accurate predictions. This phenomenon is called data leakage, and it must be avoided in order to enable accurate analysis of training results.

In order to quantify the error between the network predictions $\hat{y}$ and the true values y, a concept of **loss function** L is introduced. The value of loss function $L(\hat{y}, y)$ for given arguments $\hat{y}, y$ is called **loss**, and is proportional to the prediction error. For correct predictions, that is, when $\hat{y} = y$, its value is equal to zero. The choice of loss function is tied to the output layer of the network, and differs across various tasks of deep learning. The global loss function J across all data D is called the **cost function** and is defined as an average loss on all points in the set

$$J(\theta; D) = \frac{1}{|D|} \sum_{(x,y) \in D} L(\hat{y}, y).$$

Using this term, it is now possible to frame the training process as an optimization problem of minimizing the cost function value through an iterative process of updating parameters θ, in order to find the best possible set of parameters

$$\theta^* = \underset{\theta}{arg\,min}\, J(\theta; D_{train}).$$

One iteration across the whole train set D_{train} in which parameters θ are

updated is called an **epoch**. The optimization method used for updating the parameters θ is gradient descent, which is based on updating the parameters in the direction of negative gradient $-\nabla_\theta J(\theta; D_{train})$.

The most computationally expensive part of gradient descent is calculating the gradient of the cost function $\nabla_\theta J(\theta)$. This is done by using a procedure called **backpropagation**, which relies on the chain rule:

Theorem 3.6 (Chain rule) *Let $g : \mathbb{R}^m \longrightarrow \mathbb{R}^n$, $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be functions. If $y = g(x)$ for some x, and $z = f(y)$ for some y, then*

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

The needed partial derivatives are calculated starting from the output layer, propagating back until reaching the input layer.

3.2 Recurrent neural networks

A recurrent neural network is any network that contains a cycle in its network connections. This means that values of some units are either directly or indirectly dependent on their own earlier outputs. Even though many recurrent neural network architectures exist, in this thesis a specific type is going to be explained: the simple recurrent neural network, first proposed in 1990 by Elman[]. These networks serve as a basis for more complex models, one of which is especially important and will be explained in further detail later - the Long Short-Term Memory. Recurrent neural networks are used for several purposes regarding language modelling: sequence classification (tasks like sentiment analysis and topic classification), sequence labelling (tasks like part-of-speech tagging), and text generation (tasks like text summarization and machine translation).

3.2.1 The structure of recurrent neural networks

Figure 3.1 illustrates the structure of a simple recurrent neural network. An input vector x_l representing the current input is multiplied by the weight matrix $W^{[l]}$ and then passed through the activation function $g^{[l]}$ to compute the values for the next layer. Subscripts will be used in order to represent time, so x_t will denote the input vector x (of any which layer) at a time t. The key difference from a feed-forward network, where values from neurons are propagated exclusively forward, is in the recurrent link shown in the figure with a dashed line. This link augments the input to the computation at the hidden layer with the value of the hidden layer from the past.

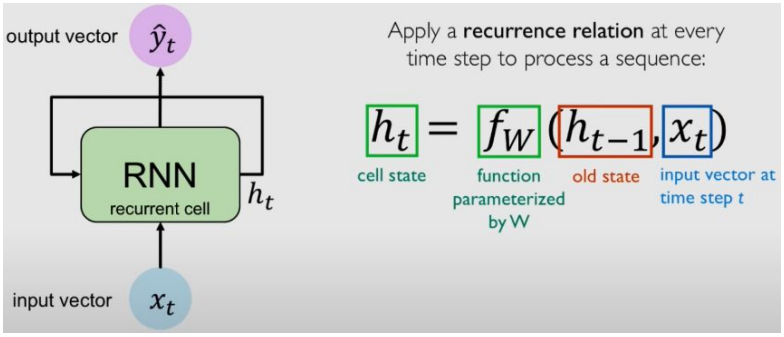

Figure 3.1: Recurrent cell of a neural network. Figure from [22].

The hidden layer from the preceding point in time provides a form of memory or context that saves results of earlier processing and helps inform decisions that are made later. This approach does not impose a fixed-length limit on the prior context. The context given by the previous hidden layer can include information from the beginning of the sequence.

The addition of the temporal dimension to the neural network makes recurrent neural networks appear more complex, but they are not that different in their mechanics from regular feed-forward networks. The standard feed-forward calculations are still performed. The most significant change is

the introduction of the new set of weights U that connect the hidden layer from the previous step to the current hidden layer. These weights determine how the past context is used in calculating the output for the current layer. These connections are trained with backpropagation, as other weights in the network.

Arriving at the sequence of outputs y_t from the sequence of inputs x_t is done similarly as in standard feed-forward networks, by applying the activation function to the sum of multiples of weights and inputs:

$$h_t = g(Uh_{t-1} + Wx_t)$$

$$y_t = f(Vh_t)$$

where h_t denotes the current hidden layer neuron values, h_{t-1} the values of neurons from the previous layer, $W \in \mathbb{R}^{d_h \times d_{in}}$, $U \in \mathbb{R}^{d_h \times d_h}$, $V \in \mathbb{R}^{d_{out} \times d_h}$, with d_{in}, d_h, d_{out} being the dimensions of the input, hidden, and output layers, respectively. The most commonly used activation function f is softmax, which gives a probability distribution over the set of possible output classes.

