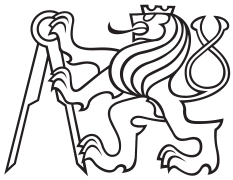
Bachelor Project



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Optimization of Text Tokenization for Efficient Language Models Training

Alina Haitota

Supervisor: Ing. David Herel
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2025

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Haitota Alina** Personal ID number: **518303**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Optimization of Text Tokenization for Efficient Language Models Training

Bachelor’s thesis title in Czech:

Optimalizace tokenizace textu pro efektivní trénování jazykových modelů

Guidelines:

Tokenization is a critical step in preparing textual data for training language models. This thesis explores advanced tokenization strategies, including the integration of multiple token streams from diverse algorithms. Additionally, it investigates the combination of subword and multiword units to enhance computational efficiency and reduce the need for full forward and backward passes for every token in the training dataset.
1. Conduct a literature review on tokenization techniques and their applications in natural language processing (NLP).
2. Implement a tokenization pipeline combining multiple streams of tokens (e.g., original text and lowercased versions).
3. Experiment with the integration of subword and multiword units and analyze their impact on computational efficiency.
4. Evaluate the proposed approaches using state-of-the-art NLP models and datasets, focusing on training time and performance metrics.
5. Provide a comparative analysis of the results, highlighting trade-offs and recommendations for future research.

Bibliography / sources:

[1] David Herel, & Tomas Mikolov. (2024). Thinking Tokens for Language Modeling, AITP 2023
[2] David Herel, & Tomas Mikolov. (2023). Advancing State of the Art in Language Modeling, preprint
[3] David Herel, & Tomas Mikolov. (2024). Collapse of Self-trained Language Models. ICLR 2024
[4] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., … Amodei, D. (2020). Language Models are Few-Shot Learners. CoRR, abs/2005.14165. Retrieved from https://arxiv.org/abs/2005.14165
[5] MIKOLOV, Tomáš. STATISTICAL LANGUAGE MODELS BASED ON NEURAL NETWORKS. Brno, 2012. Ph.D. Thesis. Brno University of Technology, Faculty of Information Technology. 2012-10-02. Supervised by Černocký Jan. Available from: https://www.fit.vut.cz/study/phd-thesis/283/

Name and workplace of bachelor’s thesis supervisor:

Ing. David Herel Foundational AI CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **22.01.2025** Deadline for bachelor thesis submission: **23.05.2025**

Assignment valid until: **20.09.2026**

prof. Dr. Ing. Jan Kybic
Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Haitota Alina
Personal number: 518303
Programme name: Open Informatics

declare that I have elaborated the bachelor's thesis entitled

Optimization of Text Tokenization for Efficient Language Models Training

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 17.05.2025

Alina Haitota
.....
student's signature

Acknowledgements

Heartfelt appreciation to Roma for your endless support, and infinite gratitude to my family, whose faith in me has been my guiding light.

Abstract

Tokenization is a critical step in preparing textual data for training language models. This thesis explores advanced tokenization strategies, including the integration of multiple token streams from diverse algorithms. Additionally, it investigates the combination of subword and multiword units to enhance computational efficiency and reduce the need for full forward and backward passes for every token in the training dataset.

Keywords: Tokenization, Language Models, Subword, Multiword, Token Streams, DualStream Architecture, Text Preprocessing

Supervisor: Ing. David Herel
CIIRC CTU,
FAI: základní výzkum AI,
Jugoslávských partyzánů 1580/3,
160 00 Praha 6 - Dejvice

Abstrakt

Tokenizace je klíčovým krokem při přípravě textových dat pro trénování jazykových modelů. Tato práce zkoumá pokročilé strategie tokenizace, včetně integrace více tokenových proudů z různých algoritmů. Dále se zabývá kombinací podslovních a víceslovných jednotek s cílem zvýšit výpočetní efektivitu a snížit potřebu provádění úplných dopředných a zpětných průchodů pro každý token v trénovací datové sadě.

Klíčová slova: Tokenizace, Jazykové Modely, Podslovo, Víceslov, Tokenové Toky, Architektura DualStream, Předzpracování Textu

Překlad názvu: Optimalizace tokenizace textu pro efektivní trénování jazykových modelů

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Natural Language Processing

Natural language processing (NLP) [1] is a field of Artificial intelligence (AI) that uses machine learning to help computers understand human language. The complexity, uncertainty and variety of human language make NLP a challenging field. Structured data generally has a clear pattern, whereas natural language is often unstructured and requires complicated models to understand the structure of human language. NLP is a key part of modern software because it can process and analyse huge amounts of text effectively.

