Neural Machine Translation

Published on

Work in progress version

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095

Funded by
the European Union

TABLE OF CONTENTS

1 Introduction 6
1.1 Introduction to machine translation 7
2 Acquiring linguistic data 12
2.1 Acquiring linguistic data - intro 13
2.2 Acquiring linguistic data from the Internet 13
2.3 Acquiring linguistic data from the PDF documents 16
2.4 Task - data extraction 21
2.5 Text preprocessing 22
2.6 Corpus alignment 23
2.7 Task - sentence alignment 28
3 Language models 29
3.1 Language models 30
4 Neural network in machine translation 32
4.1 Neural networks 33
5 Neural language models 50
5.1 Neural language models 51
Word Representation 51
5.2 Word embeddings 54
5.3 Reccurent neural language models 55
6 Neural translation models 58
6.1 Encoder-decoder 59
6.2 Alignment model 59
6.3 Neural translation model - implementation 62
Machine Translation Evaluation 74
1 Natural Language Processing 75
1.1 Introduction to NLP 76
2 Preparation of texts 78
2.1 Tokenization 79
2.2 Stemming 83
2.3 Lematization 86
2.4 Additional features 92
3 Sequence-to-Sequence Model 100
3.1 Introduction 101
3.2 Machine translation - text preparation 102

3.3 Machine translation - model creation 111
3.4 Machine translation - model deployment...121
4 Machine Translation Evaluation 124
4.1 Basics of evaluation 125
4.2 Automatic evaluation metrics..126

Introduction

Chapter 1

1.1 Introduction to machine translation

📖 1.1.1

What is a machine translation?

Machine translation is the automatic conversion of text from one natural language to another, with the quality produced by systems or machine translation tools ranging from very low to very high depending on the internal (linguistic) characteristics of the text and the language pair, i.e., from the processing of the natural language itself and transfer rules.

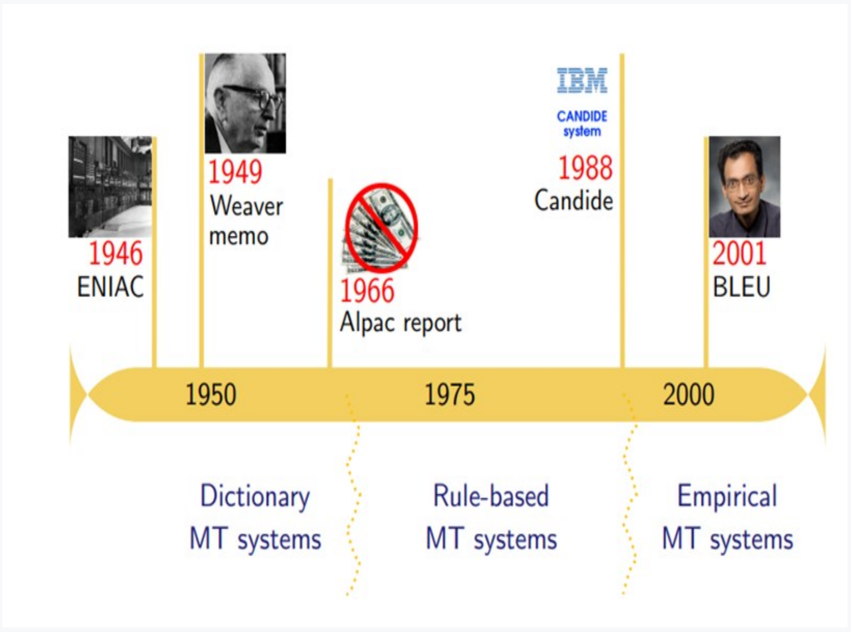
Automatic translations can be divided into three categories. The first category requires a quality suitable for publishing, but this is not possible without intervention by a translator (either by pre-editing the text input or by post-editing the text output). These translations are mainly used for disseminating information in multiple languages (e.g. product brochures). Given the complexity of language, this goal has only been achieved in some applications such as weather forecasting (the Meteo system, which is still in use) or summaries of sports events. The set of vocabulary and grammar in these domains is sufficiently limited to allow for transfer rules to be written.

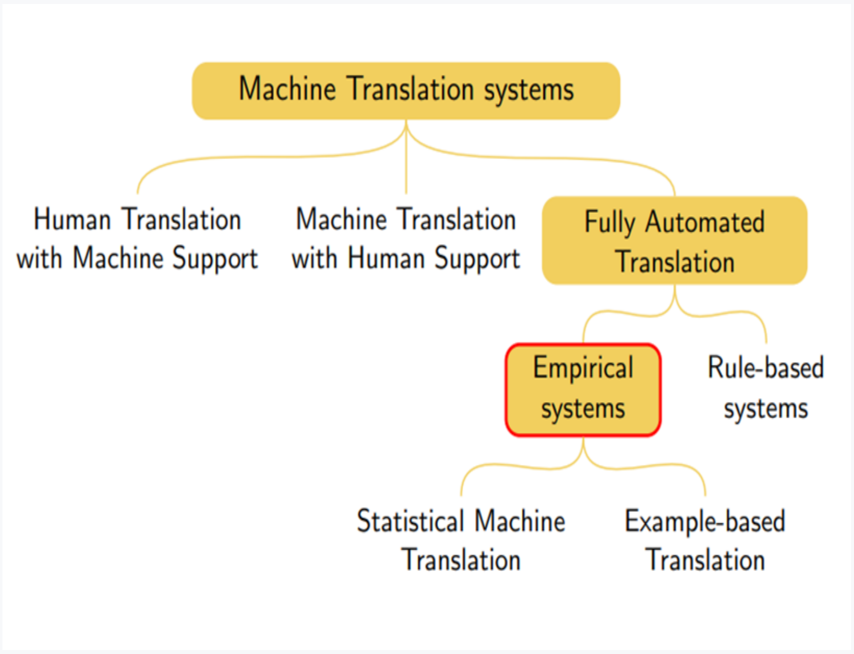
The second category of machine translations consists of "short-term" texts that are not translated by professional translators and are only used for assimilation, i.e., understanding the content of a given text. For example, online translation services are used by users themselves to translate web pages in order to understand what is written in a foreign language. If the user understands the information well enough to solve their technical problem, then machine translation has proven its usefulness. Gisting is the most widespread use of machine translation.

The last category covers machine translations for fast communication (e.g. email), which uses automatic translation for exchanging information.

Koehn (2010) argues that machine translation does not have to be perfect to be usable. He says that even "poor" machine translation has its uses. He identifies with the first two categories, but views the application of the last category as being used in intercultural communication, which includes online communication in synchronous or asynchronous form, as well as in written or spoken form.

An interesting direction for machine translation is its combination with speech technologies (its integration). This opens up wide possibilities for use, such as translation of phone conversations or audio broadcasts. Current test applications include translations of broadcast messages, parliamentary speeches, and interviews in the field of travel. Currently, systems are used that monitor news broadcasts in foreign languages and perform real-time speech translation.




Machine Translation systems
Human Translation with Machine Support
Machine Translation with Human Support
Fully Automated Translation
Empirical systems
Rule-based systems
Statistical Machine Translation
Example-based Translation

📖 1.1.2

Advantages of machine translation

Machine translation is an indispensable tool in the translation process. It can be used on its own or in combination with human post-editing.

Machine translation offers three main advantages:

1. Saves time - machine translation can save a lot of time because it can translate entire text documents within a few seconds. However, keep in mind that translators should always edit machine-translated texts afterwards
2. Reduces costs - machine translation can significantly reduce your costs because it requires less human involvement.
3. It remembers expressions - another advantage of machine translation is its ability to remember key terms and use them again wherever they might be relevant.

📖 1.1.3

Types of machine translation

There are four different types of machine translation - statistical machine translation (SMT), rule-based machine translation (RBMT), hybrid machine translation (HMT), and neural machine translation (NMT)

Rule-based machine translation (RBMT)

RBMT - the oldest form of MT - translates content based on grammatical rules. Since the advent of RBMT, there has been significant progress in machine translation technology, so it has several disadvantages. These disadvantages include the need for a large amount of human post-editing and manual language addition. Despite this low translation quality, RBMT is useful in basic situations where only a quick understanding of the meaning is required.

Statistical machine translation (SMT)

SMT works by creating a statistical model of relationships between words, phrases, and sentences in the text. Then, this translation model is applied to the second language, converting the same elements into the new language. SMT improves RBMT to some extent, but still has many of the same problems.

This method is also one of the simpler ones. It does not use any language-specific information, so it can be used for any combination of languages. To create the dictionary we use for translation, we need a bilingual aligned corpus. From this corpus, we can determine all possible translations of a given word and their frequency of occurrence. We then store a pair in the dictionary: the word and its most frequent translation. Using this dictionary, we can then translate. For example, if we were creating a Czech-Slovak translator and found that the word "hovoriť" can be translated as "mluvit" or "říkat," and the word "mluvit" appears more frequently in the corpus, we would translate all instances of "hovoriť" to "mluvit." If we want to achieve higher accuracy, we count not only the occurrence of individual words but also bigrams, trigrams, and so on. This method provides a higher accuracy dictionary, but its size begins to grow significantly, which can also affect the performance of the translator. However, this method also has many disadvantages. For example, it requires the creation of a bilingual aligned corpus from which frequency of occurrence is calculated. To obtain sufficiently large and reliable data, we need huge corpora, which are not easy or cheap to create.

Another problem is their creation itself. Often we are not able to align them correctly. For example, it often happens that we do not use a single word but a whole phrase to translate a word. Another problem is the different stylistic variation of the sentence.

Translation using a third language

Another way to translate from one language to another is to use a third language. This can be useful if we only have two translators available, which are able to translate from our language to a third language, and then from that language to the language we want. The problem with this method is that it has two weak points, which are the two translators. The overall translation may be worse than the quality of the weaker of the two. In the worst case scenario, both will make mistakes in different places. This method is only applicable if we have two quality translators available.

Another variation of this type of translation can be the use of an "interlanguage" (interlingua). The principle is almost identical to the use of a third language. The difference is the translation into a specific form of language, which does not have a particular communicative form, but only contains the meaning of the sentence and the semantics of the individual words. This type of translation has the advantage that after translating the text into an interlanguage, it can be translated into any language for which we have created rules.

Hybrid machine translation (HMT)

HMT is a combination of RBMT and SMT. HMT uses a translation memory, which makes it much more efficient in terms of quality. However, HMT also has its disadvantages, the biggest of which is the need for human editing.

Neural machine translation (NMT)

NMT (Neural Machine Translation) uses artificial intelligence to learn languages and continuously improve these skills. In this way, it tries to mimic the neural networks in the human brain. NMT is more accurate than other types of AI-based translation. With NMT, it is easier to add languages and translate content. As NMT provides better translations, it quickly becomes the standard in MT tool development.

NMT works by including training data. Depending on the user's needs, these data can be general or custom:

- General data: These are a collection of all the data learned from translations performed over time by machine translation (MT). These data create a general translation tool for various applications including text, speech, and documents.

- Custom or specialized data: These are trained data that are supplied to the machine translator in order to create specialization in a specific field. Subjects include engineering, design, programming, or any discipline with its own specialized dictionaries and lexicons.

Deep learning is just one of many popular methods for solving machine learning problems.

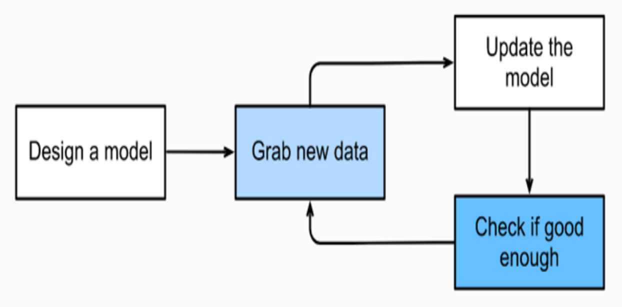
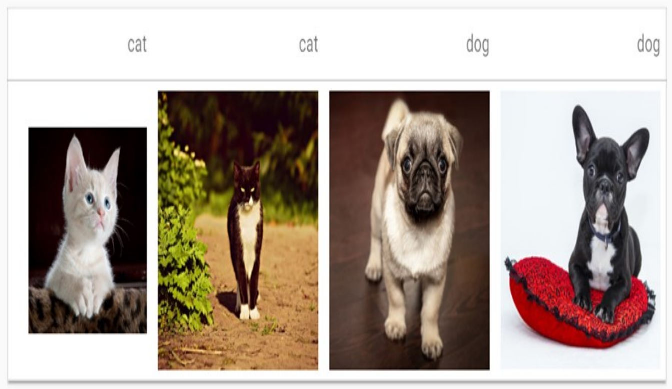


Fig. 1.1.2 A typical training process.



Acquiring linguistic data

Chapter 2

2.1 Acquiring linguistic data - intro

📖 2.1.1

Acquiring linguistic data

The first step is to extract text from its native form (in our case, pdf or html files) into text files.

Many websites are translated into multiple languages and some news organizations, such as the BBC, publish their articles for multilingual audiences. Extracting parallel texts from the web is sometimes straightforward, but often complicated for various reasons.

Working with a good dataset is crucial for ensuring good performance of a machine learning model, so adopting a good data extraction method can bring countless benefits for subsequent processes.

A parallel corpus is a set of texts aligned with translations into another language. Currently, there are several parallel corpora available. Some are available on the internet, many others are distributed by the Linguistic Data Consortium at the University of Pennsylvania, and there is also the Europarl corpus and the national corpus for the Slovak language (https://korpus.sk/).

📝 2.1.2

Does the performance of a machine translation model depend on the quality of the language corpus?

- Yes
- No

2.2 Acquiring linguistic data from the Internet

📖 2.2.1

Acquiring linguistic data from the Internet

Web scraping is a technique used to gather content and data from the internet. This data is usually stored in a local file to be manipulated and analyzed as needed. Web scraping applications (or "bots") are programmed to visit websites, capture relevant pages, and extract useful information. By automating this process, these robots can extract huge amounts of data in a very short amount of time. This has obvious advantages in the digital age, where big data - which is constantly updating and changing - plays a significant role.

Information that can be obtained from the web includes:

- pictures,
- videos,
- texts,
- product information,
- reviews (eg. TripAdvisor),
- etc.

For the purpose of creating a corpus, text data is the most interesting.

📖 2.2.2

How web scraper works

Although the specific method may vary depending on the software or tool used, all web scrapers follow three basic principles:

1. sending an HTTP request to the server,
2. extraction and parsing (or translation) of the website's code,
3. saving relevant information to local storage.

HTTP request

When you visit a website as an individual through a browser, you send a so-called HTTP request. This is essentially the digital equivalent of knocking on the door and requesting entry. After your request is approved, you can access the webpage and all the information on it. Like a person, a web scraper needs permission to access a webpage. Therefore, the first thing a web scraper does is send an HTTP request to the website it is targeting.

Extracting and parsing web site code

Once the web page grants access to the scraper, the bot can read and extract the HTML or XML code of the page. This code determines the structure of the content of the web page. The scraper then parses the code (which basically means dividing it into individual parts) to identify and extract elements or objects that were predefined by the person who launched the bot. They can include specific text, ratings, classes, tags, identifiers, or other information.

Saving relevant information to local storage

After accessing, retrieving, and analyzing the HTML or XML, the relevant data is stored locally in the web scraper. As mentioned, the extracted data is predefined - the bot is instructed on what to gather. The data is usually stored as structured data, often in an excel file, such as .csv or .xls format. This process is not performed just once, but countless times. This comes with its own set of problems that need to be addressed. For example, poorly coded scrapers can send too many HTTP requests, which can overwhelm the website. Each website also has different rules for what robots can and cannot do. Running the code for web scraping is just one part of the process.

📖 2.2.3

Web scraping tools

Knowledge of a programming language is necessary for web data mining. Currently, the most commonly used language is Python, which has several libraries implemented for web scraping.

Beautiful Soup

Beautiful Soup is a Python library for data mining from HTML and XML files. It uses a parser and provides idiomatic ways of navigating, searching, and modifying the parsed tree. It typically saves programmers hours or days of work.

Link: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Scrapy

Scrapy is an application framework based on Python that crawls and extracts structured data from the web. It is commonly used for deep data analysis, information processing, and historical content archiving. In addition to web scraping (for which it was specifically designed), it can also be used as a universal web indexing search engine or for data extraction through APIs.

Link: https://scrapy.org/

Pandas

Pandas is another versatile Python library used for data manipulation and indexing. It can be used for web scraping in conjunction with BeautifulSoup. The main advantage of using pandas is that analysts can perform the entire data analysis process using one language (and avoid the need to switch to other languages, such as R).

Link: https://pandas.pydata.org/

2.3 Acquiring linguistic data from the PDF documents

📖 2.3.1

Extracting text data from PDF documents

Extracting text from PDF documents is a challenging process as it is a "layout-based" format that does not contain semantic information about the parts of the text.

PDF is one of the most popular formats for electronic documents. Currently, the Google search engine indexes more than 3 billion PDF documents, which is more than any other document format except for HTML. However, PDF is a "layout-based" format, which means that it specifies the position and font of each character on each page that makes up the text. Page description languages (PDLs) are used as a communication between the formatted page or document description in a composition system and the output device, such as a monitor or printer. The advantage of a document written in Page Description Language (PDL) format is full control over the appearance. The disadvantage of these documents is the file size and lack of logical structure compared to the markup language HTML.

Currently, data-driven techniques dominate natural language processing. The significant progress in several natural language processing fields can be seen due to advances in machine learning and increasing data sets. However, the availability of clean data sets is still a persistent problem for most languages in the world. Data is commonly obtained by mining the World Wide Web to expand data sets. With the growing demand for increasingly large data sets, it is necessary to look for other sources of text. The PDF format essentially represents paper in digital form and naturally contains a large amount of text. This format is optimized for printing on paper and therefore focuses mainly on the visual side of documents. The format is not structured and does not distinguish the semantic meaning of objects on the page. This means that when extracting information, no distinction is made between headings, paragraphs, text in tables, or image captions.

The PDF Association, which defines the ISO standard for the PDF format, is currently focused on making PDF documents accessible. The goal is to provide people with disabilities (such as visual impairments) with full access to the information in the document. The foundation is the ability to add logical structure to the format, which refers to specific parts of the page in the document and adds semantic meaning to them. Like in HTML, the semantic meaning of content in PDF documents is represented by tags. The tagged PDF document subsequently allows for precise extraction of paragraphs, annotations, headings, as well as the correct reading order for multi-column documents. Adding logical structure to the PDF format is not mandatory, so there are few documents with this structure. Some programs, such as Apple's office suite, automatically add tags when exporting to PDF.

📖 2.3.2

Problems with extracting text from PDF documents

Since PDF displays text based on font and position on a given page for individual characters, text extraction tools face several problems. Here are some of them:

- Word identification
- Word order
- Paragraph boundries
- Ligatures
- Word boundries
- Semantical meaning

📖 2.3.3

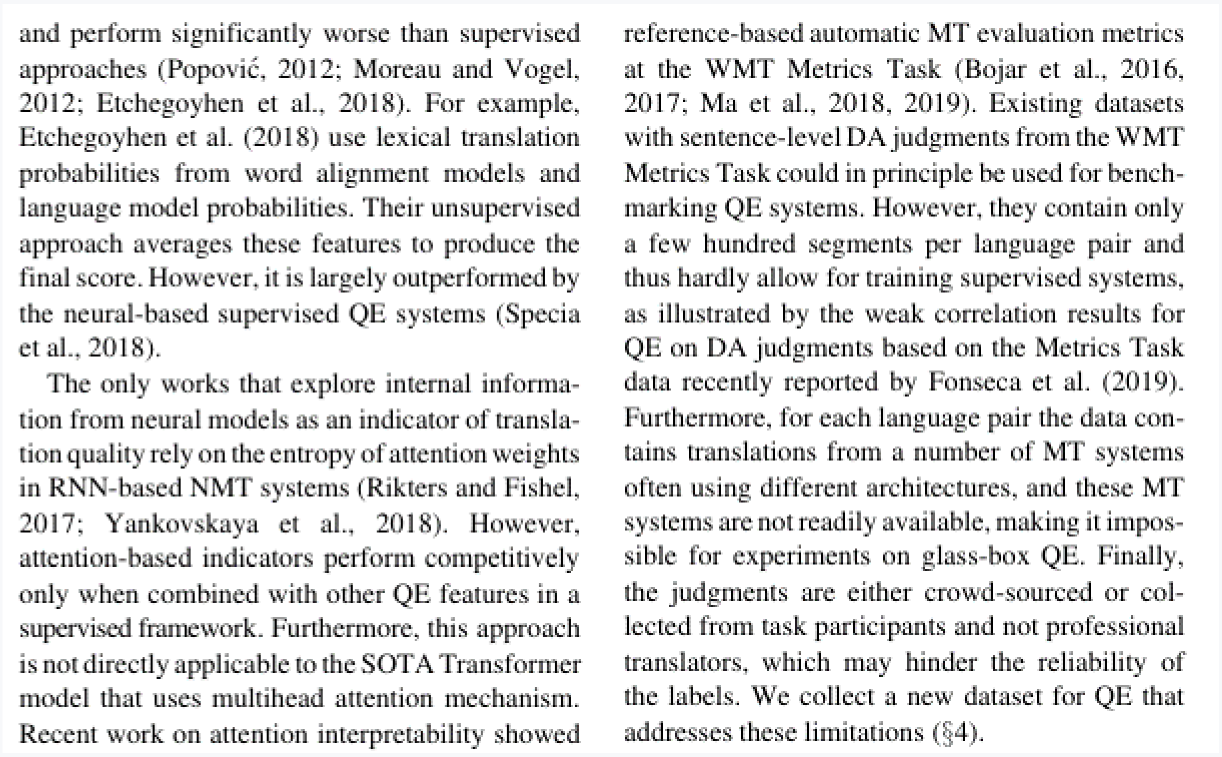
Word identification

Important for searching for words in the document. If a word is not correctly identified, the searched word may not be found. The problem with identifying words is that the spaces between letters can vary not only between lines but also within a single line. Long words can be separated at the end of a line by a hyphen, visually splitting one word into two and appearing twice in the document.

📖 2.3.4

Word order

Determining the order of words is essential for applications such as e-book readers or when we want to obtain unformatted text. The order of words can be determined based on their position in the PDF. However, in documents with multi-column layouts, the order cannot be determined based on position alone.



and perform significantly worse than supervised approaches (Popović, 2012; Moreau and Vogel, 2012; Etchegoyhen et al., 2018). For example, Etchegoyhen et al. (2018) use lexical translation probabilities from word alignment models and language model probabilities. Their unsupervised approach averages these features to produce the final score. However, it is largely outperformed by the neural-based supervised QE systems (Specia et al., 2018).

