# A COMPREHENSIVE ANALYSIS OF SUBWORD TOKENIZERS FOR MORPHOLOGICALLY RICH LANGUAGES

by

Erencan Erkaya

B.S., Computer Engineering, Marmara University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2022

A COMPREHENSIVE ANALYSIS OF SUBWORD TOKENIZERS FOR MORPHOLOGICALLY RICH LANGUAGES

APPROVED BY:

| | |
|---|---|
| Prof. Tunga Güngör<br>(Thesis Supervisor) | .................... |
| Assoc. Prof. Arzucan Özgür | .................... |
| Assoc. Prof. Murat Can Ganiz | .................... |

DATE OF APPROVAL: DD.MM.YYYY

# ACKNOWLEDGEMENTS

I would like to thank Prof. Tunga Güngör for guidance and support.

The models were trained thanks to Google TPU Research Cloud (TRC) Program.

# ABSTRACT

## A COMPREHENSIVE ANALYSIS OF SUBWORD TOKENIZERS FOR MORPHOLOGICALLY RICH LANGUAGES

Transformer language models have paved the way for outstanding achievements on a wide variety of natural language processing tasks. The first step in transformer models is dividing the input into tokens. Over the years, various tokenization approaches have emerged. These approaches have further evolved from character and word-level representations to subword-level representations. However, the impact of tokenization on models performance has not been thoroughly discussed, especially for morphologically rich languages. In this thesis, we comprehensively analyze subword tokenizers for Turkish, which is a highly inflected and morphologically rich language. We define various metrics to evaluate how well tokenizers encode Turkish morphology. Also, we examine how the tokenizer parameters like vocabulary and corpus size change the characteristics of tokenizers. Additionally, we propose a new tokenizer for agglutinative and morphologically rich languages. We demonstrate that our tokenizer reduces overall perplexity and enables better generalization performance. Downstream task experiments show that morphology supervision in tokenization improves model performance.

# ÖZET

## MORFOLOJİSİ ZENGİN DİLLER İÇİN KELİME BÖLÜMLEME ALGORİTMALARININ KAPSAMLI BİR ANALİZİ

Dönüştürücü dil modelleri, çok çeşitli doğal dil işleme görevlerinde olağanüstü başarıların yolunu açmıştır. Dönüştürücü dil modellerinde ilk adım, girdiyi jetonlara bölmektir. Yıllar boyunca, çeşitli bölümleme yaklaşımları ortaya atılmıştır. Bu yaklaşımlar, karakter ve kelime seviyesindeki temsillerden alt kelime seviyesindeki temsillere doğru daha da gelişmiştir. Bununla birlikte, özellikle morfolojik olarak zengin diller için, kelime bölümleme algoritmalarının model performansı üzerindeki etkisi tam olarak tartışılmamıştır. Bu tezde, çekimli ve morfolojik açıdan oldukça zengin bir dil olan Türkçe için alt kelime bölümleme algoritmalarının kapsamlı bir şekilde analizi yapılmıştır. Bölümleme algoritmalarının Türkçenin morfolojisini ne kadar iyi kodladığını değerlendirmek için çeşitli metrikler tanımlanmıştır. Ayrıca, sözcük dağarcığı ve derlem boyutu gibi farklı belirteç parametrelerinin belirteçlerin özelliklerini nasıl değiştirdiği incelenmiştir. Ek olarak, sondan eklemeli ve morfolojik olarak zengin diller için yeni bir bölümleme algoritması önerilmiştir. Önerilen kelime bölümleme algoritmasının daha iyi genelleme performansı sağladığı gösterilmiştir. Doğal dil işleme deneyleri, kelime bölümlemede morfoloji denetiminin model performansını iyileştirdiğini göstermektedir.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF SYMBOLS

| | |
|---|---|
| $d_k$ | Dimension of Key Vector |
| K | Key |
| Q | Query |
| V | Value |
| [CLS] | Class Token |
| [MASK] | Mask Token |
| [SEP] | Separator Token |

# LIST OF ACRONYMS/ABBREVIATIONS

| | |
|---|---|
| BERT | Bidirectional Encoder Representations from Transformers |
| BPE | Byte-Pair Encoding |
| CC | Common Crawl |
| CLS | Class Token |
| ELECTRA | Efficiently Learning an Encoder that Classifies Token Replacements Accurately |
| EM | Exact Match |
| F1 | F-measure |
| GAN | Generative Adversarial Network |
| GB | Gigabyte |
| GPT | Generative Pre-training |
| GPU | Graphics Processing Unit |
| LSTM | Long Short Term Memory |
| MLM | Masked-Language Modeling |
| NER | Named-entity Recognition |
| NLI | Natural Language Inference |
| NLP | Natural Language Processing |
| NSP | Next Sentence Prediction |
| OOV | Out-of-vocabulary |
| OSCAR | Open Super-large Crawled Aggregated coRpus |
| PoS | Parts-of-Speech |
| QA | Question Answering |
| RNN | Recurrent Neural Network |
| RTD | Replaced Token Detection |
| SEP | Separator Token |
| SOV | Subject Object Verb |
| SQuAD | The Stanford Question Answering Dataset |
| T5 | Text-to-Text Transfer Transformer |

| | |
|---|---|
| TPU | Tensor Processing Unit |
| TRC | TPU Research Cloud |
| ULM | Unigram Language Model |

# 1. INTRODUCTION

In natural language processing (NLP), tokenization is the process of dividing a text into pieces, called tokens. These tokens are represented as fixed-length vectors and then fed into models. Over the years, various approaches have emerged to address this problem.

One of the most straightforward solutions to the tokenization problem is to separate text by white space and punctuation marks. However, this approach brings its own problems, namely the problems of vocabulary size and out-of-vocabulary words. As the corpus size increases, the number of unique words increases too and this leads to a larger vocabulary size which causes memory and performance problems during processing. As a solution, reducing the vocabulary size gives rise to an increase in the number of unknown words. The other problem is out-of-vocabulary words that are seen in the pre-training phase but not in the test phase. This problem is commonly seen in morphologically rich languages like Turkish as a result of lexical sparseness and the wide diversity of grammatical properties which are expressed within morphology.

Character level representation is another solution that handles text as a sequence of characters. The size of the vocabulary is drastically decreased to the number of unique characters. In this way, unknown words can be encoded since all the characters are represented in the vocabulary. Despite that the method solves the out-of-vocabulary problem, reducing the size of the vocabulary produces way more tokens compared to word level representation. Also, character units do not carry semantic information, while word level representation keeps complete word semantics. The model that employs a character level tokenizer should be complex enough to generate plausible representations.

An intermediate approach between character and word level tokenization is subword level tokenization which splits text into subwords. Subwords are coarse-grained and generally shorter than word tokens which may encode morphological information like roots, prefixes, and suffixes, unlike character tokens. The concept of subword level tokenization was initially used in a data compression algorithm [1]. In the past years, with the development of transformer models, subwords level tokenization approaches have been quite popular, and they have been used in state-of-the-art models like Bidirectional Encoder Representations from Transformers (BERT) [2], Generative Pre-training (GPT) [3], and Text-to-Text Transfer Transformer (T5) [4] to alleviate the large vocabulary problem. Subword level tokenization enables a model to cover any given text in a reasonable vocabulary size.

From past to present, tokenization approaches have been further developed from word and character level representation to subword level representation. In the literature, byte-pair encoding [5], unigram language model [6], and Wordpiece [7] are the most widely used subword tokenization algorithms. Each of these algorithms has its own advantages and disadvantages.

It is not a straightforward process to choose an optimal tokenizer. Most state-of-the-art models have their own custom and optimized tokenizer; therefore, input segmentations of different models may be different for the same input. In addition to that, subword tokenizers do not always guarantee linguistically meaningful subwords. The extent to which tokenizer encodes morphology and produces meaningful subwords is a subject of research. The effects of the tokenizers on the model performance have not been generally discussed thoroughly in the literature, especially for morphologically rich languages.

In this thesis, we aim to analyze existing tokenization algorithms for Turkish which is a morphologically rich and highly inflected language. We define various metrics to evaluate how well tokenizers encode Turkish morphology. Also, we examine how the tokenizer parameters like vocabulary and corpus size change the characteristics of tokenizers.

Lastly, we propose a new tokenization approach for agglutinative and morphologically rich languages and compare it with Wordpiece by testing on four downstream tasks which are named-entity recognition, parts-of-speech tagging, question answering, and sentiment analysis.

The rest of the thesis is organized as follows. In Chapter 2, we provide background information about the Transformer architecture, subword algorithms, and Turkish language. In Chapter 3, we discuss the related works on transformer and input representations. In Chapter 4, we provide the details of datasets, the metrics that we use to compare tokenizers, and the details of morphologically optimized tokenizer. In Chapter 5, we give the experimental setup and discussion of the outcomes. Lastly, in Chapter 6, we give a brief overview of the findings and potential future works.

# 2. BACKGROUND INFORMATION

## 2.1. Transformer Model

The transformer architecture has paved the way for advanced language models. In the literature, there are a lot of models based on the Transformer model that show outstanding performance on a wide variety of natural language processing tasks such as named entity recognition, question answering, natural language inference [2,3,8].

Before the Transformer, Recurrent Neural Network (RNN) and Long Short Term Memory (LSTM) have dominated NLP literature; the most advanced and complex models were built using RNN and LSTM. However, RNN and LSTM can just process one word in a sequence at a time and progressively expands knowledge of the sequence. This drawback makes them hard to learn long-range dependencies. The nature of RNNs and LSTMs limits parallel computing. The Transformer enables to process a sequence at once in a parallel manner on the contrary to RNN, LSTM. As a result of the parallelization, the Transformer can process way more data than the others and does not suffer from the long dependency problem. Also, the Transformer has paved the way for a much more efficient use of computing devices like Graphics Processing Units (GPUs).

The transformer architecture consists of two major components: encoder and decoder as shown in Figure 2.1. The encoder takes the whole input sequence $(x_1, ..., x_n)$ and encodes it into a sequence of continuous representations z = $(z_1, ..., z_n)$. The decoder uses the output of the encoder z and the previously generated tokens as input to generate output $(y_1, ..., y_m)$. At each step, the decoder generates the next token.

In the original implementation, the encoder consists of six identical layers. Each layer is broken down into two components: a multi-head attention mechanism and a feed-forward network. The multi-head attention mechanism is used to reflect various attention patterns. The feed-forward network provides nonlinear transformation and operates as key-value memories [9]. These two components have residual connections followed by layer normalization. Residual connections primarily contribute to alleviating the vanishing gradient problem.

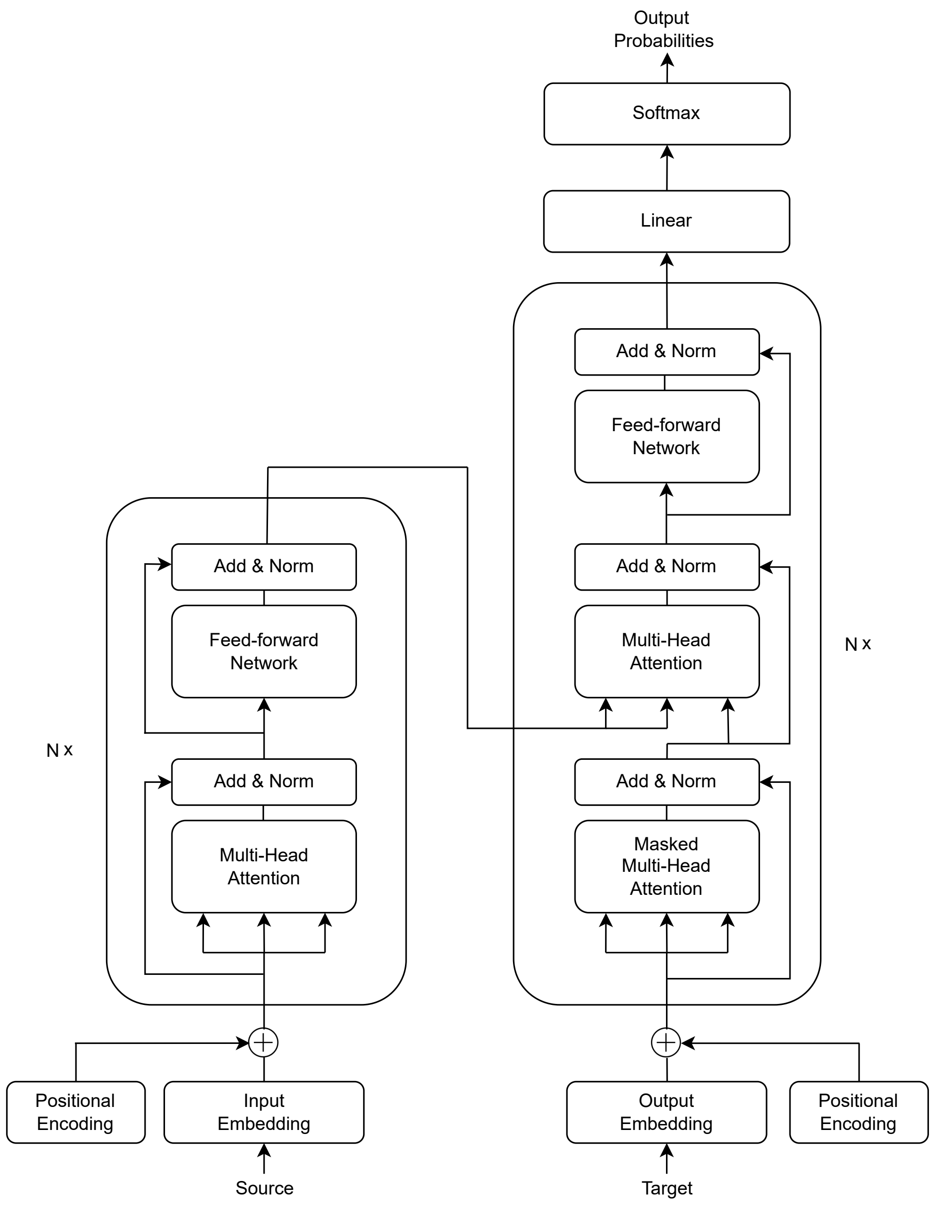

Figure 2.1. Transformer model

The decoder is a stack of six equal layers similar to the encoder. The decoder also has an additional multi-head attention layer after the first layer. The second layer carries out the multi-head attention operation over the output of the encoder's last layer. Similar to the encoder, the three components have residual connections followed by layer normalization. On the contrary to the encoder, the decoder partially masks the input to prevent attending to subsequent tokens since only preceding tokens can be used for the prediction.