Definition 3.7 *Softmax activation function is a function* $g : \mathbb{R}^K \longrightarrow \mathbb{R}^K$ *defined by*

$$g(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}; i = 1, \ldots, K,$$

where K *is the number of possible classes of output.*

Remark 3.8 *Softmax is a generalization of the sigmoid function:*

$$g : \mathbb{R} \longrightarrow \mathbb{R},$$

$$g(z) = \frac{1}{1 - e^{-z}}.$$

The softmax function gives a probability distribution of a discrete random variable which takes K different values (also called classes). Finally, this makes the output $y_t = softmax(Vh_t)$.

Recurrent neural networks are trained like simple feed-forward networks; by using backpropagation to update weights in the direction of the negative gradient, in order to minimize the cost function. There are three sets of weights to update: W - weights from the input layer to the hidden layer, U - weights from the previous hidden layer to the current hidden layer, and V - weights from the hidden layer to the output layer. However, there is a difference, thanks to the cycles present in the network architecture. Now, in order to calculate the loss function for the output at time t, the hidden layer from time $t-1$ is needed, but also the hidden layer from time $t+1$ because the hidden layer at time t influences *both* hidden layer at time $t-1$ and $t+1$.

It is necessary to change the backpropagation algorithm into a two-pass version. In the first pass, forward inference is performed (computing h_t, y_t and accumulating the loss at each step in time). In the second pass, the sequence is processed in reverse, computing the required gradients, saving the error terms for use in the hidden layer backward in time. This approach is called **backpropagation through time** [23] [24].

The input $x = (x_1, \ldots, x_t, \ldots, x_{|V|})$ consists of a series of word embeddings which are represented by one-hot vectors of size $|V| \times 1$, and the output y is a probability distribution over the vocabulary V. At time t, the following is done:

$$e_t = Ex_t$$

$$h_t = g(Uh_{t-1} + We_t)$$

$$y_t = softmax(Vh_t),$$

where E is a word embedding matrix of type $(d_h \times |V|)$, where d_h is the

size of the current hidden layer. The probability that a particular word i from the vocabulary is the next word is represented by $y_t[i]$, the i-th component of y_t:

$$P(w_{t+1} = i | w_1, \ldots, w_t) = y_t[i].$$

The probability of the entire sequence is a product of probabilities of each item from the sequence:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) = \prod_{i=1}^{n} y_i[w_i].$$

The loss function which is used for training the network is **cross-entropy loss**, which measures the difference between a predicted probability distribution and the true distribution:

$$L_{CE} = -\sum_{w \in V} y_t[w] \log \hat{y}_t[w].$$

The correct distribution y_t comes from knowing the next word in the sequence, and so, the cross-entropy loss is determined by the probability the model assigns to the correct next word. This implies

$$L_{CE}(\hat{y}_t, y_t) = -\log \hat{y}_t[w_{t+1}].$$

The idea of always giving the model the correct history sequence to predict the next word, instead of giving it the best case from the previous step, is called **teacher forcing**.

3.2.2 Recurrent neural networks for text generation

Text generation tasks cover any situation where a system needs to produce text conditioned on some other text. The approach of using a language model to generate words by repeatedly sampling the next word conditioned

on previous choices is called **autoregressive generation**. An autoregressive model is a model that predict the output at time t by taking into account the linear function of previous values in times $t-1, t-2$ etc. Even though recurrent neural networks are highly non-linear systems, this label is still used to describe such a language model because the logic holds, since the word generated at each time step depends on previously seen words.

Since the structure of recurrent neural networks is flexible, it is possible to combine several of them into new model architectures: stacked and bidirectional ones.

Stacked recurrent neural network consists of several networks where the output of one layer serves as an input to the next one; thus using entire sequences as inputs and outputs, instead of just one word at a time. The main advantage of this model, and the probable reason for its superiority over single-layer networks, is induction of different levels of abstraction across layers. The initial layers of a stacked network can induce representations that prove to be useful abstractions for the following layers, which is difficult to achieve with a single-layer network. However, this multi-layered approach increases the computational cost significantly and makes the training process a much more elaborate process.

Bidirectional recurrent neural network is a network which consists of two separate recurrent neural networks, where one operates "from left to right", and the other one "from right to left". More concretely, this means that one of them uses information from the previous points in time (past) in order to make a decision at the current time, and the other one uses information from the following points in time (future). The state of a left-to-right network is a function of inputs which represent the context to the

left of the current time t:

$$h_t^f = RNN_{forward}(x_1, \ldots, x_t),$$

where h_t^f corresponds to the hidden state at time t. The right-to-left network operates on a reversed input sequence, in order to take advantage of the context to the right from the current time t:

$$H_t^b = RNN_{backward}(x_t, \ldots, x_n),$$

where h_t^b represents all the information discerned about the sequence from time t to the end. A bidirectional recurrent neural network combines two independent recurrent neural networks and concatenates their outputs, thus capturing the context at time t from both the past and the future.

3.2.3 Long short-term memory networks

One issue that arises in practice with recurrent neural networks is that the information in hidden states is local, even though the network has access to the entire preceding sequence. Ideally, a network is desired to keep the distant information from parts of the sequence which are further away in time, be it in the past, or the future.

