M.Sc. Computerlinguistik
Center for information and language processing (CIS)
Faculty of language and literature sciences
Ludwig-Maximiliams-Universität München

Master thesis

This is the still unknown title of my master thesis

Ane Berasategi
Matriculation number: 12006250

Supervisor: Masoud Jalil Sabet
Supervisor: Prof Dr.Hinrich Schütze
Submitted on: never

Version 1, 25 June 2020

Abstract

An *abstract* is a brief summary of a research article, thesis, review, conference proceeding or any in-depth analysis of a particular subject or discipline, and is often used to help the reader quickly ascertain the paper's purpose. When used, an abstract always appears at the beginning of a manuscript, acting as the point-of-entry for any given academic paper or patent application. Abstracting and indexing services for various academic disciplines are aimed at compiling a body of literature for that particular subject.

-

Task of the Thesis in the Original:

Declaration by the candidate

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been marked.

The work has not been presented in the same or a similar form to any other testing authority and has not been made public.

I hereby also entitle a right of use (free of charge, not limited locally and for an indefinite period of time) that my thesis can be duplicated, saved and archived by the Otto von Guericke University of Magdeburg (OvGU) or any commissioned third party (e. g. *iParadigms Europe Limited*, provider of the plagiarism-detection service "Turnitin") exclusively in order to check it for plagiarism and to optimize the appraisal of results.

Magdeburg, June 30, 2020

Contents

List of Acronyms

NLP Natural language processing

List of Figures

List of Tables

1 Introduction

The introduction should present the topic of the thesis to specify the purpose and importance of the work. Other possible contents of an introduction are described in section 1 on page 5.

2 Goals of the thesis

no goals whatsoever

3 Tokenization

3.1 Introduction

Tokenization is the first major step in language processing. The main idea is simplifying or compressing the input text into meaningful units, called tokens, creating a big vocabulary of tokens and a shorter sequence. [1]

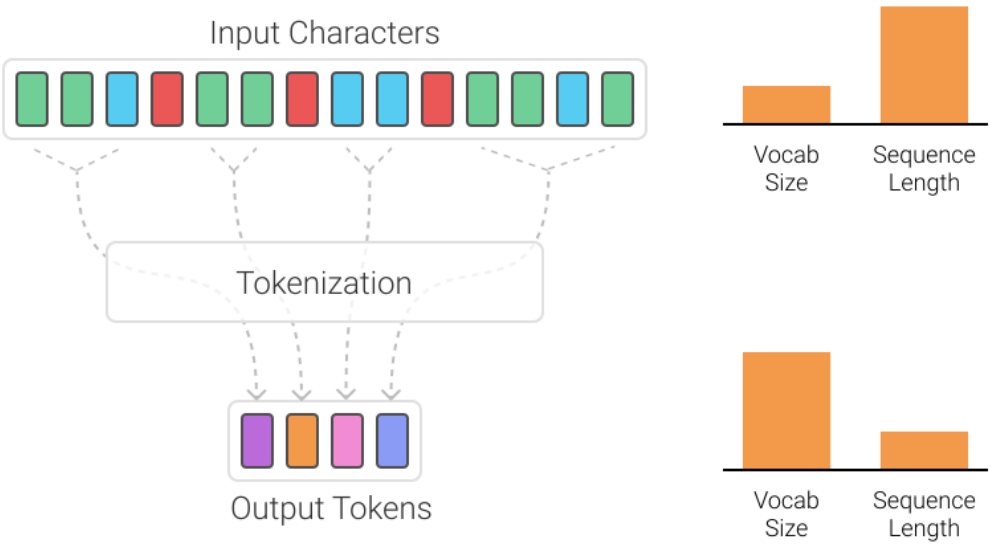

Figure 3.1: Tokenization of a sequence of text

Tokens in language are defined as such units which have a semantic meaning, be it words, phrases, symbols or other elements. Here is an example of a simple way to tokenize text:

> Raw text: I ate a burger today, and it was good.
> Text after tokenization: ['I', 'ate', 'a', 'burger', 'today', 'and', 'it', 'was', 'good']

In this example, the way to obtain tokens looks simple: just locate word boundaries, split by whitespace and get the symbols, and remove the punctuation marks, since punctuation marks have no definite meaning. However, it's not always that easy. For example, some punctuation marks are relevant to the meaning of the words around them.

How to deal with punctuation marks?

Each language has its tricky cases, for example in English there are possession and contractions, words like *aren't*, *Sarah's* and *O'Neill*. Because of this, it is imperative to

know the language of the text to be tokenized. *Language identification* is the task of identifying the language of the input text. From the initial k-gram algorithms used in cryptography (Konnheim, 1981), to more modern n-gram methods (Dunning, 1994) have been used. Once the language is known, we can follow the rules for each case and deal with punctuation marks appropriately.

Other types of tokens

In the simple example above, tokens were words. Additionally, tokens can be groups of words, characters or subwords (parts of a word). For example, take the word *smarter*:

- Sentence: the smarter computer
- Word tokens: the, smarter, computer
- Character tokens: t, h, e, s, m, a, r, t, e, r, c, o, m, p, u, t, e, r
- Subword tokens: the, smart, er, comput, er
- Subword tokens without word boundaries: the smart, er comput, er

The major question in the tokenization phase is: **what are the correct tokens to use?**. The following section explores these 4 types of tokenization methods and delves into the algorithms and code libraries available.

3.2 Tokenization algorithm types

The way to tokenize heavily depends on the task afterwards. Different applications might require different tokenization algorithms. Nowadays, most deep learning architectures in NLP process the raw text at the token level and as a first step, create embeddings for these tokens, which will be explained in more detail in the following section. In short, *the type of tokenization depends on the type of embedding*. There are several tokenization methods, explained further in the following sections, and each has its advantages and drawbacks.