### 2.1.1. Scaled Dot-Product Attention

Attention was initially used in the encoder-decoder architecture in order to overcome the inability to decode long inputs. Attention enables the decoder to reach any state from the encoder's outputs; as a result of this, the encoder does not have to compress the whole input sequence information into a single context vector. There are different types of attention mechanisms. The Transformer uses the self-attention mechanism to compute the attention scores of a token to a sequence. These scores indicate how much attention to give to each context token.

Transformer basically relies on self-attention. The input embeddings are projected onto three vector spaces called query (Q), key (K), and value (V) to calculate self-attention scores. Then, the dot products of the query and all keys are calculated. The dot products provide information about how much attention the query has over the keys. The scores are divided by the square root of the dimension of the key vector $d_k$ and applied a softmax function. The scores are scaled because the dot products might be large in magnitude. As a final step, the scaled scores are multiplied by values. In practice, these scores are computed simultaneously for all the keys in a sequence as shown below:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

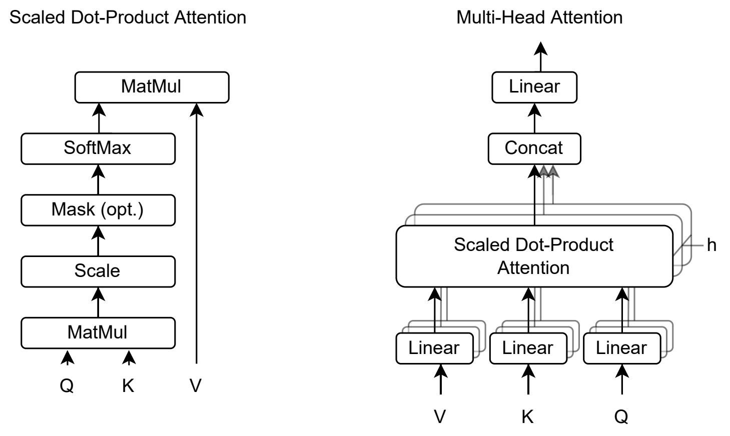

Figure 2.2. Scaled Dot-Product Attention & Multi-Head Attention

### 2.1.2. Multi-Head Attention

Scaled dot-product attention only provides a single representation; however, performing multiple self-attention functions with a different key, query, and value parameters might provide different representation sub-spaces. For example, one attention pattern can reflect the subject-verb agreement, and another can show affixes. The Transformer uses multiple-head attention instead of performing a single attention operation. The projection operation is performed h times as shown in Figure 2.2. Each time different projection vectors are used. Finally, outputs are concatenated and projected onto $W^O$ as shown below:

$$\begin{gathered} MultiHead(Q,K,V) = Concat\,(\,head_1, \ldots, head_{\,\mathrm{h}})\,W^O \\ \text{where head}_{\,i} = \text{ Attention } \left(QW_i^Q, KW_i^K, VW_i^V\right) \end{gathered} \tag{2.2}$$

In the original implementation of Transformer, 8 parallel attention layers are used.

### 2.1.3. Feed-Forward Networks

Within a single Transformer layer, there is also another sub-layer called feed-forward network. The feed-forward neural network applies two nonlinear transformations and the output of the network is passed to the subsequent layer as input.

$$\mathrm{FFN}(x) = \max\left(0, xW_1 + b_1\right)W_2 + b_2 \tag{2.3}$$

## 2.2. BERT

The original transformer architecture [10] was initially designed for machine translation tasks; for this reason, the model needs the decoder to generate text. However, some NLP tasks such as classification and part-of-speech tagging do not require text generation. This makes the decoder part unnecessary for some tasks.

BERT [2] is the only encoder part of the original transformer model with small modifications. BERT consists of stacked transformer layers, where each layer includes multiple self-attention heads. For every input of attentions, query, value, and key vectors are computed and used to correlate the relationship between current and context words. The outputs of the self-attention heads are fed into a fully connected layer. Each layer has skip-layer connections and is accompanied by layer normalization.

As shown in Figure 2.3, the input embeddings of BERT consist of position embeddings, segments embeddings, and token embeddings. Each embedding layer encodes a different type of information extracted from the input.

The input text is tokenized by using the WordPiece algorithm [7, 11]. Also, the BERT model takes a position and segment embedding for each token in the input. Since

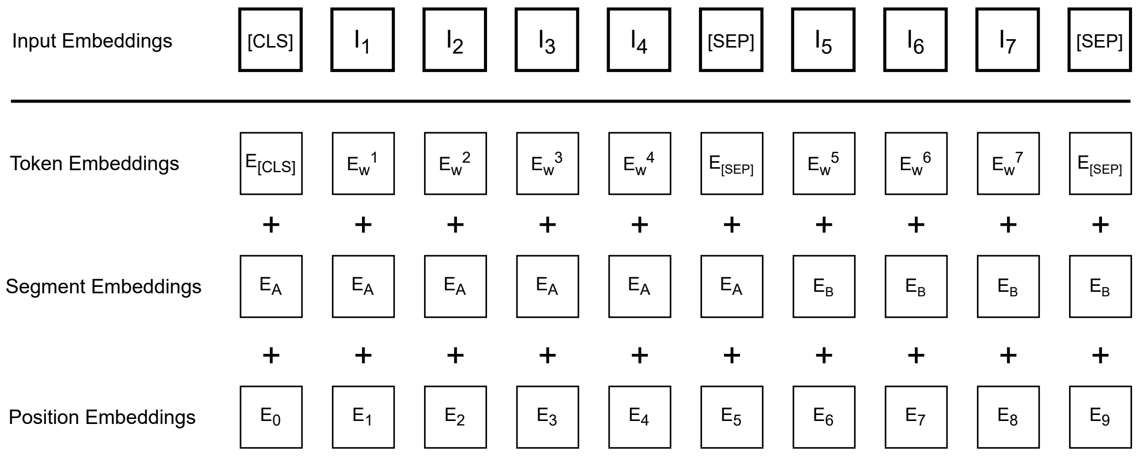

Figure 2.3. BERT Input Representation

the model does not have any position information of the token, position embedding is used to encode order for each word. Some tasks require two input segments, like question answering. In a similar way, segment embedding is used to distinguish two segments given a pair. The tree embeddings are combined and fed into the model. Every input sequence starts with a special token [CLS], which is used as an aggregation of the entire input tokens. In tasks that take two segments, the input is separated by token [SEP].

BERT is trained on two pretraining tasks; Masked Language Modelling (MLM) and Next Sentence Prediction (NSP). MLM tries to predict input tokens that are replaced by the [MASK] token. However, the [MASK] token does not appear during the fine-tuning process. If the model is trained only by masking tokens, the model only learns contextual representations of the [MASK] token. To alleviate this issue, 15% of the input tokens are randomly selected. Only 80% of the selected tokens are replaced by the [MASK] token, 10% of the selected tokens are kept unchanged, and 10% of the selected tokens are replaced with a random token. Many NLP tasks such as question answering (QA), natural language inference (NLI) require understanding the relationship between two sentences. To model this relationship, BERT uses the NSP task, which tries to predict whether two sentences are consecutive or not. 50% of the input sentences are consecutive sentences, and the remaining 50% are randomly combined sentences. The BERT model develops a solid language comprehension ability through the pretraining tasks.

## 2.3. ELECTRA

BERT is trained on the MLM task, which replaces a certain percentage of input tokens with [MASK], and the trained model is optimized to predict the original tokens. However, this approach requires a massive amount of computational power to acquire successful results. Efficiently Learning an Encoder that Classifies Token Replacements Accurately (ELECTRA) uses an alternative task to train a model called replaced token detection (RTD), as shown in Figure 2.4. Some tokens are replaced with alternative tokens that are produced by the generator. Then, another network called discriminator tries to predict whether each token is replaced or original. RTD is more effective than MLM because RTD uses all input tokens, unlike MLM, which uses just masked tokens.

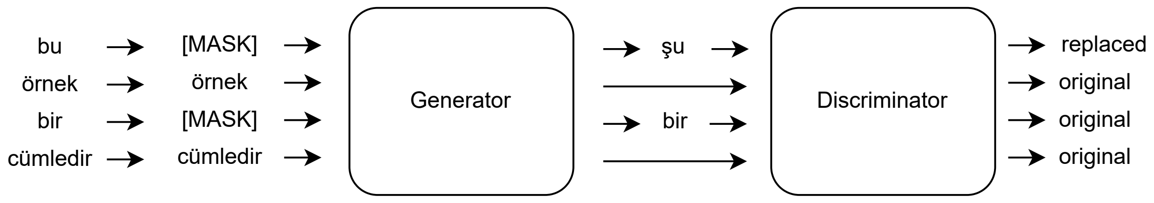

Figure 2.4. Replaced token detection.

ELECTRA consists of the generator G and the discriminator D. Both of them are encoders like BERT. The generator takes input tokens x = $[x_1, ..., x_n]$ and produces contextualized representations of the input h(x) = $[h_1, ..., h_n]$ like BERT. The generator produces a token distribution for a masked token as shown in Eq. 2.4, where t is the position of the masked token and e is token embeddings.

$$p_G\left(x_t \mid \boldsymbol{x}\right) = \exp\left(e\left(x_t\right)^T h_G(\boldsymbol{x})_t\right) / \sum_{x'} \exp\left(e\left(x'\right)^T h_G(\boldsymbol{x})_t\right) \tag{2.4}$$

The discriminator D is trained to distinguish whether a token x is replaced or not for a given position t.

$$D(\boldsymbol{x}, t) = \text{sigmoid}\left(w^T h_D(\boldsymbol{x})_t\right) \tag{2.5}$$

The objective function of ELECTRA is similar to generative adversarial networks (GAN). The generator implements MLM. It selects 15% of input tokens and replaces them with [MASK]. Although the ELECTRA architecture is similar to GAN, a few points are different. The generator is trained with maximum likelihood, not in an adversarial manner. The combination of loss $\mathcal{L}_{\text{MLM}}$ and $\mathcal{L}_{\text{Disc}}$ is minimized during training as shown in Eq. 2.6 where $\lambda$ is the weight for the discriminator objective. For downstream tasks, the discriminator is finetuned.

$$\min_{\theta_G, \theta_D} \sum_{\boldsymbol{x} \in \mathcal{X}} \mathcal{L}_{\text{MLM}}\left(\boldsymbol{x}, \theta_G\right) + \lambda \mathcal{L}_{\text{Disc}}\left(\boldsymbol{x}, \theta_D\right) \tag{2.6}$$

ELECTRA works especially well under conditions where data and computational resources are limited. However BERT requires more computational power since BERT masks only 15% of the input tokens for each example. The studies [12] show that the training process of ELECTRA is considerably faster than BERT to reach the same performance and ELECTRA performs better when it is completely trained.

## 2.4. An Overview of Turkish Language

Turkish is a highly inflected and morphologically rich language. A considerable amount of grammar is expressed with suffixes that are added to words. These suffixes may have different forms depending on what vowels are in the base word. This phenomenon in the Turkish language is known as vowel harmony. Suffixes are attached, considering which type of vowels are used. Vowel harmony rules decide which forms of a suffix are attached to preserve harmony. For example, -lar and -ler are the plural suffixes used in Turkish. Adding either -ler or -lar makes a word plural, but which one to add depends on vowel harmony. Words with one of the back vowels (a, ı, o, u) in the last syllable get the suffix -lar, while words with one of the front vowels (e, i, ö, ü) in the last syllable get the suffix -ler. Although the form of the suffix is different, the meaning it gives is the same.

There are mainly two categories of suffixes: inflectional and derivational suffixes. Inflectional suffixes are used to inflect grammatical properties of words and produce alternate forms of words without changing the essential meaning of words. Unlike inflectional suffixes, derivational suffixes make new words and change the meanings of words.

Prefixes are quite exceptional in Turkish. Most of the words with suffixes were taken from Persian and Arabic, and later from other foreign languages such as English and French [13]. For example; a-normal (anormal), dez-avantaj (disadvantage), anti-tez (antithesis). Also, there are some prefixes to intensify meaning, like mas-mavi (deep blue), pes-pembe (rose-pink).

A typical Turkish sentence mostly appears in Subject Object Verb (SOV) word order; the subject appears at the beginning of the sentence, the object comes next, and the verb generally comes at the end of the sentence. However, the word order is relatively flexible in Turkish. Various word orders like OSV and OVS can be seen in the written and spoken language. There are five main elements to make a sentence: verb, subject, object, indirect object, adverbial clause [14]. When all of these elements are taken into account, there are 24 possible sentences in the canonical form in which the verb comes at the end of the sentence. As shown in Table 2.1 [15], most sentences follow SOV and SVO word orders, and sentences that start with verbs are quite exceptional.

Also, changing the word order enables emphasizing a specific part of a sentence. The emphasis is on the word that is closer to the verb.