The only works that explore internal information from neural models as an indicator of translation quality rely on the entropy of attention weights in RNN-based NMT systems (Rikters and Fishel, 2017; Yankovskaya et al., 2018). However, attention-based indicators perform competitively only when combined with other QE features in a supervised framework. Furthermore, this approach is not directly applicable to the SOTA Transformer model that uses multihead attention mechanism. Recent work on attention interpretability showed

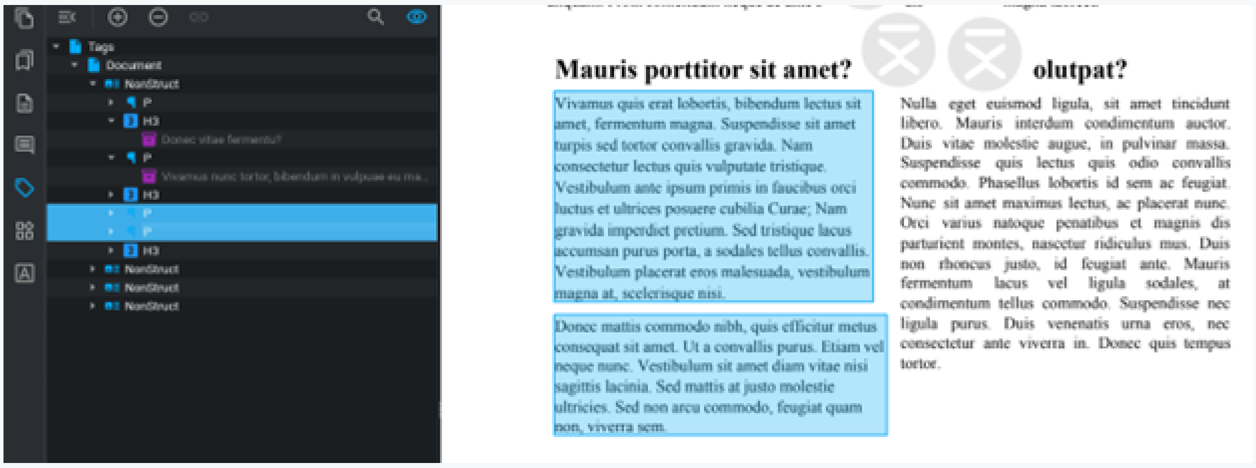
reference-based automatic MT evaluation metrics at the WMT Metrics Task (Bojar et al., 2016, 2017; Ma et al., 2018, 2019). Existing datasets with sentence-level DA judgments from the WMT Metrics Task could in principle be used for benchmarking QE systems. However, they contain only a few hundred segments per language pair and thus hardly allow for training supervised systems, as illustrated by the weak correlation results for QE on DA judgments based on the Metrics Task data recently reported by Fonseca et al. (2019). Furthermore, for each language pair the data contains translations from a number of MT systems often using different architectures, and these MT systems are not readily available, making it impossible for experiments on glass-box QE. Finally, the judgments are either crowd-sourced or collected from task participants and not professional translators, which may hinder the reliability of the labels. We collect a new dataset for QE that addresses these limitations (§4).

The sample shows a part of the text from a scientific article that is written in two columns. Some extraction tools cannot recognize such a layout and the result is text extracted by rows.

2.3.5

Paragraph boundries

Determining the beginning and end of a paragraph is again important for applications that read documents. This problem is caused, for example, if an image or table is inserted in the paragraph. The application may incorrectly evaluate the end of the paragraph even if it encounters the end of the page. Despite the fact that the paragraph continues on the next page, the application evaluates it as two different paragraphs.



Mauris porttitor sit amet?

Vivamus quis erat lobortis, bibendum lectus sit amet, fermentum magna. Suspendisse sit amet turpis sed tortor convallis gravida. Nam consectetur lectus quis vulputate tristique. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nam gravida imperdiet pretium. Sed tristique lacus accumsan purus porta, a sodales tellus convallis. Vestibulum placerat eros malesuada, vestibulum magna at, scelerisque nisi.

Donec mattis commodo nibh, quis efficitur metus consequat sit amet. Ut a convallis purus. Etiam vel neque nunc. Vestibulum sit amet diam vitae nisi sagittis lacinia. Sed mattis at justo molestie ultricies. Sed non arcu commodo, feugiat quam non, viverra sem.

olutpat?

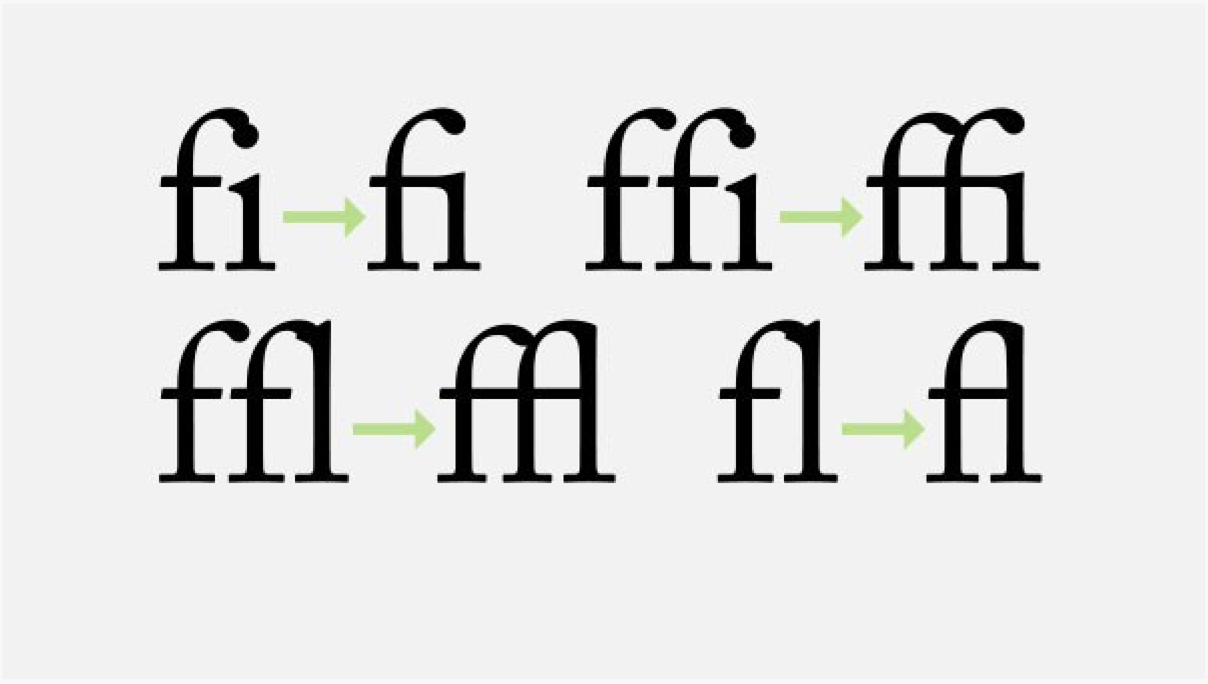
Nulla eget euismod ligula, sit amet tincidunt libero. Mauris interdum condimentum auctor. Duis vitae molestie augue, in pulvinar massa. Suspendisse quis lectus quis odio convallis commodo. Phasellus lobortis id sem ac feugiat. Nunc sit amet maximus lectus, ac placerat nunc. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Duis non rhoncus justo, id feugiat ante. Mauris fermentum lacus vel ligula sodales, at condimentum tellus commodo. Suspendisse nec ligula purus. Duis venenatis urna eros, nec consectetur ante viverra in. Donec quis tempus tortor.

The image shows an example of a tagged PDF that adds information to the document such as paragraph boundaries or reading order. In this case, extracting text is unambiguous in terms of paragraphs, but it also solves the problem of word order.

🕮 2.3.6

Ligatures

Another problem is character recognition. For example, the "fi" character (Unicode U+FB01) represents only one character. When extracting the word "fialka", the first two letters "f" and "i" may be incorrectly identified and will not appear in the output at all.



2.3.7

Word boundries

Hyphenation of words poses a problem in cases where we want to reuse the extracted text, for example, in natural language processing. Most tools use heuristic rules to distinguish the hyphen in terms of word splitting.

[illegible] program and get errors indi-
cating that a linker [illegible]

The highlighted word "indicating" in the image is divided by a hyphen. When extracting text from a PDF, we get the result exactly in this form. Such data is not suitable for purposeful training of artificial intelligence because the words "indi" and "cating" do not exist.

2.3.8

Semantical meaning

As PDF documents are not structured formats, they do not distinguish whether the text is a paragraph, a heading, a footnote, or a page number. PDF addresses this deficiency precisely by allowing the structure of the document to be supplemented and creating tagged PDF documents.

2.3.9

PDF extraction tools

Pdftotext

One of the most commonly used tools for extracting data from PDFs. It converts PDFs to plain text but does not identify paragraphs or the semantic meaning of text.

Program: https://pypi.org/project/pdftotext/

Pdfix

This is an SDK for reading, writing, rendering, and manipulating PDF files. Thanks to advanced algorithms, it can recognize the logical structure of text, headings, images, tables, or headers and footers of the document. All this data is then available in formats such as HTML, CSV, JSON, or XML.

Program: https://pdfix.net/download/

PdfMiner

Tool that can analyze the structure of a PDF file and convert it into textual, XML or HTML output. The text is divided into paragraphs, lines, and characters.

Program: https://pypi.org/project/pdfminer/

Pdfbox

A library from Apache, written in Java. Like the previous tool, it converts PDF to text format without distinguishing paragraphs or semantic meaning of texts.

Program: https://pdfbox.apache.org/download.html

Pdfforge

This is a freely available online tool whose output is a text file.

Program: https://download.pdfforge.org/

2.4 Task - data extraction

📖 2.4.1

Based on the information obtained in the previous chapters, create a parallel corpus consisting of at least 1000 sentences (Slovak and English translations).

Remove noisy data from the data - headings, page numbers, footnotes, image descriptions, graphs, and tables.

The goal is to obtain only Slovak and English sentences, which will be subsequently aligned and preprocessed, resulting in a parallel corpus.

2.5 Text preprocessing

📖 2.5.1

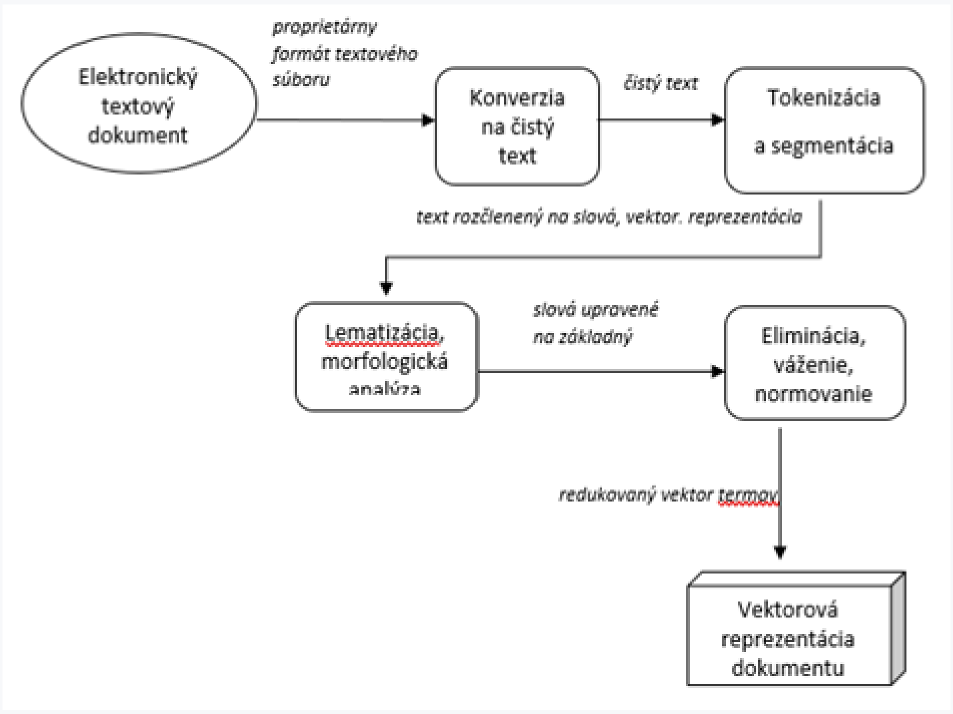
The main goal of text preprocessing is to divide the text into a form that machine learning algorithms can process. Text data obtained from natural language is unstructured and contains noisy data.

Text preprocessing involves transforming the text into a clean and consistent format that can then be inserted into a model for further analysis and learning.

Preprocessing of text data is generally referred to as obtaining feature descriptions for all text documents in the examined corpus.

Electronic text documents in various formats are assumed as input. The first step is to remove redundant formatting characters and convert the document to so-called "plain text." This text is then divided into elementary text units, called tokens. Words, or lexical units, are then identified in the text, for which the corresponding base form (lemma) and morphological categories are determined. Finally, non-semantic words, those that are presumed to contribute little to the overall context of the document, are removed. The remaining semantic words, weighted by a suitable weight function, then form the desired vector representation of the input document.

Stages of text data preprocessing:


Elektronický textový dokument
proprietárny formát textového súboru
Konverzia na čistý text
čistý text
Tokenizácia a segmentácia
text rozčlenený na slová, vektor. reprezentácia
Lematizácia, morfologická analýza
slová upravené na základný
Eliminácia, váženie, normovanie
redukovaný vektor termov
Vektorová reprezentácia dokumentu

2.6 Corpus alignment

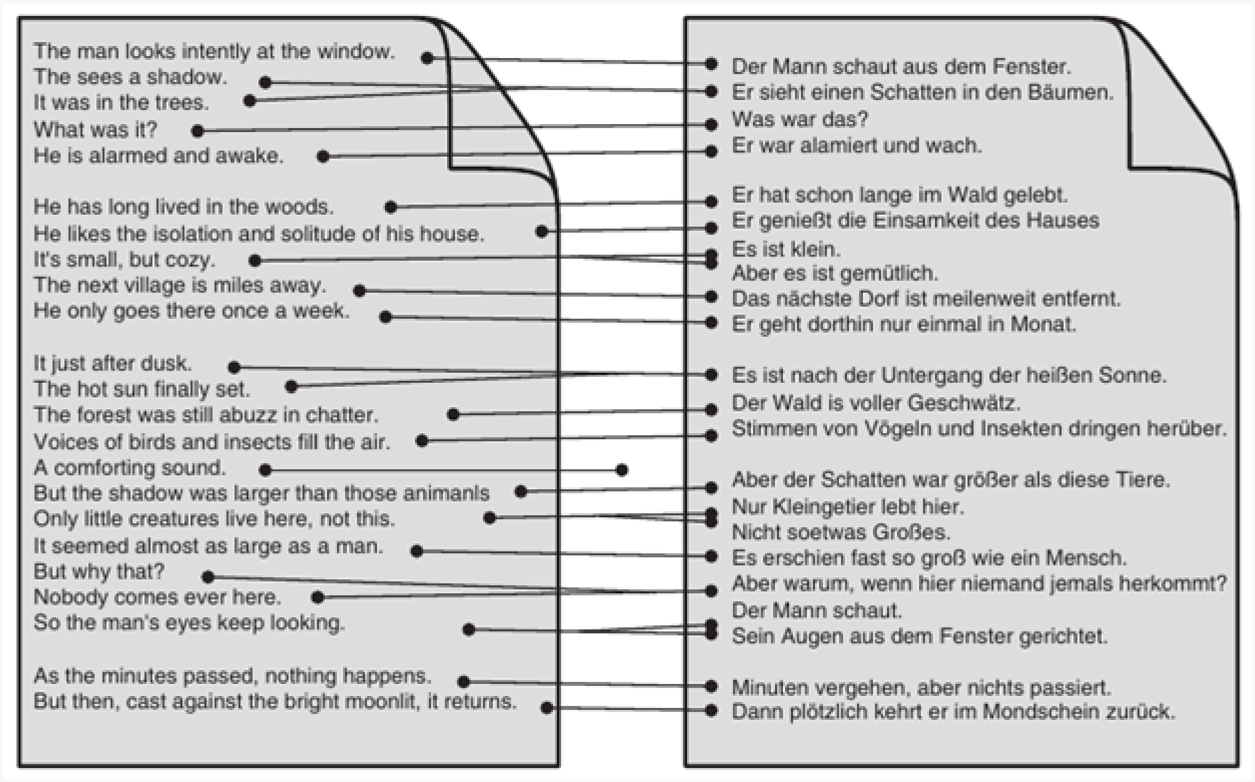
2.6.1

Alignment

Sentence alignment is a task where corresponding documents (original and its corresponding translation) are divided into sentences, and a bipartite graph is searched for that corresponds to the minimum groups of sentences that are mutual translations.

Bitext (two corresponding texts - original and translation - aligned by sentence) is used to train almost all machine translation (MT) systems. It has been found that errors in alignment have little impact on the performance of statistical MT (SMT), but it has been shown that incorrectly aligned sentences have a much greater impact on the performance of neural MT (NMT).

Rarely is text translated word for word and not always is a sentence translated sentence by sentence. Long sentences may be split or short sentences may be merged. Good alignment is also crucial for lexicography as it can be used to display parallel correspondences and to find translation equivalents, as well as to extract terminology. Aligning parallel corpora is also used in digital humanities (DH) for various purposes, such as learning historical languages or aligning versions of medieval texts.


The man looks intently at the window.
The sees a shadow.
It was in the trees.
What was it?
He is alarmed and awake.
He has long lived in the woods.
He likes the isolation and solitude of his house.
It's small, but cozy.
The next village is miles away.
He only goes there once a week.
It just after dusk.
The hot sun finally set.
The forest was still abuzz in chatter.
Voices of birds and insects fill the air.
A comforting sound.
But the shadow was larger than those animanls
Only little creatures live here, not this.
It seemed almost as large as a man.
But why that?
Nobody comes ever here.
So the man's eyes keep looking.
As the minutes passed, nothing happens.
But then, cast against the bright moonlit, it returns.
Der Mann schaut aus dem Fenster.
Er sieht einen Schatten in den Bäumen.
Was war das?
Er war alamiert und wach.
Er hat schon lange im Wald gelebt.
Er genießt die Einsamkeit des Hauses
Es ist klein.
Aber es ist gemütlich.
Das nächste Dorf ist meilenweit entfernt.
Er geht dorthin nur einmal in Monat.
Es ist nach der Untergang der heißen Sonne.
Der Wald is voller Geschwätz.
Stimmen von Vögeln und Insekten dringen herüber.
Aber der Schatten war größer als diese Tiere.
Nur Kleingetier lebt hier.
Nicht soetwas Großes.
Es erschien fast so groß wie ein Mensch.
Aber warum, wenn hier niemand jemals herkommt?
Der Mann schaut.
Sein Augen aus dem Fenster gerichtet.
Minuten vergehen, aber nichts passiert.
Dann plötzlich kehrt er im Mondschein zurück.

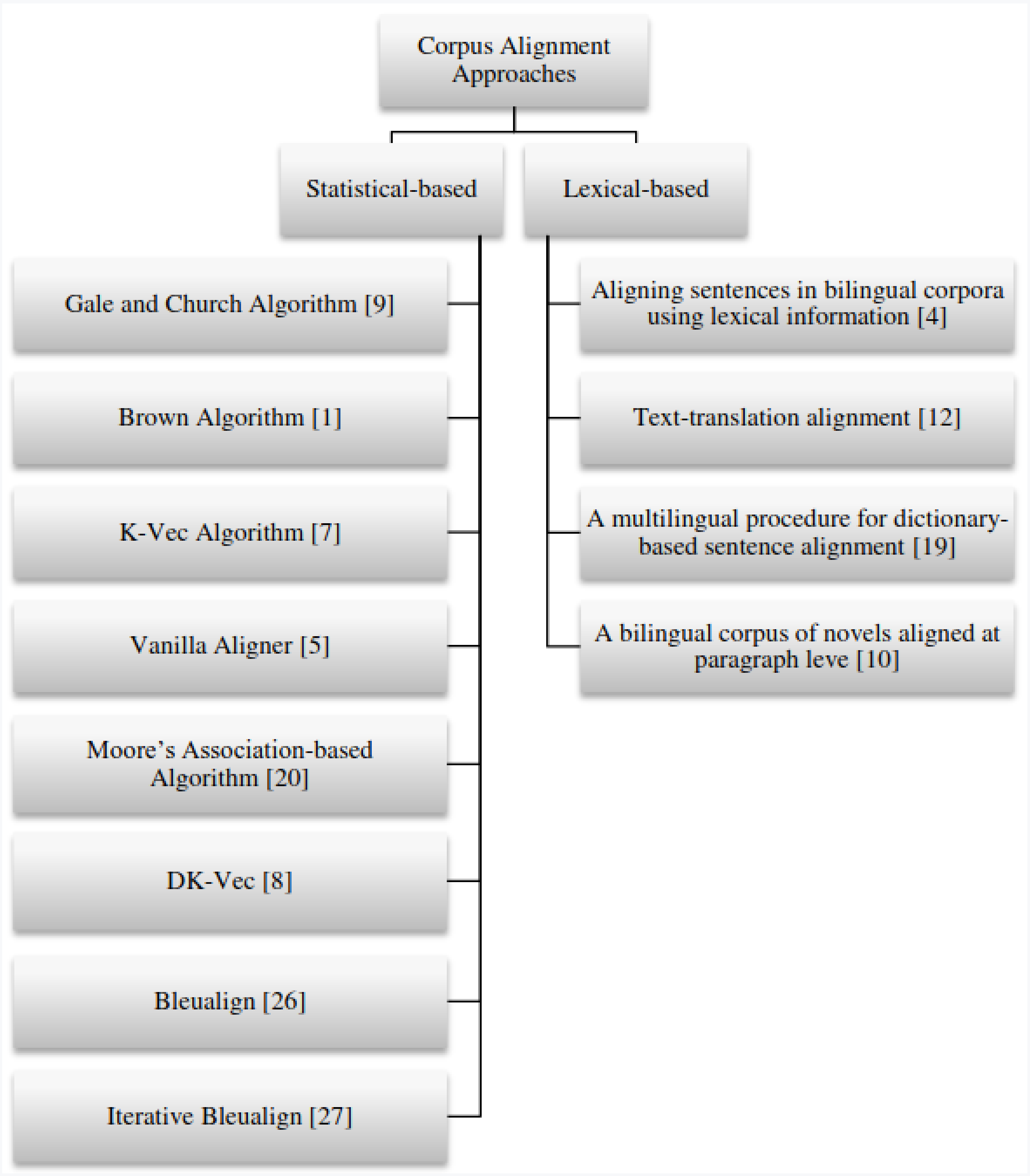
There are two main approaches to text alignment:

1. Statistical approach,
2. The approach applies linguistic-lexical knowledge.

Based on these approaches, several techniques have been developed, each with its own advantages and disadvantages.

Approaches based on lexical properties of the text rely on existing lexical resources, such as extensive bilingual dictionaries and glossaries. These techniques are slower than statistical techniques and are language-dependent. The main disadvantage of these techniques is that their performance heavily relies on the lexical information used in the alignment process. However, many of these methods and techniques are still being developed, as they are expected to generate better results than statistical methods.

Approaches based on statistics rely on extralinguistic quantitative characteristics, such as sentence length, sentence position, co-occurrence frequency, length ratio of sentences in two languages, etc. These techniques speed up the alignment process and are generally language-independent. However, the main disadvantage of these techniques is that their performance heavily relies on the structural similarity between the target and source text (bitext).



Corpus Alignment Approaches
Statistical-based
Lexical-based
Gale and Church Algorithm [9]
Brown Algorithm [1]
K-Vec Algorithm [7]
Vanilla Aligner [5]
Moore's Association-based Algorithm [20]
DK-Vec [8]
Bleualign [26]
Iterative Bleualign [27]
Aligning sentences in bilingual corpora using lexical information [4]
Text-translation alignment [12]
A multilingual procedure for dictionary-based sentence alignment [19]
A bilingual corpus of novels aligned at paragraph leve [10]

📖 2.6.2

Gale a Church algorithm

This is one of the first algorithms for automatic text alignment. Although it is not one of the best, it is language-independent.