1.2 Tokenization

Tokenization is a fundamental process of dividing text into smaller parts known as tokens. These tokens can be any size: individual words, subwords, or even characters. Afterwards, these tokens are represented by a vector of numbers by embedding layer. Tokenization helps the model to treat text as numbers. The efficiency of tokenization has a direct impact on how language models understand, interpret, and generate human language.

There are several types of tokenization 1.1: Word-level - splits into words, Character-level - splits into characters. Subword tokenization processes the data by breaking words into smaller, meaningful subword units. This approach is helpful for dealing with uncommon or new words by breaking them down into smaller parts, which are more likely to already be in the vocabulary.

Byte-Pair Encoding, or BPE [2], is a way to break down text into smaller pieces, called subwords. BPE scans a large amount of text, finds the pair of characters (or bytes) that appears together most frequently, and merges them into a single new unit or ”token.” It does this repeatedly until it has built up a vocabulary of a specific size.

WordPiece [3] is quite similar to BPE, but it relies on a probabilistic, likelihood-based approach rather than solely on frequency counts. It builds a vocabulary by combining smaller parts of words in a way that maximizes the likelihood of the training data. This approach has been used in models like BERT.

SentencePiece [4] stands out as a tokenizer designed to be language-neutral because it works directly with the raw character stream, avoiding any need for initial word-splitting based on spaces or other delimiters. This characteristic is especially valuable for languages that do not use clear word separators.

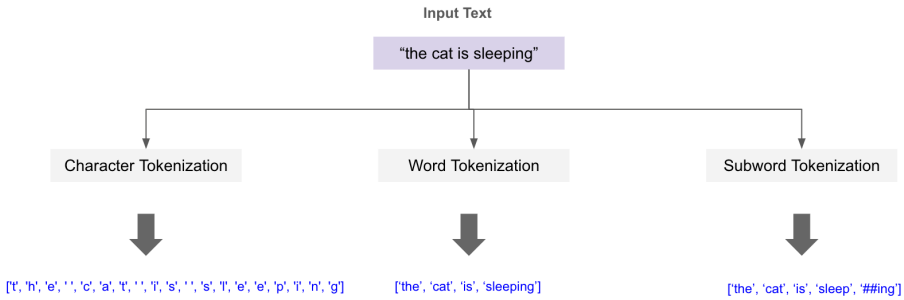


Figure 1.1: Classifications of tokenization

1.3 Motivation

Currently, the Large Language Models (LLMs) are used for the most NLP task. LLM is a machine learning transformer-based model that is trained on huge amounts of data. It takes a great deal of time, resources and finance investments to train such a model. One of the common ways to make the model more accurate is to increase the number of parameters. The more the model grows, the better it is able to memorize new complex patterns and the better it is generalizable across different language tasks. The drawback of it is that it results in longer training and more demand on memory. Furthermore, bigger models are prone to overfitting if there is not enough diverse data.

While these LLMs are powerful, their effectiveness and efficiency are significantly influenced by the initial text processing steps, particularly tokenization. Despite progress in tokenization techniques, standard methods still have limitations with some of the NLP tasks. One of the issues is how they comprehend the context in which a word is being used, which leads to incorrectly split data.

Additionally, traditional tokenizers have trouble with out-of-vocabulary words (OOV), which is a real complication for rare or terms specific words. Also it is tricky for standard methods to deal with all the different ways words are formed in various languages.

Furthermore, we need tokenizers that can handle all the different characters (like emojis or accents in Unicode) and work across many languages without needing a separate dictionary for each one. Simultaneously, there is a immense demand for models to run faster and use less computing power and memory. Using efficient tokenization can be a solution for that.

Given these limitations and the potential for efficient tokenization to address the growing demands on computational resources, this paper explores

advanced tokenization strategies, including the integration of multiple token streams from diverse algorithms. Additionally, it investigates the combination of subword and multi-word units to enhance computational efficiency.

1.4 Work Structure

This thesis is organized into the following chapters, each building upon the last to present a comprehensive study:

Chapter 2: Related Work

This chapter takes a look at what others have already done in this area. It reviews important past studies, key ideas, and common methods used, which helps to show where this current research fits into the bigger picture.

Chapter 3: Methodology

Here, the plan for how this study was carried out is explained. This means talking about the specific ways data was gathered, worked with, and looked at. The reasons for choosing these particular methods are also laid out, showing why they were a good fit for answering the research questions.

Chapter 4: Implementation

This chapter gets into how the plan from Chapter 3 was put into code. It covers the details of the dataset used, the design of any models that were built or used, and how the different techniques were actually coded and applied.

Chapter 5: Results

This is where the findings from all the experiments are presented and carefully examined. There is also a discussion about what these results mean and why they matter.