Table 2.1. Word Order Frequencies.

| | |
|---|---|
| **SOV** | 48% |
| **OSV** | 8% |
| **SVO** | 25% |
| **OVS** | 13% |
| **VSO** | 6% |
| **VOS** | <1% |

# 3. RELATED WORK

After transformer-based models have achieved SOTA results, a lot of studies that focus on ways to improve and interpret models have been published in the literature. In this chapter, we explained some important studies.

Kudo et al. [16] proposed a sub-word tokenizer called SentencePiece. It is an extension of two sub-word segmentation algorithms, byte-pair encoding, and uni-gram language model. SentencePiece does not need pre-tokenized word sequences, unlike BPE and ULM. It treats the input just as a sequence of Unicode characters, even for white spaces. It enables to build an end-to-end system that does not depend on any language-specific processing. It includes four main components: normalizer, trainer, encoder, and decoder. Normalizer is a module to normalize semantically equivalent unicode characters into canonical forms. Trainer trains the sub-word segmentation model from the normalized corpus. Encoder internally executes Normalizer to normalize the input text and tokenizes it into a sub-word sequence with the sub-word model trained by Trainer. Decoder converts the sub-word sequence into the normalized text. It also employs some speed-up methods such as a priority queue. The experiments show that Sentencepiece consistently outperforms.

Choe et al. [17] developed a tokenizer-free language model based on byte representations. The proposed model takes UTF-8 bytes instead of tokenized input text. The ASCII range covers the most common characters. Two or three UTF-8 bytes are mostly sufficient for non-ASCII characters. The experiments show that byte-level representations improve performance. Tokenizer-free language models like byte-level LM can perform better.

Sub-word tokenization algorithms do not evenly suit all languages. Clark et al. [18] proposed a neural encoder that operates on character sequences called CANINE. It is an architecture that consists of a deep transformer stack at its core that uses

Unicode characters to represent input without a vocabulary. Input characters are represented by Unicode, which does not change over time, unlike vocabulary-based IDs. CANINE is the first pre-trained encoder that uses a tokenization-free, vocabulary-free model. The pure character level model is 10x slower than the original BERT implementation. However, CANINE performs similarly in terms of speed. It may encode better, especially morphologically rich languages.

Character language models do not need a fixed word vocabulary and enable to model sub-word information. Nevertheless, even large corpora do not cover most inflection forms, especially for morphologically rich languages. Blevins et al. [19] developed a character-based language model that takes morphology into account. The morphology model outperforms the character language model, and languages with lower inflection rates benefit from morphology supervision.

Alyafeai et al. [20] analyzed various tokenization approaches for Arabic which is a morphologically rich language. They compared six tokenizers: character tokenizer, word tokenizer, morphological tokenizer, stochastic tokenizer, disjoint-letter tokenizer, and SentencePiece tokenizer. The results indicate that the word tokenizer performs better on sentiment analysis and news classification tasks. Increasing vocabulary size leads to fewer out-of-vocabulary words and generally improves accuracy. Also, stochastic tokenizer, which may produce meaningless tokens, performs close to the other tokenizers on Arabic. They commented that morphological information and coarse-grained tokens provide better results for low resource datasets, but tokenizer does not contribute too much for high resource datasets.

Matthews et al. [21] propose an open-vocabulary language model. The model combines character, word-level approaches, and morphological information in a neural network. The results show that the proposed model performs better than an n-gram model, character level RNNLM. Morphological information improves the performance of language models, especially for morphologically rich languages.

Domingo et al. [22] show the impact of various tokenization algorithms on neural machine translation. They conducted experiments on five tokenizers over ten language pairs. The experiments show that tokenization has a major impact and improves translation quality. There is not the best tokenizer for all languages. Each method gives the best performance for specific languages.

Mielke et al. [23] survey comprehensively the history of tokenization approaches and show how they evolved throughout the years. They claimed that each method has its own drawbacks and problems, and each algorithm performs better for certain applications. There are some applications that need larger tokens for efficiency and performance. On the other hand, morphologically rich languages may necessitate fine-grained tokens to represent each word adequately.

# 4. METHODOLOGY

In this chapter, we explain the most commonly used tokenization algorithms and define some key metrics to compare tokenizers. We also explain the details of the morphologically optimized tokenizer. Lastly, an intrinsic evaluation metric Pseudo-perplexity is presented to evaluate how well a model models a given input.

## 4.1. Subword Algorithms

In the following subsections we explain subword algorithms.

### 4.1.1. Byte-pair Encoding

Initially, BPE was designed as a data compression algorithm [1], then it became popular in the neural machine translation field [5, 11] and advanced language models [24,25]. Byte-pair encoding is a text segmentation algorithm that can represent a word in smaller pieces. Pretrained language models might need a representation of rare or unknown word. Since it is not possible to handle all words, pretrained language models use this kind of approaches as a solution for the out-of-vocabulary problem. Models tokenize input before processing it and generate output using subword vocabulary.

```
Input: strings S, vocabulary size k

procedure BPE(S, k)
    V ← unique characters in S
    while |V| < k do (Merge tokens)
        $a_1, a_2 \leftarrow$ most frequent bigram in S
        $a_{new} \leftarrow a_1 \oplus a_2$ (new token)
        V ← V ∪ {$a_{new}$}
        Replace $a_1, a_2$ with $a_{new}$
    end while
    return V
end procedure
```

Figure 4.1. Byte-Pair Encoding Algorithm.

The BPE algorithm is shown in Figure 4.1. Byte-pair encoding iterates over each character in a corpus and finds the pair that has the highest frequency. Then it replaces each occurrence of the pair with a new symbol of the pair (merging) and adds the merged subword to vocabulary. The process continues until reaching the number of iterations or subword vocabulary size. The motivation of BPE is to represent a corpus with a fewer number of tokens so that a model can handle the rare words problem.

### 4.1.2. Unigram Language Model

ULM is another text segmentation algorithm which is proposed by Kudo [6]. It is based on a unigram language model which makes an assumption that the probability of each subword is independent of other subwords. Therefore a sequence can be tokenized as the product of the subword $(x_1, ..., x_M)$ probabilities as shown in Eq. 4.1. The highest probability gives the most likely segmentation for a sequence (Eq. 4.2). The probabilities are calculated using the loss function (Eq. 4.3) that is trained on data.

$$P(\mathbf{x}) = \prod_{i=1}^{M} p\left(x_i\right) \tag{4.1}$$

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathcal{S}(X)}{\arg\max} P(\mathbf{x}) \tag{4.2}$$

$$\mathcal{L} = -\sum_{i=1}^{N} \log \left( \sum_{x \in S(x_i)} p(x) \right) \tag{4.3}$$

The ULM algorithm is shown in Figure 4.2. ULM starts with an initial seed vocabulary which is significantly greater than the target vocabulary size, gradually reduces to reach target vocabulary size. An initial seed vocabulary might consist of unique characters, common substrings, affixes that are extracted from training corpus. The expectation maximization algorithm is used to compute subword probabilities. The ULM algorithm calculates a loss for each subword in vocabulary at each training step. This step shows how much a particular subword affects overall score. Then, it sorts the subwords by loss and removes subwords below a certain percentage. This process continues until reaching the defined target vocabulary size. In the final version of the vocabulary, single characters are always kept.

```
Input: strings S, vocabulary size k, shrinking factor p

procedure ULM(S, k)
  V ← initial seed vocabulary (unique characters, common substrings, affixes)
  while |V| > k do
    Compute unigram language model $\theta$ for S
    for x ∈ V do
      $L_x \leftarrow P_\theta - P_{\theta'}$ (where $\theta$' is the LM without substring x)
    end for
    Remove (1-p)% of the substrings that have lowest L
  end while
  Fit final unigram LM $\theta$ to D
  return V, $\theta$
end procedure
```

Figure 4.2. Unigram Language Model Algorithm.

### 4.1.3. Wordpiece

Wordpiece was introduced by Schuster and Nakajima [7] to deal with an infinite vocabulary, then with the rapidly developing NLP studies, it has been used in transformer models like BERT [10], ELECTRA [12], DistillBERT [26]. Wordpiece uses all unique characters in the language as initial seed vocabulary, then iteratively extends vocabulary by merging the pair that causes the largest increase in likelihood. This makes Wordpiece an approach between BPE and ULM.

```
**Input:** strings S, vocabulary size k

procedure Wordpiece(S, k)
    V ← unique characters in S
    while |V| < k do (Merge tokens)
        $a_1, a_2$ ← pair that causes largest increase in likelihood
        $a_{new} \leftarrow a_1 \oplus a_2$ (new token)
        $V \leftarrow V \cup \{a_{new}\}$
        Replace $a_1, a_2$ with $a_{new}$
    end while
    return V
end procedure
```

Figure 4.3. Wordpiece Algorithm.

The Wordpiece algorithm is shown in Figure 4.3. Unlike BPE, Wordpiece combines pairs of subwords according to the language model probability instead of frequency. At each iteration of the algorithm, Wordpiece chooses the pair that gives the largest increase in the language model probability. For example, if the token *kalem* is more probably to occur than the tokens *kal* + *em*, the tokens are merged and added to the vocabulary list. The process continues until reaching the target vocabulary size. Using language model rather than frequency enables Wordpiece to consider the impact of merging a pair.

The main difference between BPE and Wordpiece is that the Wordpiece algorithm chooses the most likely segmentation instead of the best segmentation at each iteration unlike BPE. Both methods are bottom-up and greedy approaches. On the contrary, ULM is a completely probabilistic solution which makes use of probability for choosing and merging pairs.

Table 4.1. BPE, ULM, Wordpiece Examples. Tokenizers were trained with the same corpus and parameters. # denotes the word's complement tokens.

| **Word** | **BPE** | **ULM** | **Wordpiece** |
|---|---|---|---|
| büyükadaya | büyü, #ka, #daya | büyük, #aday, #a | büyük, #ada, #ya |
| dünyalıyım | dün, #yalı, #yım | dünya, #lıyım | dünya, #lı, #yım |
| inceltmek | incel, #tmek | ince, #l, #t, #mek | incel, #tme, #k |
| kaygısını | kayg, #ısını | kaygı, #sını | kaygısı, #nı |
| yangının | yang, #ının | yangın, #ın | yangını, #n |
| modüler | mo, #düler | modül, #e, #r | modül, #er |
| tümüne | tüm, #üne | tümü, #n, #e | tümü, #ne |
| yandı | yandı | y, #andı | yan, #dı |
| okuru | ok, #uru | o, #kuru | okur, #u |
| döşek | dö, #şek | döşe, #k | döş, #ek |

## 4.2. Evaluation Metrics for Subword Algorithms

We define various evaluation metrics to evaluate tokenizers. The motivation is to evaluate morphological compatibility, especially for morphologically rich languages like Turkish. Morphology-compatible tokens, suffix precision, suffix recall, and root tokens require morphological analysis to compare tokens. In addition, single-word tokens, fertility, and average token length give information about the granularity of tokens. We analyze the subword algorithms according to the following criteria:

a) Single-word Tokens: measures the percentage of words that are tokenized as a single token in a corpus. In other words, it tells what percentage of the words in a corpus are in the vocabulary; it means that the vocabulary contains those words themselves. For example, if *kalemler* (pencils) is tokenized as *kalemler*, it can be seen as a single-word token. This metric shows how well a tokenizer represents a corpus without dividing words into subwords.

b) Morphology-compatible Tokens: measures the percentage of words that are exactly tokenized as the same as morphology. For example, if *kalemler* (pencils) is tokenized as *kalem, #ler*; it is valid since the token boundaries are the same as the morpheme boundaries, which are *kalem, -ler*. A higher percentage means that the tokenizer encodes morphology better.

c) Suffix Precision: shows how many tokens are suffixes. Suffix precision is the number of suffix-tokens, in which the token boundaries are the same as the morpheme boundaries, divided by the total number of tokens. For example, if *kalemleri* (his or her pencils) is tokenized as *kalem, #ler, #i*; *#ler* and *#i* are the suffix-tokens since they are suffixes.

$$\text{Suffix Precision} = \frac{number\ of\ suffix - tokens}{number\ of\ tokens} \tag{4.4}$$

d) Suffix Recall: shows how many suffixes are tokenized properly. Suffix recall is the number of suffix-tokens, in which the token boundaries are the same as the morpheme boundaries, divided by the total number of suffixes.

$$\text{Suffix Recall} = \frac{number\ of\ suffix - tokens}{number\ of\ suffixes} \tag{4.5}$$

e) Root Tokens: measures the percentage of words in which roots are represented as a single token. Unlike the morphology-compatible tokens metric, roots are only taken into account. For example, *kalem, #leri* can be seen as a valid case, even if it does not align with morphology like *kalem, -ler, -i*. This metric ignores suffixes and focuses only on how successfully roots are represented as a whole.

f) Fertility: measures the average number of tokens per word. It shows how frequently a tokenizer splits words. A tokenizer with high fertility tends to produce fine-grained tokens.

g) Average Token Length: measures the average number of characters per token in a vocabulary.

## 4.3. Morphologically Optimized Tokenizer

The tokenization algorithms do not take morphology into account by default. As shown in Table 4.1, the words are divided into tokens that do not mostly consider morpheme boundaries. We propose the morphologically optimized tokenizer to inject morphological information into the tokenization process. Our approach needs morphological analyzer to extract suffixes. Then the training corpus is updated in a way that the suffixes are enclosed with special characters so that the tokenizer recognizes suffixes during the training phase. Extracted suffixes are represented as undivisible symbols in inputs, so suffixes are not broken up into smaller tokens. For example, *kalemleri* was updated as kalem<ler><i>. Thus, the tokenizer can distinguish the suffixes, which are -ler and -i. Suffix symbols are not included in the merge process and are always kept separate from roots.

```
**Input:** strings S, vocabulary size k

**procedure** MorphologicallyOptimizedTokenizer(S, k)
    V $\leftarrow$ unique characters and affix symbols in S
    A $\leftarrow$ unique affix symbols in S
    **while** $|V| < k$ **do** (Merge tokens)
        $a_1, a_2 \leftarrow$ pair that causes largest increase in likelihood, $\underline{a_1 \text{ or } a_2 \text{ not in A}}$
        $a_{new} \leftarrow a_1 \oplus a_2$ (new token)
        V $\leftarrow$ V $\cup$ $\{a_{new}\}$
        Replace $a_1, a_2$ with $a_{new}$
    **end while**
    **return** V
**end procedure**
```

Figure 4.4. Morphologically Optimized Tokenizer Algorithm.