The algorithm takes advantage of the fact that longer sentences in one language tend to be translated into longer sentences in the other language and vice versa. Shorter sentences tend to be translated into shorter sentences. Each proposed pair of sentences is assigned a probability score based on the ratio of the lengths of the two sentences (in characters) and the variance of this ratio. This probability score is used in dynamic programming to find the most likely alignment of sentences.

Source: https://aclanthology.org/J93-1004.pdf

Task: Implement Gale and Church alg. based on the given source.

📖 2.6.3

Moore algorithm

This is an algorithm that combines techniques adapted from the previous algorithm for aligning sentences and words in a three-step process. First, the corpus is aligned using a modified version of the Brown model based on sentence length. Then, pruning is used to efficiently find pairs of sentences that will align with the highest probability without using anchor points or larger previously aligned units. Next, the pair of sentences that was assigned the highest matching probability is used to train a modified version of the IBM Model 1 translation model. Finally, the corpus is realigned, with the original alignment model extended with IBM Model 1 to create an alignment based on sentence length and word correspondence. The final search is limited only to minimal alignment segments that were assigned a non-negligible probability according to the original alignment model, reducing the search space to the point where this alignment is actually faster than the original alignment, although the model is much more demanding to apply for each segment.

Source: https://link.springer.com/chapter/10.1007/3-540-45820-4_14

Program: https://www.microsoft.com/en-us/download/details.aspx?id=52608

Command: ./align-sents-all.pl <source_text> <target_text>

📖 2.6.4

Hunalign

Hunalign aligns bilingual text at the sentence level. Its input is tokenized and segmented text in two languages (the original and its translation). In the simplest case, its output is a sequence of bilingual sentence pairs (bitext). In the case of a dictionary, Hunalign combines it with information about sentence length according to the Gale-Church algorithm. If the dictionary does not exist, it first returns to information about sentence length and then creates an automatic dictionary based on this alignment. Then, in the second pass through the text, it aligns the text using the automatic dictionary.

Source: http://mokk.bme.hu/resources/hunalign/

Dictionary source: https://github.com/coezbek/hunalign-dict-muse/tree/main/dict[1]

Program: https://github.com/danielvarga/hunalign

📖 2.6.5

BleuAlign

The basic idea of Bleualigner is to use the output of machine translation (MT) and the score of the automatic BLEU metric, which characterizes the similarity of the MT output to the reference translation, to find reliable alignments that are used as anchor points. Gaps between these anchor points are then filled using a heuristic based on the BLEU metric and length. The main alignment algorithm is computed for each text segment between two hard delimiters (including the beginning and end of the file) and consists of two steps. First, a set of anchor points is identified using the BLEU metric. In the second step, sentences between these anchor points will be aligned either using a heuristic based on the BLEU metric or an algorithm based on length according to Gale and Church.

Source: https://aclanthology.org/2010.amta-papers.14/

Program: https://github.com/rsennrich/Bleualign

Command: ./bleualign.py -s sourcetext.txt -t targettext.txt --srctotarget sourcetranslation.txt -o outputfile

📖 2.6.6

Vecalign

Vecalign is a precise sentence alignment algorithm that is fast even for very long documents. When used in conjunction with the LASER system, Vecalign works in approximately 100 languages (i.e. 100^2 language pairs) without the need for a machine translation or lexicon (dictionary) system.

Vecalign uses the similarity of multilingual embeddings and an approximation of dynamic programming based on fast dynamic time warping, which is linear in time and space with respect to the number of aligned sentences, to assess the similarity of sentences.

Source: https://aclanthology.org/D19-1136/

Source LASER: https://github.com/facebookresearch/LASER

Program: https://github.com/thompsonb/vecalign

Command: ./vecalign.py --alignment_max_size 8 --src bleualign_data/dev.de --tgt bleualign_data/dev.fr --src_embed bleualign_data/overlaps.de bleualign_data/overlaps.de.emb –tgt_embed bleualign_data/overlaps.fr bleualign_data/overlaps.fr.emb

2.7 Task - sentence alignment

📖 2.7.1

To create a parallel corpus from the extracted data obtained in the previous task, the task consists of two steps:

1. The first step is the segmentation of documents into sentences. All alignment tools require input in the form of text files, where each line corresponds to one sentence.
2. To create a parallel corpus, the text data needs to be aligned. Choose one of the tools mentioned in this chapter and align the text data.

The result should be two text files with the same number of lines.

Language models

Chapter 3

3.1 Language models

📖 3.1.1

Language models

Language modeling is the task of calculating the probability of a sequence of words. Language models are crucial for many different applications, such as speech recognition, optical character recognition, machine translation, and spell checking. There are many words, word combinations, and phrases that are almost identical in pronunciation but have completely different meanings. A good language model can distinguish which phrase is most likely correct based on context. In this chapter, we focus on an overview of word- and character-level language models and their creation using a Recurrent Neural Network (RNN).

📖 3.1.2

N-grams

A language model can calculate the probability that a given word will follow a sequence of previous words. Determining the probability of a long sequence of words w (w1, ..., wm) is usually infeasible.

The calculation of the continuous probability P(w1, ..., wm) would be performed using the following chain rule:

$$P(w_1, \ldots, w_m) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_m|w_1, \ldots, w_{m-1})$$

We will only model the joint probabilities of combinations of n consecutive words, known as n-grams. For example, in the sentence "This food is good," we have the following n-grams:

- **1-gram (unigram)**: “This”, “food”, “is” and “good”
- **2-grams (bigram)**: “This food”, “food is" and “is good”
- **3-grams (trigram)**: “This food is” a “food is good”
- **4-grams**: “This food is good”

If we have a large corpus, we can find all n-grams up to a certain n (usually 2 to 4) and count the occurrence of each n-gram in this corpus. Based on these counts, we can estimate the probability of the last word of each n-gram given the preceding n-1 words:

- **1-gram:** $P(word) = \frac{count(word)}{\text{total number of words in corpus}}$
- **2-gram:** $P(w_i|w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$
- **N-gram:** $P(w_{n+i}|w_n, \ldots, w_{n+i-1}) = \frac{count(w_n, \ldots, w_{n+i-1}, w_{n+i})}{count(w_n, \ldots, w_{n+i-1})}$

Neural network in machine translation

Chapter 4

4.1 Neural networks

📖 4.1.1

Neural networks

We can describe a neural network as a mathematical model of information processing. A neural network is not a fixed program, but rather a model (system) that processes information (inputs).

The characteristics of a neural network are as follows:

- Information processing occurs in its simplest form, over simple elements called neurons.
- Neurons are interconnected and exchange signals with each other through connecting lines. These connections can be stronger or weaker, which determines the way information is processed.
- Each neuron has an internal state that is determined by all incoming connections from other neurons.
- Each neuron has a different activation function, which is computed based on its state and determines its output signal.

The two main characteristics of a neural network are:

- Architecture of the neural network, which describes the set of connections between neurons (feedforward, recurrent, multilayer or single-layer, etc.), as well as the number of layers and neurons in each layer.
- Learning - training. The most common and typical way of training a neural network is through gradient descent and backpropagation, although it is not the only way.

📝 4.1.2

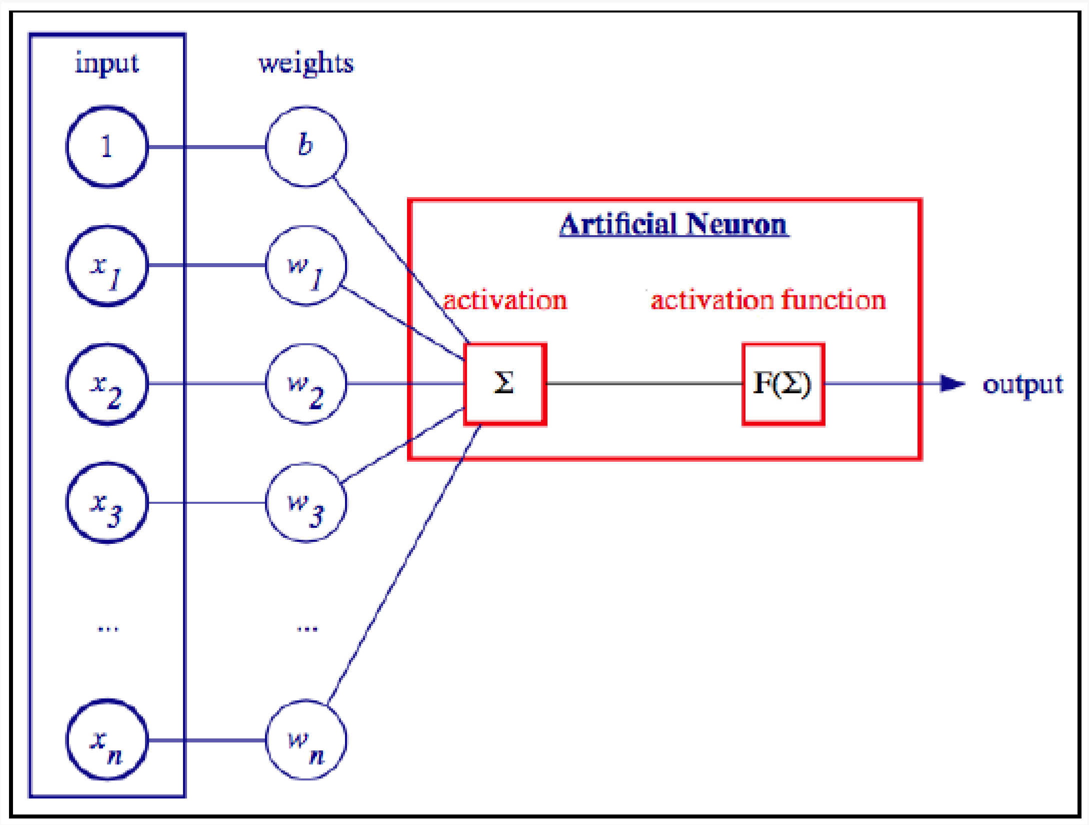
Is it true that each neuron has a different activation function that is computed based on its state and determines its output signal?

- Yes
- No

📖 4.1.3

Neuron

A neuron is a mathematical function that takes one or more input values and produces a single numerical output value:



To compute a neuron, we follow these steps:

$$y = f(\sum_i x_i w_i + b)$$

where xi is a numerical value representing input data or outputs from other neurons if the neuron is part of a neural network. The weights wi are numerical values representing either the strength of the inputs or the strength of the connections between individual neurons. The weight b is a special value called the bias, whose input is always 1.

- First, we calculate the sum of inputs xi and weights wi (called the activation value).
- Then, we use the result of the weighted sum as input to the activation function f, which is also known as the transfer function. There are many types of activation functions, but all must satisfy the requirement of nonlinearity.

The activation value can be interpreted as the dot product of two vectors w and x:

y = Prepojený obrázok sa nedá zobraziť. Súbor bol pravdepodobné presunutý, premenovaný alebo odstránený. Skontrolujte, či prepojenie smeruje k správnemu súboru a umiestneniu..

Vector x will be perpendicular to the weight vector w, where Prepojený obrázok sa nedá zobraziť. Súbor bol pravdepodobné presunutý, premenovaný alebo odst... *= 0*.

The perceptron (i.e., neuron) works only with linearly separable classes due to the definition of a hyperplane. For this reason, neurons are organized into a neural network.

Perceptron (teda neurón) pracuje len s lineárne oddeliteľnými triedami z dôvodu definovania hyperrovinu. Z tohto dôvodu sa neuróny usporiadávajú do neurónovej siete.

4.1.4

Layers in a neural network

A neural network can have an unlimited number of neurons arranged into interconnected layers. The input layer represents a set of data with initial conditions (values).

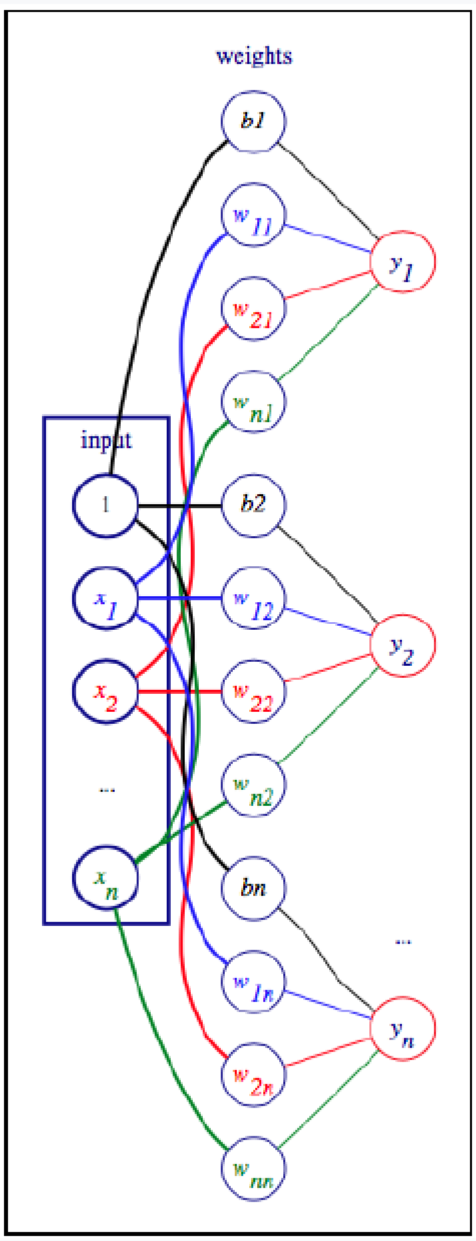
For example, if the input is a grayscale image, each neuron's output in the input layer represents the intensity of one pixel in the image.

For this reason, we generally do not count the input layer as part of the other layers. When we talk about a single-layer network, we mean that it is a simple network that has only one output layer in addition to the input layer.

The output layer can have more than one neuron. This is especially useful in classification, where each output neuron represents one class.

For example, in the case of the MNIST database (a large database of handwritten digits), we will have 10 output neurons, with each neuron corresponding to a digit from 0 to 9. In this way, we can use a single-layer network for each image to classify the digit. We determine the digit by taking the output neuron with the highest activation function value. If it is y7, we will know that the network thinks the image shows the number 7.

In the case of a single-layer feedforward network, the weights w for each connection between neurons are shown explicitly, but usually the edges connecting neurons represent weights implicitly. The weight wij connects the i-th input neuron to the j-th output neuron. The first input 1 is the bias unit, and the weight b1 is the bias weight.



The neurons on the left represent the input with bias b, the middle column represents the weights for each connection, and the neurons on the right represent the output with respect to the weights w (fig. ...).

Neurons in one layer can be connected to neurons in other layers, but not to neurons in the same layer. In this case, the input neurons x1,...,xn are only connected to output neurons y1,...,yn.

The reason for arranging neurons into a network is that a neuron can mediate only limited information (only one value). However, when we connect neurons into layers, their outputs form a vector, and instead of one activation, we can now consider the whole vector. In this way, we can mediate much more information, not only because the vector has multiple values, but also because the relative ratios between them carry additional information.

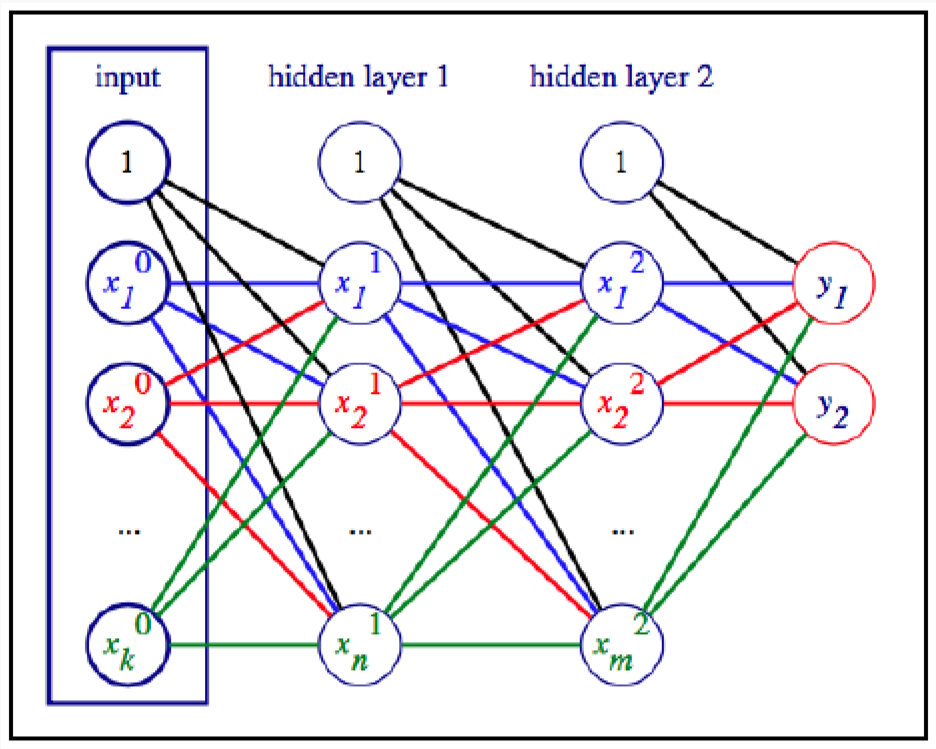
4.1.5

How many input neurons are needed for a network that recognizes digits 0-9?

4.1.6

Multi-layer neural networks

Single-layer neural networks can only classify linearly separable classes. However, nothing stops us from introducing additional layers between the input and output. These additional layers are called hidden layers. A three-layer fully connected neural network with two hidden layers is shown in the picture. The input has n input neurons, the first hidden layer has n hidden neurons, and the second hidden layer has m hidden neurons. In this case, there are two classes, y1 and y2, as the output. At the top, there is always a bias neuron turned on. A unit from one layer is connected to all units from the previous and next layer (thus fully connected). Each connection has its own weight w, which is not shown for simplicity. In this case, it is a neural network with sequential layers.



A neural network as a composition of neurons is a mathematical function, where input data represent the function arguments and network weights w represent its parameters.

📖 4.1.7

Activation functions

To turn the network into a non-linear function, we use non-linear activation functions for neurons. Usually, all neurons in the same layer have the same activation function, but different layers can have different activation functions. The most common activation functions are:

- **identity function - this function passes the activation value a.**

$$f(a) = a$$

- **threshold activity function - this function activates the neuron. If the activation is higher than a certain value a, then it is a threshold activity function.**

$$f(a) = \begin{cases} 1 \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$$

- **logistic function or logistic sigmoid - this function is one of the most commonly used, as its output is in the range between 0 and 1 and can be stochastically interpreted as the probability of neuron activation.**

$$f(a) = \frac{1}{1+exp(-a)}$$

- **bipolar sigmoid - it is basically a rescaled and shifted logistic sigmoid to have a range of (-1, 1).**

$$f(a) = \frac{2}{1+exp(-a)} - 1 = \frac{1-exp(-a)}{1+exp(-a)}$$

- **hyperbolic tangent**

$$f(a) = \frac{exp(a)-exp(-a)}{exp(a)+exp(-a)} = \frac{1-exp(-2a)}{1+exp(-2a)}$$

- **rectifier or ReLU (Rectified Linear Unit) - this activation function is probably the closest to its biological counterpart. It is a combination of the identity function and the threshold activity function. There are various variants of the ReLU activation function, such as Noisy ReLU, Leaky ReLU, or ELU (Exponential Linear Unit).**

$$f(a) = \begin{cases} a \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$$

Identity or threshold activity function were used in the creation of neural networks with implementations such as the perceptron or Adaline (adaptive linear neuron), but later on, they were not as popular as the logistic sigmoid, hyperbolic tangent, ReLU, and its variations.

📖 4.1.8

Recurrent Neural Networks

Recurrent Neural Networks (RNN) work with sequences of variable length and define a recurrent relationship over these sequences:

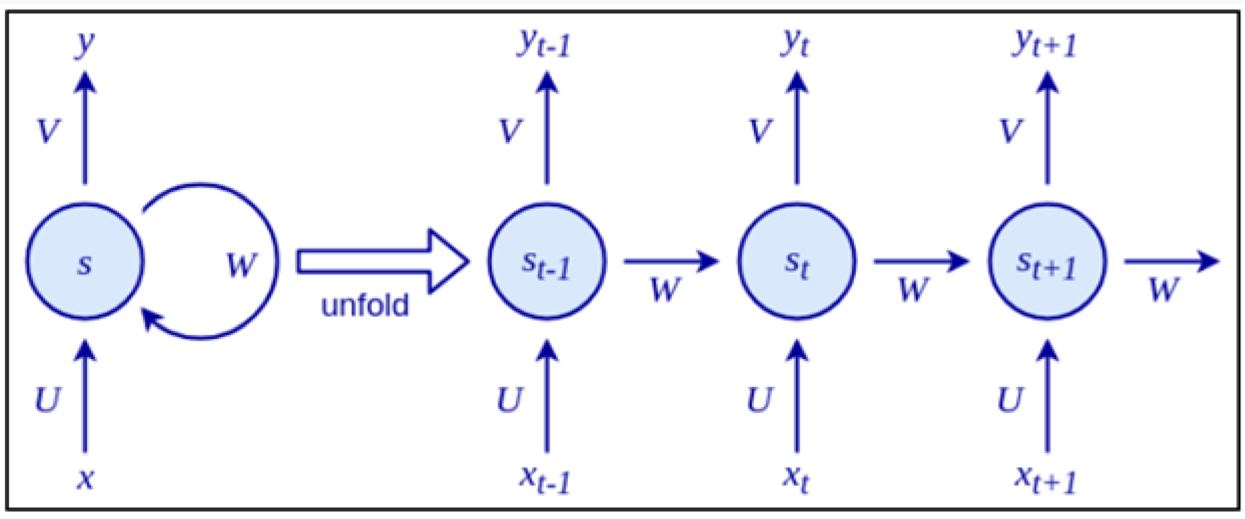
$$s_t = f(s_{t-1}, x_t)$$

where f is a differentiable function, st is a vector of values called the internal state of the network (at time step t), and xt is the input of the network at time step t. Unlike regular networks, where the state depends only on the current input (and the network weights), here st is a function of both the current input as well as the previous state st-1. We can think of the state st-1 as a summary of all previous inputs of the network.

Thanks to their ability to process arbitrary input sequences, recurrent NNs are used in natural language processing (NLP) and speech recognition tasks.

Examples of variable length data include words in a sentence or stock prices at different time intervals. Recurrent NNs are named after their ability to repeat the same function over a sequence.