3.2.1 Word level tokenization

Word level tokenization is the first type and most basic form of tokenization, and is also the most common. It splits a piece of text into individual words based on word boundaries, usually a specific delimiter, mostly whitespace ' ' or other punctuation signs.

Conceptually, splitting on whitespace can also split an element which should be regarded as a single token, for example New York. This is mostly the case with names, borrowed foreign phrases, and compounds that are sometimes written as multiple words. Tokenization without word boundaries aims to address that problem. 3.2.4 on page 16

Word level algorithms

The most simple way to obtain word level tokenization is by splitting the sentence on the desired delimeter, whitespace usually. The `sentence.split()` function in Python or a Regex command `re.findall("[\w']+", text)` achieve this in a simple way.

The natural language toolkit (NLTK) in Python provides a tokenize package which includes a *word_tokenize* function. The user can give in the language of the text; if none is given, English is taken as default.

```
from nltk.tokenize import word_tokenize
sentence = u'I spent $2 yesterday'
sentence_tokenized = word_tokenize(sentence, language='English')
>>> sentence_tokenized = ['I', 'spent', '$', '2', 'yesterday']
```

Similarly, SpaCy offers a similar functionality. It is possible to load the language model for different languages and model size. In this case, the language is English and size 'sm', small.

```
import spacy
sp = spacy.load('en_core_web_sm')
sentence = u'I spent $2 yesterday'
sentence_tokenized = sp(sentence)
>>> sentence_tokenized = ['I', 'spent', '$', '2', 'yesterday']
```

Keras also offers a similar functionality:

```
from keras.preprocessing.text import text_to_word_sequence
sentence_tokenized = text_to_word_sequence(sentence)
```

As does Gensim:

```
from gensim.utils import tokenize
sentence_tokenized = list(tokenize(sentence))
```

Depending on the applications and the frameworks the user is making use of, they might favor one algorithm over the other.

Word embeddings

As stated before, the goal of tokenization is to split the text into units with meaning. Typically, each token is assigned an embedding vector: word2vec (Mikolov et al., 2013 [2]) is a way of transforming a word into a fixed-size vector representation, as shown in the picture below.

As well as word2vec, there are other word embedding algorithms such as GloVe or fasttext. If viewed abstractly in its N dimensions, each word's representation is close to the representation of similar words. As a simple example with made up embedding numbers:

- Word: smart. Embedding: [2, 3, 1, 4]

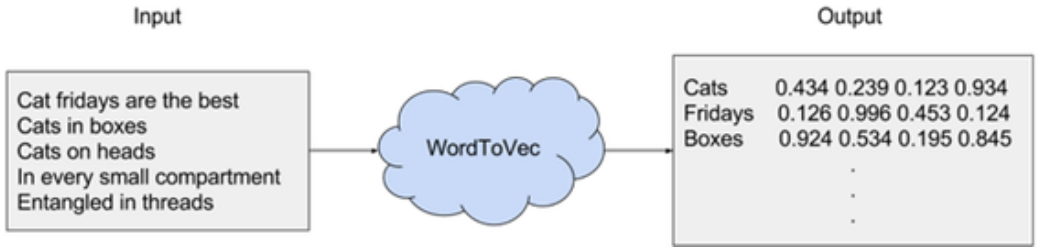

Figure 3.2: Representation of word embeddings

- Word: intelligent. Embedding: [2, 3, 2, 3]
- Word: stupid. Embedding: [-2, -4, -1, -3]

For example, the embeddings of *smart* and *intelligent* have a distance of 2, since the last 2 numbers in the vector differ by one respectively. If this was plotted in a 4 dimensional space, these words would be very close together. On the other hand, *stupid* is almost the opposite of *smart*. The distance in this case is much higher. In the plot, these words would be roughly in the opposite direction. Thus, with word embeddings, a sentence is transformed into a sequence of embedding vectors, which is very useful for NLP tasks.

Word level tokenization drawbacks

Word embeddings have some drawbacks. In many cases, a word can have more than one meaning: *well*, for example, can be used in these 2 scenarios.

> I'm doing quite well.
> The well was full of water.

In the first case, well is an adverb while in the second it's a noun. *well*'s embedding will probably be a mixture of the two, since word embeddings don't generalize to textbfhomonyms. But it won't truly represent any of the two meanings.

Another drawback is that word embeddings aren't well equipped to deal with **out of vocabulary (oov) words**. Word embeddings are created with a certain vocabulary size, that is, a certain number of words are known to the system. If afterwards a new word arrives which isn't present in the vocabulary, because it's a foreign word, or a misspelled word, it will be given an unknown <UNK> embedding, that will be the same for all unknown words. Therefore all unknown words have the same embedding, that is, the NLP task will treat all these words as if they had the same meaning. The information within these words is lost due to the mapping from OOV to UNK.

Another issue with word tokens is related to the **vocabulary size**. Generally, pre-trained models are trained on a large volume of the text corpus. As such, if the vocabulary is built with all the unique words in such a large corpus, it creates a huge vocabulary. This

opens the door to *character tokenization*, since in this case the vocabulary depends on the number of characters, which is significantly lower than the number of all different words.

These problems are not to be mistaken with tokenization problems, tokenization is merely a way to an ends. In most cases however, they're used to create embeddings. And if embeddings from word tokens have drawbacks, the tokenization method is changed in order to create different tokens, in order to create other types of embeddings.