One reason why this is not achieved with standard recurrent neural networks is the fact that each hidden layer is required to do two tasks at the same time: providing information relevant to the current decision, and carrying forward information required for future decisions. Another reason for difficulty in training recurrent neural networks is the need to backpropagate the error back through time. During the backward pass of training, hidden layers are subject to repeated multiplications which frequently results in driving the gradients to zero. This phenomenon is called the **vanishing gradients** problem.

In order to addess these problems, managing of relevant context over time, explicitly, by enabling the network to "forget" information that is no longer needed, has been introduced in more complex models. The most common improvement upon the standard recurrent neural network is a Long short-term memory network [25]. Long short-term memory networks divide the context management problem into two parts: removing information which is no longer needed (forgetting), and adding information which is likely to be needed later down the line. They solve these sub-problems by using specialized neurons called **gates** which control the flow of information into and out of neurons that comprise the network layers. The gates are implemented by using additional weights, and they share a common design: each consists of a feed-forward layer, followed by sigmoid activation function, followed by pointwise multiplication with the gated layer. Sigmoid function is chosen because it has a tendency to push the outputs to either zero or one, which, combined with pointwise multiplication, has an effect similar to a binary mask. Values in the gated layer that align with values near one are passed through barely changed, and the values corresponding to gate values closer to zero are erased.

There are three types of gates a long short-term memory network uses:

- **forget gate** - in charge of deleting information irrelevant to the current decision,
- **add gate** - in charge of selecting information to add to the current context which needs to be preserved for future decisions,
- **output gate** - in charge of deciding what information is required for the current hidden state.

The forget gate computes a weighted sum of the previous state hidden

layer output and the current input, and then passes this result through the sigmoid activation function σ:

$$f_t = \sigma(U_f h_t + W_f x_t).$$

This mask is then multiplied element-wise by the context vector to remove the information which is no longer necessary:

$$k_t = c_{t-1} \odot f_t,$$

where $\odot$ denotes the Hadamard product. The following step is calculating

$$g_t = \tanh(U_g h_{t-1} + W_g x_t)$$

in order to extract the needed information from the previous hidden states and the current input.

Remark 3.9 *The function used in the above expression is tangent hyperbolic activation function* $\tanh : \mathbb{R} \longrightarrow \mathbb{R}$ *defined with*

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

Next, a mask for the add gate is generated:

$$i_t = \sigma(U_i h_t + W_i x_t)$$

$$j_t = g_t \odot i_t,$$

and is added to the modified context vector in order to get the updated context vector:

$$c_t = j_t + k_t.$$

Finally, the output gate is used:

$$o_t = \sigma(U_o h_{t-1} + W_o x_t)$$

$$h_t = o_t \odot \tanh(c_t).$$

3.3 Transformer neural networks

While the long short-term memory network is an improvement upon the recurrent neural network, it still does not solve the problem of information loss and training difficulty, caused by passing information through a series of recurrent connections. Another side-effect of the sequential architecture of the recurrent neural network is difficulty in implementing parallel computing techniques. Transformers are designed without recurrent layers and are a come-back to the original structure of neural networks.

Transformers map sequences of input vectors $x = (x_1, \ldots, x_n)$ to sequences of output vectors $y = (y_1, \ldots, y_n)$ of the same length. They are made up of stacks of transformer blocks - multilayer networks that combine feed-forward linear layers and **self-attention layers**. These layers are the key innovation in this model that allow the network to directly take information from contexts of arbitrary size, without the need to pass it through intermediate connections.

3.3.1 Self-attention layers

Self-attention layers map sequences of input vectors $x = (x_1, \ldots, x_n)$ to sequences of output vectors $y = (y_1, \ldots, y_n)$ of the same length. While processing items in the input vector, the layer has access to all input points preceding the current one, but not to the following ones. This ensures the usefulness of transformer models for autoregressive text generation. The computations done at each point are mutually independent. This property ensures easy parallelization of forward inference and training.

The attention-based approach is rooted in the possibility of comparing an item of interest to a whole collection of other items, while also revealing

their relevance in the current context. When talking about self-attention, this mechanism is used on a single sequence, that is, items from a sequence are compared to each other, and not to items from some other collection.

The simplest form of comparison is the dot product. Let *score* denote the result of a dot product of two items:

$$score(x_i, x_j) = x_i \cdot x_j.$$

The larger the value, the higher the score, and the greater the similarity between items being compared x_i, x_j. In order to make effective use of scores, a softmax activation function is used to normalize the results and to create the weights vector α_{ij}, that indicates the relevance of each input:

$$\alpha_{ij} = softmax(score(x_i, x_j)); \forall j \leq i.$$

An output value y_i is generated by summing up the inputs thus far, weighted by their α values:

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j.$$

In order to capture different roles of word embeddings which naturally arise, transformers use weight matrices:

- W^Q - role of embeddings as the current focus of the attention while being compared to all other inputs (*query*),
- W^K - role of embeddings as the preceding input being compared to the item which is the current focus of attention (*key*),
- W^V - role of embeddings as the values used to compute the output for the current focus of attention (*value*).

These weights are used for projecting each input vector x_i into a representation of its role as a query q_i, key k_i, or value v_i:

$$q_i = W^Q x_i,$$

$$k_i = W^K x_i,$$

$$v_i = W^V x_i.$$

Let the type of weight matrices be denoted as d, that is $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$. When introducing multi-headed attention, an improvement upon the current procedure, separate dimensions for these matrices will be introduced: d_k for query and key vectors, and d_v for value vectors, meaning $W^Q, W^K \in \mathbb{R}^{d \times d_k}, W^V \in \mathbb{R}^{d \times d_v}$. For now, let $d_k = d_v = d$.