The recurrence relationship is defined by how the state evolves step by step through the sequence via feedback through previous states, as shown in the following diagram:



RNN has three sets of parameters (or weights):

- *U* transforms input xt into state st
- *W* transforms previous state st-1 into current state st
- *V* maps (transforms) the newly computed internal state st into output y

The parameters U, V, and W apply a linear transformation to their respective inputs. The most basic case of transformation is a weighted sum. Now we can define the internal state and output of the network as follows:

$$s_t = f(s_{t-1} * W + x_t * U)$$

$$y_t = s_t * V$$

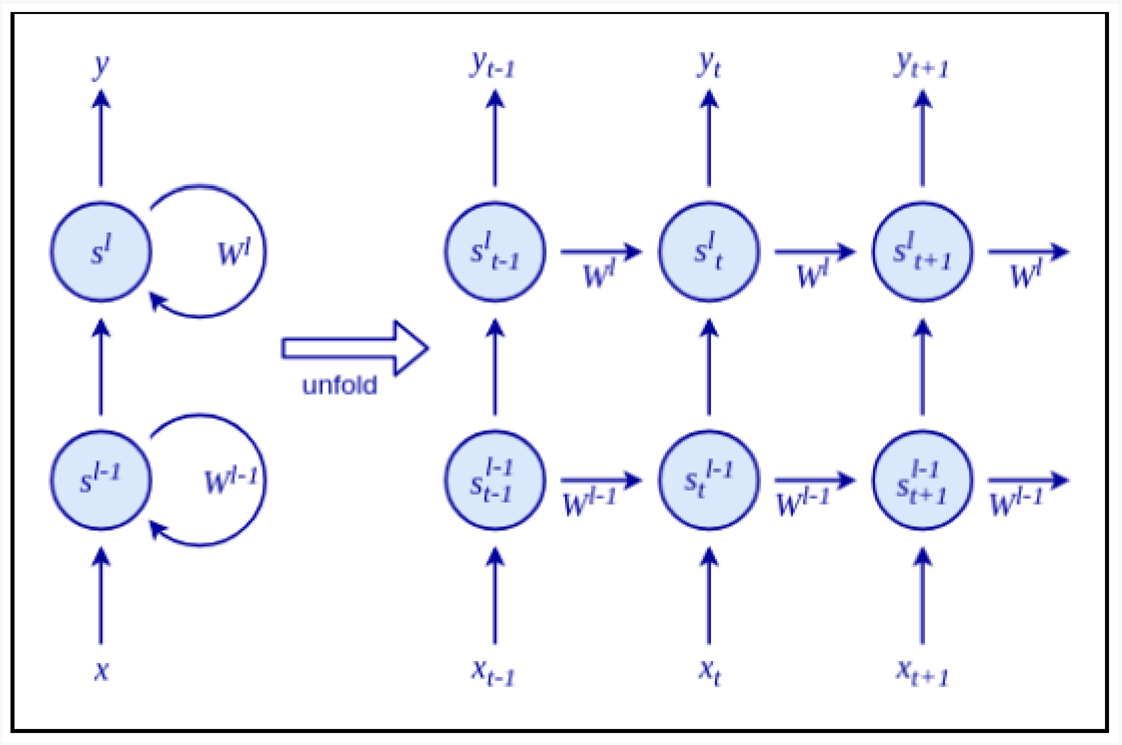
where f is a nonlinear activation function (e.g., hyperbolic tangent, sigmoid, or ReLU).

In a word-level language model, the input x will be a sequence of words encoded in input vectors (x1 ... xt ...). The state s will be a sequence of state vectors (s1 ... st ...) and the output y will be a sequence of probability vectors (y1 ... yt ...) for the following words in the sequence.

In an RNN, each state depends on all previous computations via the recurrent relationship. An important consequence of this is that RNNs have memory over time, since the states s contain information based on previous steps. Even RNNs can remember information over arbitrarily long periods, but in practice, they are limited to remembering only a few steps back.

As with regular neural networks, we can stack multiple RNNs to create a stacked RNN.

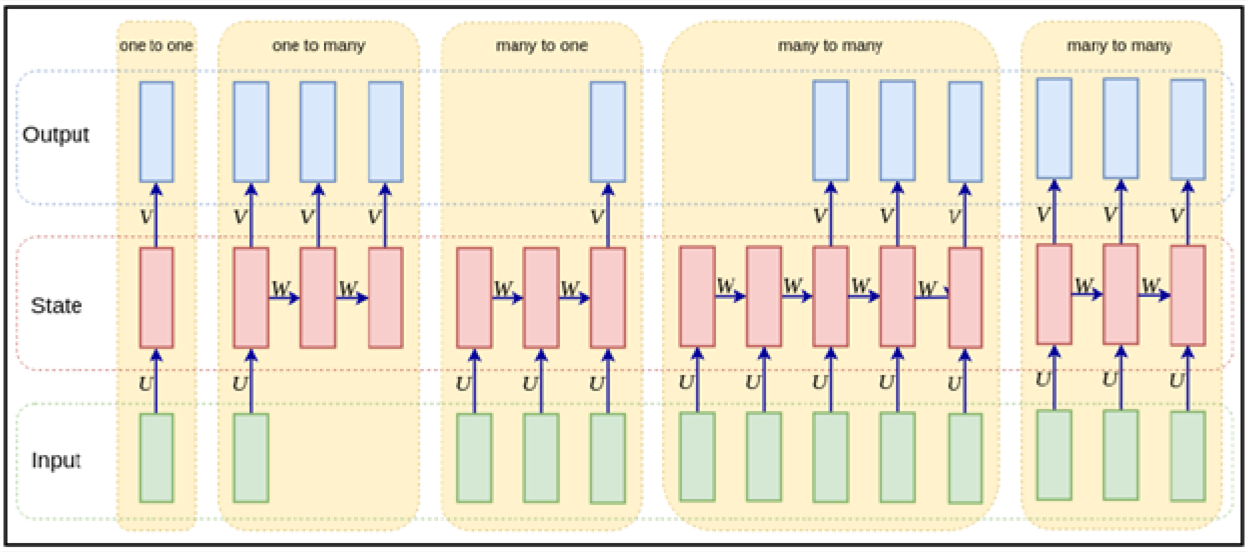
In the following figure, we see an unrolled, stacked RNN:



Since RNNs are not limited to processing inputs of fixed size, they significantly expand the possibilities of what we can compute with neural networks, such as sequences of different lengths or images of different sizes. We provide several combinations of processing:

- **One-to-one**: It is about sequential processing, such as feedforward neural networks and convolutional neural networks. There is not much difference between a feedforward network and using an RNN for a single time step. An example of one-way processing is image classification.
- **One-to-many**: It generates a sequence based on a single input, such as generating captions from an image.
- **Many-to-one**: The output is a single result based on a sequence, such as sentiment classification from text.
- **Many-to-many indirect**: The sequence is encoded into a state vector, which is then decoded into a new sequence, such as language translation.
- **Many-to-many direct**: This approach outputs a result for each input step, such as phoneme labeling within speech recognition.

The following image illustrates the previous combinations of inputs and outputs:



4.1.9

Implementation and training of a recurrent NN

Task: Teach an RNN to count ones in a sequence.

We will demonstrate using the Python language and the NumPy library how to teach a simple RNN to count the number of ones in the input and display the result at the end of the sequence. This is an example of a "many-to-one" relationship.

The example input and output are as follows:

```
Vstup: (0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0)
Výstup: 3
```

The network will have only two parameters: the input weight U and the recurrent weight W. The output weight V is set to 1, so that we can read only the last state as the output y.

The first step is to import the NumPy library, define the training data x and labels y. x is a two-dimensional array, as the first dimension represents the sample in our small dataset (we will use a small dataset with one sample):

```
import numpy as np
# The first dimension represents the mini-batch
```

```
x = np.array([[0, 0, 0, 0, 1, 0, 1, 0, 1, 0]])
y = np.array([3])
```

The recurrence relation defined by this network is $s_t = s_{t-1} * W + x_t * U$. As this is a linear model, we implement the recurrence relation as follows:

```
def step(s, x, U, W):
  return x * U + s *
```

The states st and weights W and U are individual scalar values. We just need to obtain the sum of inputs over the entire sequence. If we set U=1, then every input received will have its full value. If we set W=1, then the value that we would accumulate would never decrease. In this example, we would therefore get the desired output: 3.

Backpropagation Through Time

Backpropagation through time (BPTT) is a typical algorithm used for training recurrent neural networks.

The difference between standard backpropagation and backpropagation through time is that the recurrent NN is unfolded through time for a certain number of time steps. After the unfolding process, we obtain a model that is quite similar to a standard feedforward NN. One hidden layer of this network represents one time step. The only differences are that each layer has multiple inputs: the previous state s_{t-1} and the current input x_t. The parameters *U* and *W* are shared across all hidden layers.

The forward pass unfolds the RNN along the sequence and creates a stack of states for each time step. Here is the implementation of the forward pass that returns the activations for each recurrent step and each sample in the batch:

```
def forward(x, U, W):
  # Number of samples in the mini-batch
  number_of_samples = len(x)

  # Length of each sample
  sequence_length = len(x[0])

  # Initialize the state activation for each sample along the
sequence
  s = np.zeros((number_of_samples, sequence_length + 1))
  # Update the states over the sequence
  for t in range(0, sequence_length):
  s[:, t + 1] = step(s[:, t], x[:, t], U, W)
   # step function

  return s
```

When we have a step forward and a loss function, we can define the backward propagation gradient. Since the unfolded RNN is equivalent to a regular feedforward network, we can use the chain rule.

The weights *W* and *U* are shared across all the layers, so we will accumulate the derivative error for each recurrent step, updating the weights at the end with the accumulated value.

The implementation of the backward pass is as follows:

1. Gradients for *U* and *W* are accumulated in *gU* and *gW*:

```
def backward(x, s, y, W):
  sequence_length = len(x[0])

  # The network output is just the last activation of sequence
  s_t = s[:, -1]

  # Compute the gradient of the output w.r.t. MSE cost
function at final state
  gS = 2 * (s_t - y)

  # Set the gradient accumulations to 0
  gU, gW = 0, 0

  # Accumulate gradients backwards
  for k in range(sequence_length, 0, -1):
    # Compute the parameter gradients and accumulate the
results.
    gU += np.sum(gS * x[:, k - 1])
    gW += np.sum(gS * s[:, k - 1])

    # Compute the gradient at the output of the previous layer
    gS = gS * W
  return gU, gW
```

We will use gradient descent to optimize our network. We will use the mean squared error:

```
def train(x, y, epochs, learning_rate=0.0005):
  """Train the network"""
  # Set initial parameters
  weights = (-2, 0) # (U, W)
  # Accumulate the losses and their respective weights
  losses = list()
  weights_u = list()
  weights_w = list()
  # Perform iterative gradient descent
  for i in range(epochs):
    # Perform forward and backward pass to get the gradients
    s = forward(x, weights[0], weights[1])
    # Compute the MSE cost function
    loss = (y[0] - s[-1, -1]) ** 2
    # Store the loss and weights values for later display
    losses.append(loss)
    weights_u.append(weights[0])
    weights_w.append(weights[1])
    gradients = backward(x, s, y, weights[1])
    # Update each parameter `p` by p = p - (gradient *
learning_rate).
    # `gp` is the gradient of parameter `p`
    weights = tuple((p - gp * learning_rate) for p, gp in
zip(weights, gradients))
    print(weights)
  return np.array(losses), np.array(weights_u),
np.array(weights_w)
```

Next, we implement the related function `plot_training`, which displays the weights and loss:

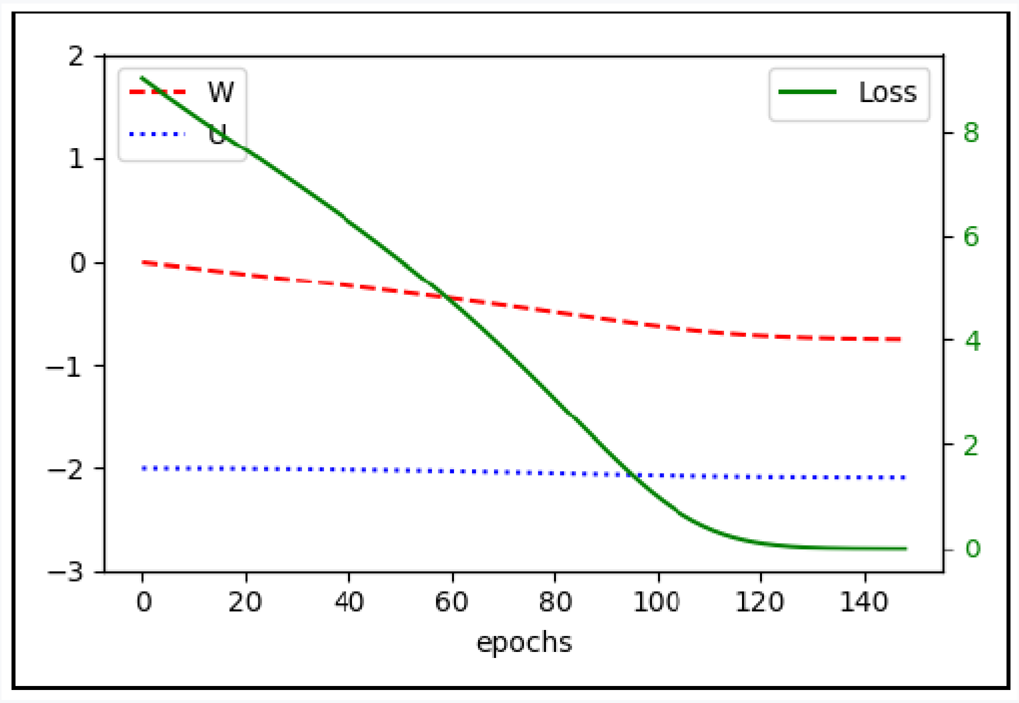
```
def plot_training(losses, weights_u, weights_w):
           import matplotlib.pyplot as plt

           # remove nan and inf values
           losses = losses[~np.isnan(losses)][:-1]
           weights_u = weights_u[~np.isnan(weights_u)][:-1]
```

```
            weights_w = weights_w[~np.isnan(weights_w)][:-1]
            # plot the weights U and W
            fig, ax1 = plt.subplots(figsize=(5, 3.4))
            ax1.set_ylim(-3, 2)
            ax1.set_xlabel("epochs")
            ax1.plot(weights_w, label="W", color="red",
linestyle="--")
            ax1.plot(weights_u, label="U", color="blue",
linestyle=":")
            ax1.legend(loc="upper left")
            # instantiate a second axis that shares the same x-
axis
            # plot the loss on the second axis
            ax2 = ax1.twinx()
            # uncomment to plot exploding gradients
            ax2.set_ylim(-3, 10)
            ax2.plot(losses, label="Loss", color="green")
            ax2.tick_params(axis="y", labelcolor="green")
            ax2.legend(loc="upper right")
            fig.tight_layout()
            plt.show()
```

We will run the following code:

```
losses, weights_u, weights_w = train(x, y, epochs=150)
plot_training(losses, weights_u, weights_w)
```

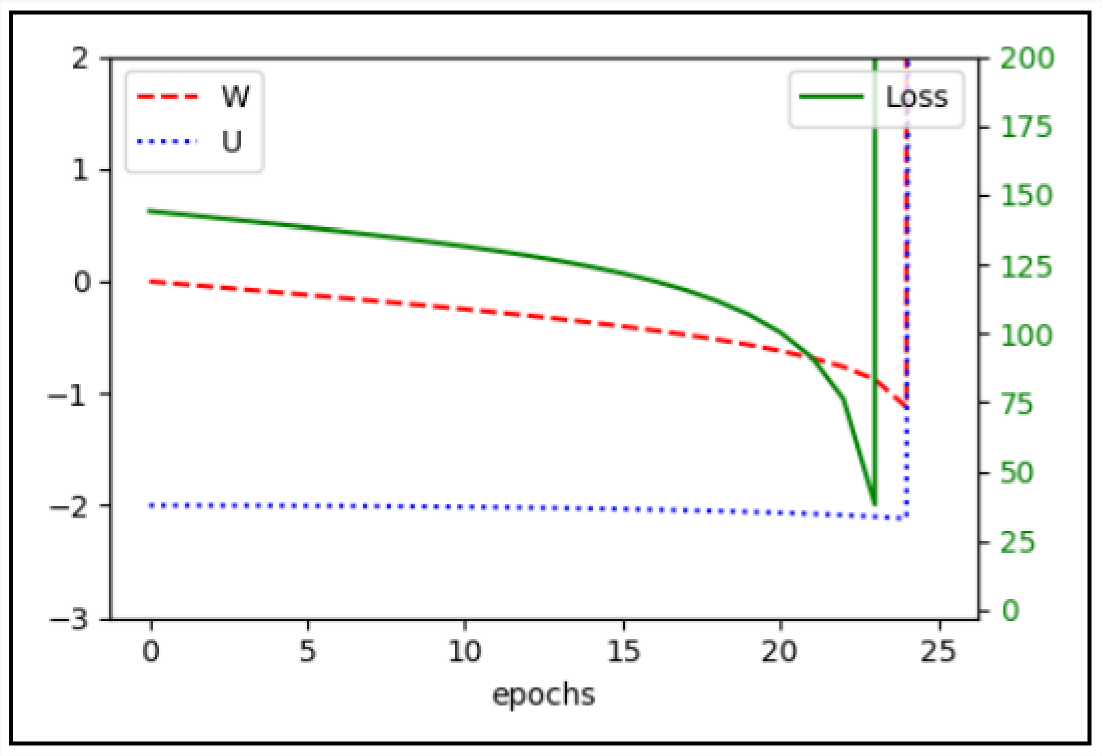
The result will be the following graph:



In the plot_training function, it is necessary to set ax2.set_ylim(-3, 200) from the original ax2.set_ylim(-3, 10).

The output will be a warning:

```
rnn_example.py:5: RuntimeWarning: overflow encountered in
multiply return x * U + s * W
rnn_example.py:36: RuntimeWarning: invalid value encountered
in multiply gU += np.sum(gS * x[:, k - 1])
rnn_example.py:37: RuntimeWarning: invalid value encountered
in multiply gW += np.sum(gS * s[:, k - 1])
```



The weights slowly approach the optimal value and the loss decreases until it reaches the value in epoch 23 (although the exact epoch is not important). There will be a situation where the cost surface on which we are training is very unstable. With small steps, we can move to the stable part of the cost function where the gradient is low, and suddenly we encounter a jump in costs and the corresponding huge gradient. Since the gradient is increasing so rapidly, it will have a large impact on our weights through weight updates - they will become NaN (Not a Number) (as illustrated by the jump off the graph). This problem is known as the exploding gradient.

There is also the problem of vanishing gradients (the opposite of exploding gradients). The gradient exponentially decays over the course of steps until it becomes extremely small in earlier states. Essentially, they are overshadowed by larger gradients from more recent time steps, and the network's ability to retain the history of these earlier states is lost. This problem is harder to detect because training will still proceed, and the network will produce valid outputs (unlike exploding gradients). It just won't be able to learn long-term dependencies.

Although vanishing and exploding gradients are present in common neural networks, they are particularly pronounced in RNNs. The reasons are as follows:

1. Depending on the length of the sequence, an RNN can be much deeper than a regular network.
2. The weights W are common to all steps. This means that the recurrent relationship that propagates the gradient back in time forms a geometric sequence:

$$\frac{\partial s_t}{\partial s_{t-m}} = \frac{\frac{\partial s_t}{\partial s_{t-1}} * \ldots * \partial s_{t-m+1}}{\partial s_{t-m}} = W^m$$

In our simple linear RNN, the gradient grows exponentially if |W| > 1 (exploding gradient). For example, 50 time steps with W=1.5 is W50 = 1.550 ≈ 6 * 108. The gradient decays exponentially if |W| <1 (vanishing gradient). For example, 20 time steps with W=0.6 is W20 = 0.620 ≈ 3*10-5. If the weight parameter W is a matrix and not a scalar, this exploding or vanishing gradient is related to the largest eigenvalue (ρ) of W (also known as the spectral radius). If ρ < 1, the gradients disappear, and if ρ>1, the gradients grow rapidly.

Neural language models

Chapter 5

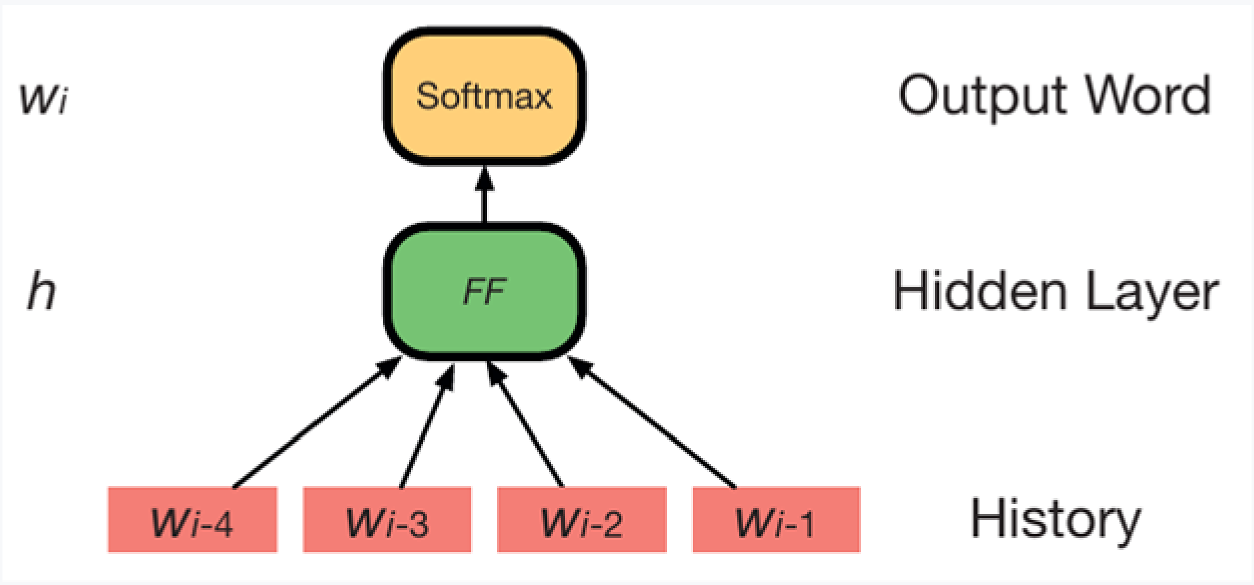
5.1 Neural language models

📖 5.1.1

Neural Language Models

Feedforward Neural Language Models

Example of a 5-gram neural network language model. Nodes in the network representing contextual words are connected to a hidden layer, which is connected to the output layer for predicting the next word.