3.2.2 Character level tokenization

In this type of tokenization, instead of splitting a text into words, the splitting is done into characters, whereby *smarter* becomes *s-m-a-r-t-e-r* for instance. Karpathy, 2015 was the first to introduce a character level language model.

OOV words, misspellings or rare words are handled better, since they're broken down into characters and these characters are usually known in the vocabulary. In addition, the size of the vocabulary is significantly lower, namely 26 in the simple case where only the English characters are considered, though one might as well include all ASCII characters. Zhang et al. (2015) [3], who introduced the character CNN, consider all the alphanumeric character, in addition to puctuation marks and some special symbols.

Character level models are unrestricted in their vocabulary and see the input "as-is". Since the vocabulary is much lower, the model's performance is much better than in the word tokens case. Tokenizing sequences at the character level has shown some impressive results, as stated below.

Radfor et al. (2017) [4] from OpenAI showed that character level models can capture the semantic properties of text. Kalchbrenner et al.(2016) [5] from Deepmind and Leet et al. (2017) [6] both demonstrated translation at the character level. These are particularly compelling results as the task of translation captures the semantic understanding of the underlying text.

Character level algorithms

The previous libraries explored in the case of word tokenization (native python libraries, nltk, spacy, keras) have their own version for character level tokenization.

Character level tokenization drawbacks

When tokenizing a text at the character level, the sequences are longer, which takes longer to compute since the neural network needs to have significantly more parameters to allow the model to perform the conceptual grouping internally, instead of being handed the groups from the beginning.

It becomes challenging to learn the relationship between the characters to form meaningful words and, given that there is no semantic information among characters, characters

are semantically void. Which makes it complicated to generate character mbeddings.

Sometimes the NLP task doesn't need processing at the character level, such as when doing a sequence tagging task or name entity recognition, the character level model will output characters, which requires post processing.

As an in-betweener between word and character tokenization, subword tokenization produces subword units, smaller than words but bigger than just characters.

3.2.3 Subword level tokenization

Subword tokenization is the task of splitting the text into subwords or n-gram characters. For example, words like *lower* can be segmented as *low-er*, *smartest* as *smart-est*, and so on. In the event of an OOV word such as *eorner*, this tokenizer will divide it into *eorn-er* and effectivey obtain some semantic information. Very common subwords such as *ing*, *ion*, usually with a morphological sense, are learnt through repetition. The word *unfriendly* would be split into *un-friend-ly*.

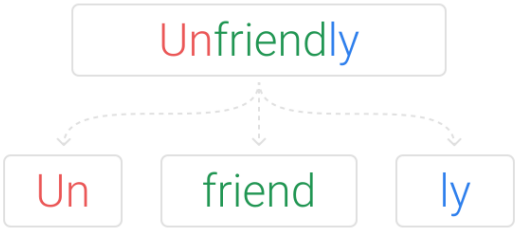

Figure 3.3: Representation of the word 'unfriendly' in subword units

At the time of writing (2020), the most powerful deep learning architectures are based on Transformers (Vaswani et al., 2017 [7]), and these rely on subword tokenization algorithms to prepare the vocabulary.

Subword level algorithms

Since Transformers are a relatively architecture in 2020, subword tokenization is an active area of research. Nowadays four algorithms stand out: byte-pair encoding (BPE), unigram LM, WordPiece and SentencePiece.

Since BPE is the basis of the thesis, it will be explained in depth in the following section. A simple explanation of BPE and the rest of the algorithms follow below.

Huggingface, an open source NLP company, released Transformers and Tokenizers (Wolf et al., 2019 [8]), two popular NLP framework which include several subword tokenizers such as *ByteLevelBPETokenizer*, *CharBPETokenizer*, *SentencePieceBPETokenizer* and *BertWordPieceTokenizer*. The first refer to the first subword level algorithm, BPE, in addition to WordPiece and SentencePiece.

BPE

BPE (Sennrich et al., 2016 [9]) merges the most frequently occurring character or character sequences iteratively. This is roughly how the algorithm works:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Split word to sequence of characters and append a special token showing the beginning-of-word or end-of-word affix/suffix respectively.
4. Calculate pairs of sequences in the text and their frequencies. For example, ('t', 'h') has frequency X, ('h', 'e') has frequency Y.
5. Generate a new subword according to the pairs of sequences that occurs most frequently. For example, if ('t', 'h') has the highest frequency in the set of pairs, the new subword unit would become 'th'.
6. Repeat from step 3 until reaching subword vocabulary size (defined in step 2) or the next highest frequency pair is 1. Following the example, ('t', 'h') would be replaced by 'th' in the corpus, the pairs calculated again, the most frequent pair obtained again, and merged again.

BPE is based on a greedy and deterministic symbol replacement, and can't provide multiple segmentations.

Unigram LM

Unigram language modeling (Kubo, 2018 [10]) is based on the assumption that all subword occurrences are independent and therefore that subword sequences are produced by the product of subword occurrence probabilities. These are the steps of the algorithm:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Optimize the probability of word occurrence by giving a word sequence.
4. Compute the loss of each subword.
5. Sort the symbol by loss and keep top X % of word (X=80% for example). To avoid oov instances, character level is recommend to be included as a subset of subwords.
6. Repeat step 3–5 until reaching the subword vocabulary size (defined in step 2) or no change (step 5).

Kudo argues that the unigram LM model is more flexible than BPE because it's based on a probabilistic LM and can output multiple segmentations with their probabilities.