The morphologically optimized tokenizer algorithm is shown in Figure 4.4. The morphologically optimized tokenizer iterates over each character in a corpus and finds the pair with the highest frequency. While choosing a pair, suffix symbols are excluded, so all the suffixes are always represented separately. For example, the tokens *kalem* and <ler> can not be merged even if the pair causes largest increase in likelihood, because the token <ler> is a suffix symbol. Then it replaces each occurrence of the pair with a new symbol of the pair (merging) and adds the merged subword to vocabulary. The process continues until reaching the number of iterations or subword vocabulary size. The motivation of the morphologically optimized tokenizer is to represent a corpus with morphologically compatible tokens.

We experimented morphologically optimized tokenizer on various downstream tasks.

## 4.4. Pseudo-perplexity

A language model is basically a probability distribution that gives a probability for the new word in a given text. Perplexity is one of the intrinsic evaluation metrics for language models. Perplexity can be defined as the normalized inverse probability of given words as shown in Eq. 4.6, where W is a sequence of words.

$$PP(W) = \sqrt[N]{\frac{1}{P\left(w_1, w_2, \ldots, w_N\right)}} \tag{4.6}$$

Intuitively, Perplexity shows how well a language model predicts a given input. The evaluation is done in sequence from left to right. If a language model gives lower perplexity, it means higher probability and less uncertainty.

However, Perplexity is not an applicable metric for masked language models like BERT and ELECTRA. It is not possible to get the probability of an input text. Salazar et al. [27] proposed Pseudo-perplexity to evaluate how well a masked language model models a given input. Pseudo-perplexity is computed by masking one token at a time, as shown in Figure 4.5. Firstly, the log-likelihood of each masked token is summed as shown in Eq. 4.7 to calculate pseudo-log-likelihood score (PLL) [28]. Then, PLL scores are summed and normalized with N which is the number of tokens in the corpus C to compute Pseudo-perplexity (PPPL).

$$\mathrm{PLL}(W) = \sum_{t=1}^{|W|} \log P_{\mathrm{MLM}}\left(w_t \mid W_{\setminus t}; \Theta\right) \tag{4.7}$$

$$\mathrm{PPPL}(C) = \exp\left(-\frac{1}{N} \sum_{W \in C} \mathrm{PLL}(W)\right) \tag{4.8}$$

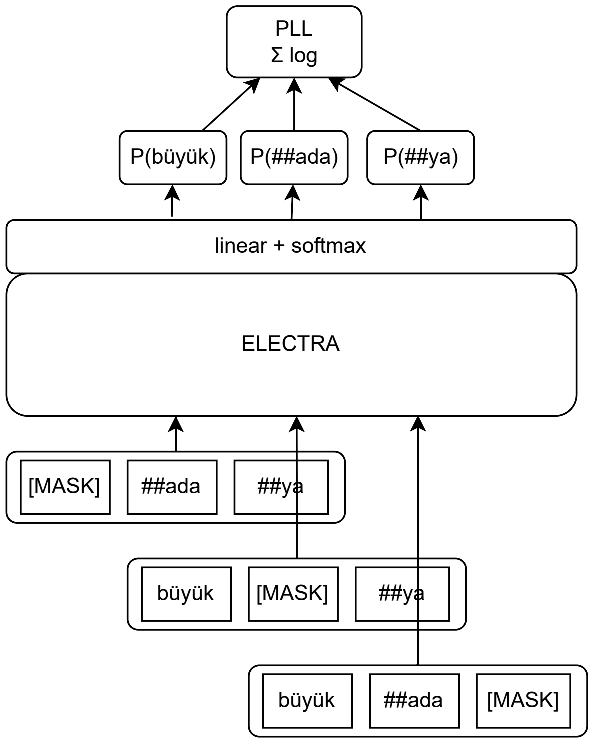

Figure 4.5. Pseudo-log-likelihood

# 5. EXPERIMENTS AND RESULTS

## 5.1. Comparison of Tokenizers

We trained BPE, ULM, and Wordpiece tokenizers with a vocabulary size of 32000 on the Turkish part of the Open Super-large Crawled Aggregated coRpus (OSCAR) corpus [29]. The corpus contains 33 gigabyte (GB) of text which is obtained by filtering the Common Crawl corpus. As shown in Table 4.1, each tokenizer can produce different tokens for the same word. To compare the tokenizers, we used BOUN Treebank [30] as the test dataset. The dataset contains 44275 words with suffixes, 97424 words in total. We tokenized the dataset using three tokenizers and analyzed them according to the metrics given in section 4.2. The comparison results are provided in Table 5.1.

Table 5.1. Comparison of the tokenizers.

| | **BPE** | **ULM** | **Wordpiece** |
|---|---|---|---|
| **Single-word Tokens** | 75% | 68% | 73% |
| **Morphology-compatible Tokens** | 52% | 51% | 54% |
| **Suffix Precision** | 49% | 44% | 51% |
| **Suffix Recall** | 36% | 38% | 39% |
| **Root Tokens** | 9% | 10% | 8% |

BPE performs better in terms of single-word tokens. BPE represents words more as a whole and without dividing them into subwords compared to the other tokenizers. ULM produces fine-grained tokens, but tokens are less morphologically compatible. Considering the results, Wordpiece is the most morphologically compatible tokenizer, and it also produces coarse-grained tokens. However, Wordpiece does not represent roots as well as ULM.

The analysis provided in Table 5.1 includes all the words in the dataset. However, examining all the words can produce misleading results since many of the words are short and do not have suffixes. We also analysed only the words with at least one suffix in the BOUN Treebank dataset. The analysis is provided in Table 5.2.

Table 5.2. Comparison of the tokenizers, only words with suffix.

| | **BPE** | **ULM** | **Wordpiece** |
|---|---|---|---|
| **Single-word Tokens** | 55% | 47% | 53% |
| **Morphology-compatible Tokens** | 6% | 8% | 10% |
| **Suffix Precision** | 21% | 24% | 27% |
| **Suffix Recall** | 12% | 16% | 16% |
| **Root Tokens** | 19% | 22% | 19% |

Similar statistics are seen when only words with suffixes are considered. About half of the words are represented as a whole. However, only 10% of the words are tokenized in a morphologically compatible way and BPE performs the worst in terms of morphology. This observation applies to the suffix metrics; BPE can not encode suffixes as well as the other methods. Roots are appropriately represented at around 20% in all the tokenizers.

Table 5.3. Comparison of the tokenizers, fertility and average token length.

| | **BPE** | **ULM** | **Wordpiece** |
|---|---|---|---|
| **Fertility** | 1.3133 | 1.4971 | 1.3413 |
| **Fertility, words with suffix** | 1.5356 | 1.7761 | 1.5813 |
| **Average Token Length** | 6.0962 | 4.4275 | 5.7159 |

Fertility and average token length are provided in Table 5.3. The average token length of ULM is less than the other tokenizers. As a result of this, ULM tends to produce more tokens for each word; ULM has higher token fertility. BPE and Wordpiece have approximately the same fertility scores considering only words with suffixes. Moreover, ULM produces more tokens for the words with suffixes since the words are generally longer.

In Table 5.4 and 5.5, example single-word and morphology-compatible tokens are provided. The tokenizers can mostly produce morphology-compatible tokens for the words with one suffix. On the other hand, the tokenizers tend to produce tokens that do not reflect morphological boundaries for the words with more than one suffix.

Table 5.4. Example single-word tokens.

| **BPE** | **ULM** | **Wordpiece** |
|---|---|---|
| sözcüğü | çalışmaları | etkinlikleri |
| hastalıklardan | desteklemek | araştırmalar |
| müşteriler | vatandaşın | maçlarda |

Table 5.5. Example morphology-compatible tokens.

| **BPE** | **ULM** | **Wordpiece** |
|---|---|---|
| editör, #ler | uygula, #mış, #lar | kahvaltı, #ya |
| rüzgar, #lık | üniversite, #de | televizyon, #da |
| deprem, #den | balık, #lar | depo, #nun |

## 5.2. Impact of Corpus Size on Tokenizer Performance

We trained tokenizers with a vocabulary size of 32000 and various corpus sizes on the Turkish part of OSCAR to measure to what extent the dataset size changes the characteristics of the tokenizers. As shown in Table 5.6, the average token length of BPE is getting lower as the corpus size increases. At the same time, fertility is getting higher. This indicates that BPE tends to produce more and shorter tokens as the corpus size increases. This analysis holds true for the other tokenizers, as can be seen in Table 5.7 and Table 5.8. After a certain point, the tokenizers become saturated, and the training corpus size does not impact that much.

The number of single-word tokens decreases as the corpus size increases, as shown in Figure 5.1. On the other hand, the number of morphology-compatible tokens increases. It means that the tokenizer is getting to encode morphology better when

Table 5.6. Impact of Corpus Size, BPE.

| **Corpus Size - GB** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1 | 1.3303 | 1.5674 | 5.5482 |
| 2 | 1.3367 | 1.5782 | 5.3147 |
| 3 | 1.3411 | 1.5851 | 5.1702 |
| 5 | 1.3485 | 1.5972 | 4.9250 |
| 10 | 1.3614 | 1.6184 | 4.5938 |
| 20 | 1.3782 | 1.6478 | 4.2437 |
| 30 | 1.3886 | 1.6653 | 4.0040 |

Table 5.7. Impact of Corpus Size, ULM.

| **Corpus Size - GB** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1 | 1.3674 | 1.6106 | 6.0719 |
| 2 | 1.3905 | 1.6337 | 5.8326 |
| 3 | 1.4007 | 1.6463 | 5.6793 |
| 5 | 1.4187 | 1.6656 | 5.4268 |
| 10 | 1.4426 | 1.6996 | 5.0931 |
| 20 | 1.4721 | 1.7418 | 4.7454 |
| 30 | 1.4924 | 1.7693 | 4.5026 |