📖 5.1.2

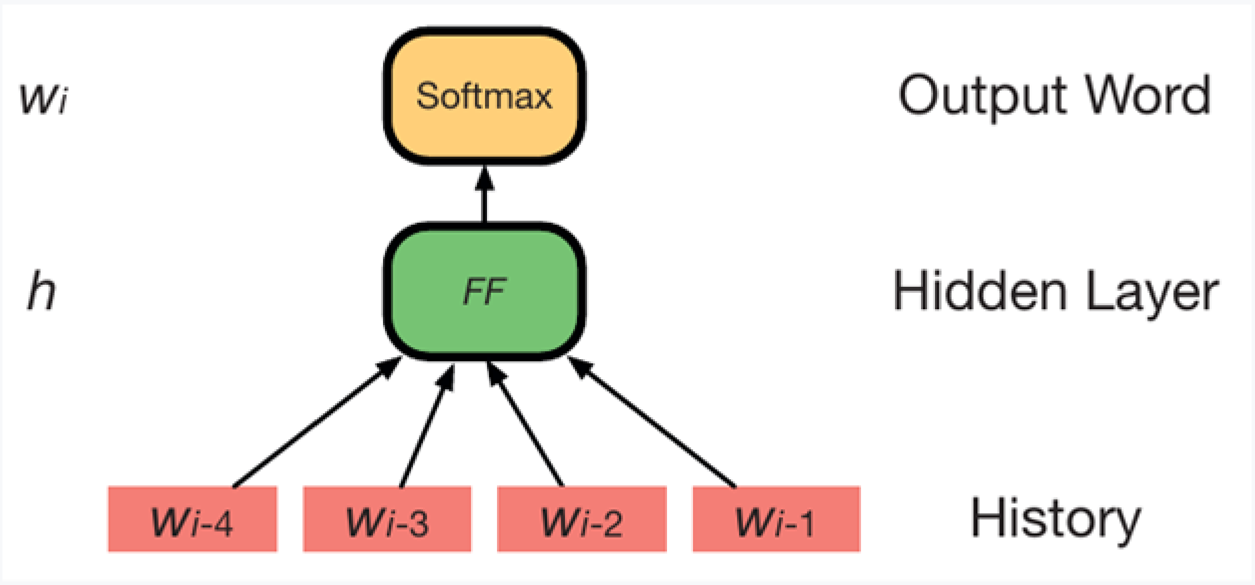
Word Representation

Nodes in a neural network take on real number values, but words are discrete items from a large vocabulary. We cannot use token identifiers because the neural network will assume that token 124 is very similar to token 125 - whereas in practice, these numbers are completely arbitrary. The same argument applies to the use of bit encoding for token identifiers. Words (1,1,1,1,1,0,0,0,0)T and (1,1,1,1,0,0,0,1)T may have very similar encoding, but may have nothing in common.

We will represent each word as a multi-dimensional vector with one dimension for each word in the dictionary. A value of 1 for the dimension that corresponds to the given word and 0 for the others. This type of vector is called a one-hot vector. For example:

- dog = (0,0,0,0,1,0,0,0,0,...)T,
- cat = (0,0,0,0,0,0,0,1,0,...)T,
- eat = (0,1,0,0,0,0,0,0,0,...)T.

To gather information between words, we introduce another layer between the input and hidden layers. In this layer, each contextual word is individually projected into a lower-dimensional space. We use the same weight matrix for each of the contextual words, creating a continuous spatial representation for each word independent of its position in the conditioning context. This representation is commonly referred to as a word embedding.



📖 5.1.3

Architecture of a neural network

The previous slide shows the architecture of a full-fledged forward neural network language model, which consists of contextual words as the input layer with a single hot vector, an embedded layer, a hidden layer, and a layer of predicted output words.

Contextual words are first encoded as one-hot vectors. They then pass through an embedding matrix E, creating a vector of floating-point numbers, known as an embedded word. This embedded vector usually has 500 or 1,000 nodes, and we use the same embedded matrix E for all contextual words.

Mathematically, not much happens in this case. Since the input to the matrix multiplication in E is a one-hot vector, most of the input values for matrix multiplication are zero. Essentially, we select one column of the matrix that corresponds to the ID of the input word. Therefore, activation is an unnecessary function. In a sense, the embedding matrix is a lookup table E(wj) for word embedding, indexed by the ID of the word wj:

$$E(w_j) = E\ w_j.$$

Mapping to the hidden layer in the model requires combining all contextual embedded words E(wj) as input to a classic forward layer, such as using hyperbolic tangent (tanh) as the activation function:

$$h = \tanh\left(b_h + \sum_j H_j E(w_j)\right).$$

The output layer is interpreted as a probability distribution over words. First, a linear combination of wij weights and hidden node values hj is calculated for each node i:

$$s = W\ h.$$

To ensure that it is a correct probability distribution, we use the softmax activation function to ensure that all values add up to one:

$$p_i = \operatorname{softmax}(s_i, \vec{s}) = \frac{e^{s_i}}{\sum_j e^{s_j}}.$$

It is similar to the neural probabilistic language model proposed by Bengio et al. (2003). This model had one more problem, adding direct connections of contextual word embeddings to word outputs, adding E (wj) word embeddings after a linear transformation with weight matrix Uj for each word wj. The equation s = W h is thus replaced by:

$$s = W\ h + \sum_j U_j\ E(w_j).$$

Their work suggests that such direct connections from contextual words to output words speed up training, although ultimately do not improve performance. They are also called residual connections, skip connections, or even the main path connections.

5.1.4

Training

We train the parameters of a neural language model (embedding matrix, weight matrices, deviation vectors) by processing all n-grams in the training corpus. For each n-gram, we input the context words into the network and compare the output with a hot vector of the correct word to be predicted. The weights are updated using backpropagation.

Language models are commonly evaluated using perplexity, which relates to the probability assigned to the correct source text. Therefore, the goal of training language models is to increase the probability of the trained data.

During training, given a context x = (w_{n-4}, w_{n-3}, w_{n-2}, w_{n-1}), we have the correct value for one hot vector →y. For each training example (x, →y), the training objective is defined based on the negative logarithmic probability as:

$$L(\mathbf{x}, \vec{y}; W) = -\sum_{k} y_k \log p_k .$$

The only value of y_k is equal to 1, and the others are equal to 0. This is the probability pk assigned to the correct word k. This definition allows us to update all weights, including those leading to incorrect output words.

5.2 Word embeddings

5.2.1

Word Embedding

In the field of NLP, the term word embedding is used to represent words in text analysis, typically in the form of a vector with real values that encodes the meaning of the word, so that words that are closer in the vector space are expected to have similar meanings.

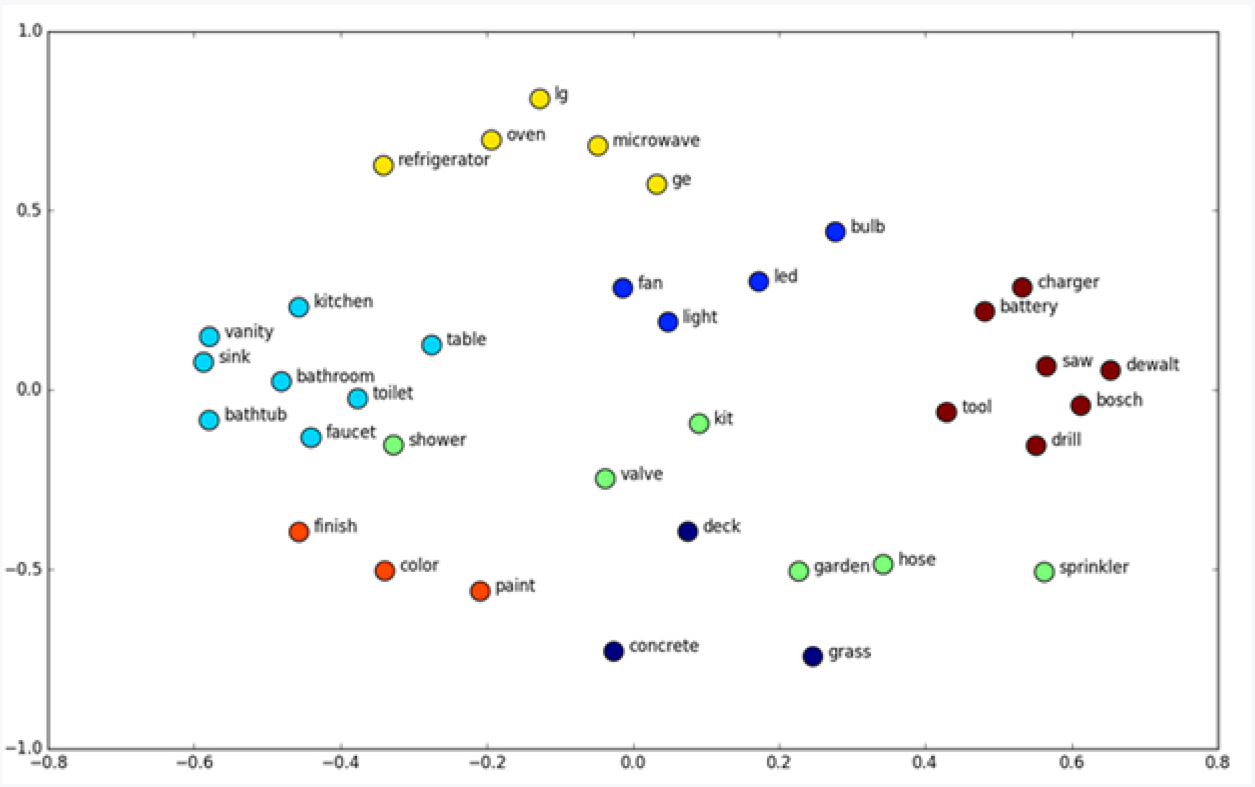
Word embedding plays an important role in neural language models. They represent contextual words that allow predicting the next word in a sequence. For example:

- but the friendly dog jumped,
- but the friendly cat jumped.

Since dog and cat occur in similar contexts, their influence on predicting the word "jumped" should be similar. It should be different from words like "car" which probably do not evoke "jumped". The idea that words occurring in similar contexts are semantically similar is a significant idea in lexical semantics.

Meaning and semantics are relatively complex concepts with mostly unresolved lexical semantic distributional definitions. The essence of distributional lexical semantics lies in defining words based on their distributional properties, i.e., the contexts in which they occur. Words that occur in similar contexts (dog and cat) should have similar representations. In vector space models, such as word embeddings, similarity can be measured by a distance function, such as cosine distance - the angle between vectors.

If we project multi-dimensional word embeddings into two dimensions, we can visualize them as shown in the figure. Words that are similar (drama, theater, festival) are grouped together.



5.3 Reccurent neural language models

📖 5.3.1

Recurrent Neural Language Models

This model uses the same architecture, i.e., words (input and output) are represented by a single hot vector, embedded words, and a hidden layer that uses, for example, 500 neurons with a real value. We use the tanh activation function on the hidden layer and the softmax function on the output layer. One of the inputs to the third word w3 in the sequence is the immediately preceding (known) word w2. However, the neurons in the network that we used to represent the beginning of the sentence are now filled with values from the hidden layer of the preceding prediction of the word w2.

In the previous feedforward NN architecture, the hidden state hi for predicting the i-th word was calculated from the preceding words $E(w_{i\text{-}j})$, for example, three words. This uses a shared embedding matrix E and a weight matrix Hj for each of the preceding words, plus a bias term bh:

$$h_i = \tanh\left(b_h + \sum_{1\leq j\leq 3} H_j E(w_{i-j})\right).$$

By modifying it to a recursive definition, we combine one preceding word wi-1 with the preceding hidden state hi-1. We add a weight matrix V to parameterize the mapping from the preceding hidden state $h_{i\text{-}1}$:

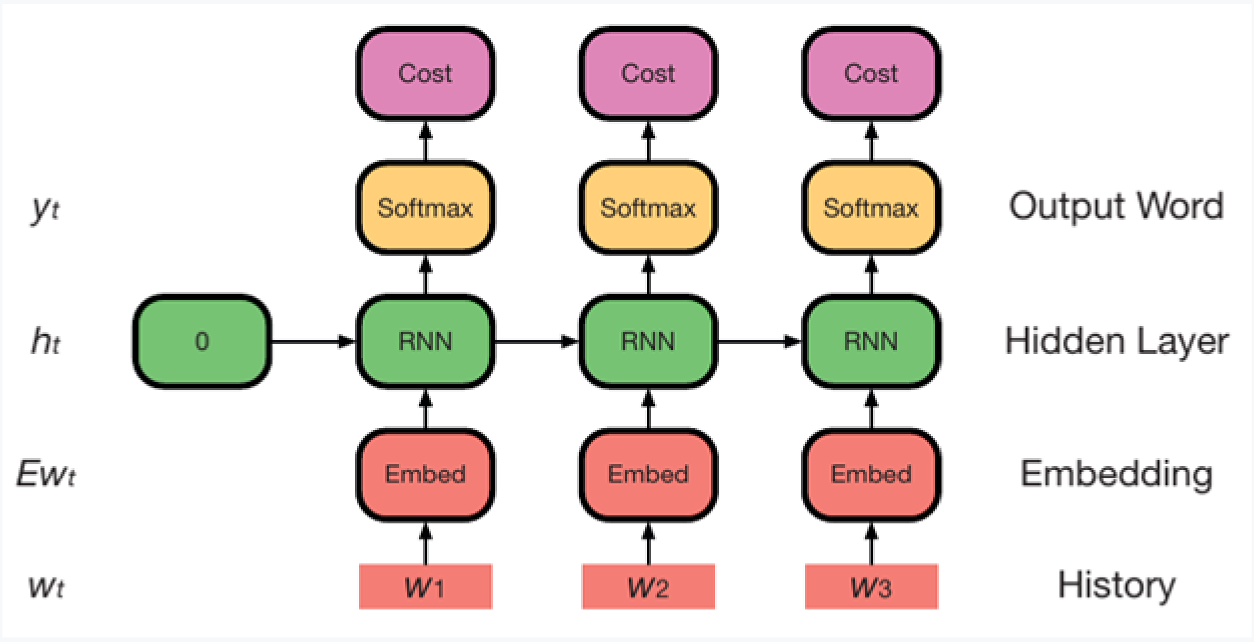
$$h_i = \tanh\left(b_h + HE(w_{i-1}) + Vh_{i-1}\right).$$

The neurons in the hidden state $h_{i\text{-}1}$ encode the context of the preceding sentence. At each step, they are enriched with information about the new input word, and thus are conditioned on the entire history of the sentence. So even the last word in the sentence is partially conditioned on the first word of the sentence. In addition, the model is simpler: it has fewer weights than a 3-gram feedforward neural language model.

🕮 5.3.2

Training with arbitrarily long contexts

The backpropagation through time procedure develops a recurrent neural network in a fixed number of steps, while returning back to, for example, predictions of five words.



Backpropagation through time can be used for each training example (time step), but it is computationally intensive. Computation must be performed at multiple steps each time. Instead, we can calculate and apply weight updates in smaller pieces. First, we process a larger number of training examples (for example, 10 to 20 or the entire sentence) and then update the weights.

5.3.3

What activation function do recurrent neural language models use on the output layer?

Neural translation models

Chapter 6

6.1 Encoder-decoder

🕮 6.1.1

A neural machine translation model is a direct extension of a language model. To train such a model, we simply concatenate the input and output sentence and use the same method as training a language model. To decode the sentence, we input the source sentence and then iterate through the model predictions until it predicts an end-of-sentence token (period, question mark, or exclamation mark).

🕮 6.1.2

Encoding Phase

When processing reaches the end of the input sentence (after predicting the end-of-sentence tag </s>), the hidden state encodes its meaning. The vector that acquires node values of this last hidden layer is the embedding of the input sentence. This is the encoding phase of the model. Then, this hidden state is used to generate the translation in the decoder phase.

🕮 6.1.3

Decoding Phase

During the encoding phase, the network must incorporate all the information about the input sentence. It cannot forget the first words at the end of the sentence. During the decoding phase, it is not necessary for the network to have enough information to predict each subsequent word. It is necessary to have an overview of what part of the input sentence has already been translated and what still needs to be translated.

6.2 Alignment model

🕮 6.2.1

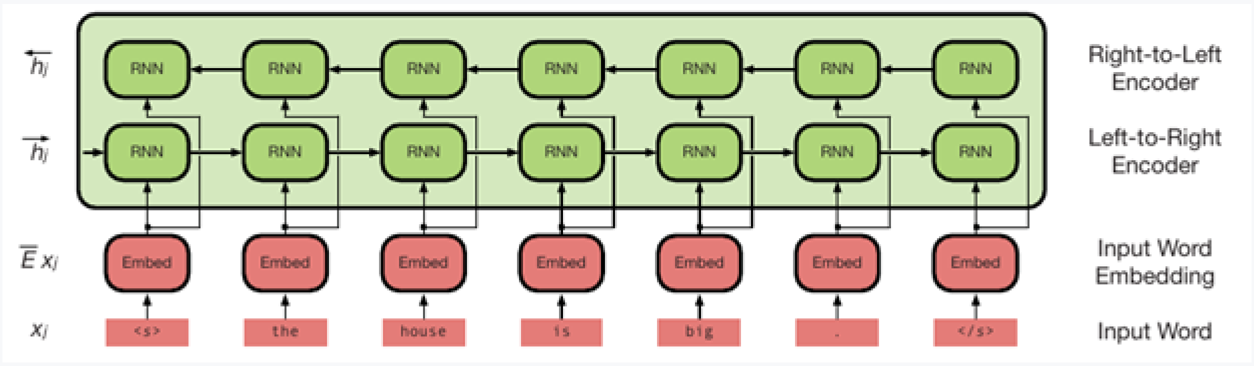
Alignment model

The first successful model of neural machine translation was the sequence-to-sequence (seq2seq) encoder-decoder with attention, which is essentially a model with an explicit alignment mechanism. In the world of deep learning, this alignment is called attention.

📖 6.2.2

Encoder

The task of the encoder is to provide a representation of the input sentence. The input sentence is a sequence of words, for which we first obtain an embedded matrix. Similar to a simple language model, we process these words using a recurrent neural network. The result is a set of hidden states that encode each word along with its left context, i.e., all the preceding words. To obtain the correct context, we also create a recurrent neural network that works from right to left, or from the end of the sentence to the beginning.



Two recurrent neural networks working in two directions are called a bidirectional recurrent neural network. Mathematically, the encoder consists of an embedded lookup for each input word x_j and a mapping that goes through the hidden states $\rightarrow h_j$ and $\rightarrow h_j$:

$$\overleftarrow{h_j} = f(\overleftarrow{h_{j+1}}, \bar{E}\, x_j)$$

$$\overrightarrow{h_j} = f(\overrightarrow{h_{j-1}}, \bar{E}\, x_j).$$

In the equations above, we use the general function f for the cell in the recurrent neural network. This function can be a simple layer of a feedforward neural network - for example, f(x) = tanh(Wx + b) - or more complex GRU or LSTM cells.

These models can be trained by adding a step that predicts the next word in the sequence, but in reality, we train them in the context of the entire machine translation model.

6.2.3

Decoder

The decoder is also a recurrent neural network. It takes a certain representation of the input context, preceding hidden states, and the prediction of the output word, and generates a new hidden state of the decoder and a new prediction of the output word. From the hidden state, we predict the output word. This prediction takes the form of a probability distribution over the entire output vocabulary. If we have a vocabulary of, say, 50,000 words, then the prediction is a 50,000-dimensional vector, where each element corresponds to the probability predicted for one word in the vocabulary.

During training, the correct output word y_i is known, so training continues with this word. The goal of training is to provide the highest possible probability for the correct output word. The production function (cost function) that controls training is therefore the negative logarithm of the probability given to the correct translation of the word:

$$\text{cost} = -\log t_i[y_i].$$

During the derivation of a new test sentence, we usually select the word y_i with the highest value in ti, i.e. the most probable translation. We use its embedding Ey_i for further inference steps.

6.2.4

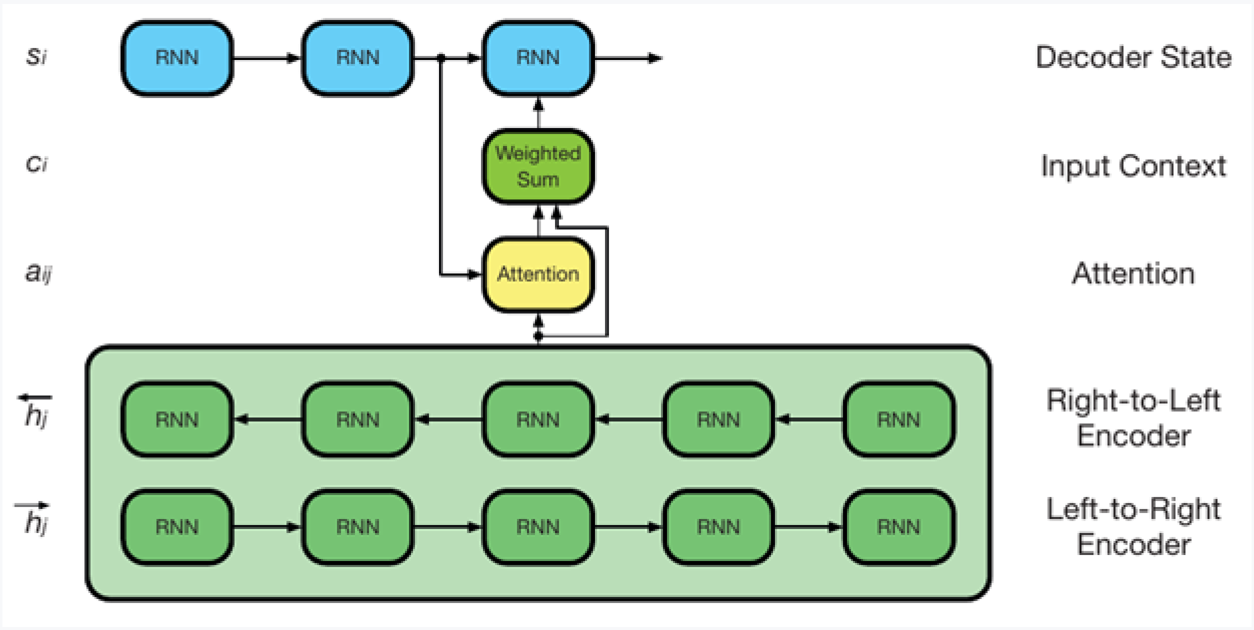
Attention mechanism

The attention mechanism in deep learning is a technique used to improve the performance of neural networks by allowing the model to focus on the most important input data when generating predictions. This is achieved by weighting the input data so that the model prioritizes certain input features over others. The result is that the model can produce more accurate predictions by considering only the most significant input variables.

Currently, we have two open ends. The encoder gave us a sequence of word representations $h_j = (\rightarrow h_j, \rightarrow h_j)$ and the decoder expects a context c_i at each step i. Now we describe the attention mechanism that brings these ends together.

The attention mechanism is difficult to illustrate using our typical neural network, but the figure below provides at least an idea of what the input and output relationships are. The attention mechanism is informed by all input word representations (→h$_j$, →h$_j$) and the previous decoder hidden state s_{i-1}, and creates a context state c_i.

The reason is that we want to compute the association between the decoder state (which contains information about where we are in the output sentence production) and each input word. Based on how strong this link is, or in other words, how relevant each specific input word is to the production of the next output word, we want to determine its impact on its word representation.



6.3 Neural translation model - implementation

6.3.1

Prerequisities:

- python
- installed libraries Numpy, TensorFlow, SkLearn

Importing libraries

```
from __future__ import absolute_import, division,
print_function # Import TensorFlow >= 1.10 and enable eager
execution
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import unicodedata
import re
import numpy as np
import os
import time

print(tf.__version__) #tensorflow version
```

Dataset preparation

We will use data (texts) that we obtained during the semester, either by extracting from the web or PDF documents.

The texts should be in tabular format, usually in (tsv - tab separated values). Since we are working with sentences, commas or underscores are not suitable separators. The easiest way to create such a file is to insert English and Slovak texts into a spreadsheet processor (MS Excel, LO Calc) and then export the table as *.tsv.