WordPiece

WordPiece (Schuster and Nakajima, 2012 [11]) was initially used to solve Japanese and Korean voice problem. It's similar to BPE in many ways, except that it forms a new subword based on likelihood, not on the next highest frequency pair. These are the steps of the algorithm:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Split word to sequence of characters.
4. Initialize the vocabulary with all the characters in the text.
5. Build a language model based on the vocabulary.
6. Generate a new subword unit by combining two units out of the current vocabulary to increment the vocabulary by one. Choose the new subword unit out of all the possible ones that increases the likelihood on the training data the most when added to the model.
7. Repeat step 5 until reaching subword vocabulary size (defined in step 2) or the likelihood increase falls below a certain threshold.

WordPiece and BPE only differ in step 6, since BPE merges the token combination that has the maximum frequency. This frequency stems from the combination of the tokens, not previous individual tokens. In WordPiece, the frequency of the two tokens separatedly is also taken into account. If there are 2 tokens A and B, the score of this combination will be the following

Score(A,B) = Frequency(A,B) / Frequency(A) * Frequency(B)

The token pair with the highest score will be selected. It might be the case that Frequency('so', 'on') is very high but their separate frequencies are also high, hence if using WordPiece, 'soon' won't be merged as the overall score is low. As another example, if Frequency('Jag','gery') might be low but if their separate frequencies are also low, 'Jag' and 'gery' might be joined to form 'Jaggery'.

BERT (Devlin et al., 2018 [12]) uses WordPiece as its tokenization method, yet the precise tokenization algorithm and/or code has not been done public. This example shows the tokenization step and how it handles OOV words.

original tokens = ["John", "Johanson", "'s", "house"] bert tokens = ["[CLS]", "john", "johan", "##son", "'", "s", "house", "[SEP]"]

SentencePiece

SentencePiece (Kudo et al. 2018 [13]) is a subword tokenization type that has an extensive Github repository with freely available code.

As the repository states, it's an unsupervised text tokenizer and detokenizer where the vocabulary size is predetermined prior to the neural model training. It implements subword units (e.g., BPE 3.2.3) and unigram LM 3.2.3) with the extension of direct training from raw sentences. It doesn't depend on language-specific pre or postprocessing.

While conceptually similar to BPE, it doesn't use the greedy encoding strategy, whereby it achieves higher quality tokenization and reduces error induced by location-dependence as seen in BPE. SentencePiece sees ambiguity in character grouping as a source of regularization for downstream models during training, and uses a simple language model to evaluate the most likely character groupings instead of greedily picking the longest recognized strings like BPE does.

Regarding ambiguity in text as a regularization parameter for downstream models results in very high quality tokenization, but comes at great cost to performance, at times making it the slowest part of an NLP pipeline. While the assumption of ambiguity in tokenization seems natural, it appears the performance trade-off is not worth it, as Google itself opted not to use this strategy in their BERT language model. 3.2.3

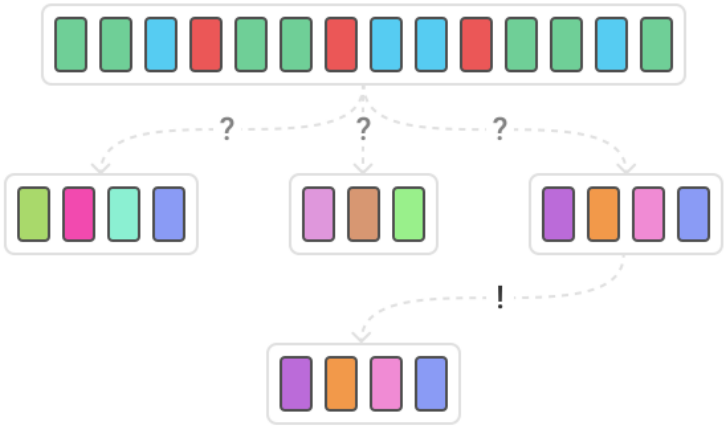

Figure 3.4: Representation of the SentencePiece tokenization in a sequence of text

The number of unique tokens in SentencePiece is predetermined, the segmentation model is trained such that the final vocabulary size is fixed, e.g., 8k, 16k, or 32k. This is different from BPE (Sennrich et al., 2015 [9]) which uses the number of merge operations instead. The number of merge operations is a BPE-specific parameter and not applicable to other segmentation algorithms, including unigram, word and character level algorithms.

3.2.4 Tokenization without word boundaries

Another type of tokenization, beyond word, character or subword, is tokenization without word boundaries. The three types of tokenization explored until now cannot create units among words, that is, they consider words separatedly.

When dealing with languages that don't include space tokenization, such as several Asian languages, an individual symbol can resemble a syllable rather than a word or letter. Most words are short (the most common length is 2 characters), and given the lack of standardization of word breaking in the writing system or lack of punctuation in certain languages, it is not always clear where word boundaries should be placed. As an example, in English:

> Input sentence: the smarter computer
> Subword tokens without word boundaries: the smart, er comput, er

An approach to handle this has been to abandon word-based indexing, and do all indexing from just short subsequences of characters (character n-grams), regardless of whether particular sequences cross word boundaries or not. Hence, at times, each character used is taken as a token in Chinese tokenization.

3.3 BPE

Byte Pair Encoding (BPE) (Sennrich et al., 2015 [9]), is a widely used tokenization method among Transformer-based models. The code is open source and there is an active repository on Github. It merges the most frequently occurring character or character sequences iteratively.

BPE enables the encoding of rare or OOV words with appropriate subword tokenization without introducing any 'unknown' tokens. Regarding the longer sequence length which was one of the drawbacks of character tokenization, in this case the length of the sentences after BPE are shorter compared to character tokenization.