Given these projections, the score between the current focus of attention x_i and an element from preceding context x_j is calculated as the dot product between the query vector of the given i-th item and the key vectors of preceding items, scaled by their dimensionality for numerical stability:

$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}},$$

where $q_i, k_j \in \mathbb{R}^{1 \times d_k}$. The output calculation is now based on a weighted sum:

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j.$$

By putting the input embeddings of a sequence of length N into a matrix form $X \in \mathbb{R}^{N \times d}$, it is possible to exploit matrix multiplication routines which are numerically more effective:

$$Q = XW^Q; K = XW^K; V = XW^V,$$

where $Q, K, V \in \mathbb{R}^{N \times d}$. Additionally, by applying the softmax function to scaled values of these scores, and then multiplying it with V, it is possible to

reduce the entire procedure for a sequence of N items to a single computation:

$$SelfAttention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \in \mathbb{R}^{N\times d}.$$

The elements in the upper-triangular portion of the matrix QK^T are set to $-\infty$ to reflect the fact that it is unnecessary to include already familiar following items into a calculation for guessing it, thus producing a matrix of the form

$$QK^T_{(N,N)} = \begin{bmatrix} q_1 \cdot k_1 & -\infty & \cdots & -\infty \\ q_2 \cdot k_1 & q_2 \cdot k_2 & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ q_N \cdot k_1 & q_N \cdot k_2 & \cdots & q_N \cdot k_N \end{bmatrix}.$$

3.3.2 Transformer blocks

The self-attention logic is the core of a transformer network unit called a **transformer block**, which also includes additional feed-forward layers, residual connections, and normalizing layers. The input and output of the blocks are matched so that the blocks can be stacked together.

Residual connections pass information from a lower layer to a higher one without going through the intermediate layer, allowing information from the activation to go forward and the gradient to go backward by skipping a layer, which improves training and gives higher layers direct access to information from lower layers. Layer normalization is applied to the sum of the layer's input and output, represented in the following calculation, assuming that layers are denoted as vectors of units:

$$z = LayerNorm(x + SelfAttention(x))$$

$$y = LayerNorm(z + FeedForward(z)),$$

where $LayerNorm$ denotes layer normalization and $FeedForward$ denotes a feed-forward layer.

Layer normalization is a variation of the standard score (z-score) applied to a single hidden layer. Given a hidden layer of dimensionality d_h, the factors are calculated as follows:

$$\mu = \frac{1}{d_h} \sum_{i=1}^{d_h} x_i$$

$$\sigma = \sqrt{\frac{1}{d_h} \sum_{i=1}^{d_h} (x_i - \mu)^2}.$$

Given these values, vectors are normalized in the standard way:

$$\hat{x} = \frac{x - \mu}{\sigma}.$$

Finally, two learnable parameters β, γ are introduced, in order to represent gain and offset values:

$$LayerNorm = \gamma \hat{x} + \beta.$$

3.3.3 Multihead attention

The concept called multihead attention is introduced for solving the problem of distinguishing and utilizing different ways in which words relate to each other simultaneously. For example, these relationships between words can be syntactic, semantic or discourse driven. It can be difficult to capture all of these intricacies using a single transformer block. Hence, several self-attention layers, called **heads**, are used, each one for a different word relationship type. They are placed at the same depth in the network, each one with its set of parameters.

Each head i is provided with its own set of query, key and value matrices W_i^Q, W_i^K, W_i^V which are used to project inputs into separate query, key and

value embeddings. The query and key embeddings have a size d_k, and the value embedding has a size d_v, resulting in $W_i^Q, W_i^K \in \mathbb{R}^{d \times d_k}, W_i^V \in \mathbb{R}^{d \times d_v}$ for each head. Just as before, these matrices are multiplied by X from the left, resulting in $Q, K \in \mathbb{R}^{N \times d_k}, V \in \mathbb{R}^{N \times d_v}$.

$$Q = XW_i^Q$$

$$K = XW_i^K$$

$$V = XW_i^V.$$

The output of each one of the h heads is of shape $N \times d_v$, producing h vectors of this size, which are combined and reduced to the size d, in order to make them suitable for further calculations in the network. This is done by using a linear projection $W^O \in \mathbb{R}^{hd_v \times d}$ on a concatenation [2] of head outputs:

$$MultiheadAttention(X) = (head_1 \oplus \ldots \oplus head_h)W^O$$

$$head_i = SelfAttention(Q, K, V).$$

3.3.4 Transformers as language models

Transformers are used as language models via semi-supervised learning: given a train set that consists of plain text, a network is learned to predict the next words in sequences by teacher forcing. The output layer of the transformer network produces a distribution over the vocabulary at each step, with the context of all the preceding words. The probability assigned to correct words during training is used to calculate cross-entropy loss for each item of the sequence. The loss for a training sequence is average cross-entropy loss.

The benefit of parallelization is seen in the possibility of training the network in such a way that each training item can be processed in parallel because outputs are computed separately.

[2] The operation of concatenation is denoted by $\oplus$.