Subsequently, we will load this file:

```
path_to_file = 'slk.tsv'
```

Dataset preprocessing

Preprocessing consists of several steps:

1. Converting the Unicode file to ASCII.
2. Adding a space between a word and punctuation. For example, a period at the end of a sentence should be separated from the last word because it is considered a token.
3. Replacing all spaces with a space, except (a-z, A-Z, ".", "?", "!", ",").
4. Adding a starting (<s>) and ending (</s>) token that precisely delimits the sentence so that the model will know when to start or end the prediction.
5. Cleaning the sentence.
6. Returning word pairs in the SOURCE:TARGET format.
7. Creating a word-to-index map. For example: "father" -> 5 and conversely 5 -> "father", for each language.

```
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
    if unicodedata.category(c) != 'Mn')

def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())
    # creating a space between a word and the punctuation
following it
    w = re.sub(r'([?.!,¿])', r' \1 ', w)
    w = re.sub(r'[“ “]+', ' ', w)
    # replacing everything with space except (a-z, A-Z, ".",
"?", "!", ",")
    w = re.sub(r'[^a-zA-Z?.!,¿]+', ' ', w)
    w = w.rstrip().strip()
    # adding a start and an end token to the sentence
    # so that the model knows when to start and stop
predicting.
    w = ' ' + w + ' '
    return w

# 1. Remove the accents
# 2. Clean the sentences
# 3. Return word pairs in the format: [ENGLISH, SLOVAK]
def create_dataset(path, num_examples):
    lines = open(path, encoding='UTF-
8').read().strip().split('\n')
    word_pairs = [[preprocess_sentence(w) for w in
l.split('\t')] for l in lines[:num_examples]]
    return word_pairs# This class creates a word -> index
mapping (e.g,. "dad" -> 5) and vice-versa
# (e.g., 5 -> "dad") for each language,
class LanguageIndex():
    def __init__(self, lang):
        self.lang = lang
        self.word2idx = {}
        self.idx2word = {}
        self.vocab = set()

        self.create_index()

    def create_index(self):
        for phrase in self.lang:
            self.vocab.update(phrase.split(' '))
        self.vocab = sorted(self.vocab)

        self.word2idx[''] = 0
        for index, word in enumerate(self.vocab):
            self.word2idx[word] = index + 1

        for word, index in self.word2idx.items():
            self.idx2word[index] = word

def max_length(tensor):
        return max(len(t) for t in tensor)

def load_dataset(path, num_examples):
    # creating cleaned input, output pairs
    pairs = create_dataset(path, num_examples)# index language
using the class defined above
    inp_lang = LanguageIndex(sk for en, sk in pairs)
    targ_lang = LanguageIndex(en for en, sk in pairs)

    # Vectorize the input and target languages

    # slovak sentences
    input_tensor = [[inp_lang.word2idx[s] for s in sk.split('
')] for en, sk in pairs]

    # English sentences
    target_tensor = [[tarRozdelenie datasetu na trénovaciu
sadug_lang.word2idx[s] for s in en.split(' ')] for en, sk in
pairs]

    # Calculate max_length of input and output tensor
    # Here, we'll set those to the longest sentence in the
dataset
    max_length_inp, max_length_tar = max_length(input_tensor),
max_length(target_tensor)

    # Padding the input and output tensor to the maximum
length
    input_tensor =
tf.keras.preprocessing.sequence.pad_sequences(input_tensor,
maxlen=max_length_inp,padding='post')
```

```
    target_tensor =
tf.keras.preprocessing.sequence.pad_sequences(target_tensor,ma
xlen=max_length_tar,padding='post')

    return input_tensor, target_tensor, inp_lang, targ_lang,
max_length_inp, max_length_tar
```

Získanie vektorov

```
num_examples = 30000
input_tensor, target_tensor, inp_lang, targ_lang,
max_length_inp, max_length_targ = load_dataset(path_to_file,
num_examples)
```

Spliting dataset on test/train sets

```
input_tensor_train, input_tensor_val, target_tensor_train,
target_tensor_val = train_test_split(input_tensor,
target_tensor, test_size=0.2)
```

Defining basic values

BUFFER_SIZE = len(input_tensor_train) BATCH_SIZE = 64 N_BATCH = BUFFER_SIZE//BATCH_SIZE embedding_dim = 256 units = 1024 vocab_inp_size = len(inp_lang.word2idx) vocab_tar_size = len(targ_lang.word2idx) dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train, target_tensor_train)).shuffle(BUFFER_SIZE) dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)

Encoder and Decoder model

In this section, we will implement an encoder-decoder model with an attention mechanism. (https://github.com/tensorflow/nmt).

```
def gru(units):
 # If you have a GPU, we recommend using CuDNNGRU(provides a
3x speedup than GRU)
 # the code automatically does that.
 if tf.test.is_gpu_available():
   return tf.keras.layers.CuDNNGRU(units,
   return_sequences=True,
   return_state=True,
   recurrent_initializer='glorot_uniform')
 else:
   return tf.keras.layers.GRU(units,
   return_sequences=True,
   return_state=True,
   recurrent_activation='sigmoid',
   recurrent_initializer='glorot_uniform')

class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units,
batch_sz):
        super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = gru(self.enc_units)

    def call(self, x, hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
        return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))


class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units,
batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = gru(self.dec_units)
        self.fc = tf.keras.layers.Dense(vocab_size)

        # used for attention
        self.W1 = tf.keras.layers.Dense(self.dec_units)
        self.W2 = tf.keras.layers.Dense(self.dec_units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, x, hidden, enc_output):
        # enc_output shape == (batch_size, max_length,
hidden_size)
        # hidden shape == (batch_size, hidden size)
        # hidden_with_time_axis shape == (batch_size, 1,
hidden size)
        # we are doing this to perform addition to calculate
the score
        hidden_with_time_axis = tf.expand_dims(hidden, 1)

        # score shape == (batch_size, max_length, 1)
        # we get 1 at the last axis because we are applying
tanh(FC(EO) + FC(H)) to self.V
        score = self.V(tf.nn.tanh(self.W1(enc_output) +
self.W2(hidden_with_time_axis)))

        # attention_weights shape == (batch_size, max_length,
1)
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size,
hidden_size)
        context_vector = attention_weights * enc_output
        context_vector = tf.reduce_sum(context_vector, axis=1)

        # x shape after passing through embedding ==
(batch_size, 1, embedding_dim)
        x = self.embedding(x)

        # x shape after concatenation == (batch_size, 1,
embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x],
axis=-1)

        # passing the concatenated vector to the GRU
        output, state = self.gru(x)

        # output shape == (batch_size * 1, hidden_size)
        output = tf.reshape(output, (-1, output.shape[2]))

        # output shape == (batch_size * 1, vocab)
        x = self.fc(output)

        return x, state, attention_weights
```

```
    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.dec_units))
```

Next, we can create encoder and decoder in main scope

```
encoder = Encoder(vocab_inp_size, embedding_dim, units,
BATCH_SIZE)
decoder = Decoder(vocab_tar_size, embedding_dim, units,
BATCH_SIZE)
```

Defininig optimizier, loss function and checkpoints

```
optimizer = tf.optimizers.Adam()

def loss_function(real, pred):
    mask = 1 - np.equal(real, 0)
    loss_ =
tf.nn.sparse_softmax_cross_entropy_with_logits(labels=real,
logits=pred) * mask
    return tf.reduce_mean(loss_)

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, 'ckpt')
checkpoint = tf.train.Checkpoint(optimizer=optimizer,
encoder=encoder, decoder=decoder)
```

Training

1. The input goes through an encoder, which returns the output of the encoder and the hidden state of the encoder.
2. The output of the encoder, hidden state of the encoder, and input of the decoder (which is the start token) are passed to the decoder.
3. The decoder returns predictions and the hidden state of the decoder.
4. The hidden state of the decoder is then passed back into the model and the predictions are used to calculate the loss.
5. Teacher forcing is used to decide the next input to the decoder.
6. Teacher forcing is a technique where the target word is passed as the next input to the decoder.
7. The final step is computing gradients and using them on the optimizer and backpropagation.

```
EPOCHS = 5
```

```
for epoch in range(EPOCHS):
    start = time.time()

    hidden = encoder.initialize_hidden_state()
    total_loss = 0

    for (batch, (inp, targ)) in enumerate(dataset):
        loss = 0

        with tf.GradientTape() as tape:
            enc_output, enc_hidden = encoder(inp, hidden)

            dec_hidden = enc_hidden

            dec_input =
tf.expand_dims([targ_lang.word2idx['']] *  BATCH_SIZE, 1)

            # Teacher forcing — feeding the target as the next
input
            for t in range(1, targ.shape[1]):
                # passing enc_output to the decoder
                predictions, dec_hidden, _ =
decoder(dec_input, dec_hidden, enc_output)

                loss += loss_function(targ[:, t], predictions)

                # using teacher forcing
                dec_input = tf.expand_dims(targ[:, t], 1)

        batch_loss = (loss / int(targ.shape[1]))

        total_loss += batch_loss

        variables = encoder.variables + decoder.variables

        gradients = tape.gradient(loss, variables)

        optimizer.apply_gradients(zip(gradients, variables))

        if batch % 100 == 0:
            print('Epoch {} Batch {} Loss {:.4f}'.format(epoch
+ 1,
            batch,
            batch_loss.numpy()))
    # saving (checkpoint) the model every 2 epochs
    if (epoch + 1) % 2 == 0:
        checkpoint.save(file_prefix = checkpoint_prefix)

    print('Epoch {} Loss {:.4f}'.format(epoch + 1, total_loss
/ N_BATCH))
    print('Time taken for 1 epoch {} sec\n'.format(time.time()
- start))
```

Implementation of the evaluation function

The evaluation function is similar to the training loop, except we do not use teacher forcing here. The input to the decoder at each time step is its previous predictions along with the hidden state and output of the encoder.

```
def evaluate(sentence, encoder, decoder, inp_lang, targ_lang,
max_length_inp, max_length_targ):
    attention_plot = np.zeros((max_length_targ,
max_length_inp))

    sentence = preprocess_sentence(sentence)
    inputs = [inp_lang.word2idx[i] for i in sentence.split('
')]
    inputs =
tf.keras.preprocessing.sequence.pad_sequences([inputs],
maxlen=max_length_inp, padding='post')
    inputs = tf.convert_to_tensor(inputs)

    result = ''
    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)

    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([targ_lang.word2idx['']], 0)

    for t in range(max_length_targ):
        predictions, dec_hidden, attention_weights =
decoder(dec_input, dec_hidden, enc_out)

        # storing the attention weights to plot later on
        attention_weights = tf.reshape(attention_weights, (-1,
))
        attention_plot[t] = attention_weights.numpy()
    predicted_id = tf.argmax(predictions[0]).numpy()

    result += targ_lang.idx2word[predicted_id] + ' '

    if targ_lang.idx2word[predicted_id] == '':
        return result, sentence, attention_plot

    # the predicted ID is fed back into the model
    dec_input = tf.expand_dims([predicted_id], 0)

    return result, sentence, attention_plot
```

Function for printing attention mechanism weights and translation function

```
def plot_attention(attention, sentence, predicted_sentence):
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(1, 1, 1)
    ax.matshow(attention, cmap='viridis')

    fontdict = {'fontsize': 14}

    ax.set_xticklabels([''] + sentence, fontdict=fontdict,
rotation=90)
    ax.set_yticklabels([''] + predicted_sentence,
fontdict=fontdict)


plt.show()


def translate(sentence, encoder, decoder, inp_lang, targ_lang,
max_length_inp, max_length_targ):
    result, sentence, attention_plot = evaluate(sentence,
encoder, decoder, inp_lang, targ_lang, max_length_inp,
max_length_targ)

    print('Input: {}'.format(sentence))
    print('Predicted translation: {}'.format(result))

    attention_plot = attention_plot[:len(result.split(' ')),
:len(sentence.split(' '))]
    plot_attention(attention_plot, sentence.split(' '),
result.split(' '))
```

Checkpoint restoration and translation - so that we don't have to train the model from scratch every time.

```
# restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

translate(u'Tom pije pivo.', encoder, decoder, inp_lang,
targ_lang, max_length_inp, max_length_targ)
```

Machine Translation Evaluation

Natural Language Processing

Chapter 1

1.1 Introduction to NLP

1.1.1

Natural Language Processing (NLP) is an interdisciplinary field of research that aims to enable machines to understand and process human languages. NLP-based applications are everywhere and chances are we've already encountered an NLP-enabled app (Alexa, Google Translate, chatbots, etc.). This course aims to provide experience to help you build NLP applications by understanding its key concepts of it. In the course, we will learn about text preprocessing, an essential part of NLP work. The most important part of the course will be an introduction to how machine translation works and its practical application.

1.1.2

Artificial intelligence is rapidly permeating various areas of our lives from the smart home to automated tech support. NLP is closely related to Machine Learning (ML). Among the most widely used tools that have emerged in the field of NLP and are encountered perhaps on a daily basis are:

- **Chatbot** - AI-based software that can hold conversations with people in natural languages. Chatbots are widely used as the first contact of customer support and are very effective in solving simple user queries.
- **Sentiment analysis** - a set of algorithms and techniques used to detect the sentiment (positive, negative or neutral) of a given text. Sentiment analysis has made it possible to gather opinions from a much wider audience at a significantly lower cost. It can also be used to identify fake news.
- **Machine translation** - one of the early tasks that NLP focused on. Nowadays, it is one of the most widely used online machine translation tools from Google. The quality of machine translation is improving with time and new approaches are already based on neural networks.

1.1.3

For example, if we wanted to create a virtual assistant we would have to train it in a human way, we would have to load a language dictionary (the easy part) into its memory, find a way to teach it grammar (speech, clause, sentence structure, etc.) and logical interpretation. This is a time-consuming task. However, what if we could transform the sentence into mathematical objects so that the computer could use mathematical or logical operations to make some sense of it? This mathematical construct could be a **vector**, a matrix, and so on. Suppose we had an n-dimensional space where each axis corresponded to a word of our language. This allows us to represent a given sentence as a vector in this space with its coordinate along each axis as the number of words representing that axis.

However, to avoid having to do the **vectorization** process manually, we can use the Python programming language and the *scikit-learn* library. Using the

CountVectorizer() function, we can vectorize a sentence and get a matrix of vectors representing that sentence.

```
from sklearn.feature_extraction.text import CountVectorizer

document = ["I like computer science","There are many computer
softwares","I have an computer with various softwares"]
vectorizer = CountVectorizer()
vectorizer.fit(document)
print("Vocabulary: ", vectorizer.vocabulary_)
vector = vectorizer.transform(document)
print("Vectorized document:")
print(vector.toarray())
```

Program output:

```
Vocabulary:  {'like': 4, 'computer': 2, 'science': 6, 'there':
8, 'are': 1, 'many': 5, 'softwares': 7, 'have': 3, 'an': 0,
'with': 10, 'various': 9}
Vectorized document:
[[0 0 1 0 1 0 1 0 0 0 0]
 [0 1 1 0 0 1 0 1 1 0 0]
 [1 0 1 1 0 0 0 1 0 1 1]]
```

Preparation of texts

Chapter 2

2.1 Tokenization

2.1.1

The dictionary is an important part of several tasks in NLP. A **lexicon** can be defined as the vocabulary of a person, language, or discipline. Roughly speaking, a lexicon can be thought of as a dictionary of terms called **lexemes**. For example, the terms used by doctors can be thought of as the lexicon of their profession. As an example, in an attempt to create an algorithm to convert a physical prescription provided by doctors into an electronic form, lexicons would consist primarily of medical terms. Lexicons are used for a variety of NLP tasks where they are provided as a word list or dictionary.

Before discussing procedures on how to create a lexicon we need to understand phonemes, graphemes and morphemes:

- Phonemes can be thought of as the sound units that can distinguish one word from another in a given language.
- Graphemes are groups of letters of length one or more that can represent these individual sounds or phonemes.
- A morpheme is the smallest unit of meaning in a language.

2.1.2

When creating a dictionary you must first divide documents or sentences into parts called **tokens**. Each token carries a semantic meaning associated with it. **Tokenization** is one of the basic stages to be performed in any text processing activity. Tokenization can be thought of as a segmentation technique in which we try to divide larger portions of text into smaller meaningful parts. Tokens generally contain words and numbers but can also be extended to include punctuation marks, symbols, and sometimes comprehensible emoticons.

The simplest approach to tokenization is certainly a simple division based on spaces. We can use the **split()** function to do this, which splits a text variable based on a specified separator. By default, the function splits based on space. However, this is a trivial function that may not work properly.

```
sent = "I like computer science"
print(sent.split())
```

Program output:
```
['I', 'like', 'computer', 'science']
```

2.1.3

While the **split()** function can split a sentence into words we may find that in some cases the result may not be correct.

```
sentence = "Slovakia's capital is Bratislava"
print(sentence.split())
```

Program output:
```
["Slovakia's", 'capital', 'is', 'Bratislava']
```

We can observe that the function does not address the apostrophe and simply takes it as part of the word. This can be a problem, especially in the case of English phrases like *I'm* or *we'll* where it is a contraction of the following word.

```
sentence = "I'm happy to visit Bratislava"
print(sentence.split())
```

Program output:
```
["I'm", 'happy', 'to', 'visit', 'Bratislava']
```

Thus, there are a number of issues that can arise if we only use basic functions. The dots indicating abbreviations or different characters can be a problem. Therefore, in the next section, we will show the different tools that can be used in tokenization.

2.1.4

Among the popular techniques used for tokenization is the use of regular expressions. Regular expressions are sequences of characters that define a search pattern. They are one of the first and still one of the most effective tools for identifying patterns in text. Imagine that you are searching for an email address in a text. These follow the same pattern and are governed by a set of rules no matter what domain they are located on. Regular expressions are a way to identify such things in textual data instead of trying machine learning-oriented techniques.

2.1.5

The Python library **nltk** provides a regular expression-based tokenization function (**RegexpTokenizer**). We can use it to tokenize or split a sentence based on a given regular expression. Consider the following sentence:

```
The average price of computers in the US is between $300 -
$500.
```

We will need expressions denoting money and alphabetic sequences. For this purpose, we can define a regular expression and give the expression to the corresponding tokenizer object.

```
from nltk.tokenize import RegexpTokenizer
s = "The average price of computers in the US is between $300
- $500."
tokenizer = RegexpTokenizer('\w+|\$[\d]+|\S+')
print(tokenizer.tokenize(s))
```

2.1.6

Use the given regular expression to tokenize the given sentences into tokens. As a result, print the output from the tokenize() function.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

For tokenization, use the following regular expression:

```
\w+(?:'\w+)?|[^\w\s]
```

```
from nltk.tokenize import RegexpTokenizer
s = "The wooden spoon couldn’t cut but left emotional scars.
She finally understood that grief was her love with no place
for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"\w+(?:'\w+)?|[^\w\s]")
print(tokenizer.tokenize(s))
```

2.1.7

Use the given regular expression to tokenize the given sentences into tokens. As a result, print the output from the **tokenize()** function.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

For tokenization, use the following regular expression:

```
(\w+|#\d|\?|!)
```

```
from nltk.tokenize import RegexpTokenizer
s = "The wooden spoon couldn’t cut but left emotional scars.
She finally understood that grief was her love with no place
for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"(\w+|#\d|\?|!)")
print(tokenizer.tokenize(s))
```

2.1.8

Another option is to use the **Treebank** tokenizer, which also uses regular expressions to tokenize text according to the Penn Treebank database (https://catalog.ldc.upenn.edu/docs/LDC95T7/cl93.html). Words are mostly split based on punctuation. The Treebank tokenizer does an excellent job of splitting abbreviations such as *doesn't* into *does* and *n't*. Further, it identifies periods at the ends of lines and removes them. Punctuation marks, such as commas, are split if they are followed by spaces. Let's look at the previous sentence and tokenise it using the Treebank tokeniser.

```
from nltk.tokenize import TreebankWordTokenizer
s = "The average price of computers in the US is between $300
- $500."
tokenizer = TreebankWordTokenizer()
print(tokenizer.tokenize(s))
```

Program output:

```
['The', 'average', 'price', 'of', 'computers', 'in', 'the',
'US', 'is', 'between', '$', '300', '-', '$', '500', '.']
```

2.1.9

The development of social media has led to the emergence of an informal language in which people tag each other using their social media accounts and use a variety of emoticons, hashtags and shortened texts to express themselves. Hence, there is a need for tokenization tools that can parse such text and make it more comprehensible. **TweetTokenizer** strongly suits this use case. So let's try to analyze the following tweet:

```
@PassedToMessi Whatever happens on Sunday. Whatever God
decides. Whoever wins. Thank you for everything Leo Messi. You
are truly the greatest ever. With or without a Worldcup. The
Worldcup may just be a reward for your hardwork. <3
```

```
from nltk.tokenize import TweetTokenizer
s = "@PassedToMessi Whatever happens on Sunday. Whatever God
decides. Whoever wins. Thank you for everything Leo Messi. You
are truly the greatest ever. With or without a Worldcup. The
Worldcup may just be a reward for your hardwork. <3"
tokenizer = TweetTokenizer()
print(tokenizer.tokenize(s))
```

Program output:

```
['@PassedToMessi', 'Whatever', 'happens', 'on', 'Sunday', '.',
'Whatever', 'God', 'decides', '.', 'Whoever', 'wins', '.',
'Thank', 'you', 'for', 'everything', 'Leo', 'Messi', '.',
'You', 'are', 'truly', 'the', 'greatest', 'ever', '.', 'With',
'or', 'without', 'a', 'Worldcup', '.', 'The', 'Worldcup',
'may', 'just', 'be', 'a', 'reward', 'for', 'your', 'hardwork',
'.', '<3']
```

2.2 Stemming

2.2.1

Imagine if we combined the words computer, computerization and computerize into one word - compute. What is happening here is called **stemming**. Part of stemming is a crude attempt to remove the inflectional forms of a word and convert them to a base form called the stem of the word. The severed parts are called *affixes*. In the previous example, the base shape is *compute* and the affixes are *r*, *rization*, and *rize*. It is important to remember that the stem of a word may not be a valid word as we know it.

2.2.2

The two most common algorithms used for stemming are the **Porter stemmer** and the **Snowball stemmer**. The Porter stemmer supports English, while the Snowball stemmer which is an enhancement of the Porter stemmer, supports multiple languages. Porter stemmer works only with strings while Snowball works with strings and also with Unicode data. Snowball stemmer allows you to use a built-in function to ignore stop words.

```
plurals = ['caresses', 'flies', 'dies', 'mules', 'died',
'agreed', 'owned', 'humbled', 'sized', 'meeting', 'stating',
'generously']
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
singles = [stemmer.stem(plural) for plural in plurals]
print(' '.join(singles))
```

Program output:

```
caress fli die mule die agre own humbl size meet state gener
```

2.2.3

Use the Porter stemmer tool to create a word stem for the input sentence. Remember that sentences must first be tokenized into tokens and thus sent for stemming. As a result, output word stems are separated by spaces.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

```
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import RegexpTokenizer
stemmer = PorterStemmer()
sent = "The wooden spoon couldn’t cut but left emotional
scars. She finally understood that grief was her love with no
place for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"(\w+|#\d|\?|!)")
tokens = tokenizer.tokenize(sent)
output = [stemmer.stem(s) for s in tokens]
print(' '.join(output))
```

2.2.4

Snowball stemmer, unlike the previous one, requires language as a parameter. In most cases, its output is similar to that of the Porter stemmer, except for the word generously, where the Porter stemmer outputs gener and the Snowball stemmer outputs generous. The example shows how Snowball stemmer makes minor changes to Porter's algorithm, achieving improvements in some cases.

```
plurals = ['caresses', 'flies', 'dies', 'mules', 'died',
'agreed', 'owned', 'humbled', 'sized', 'meeting', 'stating',
'generously']
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer(language='english')
singles = [stemmer.stem(plural) for plural in plurals]
print(' '.join(singles))
```

Program output:

```
caress fli die mule die agre own humbl size meet state
generous
```

2.2.5

Use the Snowball stemmer tool to create a word stem for the input sentence. Remember that sentences must first be tokenized into tokens and thus sent for stemming. As a result, output word stems are separated by spaces.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

```
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import RegexpTokenizer
stemmer = SnowballStemmer(language='english')
sent = "The wooden spoon couldn’t cut but left emotional
scars. She finally understood that grief was her love with no
place for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"(\w+|#\d|\?|!)")
tokens = tokenizer.tokenize(sent)
output = [stemmer.stem(s) for s in tokens]
print(' '.join(output))
```

2.3 Lematization

2.3.1

Unlike stemming in which several features are removed from words using various methods, lemmatization is a process in that context is used to convert a word into its meaningful base form. It helps in grouping words that share a common basic form and therefore can be identified as a single item. The base form is referred to as the lemma of the word and is sometimes called the dictionary form. Lemmatization algorithms try to identify the lemma form of a word by taking into account the context of the word's surroundings, part-of-speech (POS) markers, the meaning of the word, etc. The surroundings may include words in the neighbourhood, sentences or even documents.