3.3.1 Minimal algorithm to learn BPE segmentations

Subsection 3.2.3 showed a simple algorithm to build subword units. In this section it will be explained in depth with an example. These are the steps of the algorithm:

1. Get a large enough corpus.
2. Define a desired subword vocabulary size.
3. Split word to sequence of characters and append a special token showing the beginning-of-word or end-of-word affix/suffix respectively.

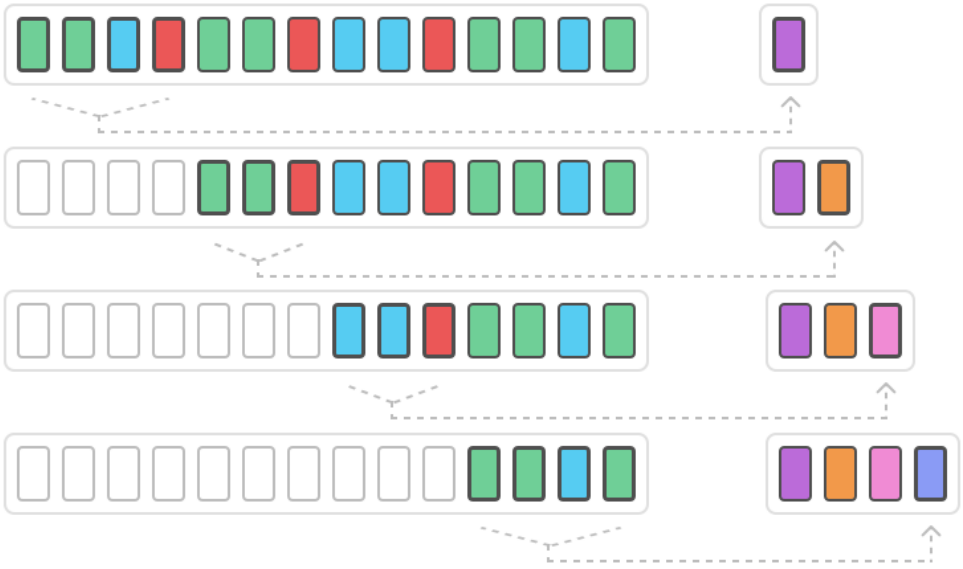

Figure 3.5: Representation of the BPE tokenization in a sequence of text

4. Calculate pairs of sequences in the text and their frequencies.
5. Generate a new subword according to the pairs of sequences that occurs most frequently, and save it to the vocabulary.
6. Merge the most frequent pair in corpus.
7. Repeat from step 4 until reaching subword vocabulary size (defined in step 2) or the next highest frequency pair is 1.

Let's consider a simple corpus with a single line, with the desired subword vocabulary size=10 for example. The character '_' marks the beginning of each word. The following code shows the steps 1-3.

```
def read_corpus(corpus):
    tokens = [("_" + " ".join(token)) for token in corpus]
    return tokens

corpus = ['this is this.']
vocab_size = 10
tokens = read_corpus(corpus)
>>> tokens = ['_t h i s _i s _t h i s .']
```

Now we can calculate the pairs of characters and their frequencies, as well as the most popular pair. These are the steps 4-5 in the algorithm above.

```
from collections import Counter

def get_stats(tokens):
    pairs = Counter()
```

```
    for sent in tokens:
        for word in sent[1:].split(' _'):
            symbols = ('_' + word).split()
            for j in range(len(symbols) - 1):
                pairs[symbols[j], symbols[j+1]] += 1
    return pairs

pairs = get_stats(tokens)
>>> pairs = Counter({('_t', 'h'): 2, ('h', 'i'): 2, ('i', 's'): 2,
                     ('_i', 's'): 1, ('i', 's,'): 1, ('_', 'i'): 1,
                     ('_i', 't'): 1, ('t', '?'): 1})

most_frequent_pair = pairs.most_common(1)[0][0]
>>> most_frequent_pair = ('_t', 'h')

vocab = []
vocab.append(most_frequent_pair)
```

There we can see each bigram and its frequency. For example, ('_t', 'h') occurs twice in the corpus, and it is taken as the most frequently occurring bigram, which we can save into the merge_list. Now it's the time to merge this pair in the corpus, step 6.

```
import re

def merge_pair_in_corpus(tokens, pair):
    # convert list of sentences into one big string
    # in order to do the substitution once
    tokens = '\n'.join(tokens)

    # regex to capture the pair
    p = re.compile(r'(?<!\S)' + re.escape(' '.join(pair)) + r'(?!\S)')

    # substitute the unmerged pair by the merged pair
    tokens = p.sub(''.join(pair), tokens)

    tokens = tokens.split('\n')
    return tokens

tokens = merge_pair_in_corpus(tokens, most_frequent_pair)
>>> tokens = ['_th i s _i s _th i s .']
```

The subword unit '_th' has been created, saved in the vocabulary and merged in the corpus. The last step is iterating until the subword vocabulary size has been reached or until there are no pairs with bigger than 1. At each step, the object *pairs* is calculated again since there might be new pairs such as ('_th', 'i') in this example. The whole minimal code would look like this:

```
corpus = ['this is this.']
vocab_size = 10
vocab = []

tokens = read_corpus(corpus)

for _ in range(vocab_size):

    pairs = get_stats(tokens)

    # frequency of the most common pair is 1, break loop
    if pairs.most_common(1)[0][1] == 1:
        break

    most_frequent_pair = pairs.most_common(1)[0][0]
    vocab.append(most_frequent_pair)
    tokens = merge_pair_in_corpus(tokens, most_frequent_pair)

>>> tokens = ['_this _i s _this .']
```

In each step of the iteration, the *get_stats* function iterates all the characters in the corpus, so the complexity is O(len(corpus) * len(sent)), for an average sentence length. Obtaining the most frequent pair takes constant time, since the object *pairs* is a Counter object and has a built-in function to retrieve the most frequent item. At the step of *merge_pair_in_corpus*, the corpus is iterated in its entirety again, with a complexity of O(len(corpus) * len(sent)). Therefore, the algorithm has a complexity of O(num_merges * len(corpus) * len(sent)). The num_merges part cannot be avoided, but operating through all the characters of the corpus is very computationally expensive. One of the contributions of this thesis is an optimization of this algorithm, as will be shown in the following chapters.