3.3.5 Text generation and summarization using transformers

A variation of autoregressive generation which underpins a large number of applications in practice uses prior context. A good illustrative task is text completion where a model is expected to generate a possible completion to the given text. The model has direct access to the prior context and its own outputs, resulting in an ability to use the knowledge of the entire earlier context and outputs at each time step, which is crucial for the effectiveness of such models.

Text summarization is nothing else than a practical application of autoregressive generation based on context. In order to train a transformer based model for this task, it is necessary to provide it with enough samples of full-length texts along with their summaries. This makes text summarization a supervised task, because labels in this case are the expected summaries, in a way. A suprisingly simple approach is effective: appending the summaries to the full-length texts with a unique marker separator d. Each text-summary pair $(x_1, \ldots, x_m), (y_1, \ldots, y_n)$ is converted into a single instance $(x_1, \ldots, x_m, d, y_1, \ldots, y_n)$ for training. These items are treated as long sentences.

After training, the texts ending with the separator are used as the context in the generation process. In contrast to the recurrent neural network language model, the transformer based one has access to the newly generated text as well as the original one throughout the process. This logic is the basis for more complex models which are used for text-to-text applications, such as machine translation, text summarization and question answering.

3.4 Encoder-decoder neural networks

Encoder-decoder neural networks, or sequence-to-sequence networks, are models capable of generating outputs of arbitrary length, which are context-based. The key idea is having two networks in one model:

- **encoder** network - in charge of taking an input sequence and creating its contextual representation,
- **decoder** network - in charge of taking said representation and producing an output specific to the task.

The encoder takes an input x_i^n and produces a representation h_i^n, which is then put through a function of the context vector, resulting in an input for the decoder c. The decoder then takes c and produces a sequence of hidden states h_i^m, from which outputs y_i^m are obtained. Encoders can be long short-term memory networks, transformers or even convolutional neural networks, while decoders can be any sequential models.

3.4.1 Encoder-decoder models with recurrent neural networks

This architecture is based on two recurrent neural networks as the encoder and the decoder. The probability of sequence $y = (y_1, \ldots, y_m)$ can be broken down to the following:

$$P(y) = P(y_1)P(y_2|y_1)\ldots P(y_m|y_1, \ldots, y_{m-1}).$$

At a given time t, the prefix of $t-1$ items is passed through the language model to produce a sequence of hidden states which ends with the hidden

state corresponding to the last word in the prefix. The final hidden state is used as a starting point in generating the following item:

$$h_t = g(h_{t-1}, x_t)$$

$$y_t = f(h_t),$$

where g is an activation function (e.g. tanh), and f is a $softmax$ over the set of possible vocabulary items. Superscripts e, d will be used to denote the hidden states of the encoder and the decoder, respectively.

Stacked architectures for the encoder are common, where the output states from the last layer of the stack are taken as the final representation outputted by the encoder h_n^e. A widely used encoder model is a bidirectional long short-term memory network. The hidden states from last layers from forward and backward passes are concatenated in order to form the needed contextualized representation, for each time step. The decoder uses the input $c = h_n^e$ as its prior hidden state h_0^d and creates the output sequence one element at a time, while keeping each hidden state conditioned on the previous one.

The influence of the context vector c weakens as the output sequence is generated. The solution is making it available at each step of the sequence generation by adding it as a parameter in the computation of the current hidden state:

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c),$$

where y_{t-1}^d is the embedding for the output sampled from the *softmax* at the previous step in time $t-1$.

Taking all of the above into account, the final form of the whole encoder-decoder model can be described by the following equations:

$$c = h_n^e$$

$$h_0^d = c$$

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c),$$

$$z_t = f(h_t^d)$$

$$y_t = softmax(z_t).$$

Finally, the most likely output at each time step is chosen as the one with the highest probability, that is, the largest value of *softmax*:

$$\hat{y}_t = \underset{w \in V}{arg\,max}\, P(w|x, y_1, \ldots, y_{t-1}).$$

3.4.2 Encoder-decoder models with transformers

The components of this architecture differ slightly from the encoder-decoder model which uses recurrent neural networks, but also from transformers themselves. Namely, a concept of the **cross-attention** layer is introduced - an extra layer added into the decoder transformer blocks, which has the same form as multihead self-attention layers, except for the fact that the key and value pairs come from the encoder outputs (instead of the previous layer of the decoder).

The final output of the encoder $H^{encoder}$ is multiplied by the key weights from the cross-attention layer W^K and value weights W^V. The output from the prior decoder layer $H^{decoder[i-1]}$ is multiplied by the cross-attention layer query weights W^Q:

$$Q = W^Q H^{decoder[i-1]}$$

$$K = W^K H^{encoder}$$

$$V = W^V H^{encoder}$$

$$CrossAttention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V.$$

Teacher forcing is used in the training process, just as before, and the training is done autoregressively, while using cross-entropy loss.

Chapter 4

Examples of using machine learning and numerical methods

In this chapter, practical examples of using the previously described methods are shown. Google Colaboratory is used, a Jupyter notebook environment in the cloud that ensures free use of Tesla K80 graphics processing units. The experiments are written in Python 3.8 programming language using the Keras deep learning librabry with Tensorflow support. The goal of experiments here is not producing the best possible results on the given task, but simply demonstrating how the described methods can be used in practice.