Also, the same words can have different lemmas depending on the context. The lemmatizer should try to identify the POS tags based on the context to determine the appropriate lemma. The most commonly used lemmatizer is the **WordNet** lemmatizer. Various other libraries also have integrated lemmatizer features, such as **Spacy**, **TextBlob**, **Gensim**, and others.

2.3.2

WordNet is an English lexical database that is freely and publicly available. WordNet includes nouns, verbs, adjectives and adverbs grouped into sets of cognitive synonyms (synsets), each expressing different concepts. These synsets are linked to each other by lexical and conceptual semantic relations. It can be easily downloaded and the nltk library offers an interface to it that allows to perform lemmatization.

```
import nltk
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
s = "She was putting efforts to heal her emotionally scarred
soul"
token_list = s.split()
print("The tokens are: ", token_list)
lemmatized_output = ' '.join([lemmatizer.lemmatize(token) for
token in token_list])
print("The lemmatized output is: ", lemmatized_output)
```

Program output:
```
The tokens are:  ['She', 'was', 'putting', 'efforts', 'to',
'heal', 'her', 'emotionally', 'scarred', 'soul']
```

```
The lemmatized output is:  She wa putting effort to heal her
emotionally scarred soul
[nltk_data] Downloading package wordnet to
/home/johny/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

We can observe that the lemmatization was not very successful and most of the words were not converted to lemma.

2.3.3

The reason why the lemmatizer didn't generate the correct lemmas in the previous lesson was that WordNet works better when it also has POS tags for the words in the input. The **nltk** library provides a method to generate POS tags for a list of words. We generate POS tags for a sentence in tuple form using the **pos_tag()** function.

```
import nltk
nltk.download('averaged_perceptron_tagger')
s = "She was putting efforts to heal her emotionally scarred
soul"
token_list = s.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

Program output:

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Unzipping
taggers/averaged_perceptron_tagger.zip.
[('She', 'PRP'), ('was', 'VBD'), ('putting', 'VBG'),
('efforts', 'NNS'), ('to', 'TO'), ('heal', 'VB'), ('her',
'PRP$'), ('emotionally', 'RB'), ('scarred', 'JJ'), ('soul',
'NN')]
```

2.3.4

However, in order for WordNet to work with the input we need to convert it to a different type of word type notation. Therefore, we can use the frequently used **get_part_of_speech_tags()** function to convert the tags into the form we need. We can then send the output of the function as a parameter to the lemmatizer.

```
import nltk
nltk.download('averaged_perceptron_tagger')
s = "She was putting efforts to heal her emotionally scarred
soul"
token_list = s.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

Program output:

```
[('She', 'PRP'), ('was', 'VBD'), ('putting', 'VBG'),
('efforts', 'NNS'), ('to', 'TO'), ('heal', 'VB'), ('her',
'PRP$'), ('emotionally', 'RB'), ('scarred', 'JJ'), ('soul',
'NN')]
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already
up-to-
[nltk_data]       date!
```

```
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

Program output:

```
She be put effort to heal her emotionally scar soul
```

We can see that the output is much better than in the case of not using POS tags.

2.3.5

Generate the lemmatized text for the specified text. Remember that the text needs to be tokenized first, followed by POS tags, and then lemmatized. There is a function to convert the tags for WordNet needs.

You are given the following text:

```
Analogous terms were later introduced for use of computers in
various fields, such as business informatics, forest
informatics, legal informatics etc. However, these fields have
more to do with digital literacy than with real informatics.
Their name is probably the result of a lack of knowledge of
the true meaning of informatics. Later in the United States,
next absurd term such as computational informatics were
developed, while all informatics is computational by its
nature.
```

```
import nltk
nltk.download('averaged_perceptron_tagger')
# create pos_tags
sentence = "Analogous terms were later introduced for use of
computers in various fields, such as business informatics,
forest informatics, legal informatics etc. However, these
fields have more to do with digital literacy than with real
informatics. Their name is probably the result of a lack of
knowledge of the true meaning of informatics. Later in the
United States, next absurd term such as computational
informatics were developed, while all informatics is
computational by its nature."
token_list = sentence.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

```
# just run this method for tagset transformation
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
# create the output of lemmatization
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

2.3.6

Generate the lemmatized text for the specified text. Remember that the text needs to be tokenized first, followed by POS tags, and then lemmatized. There is a function to convert the tags for WordNet needs.

You are given the following text:

```
Resources include individual files or an item's data, computer
programs, computer devices and functionality provided by
computer applications. Examples of consumers are computer
users, computer Software and other Hardware on the computer.
```

```
import nltk
nltk.download('averaged_perceptron_tagger')
# create pos_tags
sentence = "Resources include individual files or an item's
data, computer programs, computer devices and functionality
provided by computer applications. Examples of consumers are
computer users, computer Software and other Hardware on the
computer."
token_list = sentence.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

```
# just run this method for tagset transformation
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
# create the output of lemmatization
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

2.3.7

Generate the lemmatized text for the specified text. Remember that the text needs to be tokenized first, followed by POS tags, and then lemmatized. There is a function to convert the tags for WordNet needs.

You are given the following text:

```
An automated online assistant providing customer service on a
web page, an example of an application where natural language
processing is a major component. Natural language processing
(NLP) is a subfield of linguistics, computer science, and
artificial intelligence concerned with the interactions
between computers and human language, in particular how to
program computers to process and analyze large amounts of
natural language data. Challenges in natural language
processing frequently involve speech recognition, natural
language understanding, and natural-language generation.
```

```
import nltk
nltk.download('averaged_perceptron_tagger')
# create pos_tags
sentence = "An automated online assistant providing customer
service on a web page, an example of an application where
natural language processing is a major component. Natural
language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with
the interactions between computers and human language, in
particular how to program computers to process and analyze
large amounts of natural language data. Challenges in natural
language processing frequently involve speech recognition,
natural language understanding, and natural-language
generation."
token_list = sentence.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

```
# just run this method for tagset transformation
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
# create the output of lemmatization
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

2.4 Additional features

2.4.1

Stop words in English are words like *a, an, the, in, at,* and so on, which occur frequently in text corpora and don't carry much information in most contexts. These words are generally needed to complete sentences and make them grammatically correct. They are often the most common words in the language and can be filtered out in most NLP tasks, consequently, helping in reducing the vocabulary or search space. There is no single list of stop words available universally and they vary mostly based on use cases. However, a certain list of words is maintained for languages that can be considered language-specific stop words but should be modified based on the problem being solved.

For the use of stop words, there is a *stopwords* module in the **nltk** library that provides a list of English stop words that can be filtered out of the text under study.

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
s = "She was putting efforts to heal her emotionally scarred
soul"
token_list = s.split()
output = [token for token in token_list if token not in stop]
print(" ".join(output))
```

Program output:

```
She putting efforts heal emotionally scarred soul
[nltk_data] Downloading package stopwords to
/home/johny/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

2.4.2

For the specified text, remove stop words from it. Then print the text without stop words.

Given text:

```
An automated online assistant providing customer service on a
web page, an example of an application where natural language
processing is a major component. Natural language processing
(NLP) is a subfield of linguistics, computer science, and
artificial intelligence concerned with the interactions
between computers and human language, in particular how to
program computers to process and analyze large amounts of
natural language data. Challenges in natural language
processing frequently involve speech recognition, natural
language understanding, and natural-language generation.
```

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
s = "An automated online assistant providing customer service
on a web page, an example of an application where natural
language processing is a major component. Natural language
processing (NLP) is a subfield of linguistics, computer
science, and artificial intelligence concerned with the
interactions between computers and human language, in
particular how to program computers to process and analyze
large amounts of natural language data. Challenges in natural
language processing frequently involve speech recognition,
natural language understanding, and natural-language
generation."
token_list = s.split()
output = [token for token in token_list if token not in stop]
print(" ".join(output))
```

2.4.3

For the specified text, remove stop words from it. Then print the text without stop words.

Given text:

```
Members of the public have certain rights of access. These
include the right to access documents about the operation of
government departments and documents that are in the
possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information.
```

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
s = "Members of the public have certain rights of access.
These include the right to access documents about the
operation of government departments and documents that are in
the possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information."
token_list = s.split()
output = [token for token in token_list if token not in stop]
print(" ".join(output))
```

2.4.4

Another strategy that helps in normalizing the text is called **Case folding**. It is the unification of uppercase and lowercase with all the letters in the text corpus made lowercase. In several cases, the size of the letters plays a role and hence it is better to have all the words of the same size. This technique helps systems that deal with information retrieval, such as web search engines.

However, in situations where proper nouns are derived from common nouns, the unification of uppercase and lowercase letters becomes a hindrance because case distinction becomes an important feature here. Another problem is when abbreviations are converted to lowercase. There is a high probability that they will map to generic nouns.

A potential solution to this problem is to create machine learning models that can use features from the sentence to determine which words or tokens in the sentence should be lowercase and which should not. However, this approach is not always helpful when users mostly write in lowercase. As a result, writing everything in lowercase becomes the appropriate solution. Therefore, the strings-to-lowercase conversion function **lower()** can be used.

```
s = "She was putting efforts to heal her emotionally scarred
soul"
print(s.lower())
```

Program output:

```
she was putting efforts to heal her emotionally scarred soul
```

2.4.5

For the text you have entered, change the case of the letters to lowercase. Then print the text.

Given text:

```
Members of the public have certain rights of access. These
include the right to access documents about the operation of
government departments and documents that are in the
possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information.
```

```
s = "Members of the public have certain rights of access.
These include the right to access documents about the
operation of government departments and documents that are in
the possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information."
print(s.lower())
```

2.4.6

Sentences usually contain names of people and places and other open compound expressions, such as *living room* or *coffee mug*. These expressions convey a specific meaning when two or more words are used together. When used alone they carry a completely different meaning and the meaning of compound expressions is somehow lost. Using multiple tokens to represent such meaning can be very beneficial for NLP tasks performed. Although such occurrences are rare they still yield a lot of information. For this reason, we use techniques to preserve the meaning of compound expressions.

In general, these techniques fall under the term **n-grams**. If **n** is equal to **1**, they are referred to as **unigrams**. **Bigrams** or **2-grams** refer to pairs of words, such as *living room*. Phrases such as *United Arab Emirates*, which consist of three words are referred to as **trigrams** or **3-grams**. This naming system can be extended to larger n-grams but in most NLP tasks only trigrams or lower are used.

2.4.7

Let's try working with n-grams in practice. Let's have a sentence to describe what natural language processing is:

```
Natural language processing is an interdisciplinary subfield
of linguistics, computer science, and artificial intelligence
concerned with the interactions between computers and human
language, in particular how to program computers to process
and analyze large amounts of natural language data.
```

Since we know that natural language processing is a domain and processing these three words individually could cause a loss of meaning, we may prefer to use trigrams and preserve the meaning of the words. We can use the **nltk** library module called **ngrams** to create n-grams. The parameters for the function are the tokens of the sentence and the number of n-grams we want to generate.

```
from nltk.util import ngrams
sent = "Natural language processing is an interdisciplinary
subfield of linguistics, computer science, and artificial
intelligence concerned with the interactions between computers
and human language, in particular how to program computers to
process and analyze large amounts of natural language data."
tokens = sent.split()
trigrams = list(ngrams(tokens, 3))
print([" ".join(token) for token in trigrams])
```

Program output:
```
['Natural language processing', 'language processing is',
'processing is an', 'is an interdisciplinary', 'an
interdisciplinary subfield', 'interdisciplinary subfield of',
'subfield of linguistics,', 'of linguistics, computer',
'linguistics, computer science,', 'computer science, and',
'science, and artificial', 'and artificial intelligence',
'artificial intelligence concerned', 'intelligence concerned
with', 'concerned with the', 'with the interactions', 'the
interactions between', 'interactions between computers',
'between computers and', 'computers and human', 'and human
language,', 'human language, in', 'language, in particular',
'in particular how', 'particular how to', 'how to program',
'to program computers', 'program computers to', 'computers to
process', 'to process and', 'process and analyze', 'and
analyze large', 'analyze large amounts', 'large amounts of',
'amounts of natural', 'of natural language', 'natural language
data.']
```

2.4.8

For the given text, generate its unigrams. Print the unigrams.

Given text:

```
Members of the public have certain rights of access. These
include the right to access documents about the operation of
government departments and documents that are in the
possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information.
```

```
from nltk.util import ngrams
sent = "Members of the public have certain rights of access.
These include the right to access documents about the
operation of government departments and documents that are in
the possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information."
tokens = sent.split()
unigrams = list(ngrams(tokens, 1))
print([" ".join(token) for token in unigrams])
```

2.4.9

For the given text, generate its bigrams. Print the bigrams.

Given text:

```
Documents are also distinguished from "realia", which are
three-dimensional objects that would otherwise satisfy the
definition of "document" because they memorialize or represent
thought; documents are considered more as 2-dimensional
representations. While documents can have large varieties of
customization, all documents can be shared freely and have the
right to do so, creativity can be represented by documents,
also. History, events, examples, opinions, etc. all can be
expressed in documents.
```

```
from nltk.util import ngrams
sent = 'Documents are also distinguished from "realia", which
are three-dimensional objects that would otherwise satisfy the
definition of "document" because they memorialize or represent
thought; documents are considered more as 2-dimensional
representations. While documents can have large varieties of
customization, all documents can be shared freely and have the
right to do so, creativity can be represented by documents,
also. History, events, examples, opinions, etc. all can be
expressed in documents.'
tokens = sent.split()
bigrams = list(ngrams(tokens, 2))
print([" ".join(token) for token in bigrams])
```

2.4.10

For the given text, generate its trigrams. Print the trigrams.

Given text:

```
The Philip R. Lee Institute for Health Policy Studies is a
partner in this consortium. UCSF is home to the Industry
Documents Library (IDL), a digital library of previously
secret internal industry documents, including over 14 million
documents in the internationally known Truth Tobacco Industry
Documents, the Food Industry Documents Archive, Chemical
Industry Documents Archive and the Drug Industry Documents
Archive. The IDL contains millions of documents created by
```

```
major companies related to their advertising, manufacturing,
marketing, sales, and scientific research activities.
```

```
from nltk.util import ngrams
sent = 'The Philip R. Lee Institute for Health Policy Studies
is a partner in this consortium. UCSF is home to the Industry
Documents Library (IDL), a digital library of previously
secret internal industry documents, including over 14 million
documents in the internationally known Truth Tobacco Industry
Documents, the Food Industry Documents Archive, Chemical
Industry Documents Archive and the Drug Industry Documents
Archive. The IDL contains millions of documents created by
major companies related to their advertising, manufacturing,
marketing, sales, and scientific research activities.'
tokens = sent.split()
trigrams = list(ngrams(tokens, 3))
print([" ".join(token) for token in trigrams])
```

Sequence-to-Sequence Model

Chapter 3

3.1 Introduction

3.1.1

In this chapter, we will introduce the **Sequence-to-Sequence** model (Seq2Seq) and familiarize ourselves with encoders and decoders in the process. We will use this new knowledge to build our own machine translator using Seq2Seq modelling.

When we try to build a machine translation system we are essentially trying to convert a text sequence of arbitrary length into another text sequence of unknown length. The result is not always the text of the same length. English sentence *how are you doing?* is translated into Slovak as *ako sa máš?* The two sentences are of different lengths. Let's imagine another example: *can we do this?* is represented in Slovak as *môžeme ísť na to?* Although both English sentences contain four words, their Slovak counterparts are of different lengths. When building such systems we try to map an input sequence to an output sequence, which can be of different lengths. We use two essential things to do this: an **encoder** and a **decoder**.

3.1.2

The **encoder** is the first component in the encoder-decoder architecture. Input data is fed into the encoder and the encoder creates a representation of the input data. This low-dimensional representation of the input data is referred to as a **context vector**. The context vector attempts to capture the meaning of the input data. In essence, it tries to create an embedding of the input data, called embedding.

The encoder can be created using among other things different types of neural networks. For example, architectures based on recurrent neural networks store the context of the inputs they have seen in a hidden state. Therefore, the last hidden state will store the context of the entire sentence. The hidden state from the last time step is what we want. It is our context vector because it has seen all the inputs and preserved the context of all the input words.

The output of the encoder is a context vector that contains two parts:

- the hidden state from the last time step of the encoder,
- the neural network memory state for the input sentence.

3.1.3

Once we have obtained the context vector the next step is to insert it into the decoder and generate a translation of the input sentence. The initial hidden state for any recurrent neural network-based architecture is a randomly initialized vector. However, for decoders, the input is the context vector that we received as output from the encoder. So the initial hidden state should not be a randomly initialized vector but a context vector. The input to the decoder at the first time step is the token that marks the beginning of the sentence, *<start>*. Using this <start> token the decoder now has the task of learning to predict the first token of the target sentence. However, the decoder's job is slightly different for the learning and inference phases which are explained next.

3.1.4

During the **training phase**, the decoder received the target sequence along with the context vector as input. The input to the decoder at time step 0 is the *<start>* token. In time step 1, the input to the decoder is the predicted token or the first token of the target sequence, and so on. The task of the decoder is to learn that when it receives the context vector and the initial token <start>, it should be able to produce a set of tokens.

During the **inference phase**, we don't know what the target sequence should be and the decoder's job is to predict that target sequence. The decoder is given a context vector and an initial token, with which it should be able to predict the first token. It should then be able to predict the second token using the first predicted token and the hidden state from the first time step and continue in this way. The input at time step *t* is the predicted output from the previous time step *t-1*. The same pattern is followed for the rest of the decoder's work.

3.1.5

We already know roughly how the decoder learns, so now I just need to know how to stop sending outputs the moment a prediction occurs. Whenever the output from the decoder state is a token indicating the end of the *<end>* sentence, or we've reached a predefined maximum length for the output or target sequence, we get a signal that the decoder has finished its job of creating the output sequence and we need to stop here.

Simple LSTM neural networks at both ends allowed us to convert one data sequence to another using only the context vector. This approach for generating **Seq2Seq** can be used to create chatbots, speech recognition systems, natural language translation systems, and so on.

3.2 Machine translation - text preparation

3.2.1

In the following sections, we will describe how to create a machine translation using Seq2Seq modelling. We will focus on translation from the Slovak language to the English language. We will use a dataset from http://www.manythings.org/anki/, which contains about 11 000 sentences and their translations into English. In addition to the **nltk** library, we will also need the **tensorflow** and **keras** libraries to help us train our model. Next, we will need the **pandas** and **re** libraries to help us prepare the dataset. In the following microlessons, we'll go through the step-by-step process of how to build the model using mainly custom functions.

```
import pandas as pd
import string
import re
from urllib.request import urlopen
import numpy as np
from unicodedata import normalize
import keras, tensorflow
from keras.models import Model
from keras.layers import Input, LSTM, Dense
```

3.2.2

The first step will be to load an input file to help us train our model. We load the input file line by line into a DataFrame structure using the **pandas** library that acts like a table and will help us to access the individual data in the file. Once the file is loaded, we can examine the created DataFrame to have a better idea of what it contains.

```
import pandas as pd
from urllib.request import urlopen
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500:1510])
print(len(data))
data = data[:10000]
```

Program output:

```
["I've heard that.\tPočul som to.\tCC-BY 2.0 (France)
Attribution: tatoeba.org #2248400 (CK) & #9846917
(Dominika7)", "I've heard that.\tPočula som to.\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248400 (CK) & #9846918
(Dominika7)", 'Is he breathing?\tDýcha?\tCC-BY 2.0 (France)
Attribution: tatoeba.org #239892 (CK) & #8957974 (Dominika7)',
'Is it poisonous?\tJe to jedovaté?\tCC-BY 2.0 (France)
Attribution: tatoeba.org #2248466 (CK) & #10033911
(Dominika7)', 'Is it poisonous?\tJe jedovatý?\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248466 (CK) & #10033914
(Dominika7)', 'Is it poisonous?\tJe jedovatá?\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248466 (CK) & #10033916
(Dominika7)', 'Is she a doctor?\tJe lekárka?\tCC-BY 2.0
(France) Attribution: tatoeba.org #312527 (CK) & #9734240
(Dominika7)', 'Is this a river?\tJe toto rieka?\tCC-BY 2.0
(France) Attribution: tatoeba.org #56259 (CK) & #4642167
(Sim)', 'Is this my wine?\tTo je moje víno?\tCC-BY 2.0
(France) Attribution: tatoeba.org #1764491 (CK) & #10086221
(Dominika7)', 'Is today Monday?\tDnes je pondelok?\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248648 (CK) & #9948296
(Dominika7)']
11550
```

We can see that the dataset contained more than 11 thousand sentences. So let's take the first 10 000 sentences from the dataset, which will be used for training, and keep the rest as a test set on which we will then test our model.

3.2.3

With the input file, we saw that the sentences are separated by a tab (\t), so we can very easily use the **split()** function to split the sentences into Slovak and English.

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)
```

3.2.4

Once we have the sentences divided, we can proceed to the essential part namely the preprocessing of the texts. The goal of this feature is to remove unnecessary characters from sentences, such as punctuation or special characters. The next step is to unify the case of the letters with all words starting with a lowercase letter. The result will be a preprocessed sentence cleaned of unnecessary characters.

```
def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  return ' '.join(cleaned_sent)
```

3.2.5

Once we have prepared the sentence preprocessing function we can apply it to our English and Slovak sentences.

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)
```

3.2.6

The next phase is important because it's where we'll be creating our own dictionary. The goal will also be to obtain tokens that mark the beginning and end of the sequence, as required by the decoder. When we covered dictionary creation in previous lessons we dealt with it at the word level. In this case, we'll go one level lower and work at the character level. We'll place a tab at the beginning of our sequence and a newline label at the end. We'll also prepare a list of unique input and output characters. Our model will then attempt to predict at the character level.