3.3.2 Applying BPE to OOV words

In the event of an OOV word, such as *these*, which the corpus used in the previous example doesn't know, the BPE algorithm can create some subword units from the corpus used before.

1. Split the OOV word into characters after inserting '_' in the beginning.
2. Compute the pair of character or character sequences in the OOV word.
3. Select the pairs present in the learned operations.
4. Merge the most frequent pair.
5. Repeat steps 2-4 until merging is possible.

And this is the code in Python for such an algorithm:

```
oov = 'these'
oov = ['_' + ' '.join(list(oov))]

i = 0
while True:
    pairs = get_stats(oov)
    # find the pairs available in the vocab learnt before
    idx = [vocab.index(i) for i in pairs if i in vocab]

    if len(idx) == 0:
        print("BPE completed")
        break

    # choose the most frequent pair which appears in the OOV word
    best = merges[min(idx)]

    # merge the best pair
    oov = merge_vocab(best, oov)

>>> oov = '_th e s e'
```

_th is the only known merge in the vocabulary, the rest of the characters ('e', 's', 'e') are unknown to the vocabulary so it doesn't know how to create any subword units.

3.3.3 BPE dropout

BPE dropout (Provilkov et al., 2019 [14]) slightly changes the BPE algorithm by stochastically corrupting the segmentation procedure of BPE, producing multiple segmentations within the same fixed BPE framework.

It exploits the innate ability of BPE to be stochastic: the merge table remains the same, but when applying it to the corpus, at each merge step some merges are randomly dropped with probability p, hence the name of BPE dropout. In the paper they use p=0.1 during training and p=0 during inference. For Chinese and Japanese, they use p=0.6 to match the increase in length of segmented sentences for other languages.

The paper hypothesize that exposing a model to different segmentations might result in better understanding of the whole words as well as their subword units. The improvement with respect to normal BPE are consistent no matter the vocabulary size, but it's shown that the effect from using BPE-Dropout vanishes when a corpora size gets bigger.

These results are replicated and confirmed in later chapters.

Sentences segmented with BPE-Dropout are longer. There's a danger that models trained with BPE-Dropout might use more fine-grained segmentation in inference and hence slow inference down.

3.3.4 BPE drawbacks

Kudo (2018) [10] showed that BPE is a **greedy algorithm** that keeps the most frequent words intact, while splitting the rare ones into multiple tokens. BPE splits words into unique sequences, meaning that for each word, a model observes **only one segmentation**, meaning that if there's a segmentation error, all the following steps are erroneous. Besides, subwords into which rare words are segmented end up poorly understood.

The problem of unique segmentation was solved by BPE Dropout [14], which produces several. The other problem about BPE being greedy and fragile regarding segmentation errors is explored in the following chapters.

4 Translation

1. NMT? open vocabulary problems? 2. https://arxiv.org/abs/2004.08728 section 5 3. fastalign, eflomal

5 Methodology

Theoretical idea: what I want to do, how, algorithmic/mathematical

This chapter explains the technical content of this thesis in broad strokes, the methodology used and the general idea of the methods employed. For a more in-depth analysis, refer to the next chapter.

This thesis, first of all, aims to replicate the results of BPE and BPE dropout.

5.1 Replication of BPE results

Using Sennrich et al.'s [9] https://github.com/rsennrich/subword-nmt/, the first goal was to check the gold standard's alignment scores. For that, these steps were undertaken:

1. Write learn BPE from corpus algorithm
2. Write apply BPE to corpus algorithm
3. Write extract alignment script to align files from 2 different languages using fastalign/eflomal
4. Write a subword-word alignment script, since fastalign's output are subword alignments and we need word alignments
5. calculate alignment scores

The actual code for each step can be found in the next section, Development.

The corpus is a 10k sentence English-German corpus, containing an index number for each sentence. As an excerpt of the corpora:

> English
> 21 The Committee on Transport and Tourism has adopted four amendments for the second reading .
> 22 They will certainly enhance the feeling of the right of movement by EU citizens and they will also certainly benefit disabled drivers .
> 23 The initial Commission proposal was adopted unamended by Parliament on first reading .
>
> German

21 Der Transportausschuß hat für die zweite Lesung vier Änderungsanträge beschlossen .
22 Sie werden bei den EU-Bürgern gewiß das Gefühl für das Recht auf Freizügigkeit stärken , und sie werden gewiß auch behinderten Fahrern Vorteile bringen .
23 Der ursprüngliche Vorschlag der Kommission wurde vom Parlament in erster Lesung ohne Änderungen verabschiedet .

5.1.1 Learn BPE algorithm

Sennrich's repository's code has some additional parameters that weren't relevant for a minimal implementation of the BPE algorithm, so the script was adapted. These are the steps for a minimal learn BPE algorithm:

1. Read corpus into tokens, parse index.
2. Count pair frequencies.
3. Start loop from 1 until desired vocabulary size. In our case, 10k merges.
   a) Get most frequent pair.
   b) Append most frequent pair to vocabulary.
   c) Merge pair in corpus.
   d) Count pair frequencies in corpus.
4. Write vocabulary to a file.