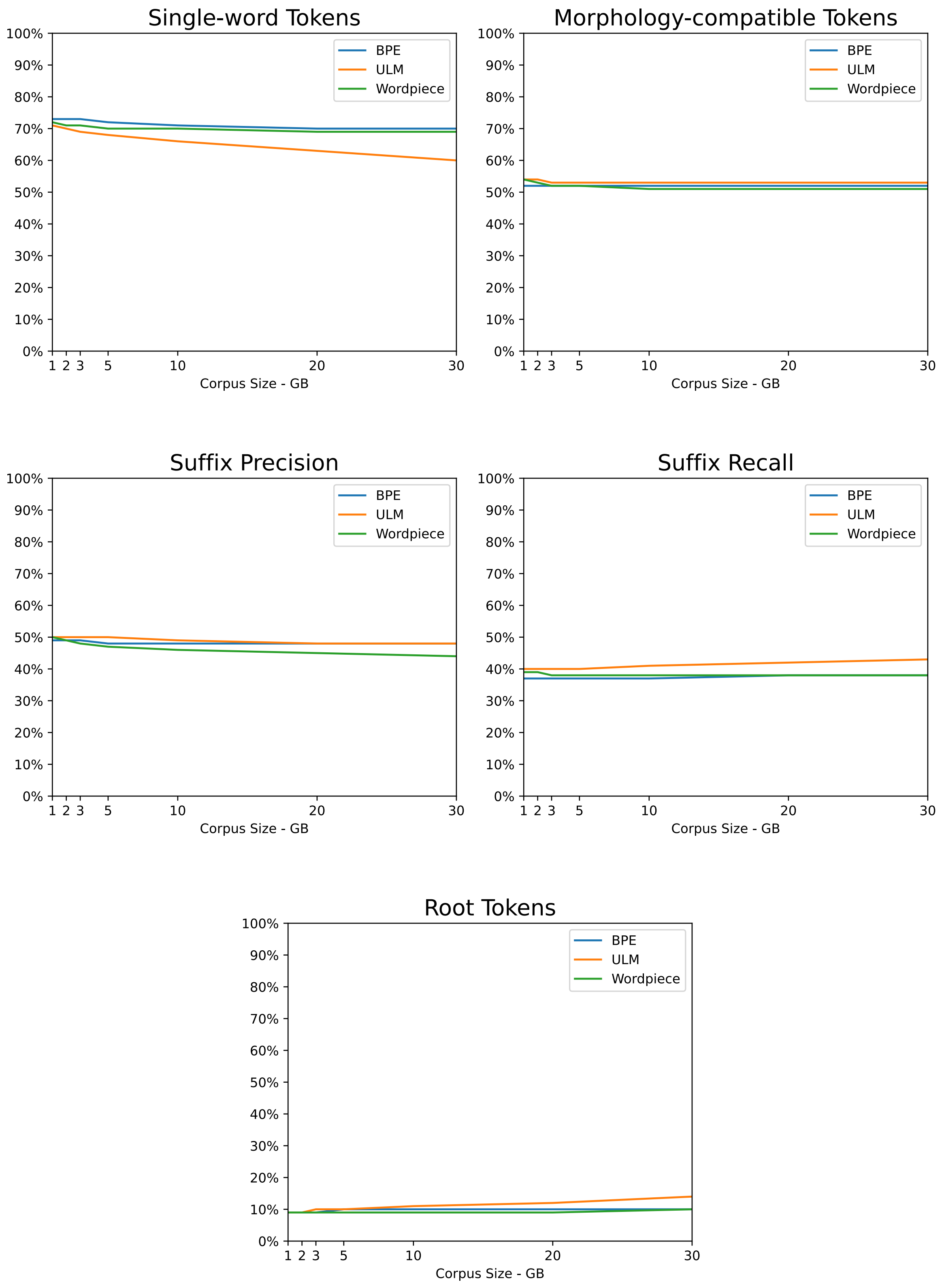

Figure 5.1. Impact of Corpus Size on Tokenizer Performance

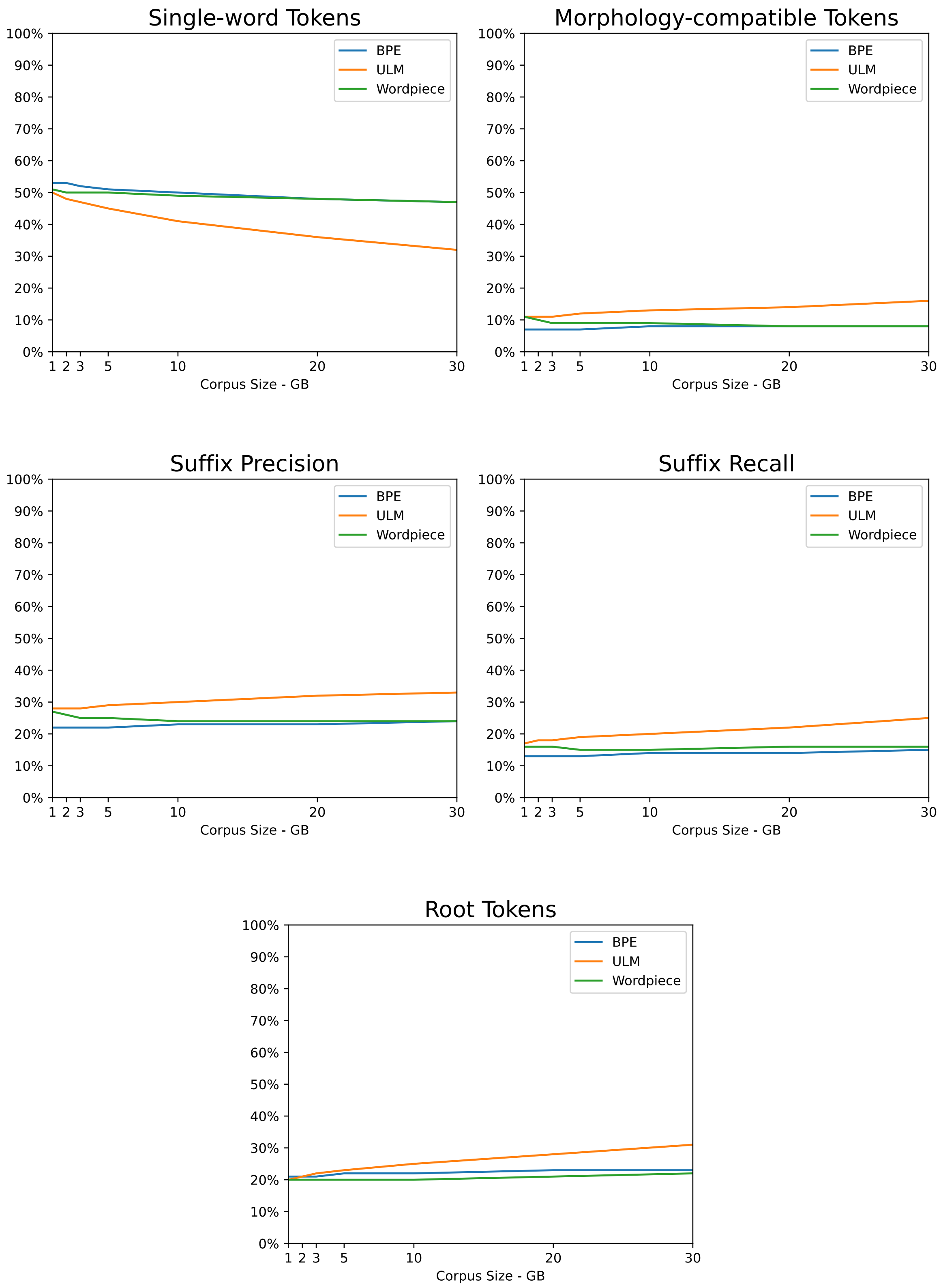

Figure 5.2. Impact of Corpus Size on Tokenizer Performance, words with suffixes

Table 5.8. Impact of Corpus Size, Wordpiece.

| **Corpus Size - GB** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1 | 1.3753 | 1.6416 | 4.8747 |
| 2 | 1.3917 | 1.6684 | 4.4702 |
| 3 | 1.4048 | 1.6887 | 4.2163 |
| 5 | 1.4253 | 1.7234 | 3.7898 |
| 10 | 1.4645 | 1.7882 | 3.2287 |
| 20 | 1.5226 | 1.8807 | 2.6431 |
| 30 | 1.5717 | 1.9620 | 2.2617 |

tokenizers are trained with more data. However, as shown in Figure 5.2, even a few gigabytes of data is enough to encode a language reasonably. ULM is the most sensitive tokenizer to the corpus size in terms of morphology. Training with more data causes a decrease in the number of single-word tokens, but it drastically increases the number of morphology-compatible tokens than Wordpiece and BPE.

## 5.3. Impact of Vocabulary Size on Tokenizer Performance

We trained tokenizers with various vocabulary sizes on the Turkish part of OSCAR to measure how the vocabulary size changes the characteristics of the tokenizers. The tokenizers produce fewer tokens per word as the vocabulary size increases. A vocabulary size of 1000 is nearly identical to a character level tokenizer. Increasing vocabulary size makes tokenizers generate more and shorter tokens. When the vocabulary size is close to the number of words in a corpus, the tokenizer fertility is nearly equal to 1. As shown in Table 5.9, Table 5.10, and Table 5.11, the fertility of words with suffixes is higher since the words that have suffixes are generally longer than words without suffixes.

Table 5.9. Impact of Vocabulary Size, BPE.

| **Vocabulary Size** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1000 | 5.4698 | 8.0283 | 1.0119 |
| 2000 | 2.2568 | 3.0263 | 2.2570 |
| 3000 | 1.9909 | 2.6274 | 2.2971 |
| 5000 | 1.7632 | 2.2778 | 3.7434 |
| 10000 | 1.5548 | 1.9434 | 4.6962 |
| 20000 | 1.3967 | 1.6789 | 5.5725 |
| 30000 | 1.3238 | 1.5540 | 6.0275 |

The tokenizers are getting to produce coarse-grained tokens as the vocabulary size increases. It leads to an increase in the number of single-word tokens, as shown in Figure 5.3. On the other hand, the number of morphology-compatible tokens does not change considerably, especially after a vocabulary size of 5000. A vocabulary size of between 3000 and 5000 encodes morphology better, as shown in Figure 5.4. After that size, the number of root tokens gradually decreases.

We used the Hugging Face library [31] to train tokenizers. The library takes additional parameters like alphabet limit to limit the maximum number of different characters. The alphabet limit parameter is of significant importance in the quality of the generated vocabulary. As shown in appendix C, if the alphabet limit parameter is kept constant, the metrics do not change significantly.

Table 5.10. Impact of Vocabulary Size, ULM.

| **Vocabulary Size** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1000 | 5.4698 | 8.0283 | 1.0078 |
| 2000 | 5.3461 | 7.8720 | 1.0525 |
| 3000 | 3.0765 | 4.2064 | 1.1728 |
| 5000 | 2.2633 | 3.0117 | 1.9316 |
| 10000 | 2.0186 | 2.6142 | 2.2350 |
| 20000 | 1.7347 | 2.1583 | 2.8601 |
| 30000 | 1.5162 | 1.8087 | 4.2252 |

Table 5.11. Impact of Vocabulary Size, Wordpiece.

| **Vocabulary Size** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1000 | 5.4698 | 8.0283 | 1.0060 |
| 2000 | 5.4698 | 8.0283 | 1.0060 |
| 3000 | 2.3922 | 3.2006 | 1.8016 |
| 5000 | 1.9347 | 2.5326 | 2.8672 |
| 10000 | 1.6347 | 2.0659 | 4.0520 |
| 20000 | 1.4384 | 1.7451 | 5.0974 |
| 30000 | 1.3548 | 1.6049 | 5.6345 |