```
def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)

  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)

  return input_dataset, target_dataset,
sorted(list(input_characters)),
sorted(list(target_characters))

input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

3.2.7

The following code will serve as a revision of the functions already created. So let's take a look at what the dictionary we generated looks like. Run the individual code blocks in order. Your task is to print what the **input character** list looks like.

```
import pandas as pd
import string
import re
from urllib.request import urlopen
import numpy as np
from unicodedata import normalize
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500])
print(len(data))
data = data[:10000]
```

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)
```

```
def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  return ' '.join(cleaned_sent)
```

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)
```

```
def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)

  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)

  return input_dataset, target_dataset,
sorted(list(input_characters)),
sorted(list(target_characters))

input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

```
# write your code here
print(input_characters)
```

3.2.8

The following code will serve as a revision of the functions already created. So let's take a look at what the dictionary we generated looks like. Run the individual code blocks in order. Your task is to write out what the list of **output characters** looks like.

```
import pandas as pd
import string
import re
```

```
from urllib.request import urlopen
import numpy as np
from unicodedata import normalize
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500])
print(len(data))
data = data[:10000]
```

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)
```

```
def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  return ' '.join(cleaned_sent)
```

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)
```

```
def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)

  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)

  return input_dataset, target_dataset,
sorted(list(input_characters)),
sorted(list(target_characters))

input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

```
# write your code here
print(target_characters)
```

3.3 Machine translation - model creation

3.3.1

The results of the previous assignments show us the difference in that we have added escape sequences to the output characters indicating the beginning and end of our sequence, so the **\t** and **\n** tokens are also there. These are used for the decoder to better understand the beginning and end of the sequence. Our input and output dictionaries need not be the same for tasks such as natural language translation. In fact, sometimes even our character set may not be the same. For example, we may be trying to translate between English and Arabic, which have completely different character sets. In addition to the differences in vocabulary, we should also be aware that our input sequence and the target sequence may not be the same size. Not only the number of words in two parallel sentences may be different but also the number of characters in each word. Therefore, we need to get information about the metadata of our sentences:

- the size of the input and output vocabulary,
- the maximum length of the input and output character set.

```
def get_metadata(input_dataset, target_dataset,
input_characters, target_characters):
  num_Encoder_tokens = len(input_characters)
  num_Decoder_tokens = len(target_characters)
  max_Encoder_seq_length = max([len(data_point) for data_point
in input_dataset])
  max_Decoder_seq_length = max([len(data_point) for data_point
in target_dataset])
  print('Number of data points:', len(input_dataset))
  print('Number of unique input tokens:', num_Encoder_tokens)
  print('Number of unique output tokens:', num_Decoder_tokens)
  print('Maximum sequence length for inputs:',
max_Encoder_seq_length)
  print('Maximum sequence length for outputs:',
max_Decoder_seq_length)
  return num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length

num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length =
get_metadata(input_dataset, target_dataset, input_characters,
target_characters)
```

```
Number of data points: 10000
Number of unique input tokens: 26

Number of unique output tokens: 29

Maximum sequence length for inputs: 50

Maximum sequence length for outputs: 38
```

3.3.2

```
Number of data points: 10000

Number of unique input tokens: 26

Number of unique output tokens: 29

Maximum sequence length for inputs: 50

Maximum sequence length for outputs: 38
```

In the previous microlesson, we got information about our metadata. We discovered the following:

- there are 10 000 unique English-Slovak sentence pairs in our dataset,
- the number of unique input tokens (characters) is 26,
- the number of unique output tokens (characters) that we try to extract and predict is 29,
- our longest input sequence is 50 characters long,
- our longest output sequence is 38 characters long.

3.3.3

A very important step is to create a mapping from characters to indexes and vice versa. This will help us in the following activities:

- represent our input characters using the appropriate indices,
- convert our predicted indices to their corresponding characters when predicting.

```
def create_indices(input_characters, target_characters):
  input_char_to_idx = {}
  input_idx_to_char = {}
  target_char_to_idx = {}
  target_idx_to_char = {}
  for i, char in enumerate(input_characters):
    input_char_to_idx[char] = i
    input_idx_to_char[i] = char
  for i, char in enumerate(target_characters):
    target_char_to_idx[char] = i
    target_idx_to_char[i] = char
  return input_char_to_idx, input_idx_to_char,
target_char_to_idx, target_idx_to_char

input_char_to_idx, input_idx_to_char, target_char_to_idx,
target_idx_to_char = create_indices(input_characters,
target_characters)
```

3.3.4

We can then build our data structure based on the extracted metadata from the previous microlessons.

```
def build_data_structures(length_input_dataset,
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens):
  Encoder_input_data = np.zeros((length_input_dataset,
max_Encoder_seq_length, num_Encoder_tokens), dtype='float32')
  Decoder_input_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  Decoder_target_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  print("Dimensionality of Encoder input data is : ",
Encoder_input_data.shape)
  print("Dimensionality of Decoder input data is : ",
Decoder_input_data.shape)
  print("Dimensionality of Decoder target data is : ",
Decoder_target_data.shape)
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
build_data_structures(len(input_dataset),
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens)
```

```
Dimensionality of Encoder input data is : (10000, 50, 26)

Dimensionality of Decoder input data is : (10000, 38, 29)

Dimensionality of Decoder target data is : (10000, 38, 29)
```

3.3.5

Dimensionality of Encoder input data is : (10000, 50, 26)

Dimensionality of Decoder input data is : (10000, 38, 29)

Dimensionality of Decoder target data is : (10000, 38, 29)

Let's look at the properties of the data structure we have created:

- the dimension of the input data is (10000, 50, 26),
- the first dimension represents the number of data points: 10 000,
- the second dimension represents the maximum length of our input sequence: 50,
- the third dimension represents the size of our input character set: 26,
- the dimension of the decoder input and output data is (10000, 38, 29),
- the first dimension represents the number of data points: 10 000,
- the second dimension represents the maximum length of our output sequence: 38,
- the third dimension represents the size of our output character set: 29,

Once we have created the data structure we add data to it.

```
def add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data):
  for i, (input_data_point, target_data_point) in
enumerate(zip(input_dataset, target_dataset)):
    for t, char in enumerate(input_data_point):
      Encoder_input_data[i, t, input_char_to_idx[char]] = 1.
    for t, char in enumerate(target_data_point):
      Decoder_input_data[i, t, target_char_to_idx[char]] = 1.
      if t > 0:
        Decoder_target_data[i, t - 1,
target_char_to_idx[char]] = 1.
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data)
```

3.3.6

We used a character-to-index mapping and converted some of the entries in our data structure to 1, indicating the presence of a particular character at a particular position in each of the sentences. As a result of the mapping, the last dimension (26 in the encoder input data structure and 29 in the decoder input or target data structure) is a vector of 1's indicating which item is present at a given position in our data. We do not insert anything for the *<start>* token when building the decoder target data and it is also prefixed by a one-time step for the same reasons we mentioned in the section on decoders. Our decoder target data is the same as the decoder input data, it's just shifted by one-time step. We are now ready to set the hyperparameters of our model.

```
batch_size = 256
epochs = 100
latent_dim = 256
```

3.3.7

The next step is to create our coder. We set the *return_state* property to *True* so that the decoder will return the last hidden state and memory that will create the context vector. The states *state_h* and *state_c* represent our last hidden state and memory information. The role of the encoder is to provide a context vector that captures the context of the input sentence. However, we have no explicit target context vector defined against which we can compare the performance of the encoder. The encoder learns from the performance of the decoder, which we will describe later. The decoder error is fed back and this is how backpropagation works in the encoder and the encoder learns based on this.

```
Encoder_inputs = Input(shape=(None, num_Encoder_tokens))
Encoder = LSTM(latent_dim, return_state=True)
Encoder_outputs, state_h, state_c = Encoder(Encoder_inputs)
Encoder_states = [state_h, state_c]
```

3.3.8

Let's focus on the second part, the decoder. During training, both input and output data are provided to the decoder and the decoder is asked to predict the input data with an offset of 1. This helps the decoder understand what it should predict if it receives a context vector from the encoder. This learning method is referred to as teacher forcing. The initial state of the decoder is in the *Encoder_states* variable, which is our context vector obtained from the encoder. The neural network layer is part of the decoder, where the number of neurons is equal to the number of tokens (in our case, characters) present in the output character set of the decoder. This layer is associated with the output of the *softmax* function, which helps us obtain normalized probabilities for each output character. Also, this function predicts the target character with the highest probability.

The *return_sequences* parameter in the neural network decoder helps us to get the entire output sequence from the decoder. We want the output from the decoder at each time step and hence we set this parameter to *True*. Since we have used a layer along with the output of the *softmax* function, we get the probability distribution over our output features for each time step, selecting the feature with the highest probability. We judge the performance of our decoder by comparing its output produced at each time step.

```
Decoder_inputs = Input(shape=(None, num_Decoder_tokens))
Decoder_lstm = LSTM(latent_dim, return_sequences=True,
return_state=True)
Decoder_outputs, _, _ = Decoder_lstm(Decoder_inputs,
initial_state=Encoder_states)
Decoder_dense = Dense(num_Decoder_tokens,
activation='softmax')
Decoder_outputs = Decoder_dense(Decoder_outputs)
```

3.3.9

We have defined our encoder and decoder and now we will combine them into a model. We'll use the **Keras Model API** to define the different inputs and outputs that we'll use at different stages. The Model API provides *Encoder_input_data*; *Decoder_input_data* is the input to our model that will be used as the encoder and decoder inputs; *Decoder_target_data* is used as the decoder output. The model will try to convert *Encoder_input_data* and *Decoder_input_data* to *Decoder_target_data*.

```
model = Model(inputs=[Encoder_inputs, Decoder_inputs],
outputs=Decoder_outputs)
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy')
model.summary()
```

3.3.10

The following code will serve as a reiteration of the already created functions and deployment of the encoder and decoder. So let's take a look at what the summary of our model looks like. Run the individual code blocks in order. Your task is to list what is the number of **parameters in the LSTM**.

```
import pandas as pd
import string
import re
from urllib.request import urlopen
```

```
import numpy as np
from unicodedata import normalize
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500])
print(len(data))
data = data[:10000]
```

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)

def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  return ' '.join(cleaned_sent)
```

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)

def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)

  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)

  return input_dataset, target_dataset,
sorted(list(input_characters)),
sorted(list(target_characters))

input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

```
def get_metadata(input_dataset, target_dataset,
input_characters, target_characters):
  num_Encoder_tokens = len(input_characters)
  num_Decoder_tokens = len(target_characters)
  max_Encoder_seq_length = max([len(data_point) for data_point
in input_dataset])
```

```
  max_Decoder_seq_length = max([len(data_point) for data_point
in target_dataset])
  print('Number of data points:', len(input_dataset))
  print('Number of unique input tokens:', num_Encoder_tokens)
  print('Number of unique output tokens:', num_Decoder_tokens)
  print('Maximum sequence length for inputs:',
max_Encoder_seq_length)
  print('Maximum sequence length for outputs:',
max_Decoder_seq_length)
  return num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length

num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length =
get_metadata(input_dataset, target_dataset, input_characters,
target_characters)
```

```
def create_indices(input_characters, target_characters):
  input_char_to_idx = {}
  input_idx_to_char = {}
  target_char_to_idx = {}
  target_idx_to_char = {}
  for i, char in enumerate(input_characters):
    input_char_to_idx[char] = i
    input_idx_to_char[i] = char
  for i, char in enumerate(target_characters):
    target_char_to_idx[char] = i
    target_idx_to_char[i] = char
  return input_char_to_idx, input_idx_to_char,
target_char_to_idx, target_idx_to_char

input_char_to_idx, input_idx_to_char, target_char_to_idx,
target_idx_to_char = create_indices(input_characters,
target_characters)
```

```
def build_data_structures(length_input_dataset,
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens):
  Encoder_input_data = np.zeros((length_input_dataset,
max_Encoder_seq_length, num_Encoder_tokens), dtype='float32')
  Decoder_input_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  Decoder_target_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
```

```
  print("Dimensionality of Encoder input data is : ",
Encoder_input_data.shape)
  print("Dimensionality of Decoder input data is : ",
Decoder_input_data.shape)
  print("Dimensionality of Decoder target data is : ",
Decoder_target_data.shape)
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
build_data_structures(len(input_dataset),
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens)
```

```
def add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data):
  for i, (input_data_point, target_data_point) in
enumerate(zip(input_dataset, target_dataset)):
    for t, char in enumerate(input_data_point):
      Encoder_input_data[i, t, input_char_to_idx[char]] = 1.
    for t, char in enumerate(target_data_point):
      Decoder_input_data[i, t, target_char_to_idx[char]] = 1.
      if t > 0:
        Decoder_target_data[i, t - 1,
target_char_to_idx[char]] = 1.
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data)
```

```
import tensorflow
from tensorflow import keras

#from keras.models import Model
#from keras.layers import Input, LSTM, Dense
print('done')
```

```
batch_size = 256
epochs = 100
latent_dim = 256

Encoder_inputs = keras.layers.Input(shape=(None,
num_Encoder_tokens))
```

```
Encoder = keras.layers.LSTM(latent_dim, return_state=True)
Encoder_outputs, state_h, state_c = Encoder(Encoder_inputs)
Encoder_states = [state_h, state_c]

Decoder_inputs = keras.layers.Input(shape=(None,
num_Decoder_tokens))
Decoder_lstm = keras.layers.LSTM(latent_dim,
return_sequences=True, return_state=True)
Decoder_outputs, _, _ = Decoder_lstm(Decoder_inputs,
initial_state=Encoder_states)
Decoder_dense = keras.layers.Dense(num_Decoder_tokens,
activation='softmax')
Decoder_outputs = Decoder_dense(Decoder_outputs)
```

```
model = keras.Model(inputs=[Encoder_inputs, Decoder_inputs],
outputs=Decoder_outputs)
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy')
print(model.summary())
```

3.3.11

The last step in this phase is to train the model. We will train on 80% of the data and the remaining 20% will be used to evaluate the model. We can then save the created model using the **save()** function.

```
model.fit([Encoder_input_data, Decoder_input_data],
Decoder_target_data, batch_size=batch_size, epochs=epochs,
validation_split=0.2)
model.save('translation_slovak_to_english.h5')
```

3.4 Machine translation - model deployment

3.4.1

Once we have created our model we need to test and deploy it. To do this we need to create a few more functions to ensure that we can send the input sequence to the encoder and retrieve the initial state of the decoder. We then send the start token and initial state to the decoder to get the next output character. Then we add the predicted output character to the sequence and repeat this process until we receive the end token or reach the maximum number of predicted characters.

```
Encoder_model = Model(Encoder_inputs, Encoder_states)

Decoder_state_input_c = Input(shape=(latent_dim,))
Decoder_state_input_h = Input(shape=(latent_dim,))
Decoder_states_inputs = [Decoder_state_input_h,
Decoder_state_input_c]

Decoder_outputs, state_h, state_c =
Decoder_lstm(Decoder_inputs,
initial_state=Decoder_states_inputs)
Decoder_states = [state_h, state_c]
Decoder_outputs = Decoder_dense(Decoder_outputs)

Decoder_model = Model([Decoder_inputs] +
Decoder_states_inputs, [Decoder_outputs] + Decoder_states)
```

3.4.2

In the next step let's create a **decode_sequence()** function that will use the encoder-decoder model we created.

```
def decode_sequence(input_seq):
  states_value = Encoder_model.predict(input_seq)
  target_seq = np.zeros((1, 1, num_Decoder_tokens))
  target_seq[0, 0, target_char_to_idx['\t']] = 1.
  stop_condition = False
  decoded_sentence = ''
  while not stop_condition:
    output_tokens, h, c = Decoder_model.predict([target_seq]+
states_value)
    sampled_token_index = np.argmax(output_tokens[0, -1, :])
    sampled_char = target_idx_to_char[sampled_token_index]
    decoded_sentence += sampled_char
    if (sampled_char == '\n' or len(decoded_sentence) >
max_Decoder_seq_length):
      stop_condition = True
    target_seq = np.zeros((1, 1, num_Decoder_tokens))
    target_seq[0, 0, sampled_token_index] = 1.
    states_value = [h, c]
  return decoded_sentence
```

3.4.3

Finally, we can create a **decode()** function whose parameter is the index of a sentence from the data file.

```
def decode(seq_index):
  input_seq = Encoder_input_data[seq_index: seq_index + 1]
  decoded_sentence = decode_sequence(input_seq)
  print('-')
  print('Input sentence:', input_dataset[seq_index])
  print('Decoded sentence:', decoded_sentence)
```

Machine Translation Evaluation

Chapter 4

4.1 Basics of evaluation

4.1.1

Language plays an essential role in how we interact with the world around us. Yet few people understand what good translation requires. The sign of a good translation is the expression of ideas and emotions in the target language as they were expressed in the source and in such a way that the target text is not a simplistic imitation of the source text. Common terms we encounter or have encountered include the following:

- **source text** - the text to be translated,
- **the target text** - the text translated into the desired language,
- **hypothesis** - a machine-generated translation,
- **reference** - human translation,
- **MT** (Machine Translation) - machine translation.

4.1.2

When evaluating a translation (machine or human) we may be interested in, for example, only the comprehension of the text without previous error analysis or on the contrary an in-depth analysis of the errors detected. In most cases, a detailed error analysis is carried out for the purpose of identifying the weaknesses and strengths of machine translation systems or of the translators themselves. The quality of a translation depends on various factors. It may be influenced by the subjective view of the translation by the human translator or by his/her experience and the area from which the source text originates. The relationship between the source text, the target text and their respective textual forms may also play a role.

4.1.3

Evaluating the quality of a translation is very difficult. There are different conceptions of quality and different perspectives on the neatness of a translation, opinions on what is acceptable and what is not, as well as a wide range of translation approaches, resulting in ambiguity in defining criteria for evaluating a translation.

Models and methods for evaluating the quality of translation, human or machine, require the inclusion of human assessors in the evaluation, either to identify errors or to determine different kinds of linguistic or functional equivalence. This requires the unconditional inclusion of the understanding of the translation as well as the assessment of its correspondence with the original which actually reflects the clarity and fidelity of the translation. Both attributes are also very important for the evaluation of machine translation.

4.1.4

The quality of machine translation can be evaluated either manually or automatically. Manual evaluation is mainly based on two factors: **accuracy** and **precision** of the translation. The disadvantage of manual evaluation is that it requires an expert (ideally a professional translator) to evaluate the translation. The aim is to assess how far the translation expresses the same things as the source text and how far the context of the original text is preserved. Both categories have a scale of 5 points. The disadvantage of manual evaluation is the time-consuming nature of the process and also the possible subjectivity of the evaluator. For this reason, automatic evaluation metrics are nowadays mainly used to evaluate the quality of a translation.

4.1.5

Machine translation automatic evaluation metrics have the advantage of being non-subjective and less time-consuming. However, they have the disadvantage that they need one or more reference translations to work with which determines how good the translation is. The more reference translations available, the more accurate the assessment but in practice, it is often the case that only one reference translation is available. Nowadays there are a large number of automatic metrics that are based on different approaches. We will present the most relevant ones.

4.2 Automatic evaluation metrics

4.2.1

The basic metrics of automatic evaluation include:

- **precision,**
- **recall,**
- **F-measure.**

The above metrics are applied to the results of binary classification which classifies the input data into two classes (positivity, and negativity) with the classification result taking the value 1 (true) or 0 (false). All other automatic evaluation metrics are based on these metrics.

4.2.2

The precision determines the ratio of true (TP) and false positives (FP) which is calculated as:

$$\text{Precision} = \frac{TP}{(TP + FP)}$$

```
def my_precision(ref, hyp):
  correct = 0
  lengthO = len(hyp)
  my_ref = ref.copy()
  for i in range(0,len(hyp)):
    for j in range(0,len(my_ref)):
      if hyp[i]==my_ref[j]:
        correct += 1
        my_ref.remove(my_ref[j])
        break
  return float(correct/lengthO)
```

4.2.3

Recall focuses on identifying true positive cases. It determines the ratio of true positives (TP) and false negatives (FN) results, i.e. all relevant cases in the dataset. Recall closely relates to precision, if precision increases, recall decreases and vice versa.

We calculate the recall metric as:

$$recall = \frac{TP}{TP + FN}$$

```
def my_recall(ref, hyp):
  correct = 0
  lengthR = len(ref)
  my_ref = ref.copy()
  for i in range(0,len(hyp)):
    for j in range(0,len(my_ref)):
      if hyp[i]==my_ref[j]:
        correct += 1
        my_ref.remove(my_ref[j])
      break
  return float(correct/lengthR)
```

4.2.4

The F-measure (*F-score*) combines the metrics of precision and recall and determines their harmonic mean. Unlike the simple average, the harmonic average penalizes outliers ensuring equal weight for both metrics.

The F-measure metric is calculated as:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

```
def my_f_measure(ref, hyp):
  prec = float(my_precision(ref, hyp))
  rec = float(my_recall(ref, hyp))
  try:
    res = (prec*rec)/((prec+rec)/2)
  except:
    res = NaN
  return res
```

4.2.5

The most well-known metric for automatic machine translation evaluation is the **BLEU** (*Bilingual Evaluation Understudy*) metric, which measures *precision*, i.e., how many words (and/or n-grams) in the machine-generated translations appeared in the human reference translations. In other words, it claims that the closer a machine translation is to a professional human translation the better it is.

The issue can be that every translator has a different vocabulary and a different way of composing a sentence, so it is almost impossible to get identical translations. One way of comparing translations is at the level of so-called **n-grams**. This is a sequence of *1, 2, ..., n* words, i.e. unigrams, bigrams,..., n-grams. The correspondence of these n-grams in translations is characterized by the so-called **n-gram precision**, which can be calculated for individual n-grams or even for the whole text.

Ideally, the length of the candidate translation would be equal to the length of the reference translation. Otherwise:

- if a candidate translation is created that is too long, we penalize it using the **modified n-gram precision**,
- if too short a translation is produced, a **brevity penalty** (BP) is applied.

4.2.6

We can use the most popular natural language processing library, **nltk**, to compute the BLEU metric score. However, before we can compute the BLEU score we need to tokenize the reference text and the hypothesis. This means breaking the sentence into tokens, i.e. word units. We will also use a function from the nltk library to do this, namely **word_tokenize()**. We can then call the **sentence_bleu()** function, which will return the score for the BLEU metric. By setting the weights parameters, we can tell the function for which n-gram we want to calculate the BLEU. The closer the BLEU score is to 1, the better, and more correct the translation. Conversely, a value closer to 0 indicates a poor translation.

```
from nltk.translate.bleu_score import sentence_bleu
from nltk import word_tokenize

ref = "Computer science spans theoretical disciplines (such as
algorithms, theory of computation, information theory, and
automation) to practical disciplines (including the design and
implementation of hardware and software)."
hyp = "Computer science includes theoretical disciplines (such
as algorithms, theory of computing, theoretical computer
science and automation) and practical disciplines (including
hardware and software design and implementation)."

ref = word_tokenize(ref)
hyp = word_tokenize(hyp)

print('BLEU-1:',sentence_bleu([ref], hyp, weights=(1,0,0,0)))
print('BLEU-2:',sentence_bleu([ref], hyp,
weights=(0.5,0.5,0,0)))
print('BLEU-3:',sentence_bleu([ref], hyp,
weights=(0.33,0.33,0.33,0)))
print('BLEU-4:',sentence_bleu([ref], hyp,
weights=(0.25,0.25,0.25,0.25)))
```

Program output:

```
BLEU-1: 0.7700677691930449
BLEU-2: 0.656003030651081
BLEU-3: 0.5422370790508815
BLEU-4: 0.43076614970957955
```

PRISCILLA

priscilla.fitped.eu