This step of the pipeline only has to be done once for each corpus, afterwards the vocabulary can be used in different ways. But this minimal algorithm, since it has to count all the pairs in the whole corpus in each iteration, takes a long time. An optimization came after this.

5.1.2 Apply BPE algorithm

Once the vocabulary has been learnt, it can be applied to a corpus. In this case, we use the same corpus for training and for applying. To generate different output files, different num_merges are declared. For example, for 500 merges, only the first 500 merges of the vocabulary are considered, and there aren't many recognizable BPE units in the corpus. For bigger merge values, more and more subword units get merged. In this thesis, the following merges have been considered: [100, 200, 500, 1000, 2000, 4000, 6000, 8000]. These are the steps for this part:

1. Load data, corpus and BPE vocabulary.

2. Start loop for all numbers of merges.
    a) Start loop from 1 until desired amount of merges.
        i. Merge corpus for current most frequent pair.
    b) Write to output.bpe file.

For example, this is what the excerpt from the corpora above look like after 100 merges:

English
_the _Comm it t e e _on _T ran s p ort _and _T ouris m _has _a d o p t ed _f our _a m end ment s _for _the _sec ond _re a d ing _.
_the y _w ill _cer t a in ly _en h an ce _the _f e e ling _of _the _ri g h t _of _m o ve ment _b y _E U _citi z en s _and _the y _w ill _al s o _cer t a in ly _ben e f it _d is a bled _d ri ver s _.
_the _in iti al _Commission _pro p o s al _w as _a d o p t ed _u n a m end ed _b y _P ar li a ment _on _f ir st _re a d ing _.

German
_der _T rans p or t a ussch u ß _h at _für _die _z w eite _L es ung _v ier _Ä nder ung s ant r ä ge _besch l o ss en _.
_s ie _wer den _bei _den _E U - B ür ger n _ge w i ß _das _G e f ü h l _für _das _Re ch t _auf _F r ei z ü g i g k eit _st ä r k en _, _und _s ie _wer den _ge w i ß _au ch _beh inder ten _F a hr er n _V or tei l e _b r ingen _.
_der _ur s p r ü ng lich e _V or sch la g _der _Ko mm iss ion _w ur de _v o m _P ar la ment _in _er ster _L es ung _o h n e _Ä nder ungen _vera b sch ie det _.

Only the 100 most common units in the language have been merged, so in English we can see the merge of *_the*, *_and*, *ment*, and other very common words and affix/suffixes. As for German, we can see very common words being merged such as *_die*, *_das*, *_bei* and so on. When we do 4000 merges:

English
_the _Committee _on _Transport _and _Tourism _has _adopted _four _amendments _for _the _second _reading _.
_they _will _certainly _enhance _the _feeling _of _the _right _of _movement _by _EU _citizens _and _they _will _also _certainly _benefit _disabled _drivers _.
_the _initial _Commission _proposal _was _adopted _un amended _by _Parliament _on _first _reading _.

German
_der _T ransp ortausschuß _hat _für _die _zweite _Lesung _vier _Änderungsanträge _beschlossen _.
_sie _werden _bei _den _EU-B ür gern _gew iß _das _Ge fü hl _für _das _Recht _auf _Freizügigkeit _stärken _, _und _sie _werden _gew iß _auch _behinderten _Fahrern _Vorteile _bringen _.
_der _ursprüngliche _Vorschlag _der _Kommission _wurde _vom _Parlament _in _erster _Lesung _ohne _Änderungen _verabschiedet _.

Most words are merged, except _*dis abled* in English, and _*T ransp ortausschuß* in German for instance.

5.1.3 Obtain alignments from fastalign or eflomal

a

1.

6 Development

This chapter talks more deeply about the code and algorithms previously explained in the Methodology section. 5

6.1 Coding practices

The parameters for the pipeline, such as num_symbols, dropout, file paths, etc. have been written in *settings.py*.

```
# global variables
import os
from os.path import join
import sys

word_sep = u'\u2581'
source, target = 'eng', 'deu'

num_all_symbols = 20000
all_symbols = [100, 200, 500, 1000, 2000, 4000, 6000, 8000]

rootdir = os.getcwd()
if rootdir.split(os.sep)[-1] == 'src':
    rootdir = os.sep.join(rootdir.split(os.sep)[:-1])

datadir = join(rootdir, 'data')
inputdir = join(datadir, 'input')
bpedir = join(datadir, 'dropout_bpe' if dropout > 0 else 'normal_bpe')
baselinedir = join(rootdir, 'reports', 'scores_normal_bpe')
scoredir = join(rootdir, 'reports', 'scores_' + ('dropout_bpe' if
                                    dropout > 0 else 'normal_bpe'))
goldpath = join(inputdir, 'eng_deu.gold')
inputpath = {source: join(inputdir, source+'_with_10k.txt'),
             target: join(inputdir, target+'_with_10k.txt')}

fastalign_path = join(rootdir, "tools/fast_align/build/fast_align")
atools_path = join(rootdir, "tools/fast_align/build/atools")
```

6.2 Replication of BPE results

1. Write learn BPE from corpus algorithm

2. Write apply BPE to corpus algorithm
3. Write extract alignment script to align files from 2 different languages using fastalign/eflomal
4. Write a subword-word alignment script, since fastalign's output are subword alignments and we need word alignments
5. calculate alignment scores