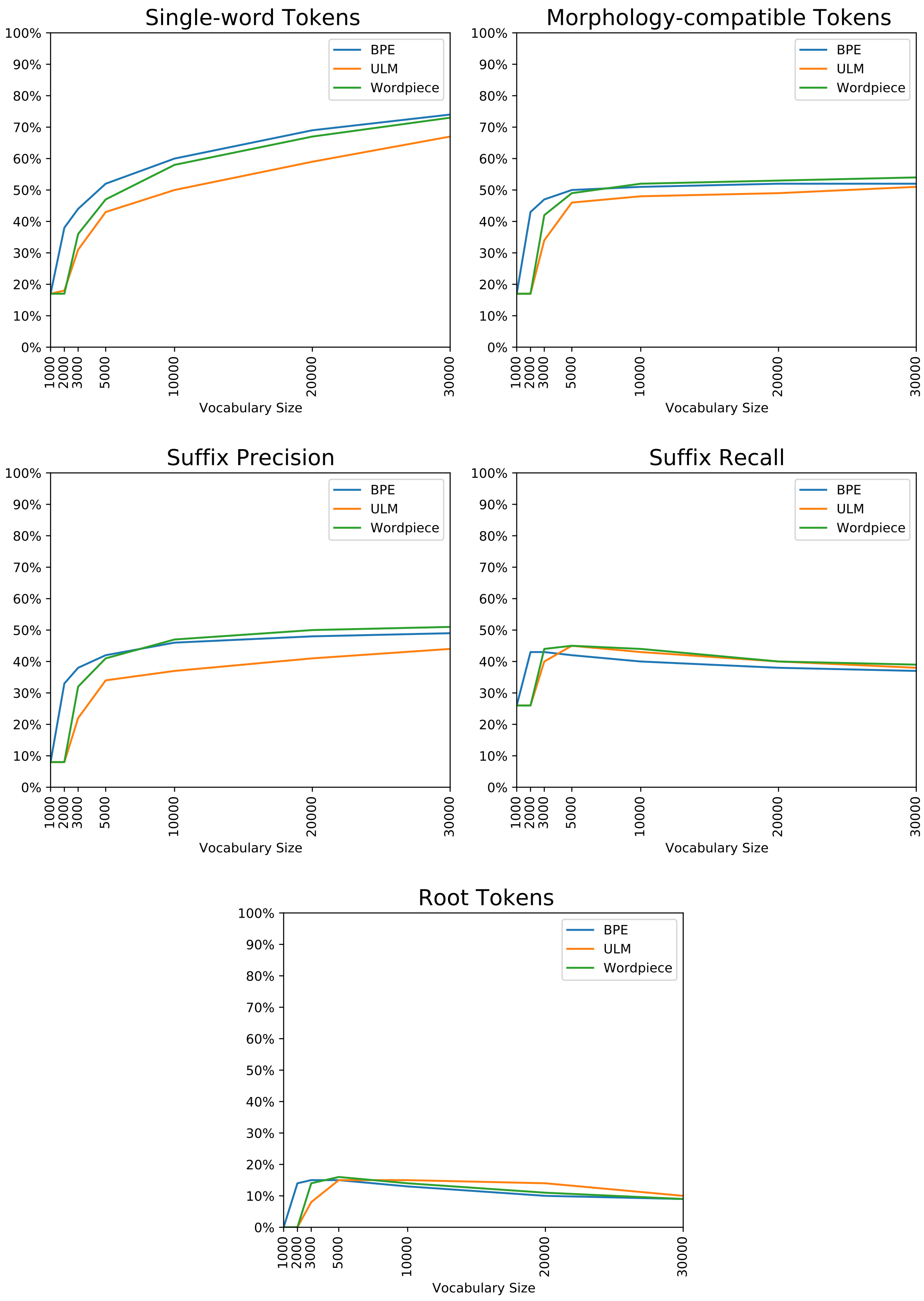

Figure 5.3. Impact of Vocabulary Size on Tokenizer Performance

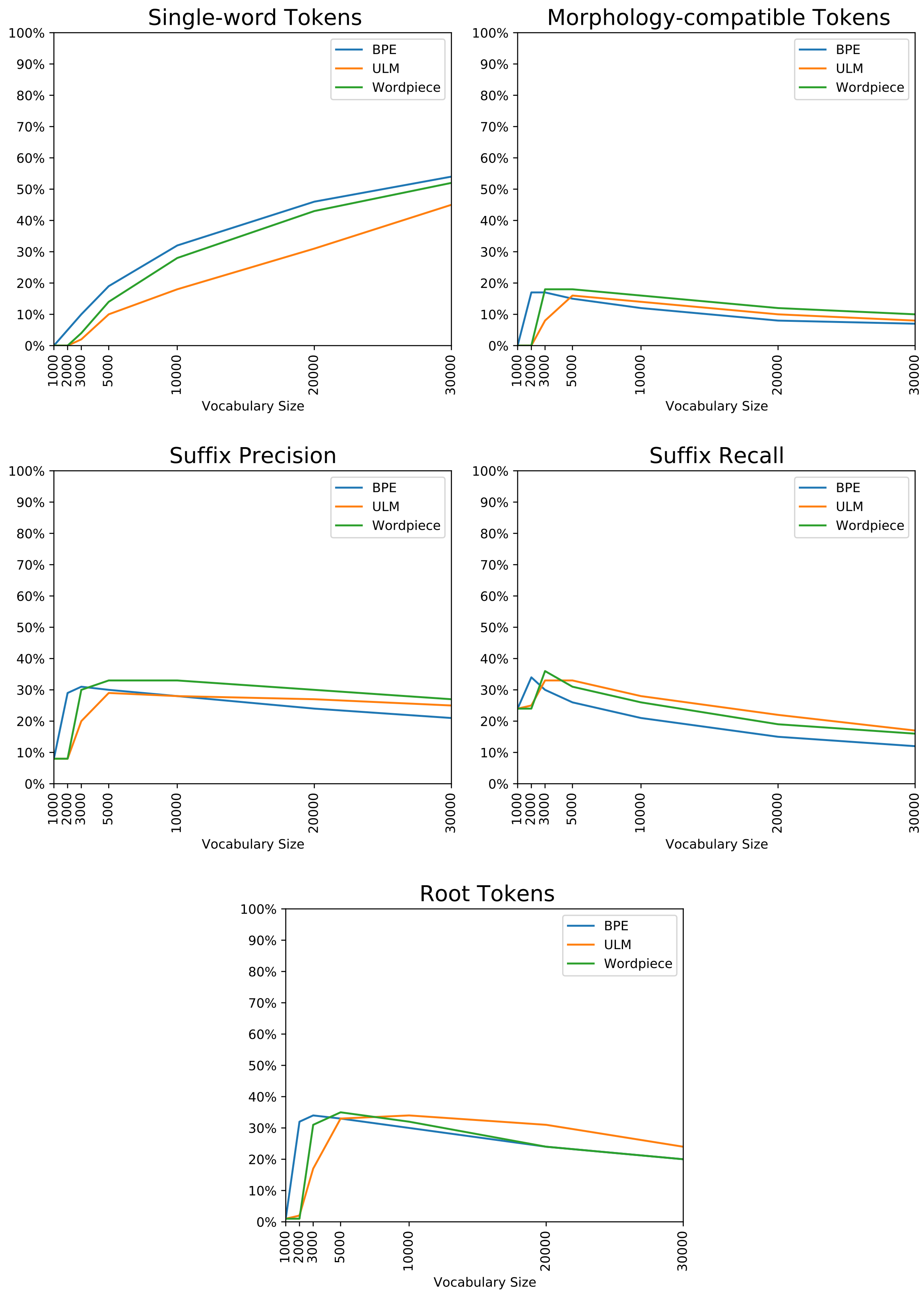

Figure 5.4. Impact of Vocabulary Size on Tokenizer Performance, words with suffixes

Table 5.12. Wordpiece tokenization example across different vocabulary sizes.

| **Vocabulary Size** | **Tokens** |
|---|---|
| 1000 | h, #a ,#z ,#i ,#n ,#e ,#s ,#i ,#d, #i, #r |
| 2000 | h, #a ,#z ,#i ,#n ,#e ,#s ,#i ,#d, #i, #r |
| 3000 | haz, #ine ,#si ,#dir |
| 5000 | haz, #ine ,#si ,#dir |
| 10000 | haz, #ine ,#si ,#dir |
| 20000 | hazin, #esi ,#dir |
| 30000 | hazine, #si ,#dir |

Table 5.12 shows how tokens change across different vocabulary sizes for the word *hazinesidir* (it is him/her treasure). As the vocabulary size increases, the Wordpiece tokenizer produces longer and fewer tokens. The WordPiece tokenizer trained with a vocabulary size of 1000 produces a token for each character. The WordPiece tokenizer trained with a vocabulary size of 30000 encodes the word exactly in accordance with the morphology of the word.

## 5.4. Morphologically Optimized Tokenizer

We trained the morphologically optimized tokenizer with a vocabulary size of 32000 on the Turkish part of the OSCAR corpus as explained in Section 4.3. We performed morphological analysis using Zemberek [32] to extract suffixes. Zemberek is an open-source library for Turkish and provides a variety of NLP solutions, including a morphological parser. The suffixes are listed in Table A.1. After the extracting suffixes, the corpus was updated in a way that the suffixes are enclosed with special characters so that the tokenizer recognizes suffixes during the training phase. For example, kalemleri was updated as kalem<ler><i>. Thus, the tokenizer can distinguish the suffixes, which are -ler and -i. The morphologically optimized tokenizer was trained using the updated corpus as explained in Section 4.3.

In addition to the morphologically optimized tokenizer, we trained the Wordpiece tokenizer with the same parameters and corpus to compare the tokenizers. Some examples from both tokenizers are given in Table 5.13. For example, the word *tesislerinde* (in their facilities) is not broken up and is represented as a single-word token with the Wordpiece tokenizer. Suppose we train a model with this tokenizer. In that case, the model can not associate *tesislerinde* (in their facilities) with *tesislerimde* (in my facilities) since the tokens of the words do not contain any common token. On the other hand, the morphologically optimized tokenizer tokenizes the word *tesislerinde* into morphology-compatible tokens. The token boundaries are exactly compatible with the morphology of the word; the morphologically optimized tokenizer can split properly the root, which is *tesis* (facility) and the suffixes, which are *ler*, *i*, and *nde*. In this way, if we train a model using the morphologically optimized tokenizer, the model can correlate *tesislerinde* (in their facilities) with *tesislerimde* (in my facilities) as the root of the words, which is *tesis* (facility), is represented separately.

For another example, the word *olmuş* (happened) is tokenized as *olm*, *#uş* with the Wordpiece tokenizer. It means that the vocabulary does not contain the word itself, and the word is split into smaller pieces, but the token *olm* is not meaningful in Turkish, neither root nor suffix. On the other side, if we tokenize the word *olmuş* using the morphologically optimized tokenizer, it can split the word into meaningful tokens which preserve morphological integrity.

We used the BOUN Treebank as the test dataset to compare the tokenizers. The dataset includes 44275 words with suffixes, 97424 words in total. We tokenized the dataset using Wordpiece and morphologically optimized tokenizers and analyzed them according to the metrics given in section 4.2. The comparison results are provided in Table 5.14.

The number of single-word tokens is higher in Wordpiece. Wordpiece represents words without splitting them into subwords compared to the morphologically optimized tokenizer. However, it does not encode morphology better, as shown in Table

Table 5.13. Morphological analysis, Wordpiece tokenization, and morphologically optimized tokenization of example words.

| **Word** | **Morphology** | **Wordpiece** | **Morphologically Optimized Tokenizer** |
|---|---|---|---|
| kumbaranın | kumbara, -nın | kum, #bara, #nın | kumbara, #nın |
| başlayan | başla, -yan | baş, #layan | başla, #yan |
| uğraştığı | uğraş, -tığ, -ı | u, #ğra, #ştı, #ğı | uğraş, #tığ, #ı |
| esiyordu | es, -iyor, -du | es, #iyordu | es, #iyor, #du |
| tesislerinde | tesis, -ler, -i, -nde | tesislerinde | tesis, #ler, #i, #nde |
| yaşamak | yaşa, -mak | yaşama, #k | yaşa, #mak |
| olmuş | ol, -muş | olm, #uş | ol, #muş |
| dağlarına | dağ, -lar, -ı, -na | da, #ğla, #rı, #na | dağ, #la, #rı, #na |
| sınarız | sına, -r, ız | sına, #rı, #z | sına, #r, #ız |
| kaptanının | kaptan, -ı, -nın | kaptanı, #nın | kaptan, #ı, #nın |

Table 5.14. Comparison of Wordpiece and Morphologically Optimized Tokenizers for all words in BOUN Treebank.

| | **Wordpiece** | **Morphologically Optimized Tokenizer** |
|---|---|---|
| **Single-word Tokens** | 73% | 47% |
| **Morphology-compatible Tokens** | 54% | 83% |
| **Suffix Precision** | 51% | 78% |
| **Suffix Recall** | 39% | 89% |
| **Root Tokens** | 8% | 38% |

5.13. The morphologically optimized tokenizer can only represent about half of the words as a whole. On the other hand, the morphologically optimized tokenizer encodes morphology better, especially for the longer words with more than one suffix. Suffix precision and recall are higher than Wordpiece. It means that the token boundaries are compatible with the suffixes of the words.

We also analysed only the words with at least one suffix in the BOUN Treebank dataset. The analysis is provided in Table 5.15.

Table 5.15. Comparison of Wordpiece and Morphologically Optimized Tokenizers for words with suffixes in BOUN Treebank.

| | **Wordpiece** | **Morphologically Optimized Tokenizer** |
|---|---|---|
| **Single-word Tokens** | 53% | 0% |
| **Morphology-compatible Tokens** | 10% | 76% |
| **Suffix Precision** | 27%. | 82% |
| **Suffix Recall** | 16% | 100% |
| **Root Tokens** | 19% | 76% |

If a word has a suffix, the morphologically optimized tokenizer always splits the word thanks to encoded morphology information. As a result of this, the morphologically optimized tokenizer does not generate a single-word token for the words with suffixes. On the other hand, the morphologically optimized tokenizer may not encode morphology perfectly. As shown in Table 5.15, the tokenizer can only encode 76% of the words in a morphologically compatible way. The rest of the words can not be fully encoded because their roots are broken up into smaller pieces. For example, the word *kasetin* (your tape) is tokenized as *kas*, *#et*, *#in*. The suffix, which is *-in*, is tokenized properly, but the root, which is *kaset* (tape), is tokenized into smaller pieces and does not preserve its integrity.

Fertility and average token length are provided in Table 5.16. Since the morpho-

Table 5.16. Comparison of Wordpiece and Morphologically Optimized Tokenizers with respect to fertility and average token length.

| | **Wordpiece** | **Morphologically Optimized Tokenizer** |
|---|---|---|
| **Fertility, all words** | 1.3413 | 1.9935 |
| **Fertility, words with suffix** | 1.5813 | 2.9480 |
| **Average Token Length** | 5.7159 | 5.0024 |

logically optimized tokenizer produces fewer single-word tokens, fertility is higher and the average token length is lower than Wordpiece.

### 5.4.1. Pretraining

We pretrained two models: ELECTRA with the Wordpiece tokenizer and ELECTRA with the morphologically optimized tokenizer. The pretraining parameters are given in Table B.1. Both models were trained on the Turkish part of OSCAR. We used the original implementation of ELECTRA [12]. We trained the models on a v3-8 Tensor Processing Unit (TPU) for 1M steps, and the training process took 12 hours for each model.

The experimental setup is shown in Figure 5.5; the models are the same apart from input vocabularies. The ELECTRA Morphology model was trained with the morphologically optimized tokenizer. The Electra model was trained with the Wordpiece tokenizer.

The ELECTRA Morphology model converges faster in fewer steps than ELECTRA, as shown in Figure 5.6.

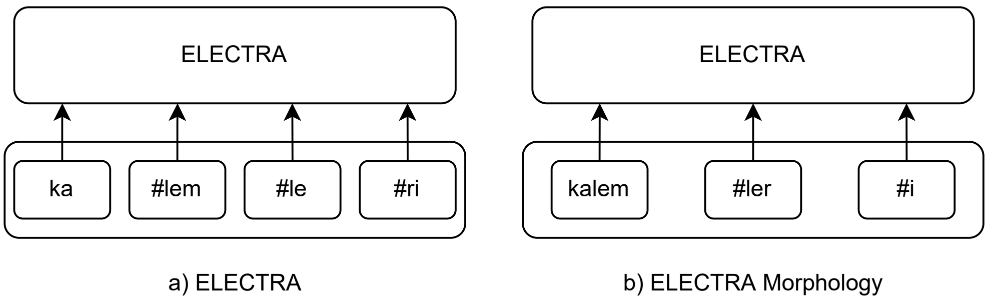

Figure 5.5. Experimental Setup

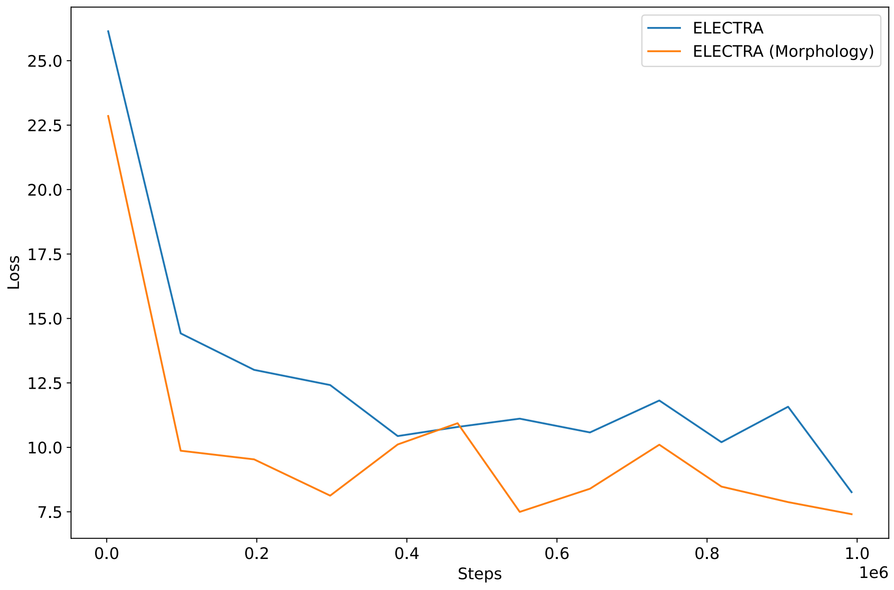

Figure 5.6. Loss of the two ELECTRA-based models

### 5.4.2. Pseudo-perplexity

We used pseudo-perplexity to compare the models intrinsically. In this experiment, we analyzed pseudo-perplexity scores for 2206 words that are tokenized in the same way. The scores are presented in Table 5.17.