Since matrix factorization methods predate machine learning methods, they are used on simpler tasks of topic modelling and key-word extraction from text. As text summarization is a much more difficult procedure, machine learning is more suitable for this task because it is more powerful.

Difficulties of producing a good summary from given text are many. Firstly, there is a need to transform the text into a numerical form of data.

This can be done naively, by using tf-idf [16] or bag-of-words methods, or in a more sophisticated way by using neural network models such as *word2vec* [14], *gloVe* [13] or *BERT* [15]. Explanations of the functionalities of these methods is beyond the scope of this thesis.

Numerical data, usually in vector form, which is produced by the chosen method from given text is called **word embedding**. Word embeddings represent words in such a way that words closer to each other in meaning are their vector representatives are closer to each other in the embedding space (by some chosen metric).

Second problem that arises is the actual formation of summaries from text inputs. Lastly, the evaluation of summaries is a complex task in itself. Currently, it is best done manually, that is, by human assessment. This is an active topic of research. There exist some metrics specifically for this purpose, but are still underdeveloped. An example of this is *ROUGE* metric [17].

4.1 Data preparation

Two methods are chosen for creating word embeddings, to demonstrate the power of each: vectorization by counting the number of occurrences of a word in the text, and vectorization by tf-idf, as previously explained.

In order to apply vectorization by counting, another data preparation step needs to be completed: **tokenization**. Tokenization refers to the procedure of transforming pieces of text into lists of so-called tokens - smallest possible semantic units, or atoms, which depend on the language. Another step of data preparation is recommended for better performance: **lemmatization**. Lemmatization is the task of determining that two words have the same root,

despite their surface differences.

Remark 4.1 *Both tokenization and lemmatization belong to a class of procedures called* ***text normalization*** *which serve the purpose of transforming text into desirable, more manageable forms.*

All numerical methods are implemented by calling the corresponding function from the *sklearn* [18] library. The reason why *sklearn* implementations have been used is in order to ensure numerical stability and algorithm optimality.

Two datasets are used:

- news headlines - table with two columns; date of publishing and headline text, containing 1103663 headlines,
- food reviews - table containing 568454 samples of review texts alongside their proposed summaries written by users of Amazon.

Matrix factorizations are done on news data and deep learning model is used on food reviews data. This choice is made because news headlines are considered to be more useful for topic modelling because they are more diverse in topics and key-words, while food reviews are more suitable for abstractive text summarization for two reasons. The first reason is that this dataset contains human-made summaries, which are needed for supervised learning. The second reason is that reviews are longer pieces of text than news headlines, thus making them more appropriate because there is more text for the model to train on. This also makes the task harder because the vocabulary is larger and the text is more unpredictable, thus showing the true power of deep learning.

4.2 Non-negative matrix factorization for key-word extraction

The algorithm is implemented by using an *sklearn* library function which uses the following objective function:

$$L(W,H) = \frac{1}{2}||V - WH||^2_{loss} + \alpha_W l_1 N||W||_1 + \alpha_H l_1 M||H||_1 +$$
$$+\frac{1}{2}\alpha_W(1-l_1)N||W||^2_F + \frac{1}{2}\alpha_H(1-l_1)M||H||^2_F,$$

where N is the number of features, M is the number of samples and l_1 denotes the norm-1 ratio, and $||\cdot||_{loss}$ can be any chosen norm supported by the back-end implementation (passed as a function argument). Parameters α_W and α_H control the strength of regularization applied to W and H, respectively.

The non-negative matrix factorization function is applied to the embedding matrix of news headlines text which is created by the use of tf-idf vectorization. It is possible to choose a desired number of topics by hand, in this case 10. This is forwarded to the function as an argument. The output is a feature matrix representing topics found in the text.

As the chosen number of topics is 10, the shapes of matrices X, W, H are (1103663, 11213), (1103663, 10), (10, 11213), that is

$$X \in M_{1103663,11213}(\mathbb{R}),$$

$$W \in M_{1103663,10}(\mathbb{R}),$$

$$H \in M_{10,11213}(\mathbb{R}).$$

It is possible to name the topics and assign to them their most relevant key-words, as shown in 4.1.

As a result, it is possible to determine the topic of any headline, as shown in 4.2, which has some relation to the data on which the model is trained.

```
For topic 7 the words with the highest value are:
crash      5.455308
car        2.480800
dies       1.841566
killed     1.837934
fatal      1.451367
woman      1.178756
road       0.986225
driver     0.875721
plane      0.778262
injured    0.651799
```

Figure 4.1: Assigning names to numerated topics according to their most relevant and common key-words.

```
my_document = documents.headline_text[105]
my_document
```

```
'national gallery gets all clear after'
```

```
pd.DataFrame(nmf_features).loc[105]
```

```
0    0.000351
1    0.000000
2    0.000000
3    0.001151
4    0.019754
5    0.000000
6    0.000000
7    0.000000
8    0.000000
9    0.001774
Name: 105, dtype: float64
```

```
pd.DataFrame(nmf_features).loc[105].idxmax()
```

```
4
```

Figure 4.2: Finding the most probable topic for the given headline (in this case the document sample 105).

The quality of the classification depends on the number of headlines in the dataset for training; the more data, the more quality model. Of course, it is possible to provide the model with any input (as long as it is in string

format), and gain the output. However, the quality of the output depends on the input text, that is, if the text has nothing to do with training data (in this case news headlines), then the output is irrelevant because the model has no knowledge of the subjects of the given text.