6.2.1 Learn BPE algorithm

```
#!/usr/bin/env python

import os
import re
import sys
import codecs
from tqdm import tqdm
from os.path import join
from collections import defaultdict, Counter

# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], '..'))
from settings import *

def read_corpus(corpus: list) -> list:
    """
    Read corpus, strip index and new line characters.
    In space mode, each word has a word_sep symbol at the beginning to
                                        signal it's the beginning of the
                                         word.
    example:
    tokens = [
        '\_w e \_d o \_n o t \_b e l i e v e
        \_t h a t \_w e \_s h o u l d
        \_c h e r r y - p i c k \_.',
        ...
    ]
    """

    tokens = []
    for line in corpus:
        line = line.split('\t')[1].strip('\r\n ')
        line = line.split()
        line[0] = str.lower(line[0])
```

```
        # add word_sep to each beginning of word and join by space
        tokens.append(' '.join([word_sep + ' '.join(word) for word in
                                                line]))

    return tokens

def get_stats(tokens: list) -> Counter:
    """
    Count frequency of all bigrams and the frequency per index.
    pairs = {
        ('s', 'h'): 5,
        ('h', 'e'): 6
    }
    The last token '.' or word_sep. isn't merged with anything.
    """

    pairs = Counter()
    for i, sent in enumerate(tokens):
        # get stats for each word independently,
        # no bigrams between different words
        for word in sent[1:].split(' '+word_sep):
          symbols = symbols.split()
          for j in range(len(symbols) - 1):
              pairs[symbols[j], symbols[j + 1]] += 1

    return pairs


def merge_token(corpus, most_frequent):
  str_corpus = '\n'.join(corpus)
  str_corpus = str_corpus.replace(' '.join(most_frequent), ''.join(
                                    most_frequent))
  return str_corpus.split('\n')


def learn_bpe(argsinput, bpe_model):
    """
    Learn BPE operations from vocabulary.
    Steps:
    1. split corpus into characters, count frequency
    2. count bigrams in corpus
    3. merge most frequent symbols
    4. Update bigrams in corpus
    """

    corpus = read_corpus(argsinput)
```

```
    most_frequent_merges = []
    for i in range(num_all_symbols):

      pairs = get_stats(corpus)

        try:
            most_frequent = pairs.most_common(1)[0][0]
        except:
            # pairs is empty
            break

        most_frequent_merges.append(most_frequent)
        corpus = merge_token(corpus, most_frequent)

    return most_frequent_merges


def write_bpe(lang, most_freq_merges):

    bpe_file = codecs.open(join(datadir, lang+'.model'), 'w', encoding='
                                        utf-8')

    bpe_file.write(f"{lang} {len(most_freq_merges)}\n")
    bpe_file.write('\n'.join(' '.join(item) for item in most_freq_merges
                                        ))

    return

if __name__ == '__main__':

    for lang in [source, target]:

        argsinput = codecs.open(inputpath[lang], encoding='utf-8')
        bpe_model = codecs.open(join(datadir, lang+'.model'), 'w',
                                            encoding='utf-8')
        most_freq_merges = learn_bpe(argsinput, bpe_model)
        write_bpe(lang, most_freq_merges)
```

6.2.2 Apply BPE algorithm

```
import os
from os.path import join
import sys
import codecs
import random
from tqdm import tqdm
```

```
# import global variables from settings.py
sys.path.insert(1, os.path.join(sys.path[0], '..'))
from settings import *
from learn_bpe import read_bpe_model, read_corpus

def load_data():

    os.chdir(datadir)
    langs = [source, target]
    bpe_models = []
    corpora = []
    for lang in langs:

        argsinput = codecs.open(inputpath[lang], encoding='utf-8')
        corpora.append(read_corpus(argsinput))

        bpe_model, _ = read_bpe_model(lang)
        if not bpe_model:
            print(f"No model found for lang={lang}")

        bpe_model = [tuple(item.strip('\r\n ').split(' ')) for (n, item)
                                                in enumerate(bpe_model)]
        bpe_models.append(bpe_model[1:])

    return langs, bpe_models, corpora


def write_bpe(lang, num_symbols, merged_corpus, i=-1):
    outputpath = join(bpedir, 'segmentations', lang+"_"+str(num_symbols)
                                        +".bpe")
    argsoutput = codecs.open(outputpath, 'w', encoding='utf-8')
    argsoutput.write(merged_corpus)
    return


def apply_bpe(langs, bpe_models, corpora):

    for lang, bpe_model, corpus in zip(langs, bpe_models, corpora):

        bpe_model = bpe_model[:max(all_symbols)]
        all_symbols_copy = all_symbols.copy()

        str_corpus = '\n'.join(corpus)
        for j, bigram in enumerate(bpe_model):
```

```
            str_corpus = str_corpus.replace(' '.join(bigram), ''.join(
                                                bigram))

            if j + 1 == all_symbols_copy[0]:
                write_bpe(lang, all_symbols_copy.pop(0), str_corpus, i)

    return


if __name__ == "__main__":

    os.makedirs(join(bpedir, 'segmentations'), exist_ok=True)
    langs, bpe_models, corpora = load_data()
    apply_bpe()
```

7 Summary

The summary is the last section of the text and summarizes the results of the work (see also section **??** from page **??**).

Bibliography

[1] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[3] Xiang Zhang and Yann LeCun. Text understanding from scratch, 2015.

[4] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment, 2017.

[5] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time, 2016.

[6] Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *Transactions of the Association for Computational Linguistics*, 5:365–378, 2017.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[8] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing, 2019.

[9] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2015.

[10] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[11] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[13] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[14] Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. Bpe-dropout: Simple and effective subword regularization, 2019.

A Diagrams

Possible contents for an attachment as well as its formal design are described