Table 5.17. Pseudo-perplexity.

| | **Pseudo-perplexity** |
|---|---|
| ELECTRA | 80921 |
| ELECTRA (Morphology) | **67259** |

Pseudo-perplexity can be interpreted as the model's uncertainty. Less pseudo-perplexity is favourable over more pseudo-perplexity, similar to the case with the perplexity metric. The result shows that morphological information reduces overall perplexity, whereas the original model has a higher pseudo-perplexity score. A lower pseudo-perplexity score indicates better generalization performance.

## 5.5. Downstream Task Experiments

Since it is not straightforward to measure the impact of the morphologically optimized tokenizer, the models are tested on several downstream tasks: part-of-speech tagging, named-entity recognition, question answering, and sentiment analysis. We fine-tuned the models on the tasks.

### 5.5.1. Parts-of-Speech Tagging

Parts-of-speech (PoS) tagging is the task of tagging each word in a sentence with its proper parts-of-speech, such as noun or verb. We added a linear classification layer on top of the models and fine-tuned the entire architecture for predicting the parts-of-speech tags for single tokens. We fine-tuned the models on 4 datasets: BOUN [30], IMST [33], Kenet [34], and Penn [35]. The hyper-parameters are given in Table B.2.

The accuracy scores are shown in Tables 5.18 - 5.21.

Table 5.18. Performance comparison of the models, PoS Tagging, BOUN.

| **Training Steps** | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| 200k | 89.97 | **90.17** |
| 400k | 90.21 | **90.38** |
| 600k | 90.40 | **90.56** |
| 800k | 90.54 | **90.59** |
| 1M | 90.62 | **90.64** |

Table 5.19. Performance comparison of the models, PoS Tagging, IMST.

| **Training Steps** | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| 200k | 93.57 | **94.73** |
| 400k | 94.31 | **95.11** |
| 600k | 94.78 | **95.20** |
| 800k | 95.41 | **95.42** |
| 1M | 95.36 | **95.56** |

Table 5.20. Performance comparison of the models, PoS Tagging, Kenet.

| **Training Steps** | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| 200k | 92.82 | **93.05** |
| 400k | 92.80 | **93.07** |
| 600k | 93.20 | **93.48** |
| 800k | 93.22 | **93.46** |
| 1M | 93.19 | **93.52** |

Table 5.21. Performance comparison of the models, PoS Tagging, Penn.

| **Training Steps** | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| 200k | 94.24 | **94.41** |
| 400k | 94.38 | **94.58** |
| 600k | 94.41 | **94.60** |
| 800k | 94.47 | **94.61** |
| 1M | **94.58** | 94.54 |

The ELECTRA Morphology model generally outperforms the ELECTRA model. Especially at earlier training steps, The ELECTRA Morphology model achieves a relatively better score than the ELECTRA model.

### 5.5.2. Named-entity Recognition

Named-entity recognition (NER) is the task of identifying entities, such as person, location, and organization. We used a similar architecture to parts-of-speech tagging. We added a linear classification layer on top of the models and fine-tuned the entire architecture for predicting entity tags for single tokens. We fine-tuned the models on the Turkish dataset from XTREME [36]. The hyper-parameters are given in Table B.2. The accuracy results are shown in Table 5.22.

Table 5.22. Performance comparison of the models, NER, XTREME.

| **Training Steps** | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| 200k | 86.02 | **86.40** |
| 400k | 87.41 | **87.48** |
| 600k | 88.21 | **89.04** |
| 800k | 89.08 | **89.44** |
| 1M | 89.27 | **89.32** |

The ELECTRA Morphology model outperforms the ELECTRA model on the named-entity recognition task.

### 5.5.3. Question Answering

The question answering task extracts an answer given a question from a passage of text. The model takes two sequences separated by the [SEP] token: a question and a passage. The model is optimized to predict the index of starting and ending answer tokens. We fine-tuned the models on TQuAD [37]. The dataset contains 9200 question-answer pairs. We used the Hugging Face question-answering pipeline. The hyper-parameters are given in Table B.3. The F-measure (F1) and Exact match (EM) scores are shown in Table 5.23.

Table 5.23. Performance comparison of the models, QA, TQuAD.

| | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| F1 | 57.38 | **58.09** |
| EM | 37.78 | **38.17** |

The ELECTRA Morphology model outperforms the ELECTRA model on the question answering task.

### 5.5.4. Sentiment Analysis

Sentiment analysis is the task of predicting the sentiment of a sentence. We added a classification layer that takes the [CLS] representation as input and fine-tuned the model to predict sentiment. We fine-tuned the models on the Turkish movie review dataset from Beyazperde [38]. The dataset contains 5331 positive and 5331 negative sentences. The hyper-parameters are given in Table B.4. The accuracy scores are shown in Table 5.24.

Table 5.24. Performance comparison of the models, sentiment analysis, Beyazperde.

| | **ELECTRA** | **ELECTRA (Morphology)** |
|---|---|---|
| Accuracy | 0.8630 | **0.8788** |

The ELECTRA Morphology model performs better on the sentiment analysis than the ELECTRA model.

# 6. CONCLUSION & FUTURE WORK

## 6.1. Conclusion

In this thesis, we analyzed the three most widely used tokenization algorithms: BPE, Wordpiece, and ULM. We defined various metrics to evaluate how well tokenizers encode a morphologically rich language. We have found that Wordpiece tends to produce more single-word tokens, along with morphology-compatible tokens, than BPE and ULM. The analysis of tokenizers trained with the same corpus and parameters has shown that the tokenizers can represent about half of the words as a whole. However, 10% of the words could be tokenized in a morphologically compatible way.

We analyzed the impact of corpus size on tokenizer performance. The tokenizers are getting to produce more and shorter tokens as the corpus size increases. However, the tokenizers become saturated after a certain point, and the training corpus size does not impact that much. Training with more data causes a decrease in the number of single-word tokens, but it drastically increases the number of morphology-compatible tokens.

The tokenizers produce fewer tokens per word as the vocabulary size increases. Increasing vocabulary size makes tokenizers generate more and shorter tokens. A vocabulary size of between 3000 and 5000 encodes morphology better. After that size, the number of root tokens gradually decreases.

We proposed a morphologically optimized tokenizer to encode morphology better. We pretrained two models: ELECTRA with the Wordpiece tokenizer and ELECTRA with the morphologically optimized tokenizer. The models are the same apart from input vocabularies. We analyzed pseudo-perplexity scores to compare the models intrinsically. The experiment has shown that morphological information reduces overall perplexity, whereas the original model has a higher pseudo-perplexity score.

We also fine-tuned the models to measure the impact of the morphologically optimized tokenizer. The downstream task experiments show that the model trained with the morphologically optimized tokenizer slightly outperforms the model trained with the Wordpiece tokenizer.

## 6.2. Future Work

For future work, we plan to train more models from scratch to compare the impact of the morphologically optimized tokenizer. As shown in various studies [39,40], adversarially fine-tuning improves performance. Similar to these approaches, training with both Wordpiece and morphologically optimized segmentations can be tested. Also, examining how morphological information is processed along the way in a model in terms of self-attention weights is a promising study.

# REFERENCES

1. Gage, P., "A new algorithm for data compression", *C Users Journal*, Vol. 12, No. 2, pp. 23–38, 1994.

2. Devlin, J., M.-W. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Association for Computational Linguistics, Minneapolis, Minnesota, Jun. 2019, `https://aclanthology.org/N19-1423`.

3. Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, "Language Models are Few-Shot Learners", H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan and H. Lin (Editors), *Advances in Neural Information Processing Systems*, Vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.

4. Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", *Journal of Machine Learning Research*, Vol. 21, No. 140, pp. 1–67, 2020, `http://jmlr.org/papers/v21/20-074.html`.

5. Sennrich, R., B. Haddow and A. Birch, "Neural Machine Translation of Rare Words with Subword Units", pp. 1715–1725, Aug. 2016, `https://aclanthology.org/P16-1162`.

6. Kudo, T., "Subword Regularization: Improving Neural Network Translation Mod-

els with Multiple Subword Candidates", *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 66–75, Association for Computational Linguistics, Melbourne, Australia, Jul. 2018, `https://aclanthology.org/P18-1007`.

7. Schuster, M. and K. Nakajima, "Japanese and korean voice search", *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5149–5152, IEEE, 2012.

8. Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach", *arXiv preprint arXiv:1907.11692*, 2019.

9. Geva, M., R. Schuster, J. Berant and O. Levy, "Transformer Feed-Forward Layers Are Key-Value Memories", *Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

10. Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention is all you need", *Advances in neural information processing systems*, pp. 5998–6008, 2017.

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

12. Clark, K., M.-T. Luong, Q. V. Le and C. D. Manning, "ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators", *ICLR*, 2020, `https://openreview.net/pdf?id=r1xMH1BtvB`.

13. Şahin, H., "Türkçe'de Ön Ek", *Uludağ Üniversitesi Fen-Edebiyat Fakültesi Sosyal Bilimler Dergisi*, Vol. 7, No. 10, pp. 65–77, 2006.

14. Mete, F., “In Turkish: Sentence Structure and Possible Sentences According to the Sequence of Elements.”, *Journal of Education and Training Studies*, Vol. 4, No. 10, pp. 221–231, 2016.

15. Hoffman, B., M. Walker, E. Prince and A. J. Oxford, “Word order, information structure, and centering in Turkish”, *Centering in discourse*, Citeseer, 1997.

16. Kudo, T. and J. Richardson, “SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing”, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, Association for Computational Linguistics, Brussels, Belgium, Nov. 2018, `https://aclanthology.org/D18-2012`.

17. Choe, D., R. Al-Rfou, M. Guo, H. Lee and N. Constant, “Bridging the gap for tokenizer-free language models”, *arXiv preprint arXiv:1908.10322*, 2019.

18. Clark, J. H., D. Garrette, I. Turc and J. Wieting, “Canine: Pre-training an efficient tokenization-free encoder for language representation”, *arXiv preprint arXiv:2103.06874*, 2021.

19. Blevins, T. and L. Zettlemoyer, “Better Character Language Modeling through Morphology”, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 1606–1613, Association for Computational Linguistics, Florence, Italy, Jul. 2019, `https://www.aclweb.org/anthology/P19-1156`.

20. Alyafeai, Z., M. S. Al-shaibani, M. Ghaleb and I. Ahmad, “Evaluating various tokenizers for arabic text classification”, *arXiv preprint arXiv:2106.07540*, 2021.

21. Matthews, A., G. Neubig and C. Dyer, “Using morphological knowledge in open-vocabulary neural language models”, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1435–1445, 2018.

22. Domingo, M., M. Garcıa-Martınez, A. Helle, F. Casacuberta and M. Herranz, "How much does tokenization affect neural machine translation?", *arXiv preprint arXiv:1812.08621*, 2018.

23. Mielke, S. J., Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot *et al.*, "Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP", *arXiv preprint arXiv:2112.10508*, 2021.

24. Radford, A. and K. Narasimhan, "Improving Language Understanding by Generative Pre-Training", , 2018.

25. Radford, A., J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language Models are Unsupervised Multitask Learners", , 2019.

26. Sanh, V., L. Debut, J. Chaumond and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter", *arXiv preprint arXiv:1910.01108*, 2019.

27. Salazar, J., D. Liang, T. Q. Nguyen and K. Kirchhoff, "Masked Language Model Scoring", *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2699–2712, Association for Computational Linguistics, Online, Jul. 2020, `https://aclanthology.org/2020.acl-main.240`.

28. Shin, J., Y. Lee and K. Jung, "Effective sentence scoring method using bert for speech recognition", *Asian Conference on Machine Learning*, pp. 1081–1093, PMLR, 2019.

29. Abadji, J., P. J. O. Suárez, L. Romary and B. Sagot, "Ungoliant: An optimized pipeline for the generation of a very large-scale multilingual web corpus", Proceedings of the Workshop on Challenges in the Management of Large Corpora (CMLC-9) 2021. Limerick, 12 July 2021 (Online-

Event), pp. 1 – 9, Leibniz-Institut für Deutsche Sprache, Mannheim, 2021, `https://nbn-resolving.org/urn:nbn:de:bsz:mh39-104688`.

30. Türk, U., F. Atmaca, Ş. B. Özateş, G. Berk, S. T. Bedir, A. Köksal, B. Ö. Başaran, T. Güngör and A. Özgür, "Resources for Turkish dependency parsing: Introducing the BOUN treebank and the BoAT annotation tool", *Language Resources and Evaluation*, pp. 1–49, 2021.

31. Wolf, T., L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest and A. Rush, "Transformers: State-of-the-Art Natural Language Processing", *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Association for Computational Linguistics, Online, Oct. 2020, `https://aclanthology.org/2020.emnlp-demos.6`.

32. Akın, A. A. and M. D. Akın, "Zemberek, an open source NLP framework for Turkic languages", *Structure*, Vol. 10, No. 2007, pp. 1–5, 2007.

33. Sulubacak, U., G. Eryiğit, T. Pamay *et al.*, "IMST: A revisited Turkish dependency treebank", *Proceedings of TurCLing 2016, the 1st International Conference on Turkic Computational Linguistics*, Ege University Press, 2016.

34. Bakay, Ö., Ö. Ergelen, E. Sarmış, S. Yıldırım, B. N. Arıcan, A. Kocabalcıoğlu, M. Özçelik, E. Sanıyar, O. Kuyrukçu, B. Avar *et al.*, "Turkish wordnet kenet", *Proceedings of the 11th global wordnet conference*, pp. 166–174, 2021.