4.3 Encoder-decoder neural network model with bidirectional long short-term memory cells

The structure of the neural network is in 9 layers, which constitute the encoder and the decoder:

- encoder input layer which accepts sequential data,
- encoder embedding layer which produces word embeddings from the input,
- 3 encoder bidirectional LSTM cells,
- decoder input layer,
- decoder embedding layer,
- decoder bidirectional LSTM cell,
- dense (fully connected) layer with *softmax* activation function which outputs probabilities for each word in the vocabulary to be the next word in the sequence.

This sequential kind of network architecture is refered to as a **sequence-to-sequence** (or colloquially *seq2seq*) model [19]. Models that belong to

this family of machine learning approaches turn one sequence into another sequence, which is refered to as sequence transformation.

This is achieved by implementing recurrent neural units in the neural network, or, more commonly, long short-term memory cells. Each step produces an output which then becomes context for the following step. Duty of the encoder is to produce each item into a hidden (*latent*) vector containing the input item and its context. The decoder then reverses this process, that is, turns the vector into an output item, using the previous output as the input context.

In the context of text summarization, this means that the encoder produces hidden vectors from input review text embeddings, and the decoder then outputs a summary sequentially from these hidden vectors, using previous outputs as its context, which is described in more detail in the previous chapter.

The reviews data is split into the training and testing portion, in the ratio 90 : 10%. During training, 20% of the training data is used for validation in each epoch. The model is trained on 50 epochs in batches of size 128. During training, the accuracy of the model is being tracked and used as an indicator of performance quality. The optimizer is *RMSProp* - Root Mean Square Propagation which is the extension of the Stochastic Gradient Descent algorithm.

The accuracy of the trained model is 98.7% on the training data and 98.3% on validation data (measured during training). Final model accuracy on test data is 98%.

The reason why training stopped after 12 epochs instead of the given 50 epochs is because a technique called **early stopping** has been implemented in order to possibly reduce the training time, while not losing a significant

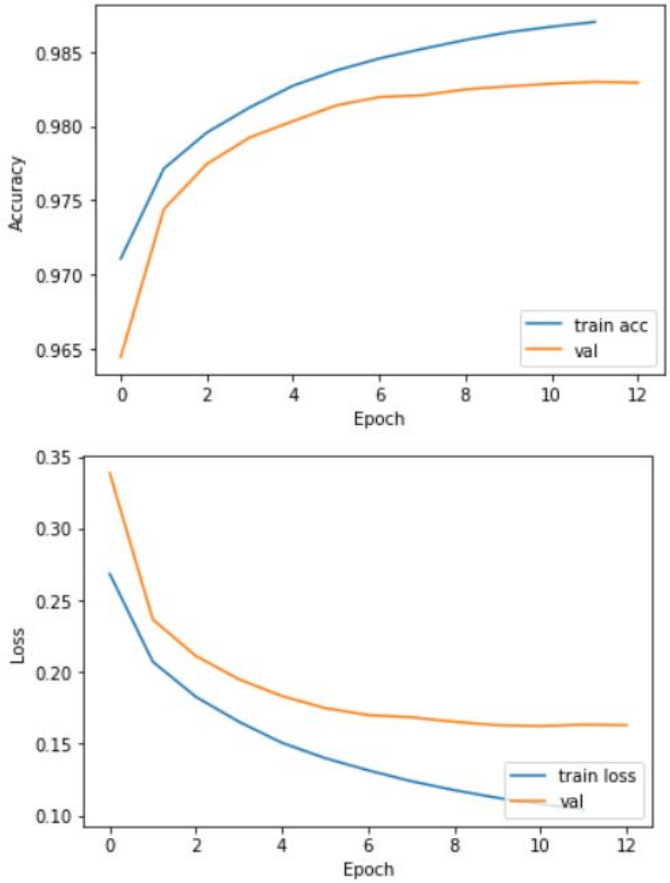

Figure 4.3: Accuracy and loss of the model during training.

amount of accuracy. The stopping criterion is accuracy on the validation data during training. More detailed explanation of this regularization technique can be found in [4].

Conclusion

This thesis covers theoretical foundations for text summarization techniques, accompanying them by practical examples of usage on real world data. Two different approaches are used: numerical, using matrix factorizations, and deep learning, using encoder-decoder neural network model with bidirectional long short-term memory cells.

The accent is on providing a thorough overview and both intuitive and formal explanations of the mentioned methods. Examples of usage do not represent any attempt at achieving *state-of-the-art* results, but are simply demonstrations of the theoretically described methods in practice.

As numerical methods of non-negative matrix factorization and singular value decomposition predate the neural network sequence-to-sequence modelling, they are suitable for extractive text summarization in the form of topic modelling and key-word extraction. The deep learning model can, however, achieve good results with both the extractive and abstractive approach to text summarizing.

Literature

[1] D. Jurafsky, J. H. Martin, Speech and Language Processing, Stanford University, Draft of January 12, 2022.

[2] N. D. Ho, Nonnegative matrix factorization algorithms and applications, Université catholique de Louvain, École polytechnique de Louvain, 2008.

[3] D. D. Lee, H. S. Seung, Algorithms for Non-negative Matrix Factorization, Advances in Neural Information Processing Systems 13 - Proceedings of the 2000 Conference, NIPS 2000.