35. Kuzgun, A., N. Cesur, B. N. Arıcan, M. Özçelik, B. Marşan, N. Kara, D. B. Aslan and O. T. Yıldız, "On building the largest and cross-linguistic turkish dependency corpus", *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pp. 1–6, IEEE, 2020.

36. Hu, J., S. Ruder, A. Siddhant, G. Neubig, O. Firat and M. Johnson, “Xtreme: A massively multilingual multi-task benchmark for evaluating cross-lingual generalisation”, *International Conference on Machine Learning*, pp. 4411–4421, PMLR, 2020.

37. Soygazi, F., O. Çiftçi, U. Kök and S. Cengiz, “THQuAD: Turkish Historic Question Answering Dataset for Reading Comprehension”, *2021 6th International Conference on Computer Science and Engineering (UBMK)*, pp. 215–220, IEEE, 2021.

38. Demirtas, E. and M. Pechenizkiy, “Cross-lingual polarity detection with machine translation”, *Proceedings of the Second International Workshop on Issues of Sentiment Discovery and Opinion Mining*, pp. 1–8, 2013.

39. Tan, S., S. Joty, M.-Y. Kan and R. Socher, “It’s Morphin’ Time! Combating Linguistic Discrimination with Inflectional Perturbations”, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2920–2935, Association for Computational Linguistics, Online, Jul. 2020, `https://aclanthology.org/2020.acl-main.263`.

40. Tan, S., S. Joty, L. Varshney and M. Kan, “Mind your inflections! Improving NLP for non-standard Englishes with base-inflection encoding”, *EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, pp. 5647–5663, Association for Computational Linguistics (ACL), 2020.

# APPENDIX A: TURKISH SUFFIXES

Table A.1. Turkish Suffixes.

| | **Suffix** | | **Suffix** | | **Suffix** | | **Suffix** |
|---|---|---|---|---|---|---|---|
| **1** | -a | **26** | -ci | **51** | -duğ | **76** | -er |
| **2** | -abil | **27** | -cik | **52** | -dü | **77** | -erek |
| **3** | -acak | **28** | -ciğ | **53** | -dük | **78** | -esi |
| **4** | -acağ | **29** | -cu | **54** | -dükçe | **79** | -esiye |
| **5** | -adur | **30** | -cuk | **55** | -dür | **80** | -eyaz |
| **6** | -agel | **31** | -cuğ | **56** | -düğ | **81** | -i |
| **7** | -agör | **32** | -cü | **57** | -dı | **82** | -ici |
| **8** | -akal | **33** | -cük | **58** | -dık | **83** | -il |
| **9** | -akoy | **34** | -cüğ | **59** | -dıkça | **84** | -im |
| **10** | -alı | **35** | -cı | **60** | -dır | **85** | -imiz |
| **11** | -am | **36** | -cık | **61** | -dığ | **86** | -imle |
| **12** | -ama | **37** | -cığ | **62** | -e | **87** | -imsi |
| **13** | -amadan | **38** | -da | **63** | -ebil | **88** | -in |
| **14** | -an | **39** | -dan | **64** | -ecek | **89** | -ince |
| **15** | -ar | **40** | -de | **65** | -eceğ | **90** | -inil |
| **16** | -arak | **41** | -den | **66** | -edur | **91** | -iniz |
| **17** | -ası | **42** | -di | **67** | -egel | **92** | -inle |
| **18** | -asıya | **43** | -dik | **68** | -egör | **93** | -ip |
| **19** | -ayaz | **44** | -dikçe | **69** | -ekal | **94** | -ir |
| **20** | -ca | **45** | -dir | **70** | -ekoy | **95** | -iver |
| **21** | -casına | **46** | -diğ | **71** | -eli | **96** | -iyor |
| **22** | -cağız | **47** | -du | **72** | -em | **97** | -iz |
| **23** | -ce | **48** | -duk | **73** | -eme | **98** | -iş |
| **24** | -cesine | **49** | -dukça | **74** | -emeden | **99** | -k |
| **25** | -ceğiz | **50** | -dur | **75** | -en | **100** | -ken |

Table A.1. Turkish Suffixes. (cont.)

| | **Suffix** | | **Suffix** | | **Suffix** | | **Suffix** |
|---|---|---|---|---|---|---|---|
| **101** | -ki | **126** | -lığ | **151** | -msı | **176** | -nul |
| **102** | -kü | **127** | -m | **152** | -muz | **177** | -nun |
| **103** | -la | **128** | -ma | **153** | -muş | **178** | -nunla |
| **104** | -lan | **129** | -maca | **154** | -müz | **179** | -nuz |
| **105** | -lar | **130** | -madan | **155** | -müş | **180** | -nü |
| **106** | -ları | **131** | -mak | **156** | -mız | **181** | -nül |
| **107** | -laş | **132** | -maksızın | **157** | -mış | **182** | -nün |
| **108** | -le | **133** | -makta | **158** | -n | **183** | -nüz |
| **109** | -len | **134** | -malı | **159** | -na | **184** | -nı |
| **110** | -ler | **135** | -mazlık | **160** | -nca | **185** | -nıl |
| **111** | -leri | **136** | -mazlığ | **161** | -nce | **186** | -nın |
| **112** | -leş | **137** | -me | **162** | -nda | **187** | -nız |
| **113** | -li | **138** | -mece | **163** | -ndan | **188** | -r |
| **114** | -lik | **139** | -meden | **164** | -nde | **189** | -sa |
| **115** | -lim | **140** | -mek | **165** | -nden | **190** | -sal |
| **116** | -liğ | **141** | -meksizin | **166** | -ne | **191** | -sana |
| **117** | -lu | **142** | -mekte | **167** | -ni | **192** | -sanıza |
| **118** | -luk | **143** | -meli | **168** | -nil | **193** | -se |
| **119** | -luğ | **144** | -mezlik | **169** | -nin | **194** | -sel |
| **120** | -lü | **145** | -mezliğ | **170** | -niz | **195** | -sene |
| **121** | -lük | **146** | -miz | **171** | -nla | **196** | -senize |
| **122** | -lüğ | **147** | -miş | **172** | -nlar | **197** | -si |
| **123** | -lı | **148** | -msi | **173** | -nlu | **198** | -sin |
| **124** | -lık | **149** | -msu | **174** | -nsuz | **199** | -siniz |
| **125** | -lım | **150** | -msü | **175** | -nu | **200** | -sinler |

Table A.1. Turkish Suffixes. (cont.)

| | Suffix | | Suffix | | Suffix | | Suffix |
|---|---|---|---|---|---|---|---|
| **201** | -siz | **226** | -tiğ | **251** | -unuz | **276** | -ydi |
| **202** | -su | **227** | -tu | **252** | -up | **277** | -ydu |
| **203** | -sun | **228** | -tuk | **253** | -ur | **278** | -ydü |
| **204** | -sunlar | **229** | -tukça | **254** | -uver | **279** | -ydı |
| **205** | -sunuz | **230** | -tur | **255** | -uyor | **280** | -ye |
| **206** | -suz | **231** | -tuğ | **256** | -uz | **281** | -yebil |
| **207** | -sü | **232** | -tü | **257** | -uş | **282** | -yecek |
| **208** | -sün | **233** | -tük | **258** | -ya | **283** | -yeceğ |
| **209** | -sünler | **234** | -tükçe | **259** | -yabil | **284** | -yedur |
| **210** | -sünüz | **235** | -tür | **260** | -yacak | **285** | -yegel |
| **211** | -süz | **236** | -tüğ | **261** | -yacağ | **286** | -yegör |
| **212** | -sı | **237** | -tı | **262** | -yadur | **287** | -yekal |
| **213** | -sın | **238** | -tık | **263** | -yagel | **288** | -yekoy |
| **214** | -sınlar | **239** | -tıkça | **264** | -yagör | **289** | -yeli |
| **215** | -sınız | **240** | -tır | **265** | -yakal | **290** | -yem |
| **216** | -sız | **241** | -tığ | **266** | -yakoy | **291** | -yeme |
| **217** | -t | **242** | -u | **267** | -yalı | **292** | -yemeden |
| **218** | -ta | **243** | -ucu | **268** | -yam | **293** | -yen |
| **219** | -tan | **244** | -ul | **269** | -yama | **294** | -yerek |
| **220** | -te | **245** | -um | **270** | -yamadan | **295** | -yesi |
| **221** | -ten | **246** | -umsu | **271** | -yan | **296** | -yesiye |
| **222** | -ti | **247** | -umuz | **272** | -yarak | **297** | -yeyaz |
| **223** | -tik | **248** | -un | **273** | -yası | **298** | -yi |
| **224** | -tikçe | **249** | -unca | **274** | -yasıya | **299** | -yici |
| **225** | -tir | **250** | -unul | **275** | -yayaz | **300** | -yim |

Table A.1. Turkish Suffixes. (cont.)

| | **Suffix** | | **Suffix** | | **Suffix** | | **Suffix** |
|---|---|---|---|---|---|---|---|
| **301** | -yimiz | **326** | -yunca | **351** | -yış | **376** | -ünce |
| **302** | -yin | **327** | -yunuz | **352** | -z | **377** | -ünül |
| **303** | -yince | **328** | -yup | **353** | -ça | **378** | -ünüz |
| **304** | -yiniz | **329** | -yuver | **354** | -çasına | **379** | -üp |
| **305** | -yip | **330** | -yuz | **355** | -çe | **380** | -ür |
| **306** | -yiver | **331** | -yuş | **356** | -çesine | **381** | -üver |
| **307** | -yiz | **332** | -yü | **357** | -çi | **382** | -üyor |
| **308** | -yiş | **333** | -yücü | **358** | -çik | **383** | -üz |
| **309** | -yken | **334** | -yüm | **359** | -çiğ | **384** | -üş |
| **310** | -yla | **335** | -yün | **360** | -çu | **385** | -ı |
| **311** | -yle | **336** | -yünce | **361** | -çuk | **386** | -ıcı |
| **312** | -yli | **337** | -yünüz | **362** | -çuğ | **387** | -ıl |
| **313** | -ymiş | **338** | -yüp | **363** | -çü | **388** | -ım |
| **314** | -ymuş | **339** | -yüver | **364** | -çük | **389** | -ımsı |
| **315** | -ymüş | **340** | -yüz | **365** | -çüğ | **390** | -ımız |
| **316** | -ymış | **341** | -yüş | **366** | -çı | **391** | -ın |
| **317** | -yor | **342** | -yı | **367** | -çık | **392** | -ınca |
| **318** | -ysa | **343** | -yıcı | **368** | -çığ | **393** | -ınla |
| **319** | -yse | **344** | -yım | **369** | -ü | **394** | -ınıl |
| **320** | -ysiz | **345** | -yın | **370** | -ücü | **395** | -ınız |
| **321** | -yu | **346** | -yınca | **371** | -ül | **396** | -ıp |
| **322** | -yucu | **347** | -yınız | **372** | -üm | **397** | -ır |
| **323** | -yum | **348** | -yıp | **373** | -ümsü | **398** | -ıver |
| **324** | -yumuz | **349** | -yıver | **374** | -ümüz | **399** | -ıyor |
| **325** | -yun | **350** | -yız | **375** | -ün | **400** | -ız |

Table A.1. Turkish Suffixes. (cont.)

| | **Suffix** | | **Suffix** | | **Suffix** | | **Suffix** |
|---|---|---|---|---|---|---|---|
| **401** | -ış | | | | | | |

# APPENDIX B: MODEL HYPERPARAMETERS

Table B.1. ELECTRA Pretraining Hyperparameters.

| **Parameter** | |
|---|---|
| Learning rate | 0.0005 |
| Batch size | 128 |
| Weight decay | 0.01 |
| Sequence length | 128 |

Table B.2. NER, PoS Tagging Fine-tuning Hyperparameters.

| **Parameter** | |
|---|---|
| Learning rate | 0.00005 |
| Epochs | 10 |
| Batch size | 16 |
| Optimizer | AdamW |
| Weight decay | 0.01 |

Table B.3. QA Fine-tuning Hyperparameters.

| **Parameter** | |
|---|---|
| Learning rate | 0.00003 |
| Epochs | 20 |
| Batch size | 32 |
| Optimizer | AdamW |
| Weight decay | 0.01 |
| Max sequence length | 384 |
| Doc stride | 128 |

Table B.4. Sentiment Analysis Fine-tuning Hyperparameters.

| **Parameter** | |
|---|---|
| Learning rate | 0.00002 |
| Epochs | 5 |
| Batch size | 32 |
| Optimizer | AdamW |
| Weight decay | 0.01 |

# APPENDIX C: ALPHABET LIMIT PARAMETER

Table C.1. Impact of Corpus Size, BPE.

| **Corpus Size - GB** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1 | 1.3155 | 1.5419 | 6.0766 |
| 2 | 1.3155 | 1.5412 | 6.0789 |
| 3 | 1.3147 | 1.5389 | 6.0856 |
| 5 | 1.3137 | 1.5372 | 6.0901 |
| 10 | 1.3134 | 1.5361 | 6.0938 |
| 20 | 1.3136 | 1.5360 | 6.0942 |
| 30 | 1.3135 | 1.5358 | 6.0974 |

Table C.2. Impact of Corpus Size, Wordpiece.

| **Corpus Size - GB** | **Fertility** | **Fertility, words with suffix** | **Average Token Length** |
|---|---|---|---|
| 1 | 1.3444 | 1.5881 | 5.6984 |
| 2 | 1.3432 | 1.5852 | 5.7056 |
| 3 | 1.3441 | 1.5865 | 5.7082 |
| 5 | 1.3432 | 1.5860 | 5.7109 |
| 10 | 1.3426 | 1.5840 | 5.7140 |
| 20 | 1.3425 | 1.5838 | 5.7146 |
| 30 | 1.3413 | 1.5815 | 5.7163 |