[4] I. Marin, Regularization in deep learning, Faculty of Science, University of Split, 2019.

[5] S. T. Dumais, Latent Semantic Analysis, Microsoft Research, Redmond, Washington, Information Science and Technology, 2005.

[6] L. Hogben, K. H. Rosen, Handbook of Linear Algebra, Discrete Mathematics and its Applications, Chapman & Hall/CRC, 2007.

[7] S. Braić, Introduction to Probability and Statistics, Faculty of Science, University of Split, 2019.

Literature

[8] A. Golemac, Basics of Graph Theory, Faculty of Science, University of Split, 2017.

[9] D. I. Martin, M. W. Berry, Mathematical Foundations Behind Latent Semantic Analysis, In Handbook of Latent Semantic Analysis (pages 35-55), 2007.

[10] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, R. Harshman, Indexing by latent semantic analysis, Journal of the American Society for Information Science, 1990.

[11] T. A. Letsche, M. Berry, Large-Scale Information Retrieval with Latent Semantic Indexing, Information Sciences, 1997, (pages 105-137).

[12] E. R. Jessup, J. H. Martin, Taking a new look at the latent semantic analysis approach to information retrieval, Conference: Proceedings of the SIAM Workshop on Computational Information Retrieval, 2005.

[13] J. Pennington, R. Socher, C. D. Manning, GloVe: Global Vectors for Word Representation, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pages 1532–1543), 2014.

[14] T. Mikolov, K. C., G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, arXiv:1301.3781, 2013.

[15] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, arXiv:1810.04805, 2018.

[16] A. Rajaraman, J. D. Ullman, Data Mining, in Mining of Massive Datasets (pages 1 - 17), Cambridge University Press, 2011.

[17] C.-Y. Lin, ROUGE: A Package for Automatic Evaluation of Summaries, in Text Summarization Branches Out (pages 74–81), Association for Computational Linguistics, 2004.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, É. Duchesnay, Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 2011.

[19] I. Sutskever, O. Vinyals, Q. V. Le, Sequence to Sequence Learning with Neural Networks, arXiv:1409.3215, 2014.

[20] P. Paatero, U. Tapper, Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values, Fourth International Conference on Statistical methods for the Environmental Sciences 'Environmetrics', 1994.

[21] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004.

[22] R. Miyamoto, RNN - Recurrent Neural Networks, `https://www.rkenmi.com/posts/rnn-recurrent-neural-networks?lang=en` (visited on September 12, 2022).

[23] P. J. Werbos, Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, 1974.

[24] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, Nature volume 323 (pages 533–536), 1986.

[25] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory, Neural computation volume 9 (pages 1735-1780), 1997.

TEMELJNA DOKUMENTACIJSKA KARTICA

PRIRODOSLOVNO–MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU
ODJEL ZA MATEMATIKU

DIPLOMSKI RAD

Numeričke metode i strojno učenje u automatskom skraćivanju teksta

Domina Sokol

Sažetak:

Skraćivanje teksta je zadatak obrade prirodnog jezika s ciljem stvaranja smislenih sažetaka zadanog teksta uz pomoć računala. Postoje dva glavna pristupa ovome zadatku: numeričke metode za ekstraktivno sažimanje i metode strojnog učenja za ekstraktivno i apstraktivno sažimanje. Cilj ovog diplomskog rada je dati osvrt na obje ove metode, predstaviti njihovu teorijsku pozadinu i intuiciju, dati povijesni kontekst za njih te pokazati praktično korištenje na primjerima.

Ključne riječi:

nenegativna matrična faktorizacija, dekompozicija singularnih vrijednosti, povratna neuronska mreža, latentna semantička analiza

Podatci o radu:

60 stranica, 5 slika, 25 literaturnih navoda, napisano na engleskom jeziku

Mentor: *prof. dr. sc. Saša Mladenović*

Članovi povjerenstva:

prof. dr. sc. Milica Klaričić Bakula

Stipe Marić, mag. math

TEMELJNA DOKUMENTACIJSKA KARTICA

Povjerenstvo za diplomski rad je prihvatilo ovaj rad *u rujnu 2022.*

TEMELJNA DOKUMENTACIJSKA KARTICA

FACULTY OF SCIENCE, UNIVERSITY OF SPLIT

DEPARTMENT OF MATHEMATICS

MASTER'S THESIS

NUMERICAL AND MACHINE LEARNING METHODS IN AUTOMATIC TEXT SUMMARIZATION

Domina Sokol

Abstract:

Text summarization is a task in natural language processing with the goal of producing meaningful summaries from given text, by the use of a computer. There are two main approaches to this task: numerical methods for extractive summarizing, and machine learning methods for extractive and abstractive summarizing. The objective of this master's thesis is to provide an overview of both of these methods, present their theoretical background and intuition, give a historical context for them, and show practical usage on examples.

Key words:

non-negative matrix factorization, singular value decomposition, recurrent neural network, latent semantic analysis

Specifications:

60 pages, 5 figures, 25 references, written in English

Mentor: *professor Saša Mladenović*

Committee:

professor Milica Klaričić Bakula

Stipe Marić

TEMELJNA DOKUMENTACIJSKA KARTICA

This thesis was approved by a Thesis commettee in *September 2022.*