Chapter 6: Conclusion

The final chapter wraps everything up. It gives a quick summary of the whole project, highlighting the main discoveries and what they add to the field. It also suggests ideas for what could be explored next to build on this research.

Chapter 2

Related Work

As NLP models get more complex, tokenization needs to keep up so efficiency or performance will not be lost. Simple word splitting was not enough, leading to the more advanced methods we see today.

2.1 Integration of Subword and Sultiword Units

2.1.1 From Word-Level to Subword Units

The traditional approach of word-level tokenization, which simply divides text based on spaces and punctuation (tokens like ”I”, ”am”, ”learning”, ”NLP”), presented significant challenges as NLP developed. The main drawback is the creation of extremely large vocabularies. Furthermore, there is a significant problem with out-of-vocabulary (OOV) words, which do not present in the training data causing the model not to process them effectively. It also treated related word forms, such as ’run’, ’running’, and ’ran’, as entirely distinct units, which led to inability to see the relation and semantic connection.

Subword tokenization 2.1 (BPE [2], WordPiece [3], SentencePiece [4]) was developed as the result to overcome limitations of previous methods. This technique segments words into smaller, meaningful units, also known as subwords. This methodology presents several key advantages. It effectively deals with the OOV problem by allowing models to represent unknown words as sequences of known subword units. This helps the models to processe previously unseen words.

Subword tokenization helped to find a balance: it avoids the massive vocabularies typical of word-level methods while also preventing the excessively long sequences that could be produced by character-level tokenization. By breaking down less common words into their more frequent subwords, it improves the ability to interpret words absent from the training set.

Additionally, this approach has a better view to understanding linguistic variations. Because related word forms (like ”love,” ”loving,” ”loved”) often have common subwords, the model can recognize and learn these relationships. It creates the reduction in vocabulary size, which contributes to greater computational efficiency and allows models to generalize more effectively to new data.

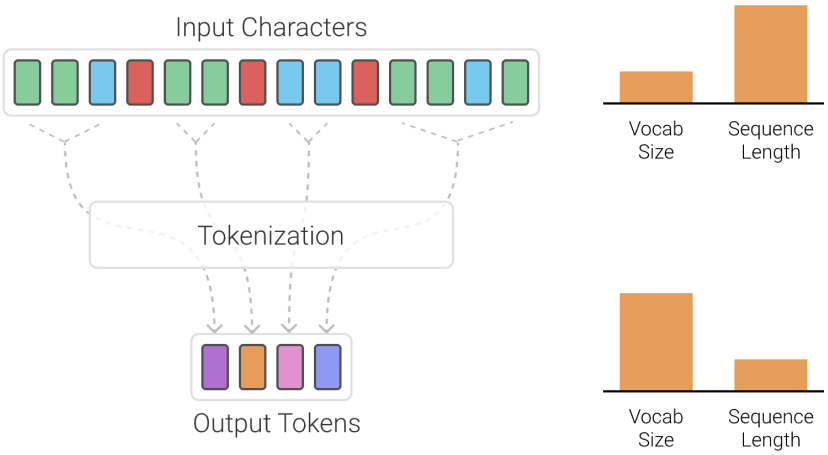


Figure 2.1: Subword tokenization

2.1.2 Character-Level Tokenization

Character-level tokenization 2.2 operates differently. It treats every single character in the text (letters, numbers, punctuation, spaces) as its own separate token.

The primary strength of this method that it is able to process any text input without 'out-of-vocabulary' issues. Because the entire set of possible characters forms the vocabulary, no character sequence is unknown. However, the primary difficulty with this character-by-character approach is that it produces much longer sequences of tokens. When compared to methods that use whole words or subwords, breaking everything down into individual characters results in considerably more tokens for the same amount of text, as even short words become multi-token sequences.

This growth directly affects the learning process. When words are divided into characters, the semantic information is lost. The model must then learn the context and meaning from these one-character sequences. Often, this requires more refined model architectures and increases the computational resources needed for the model to effectively learn the patterns.

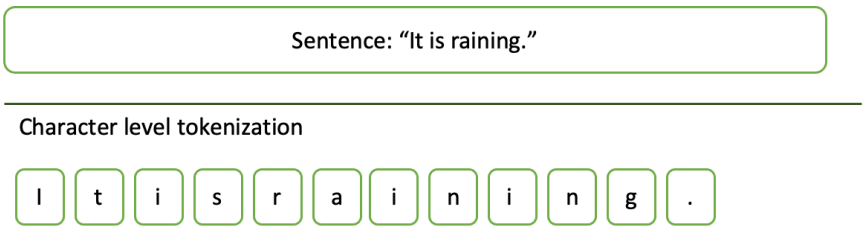


Figure 2.2: Character-Level tokenization

2.1.3 Subword and Multi-Word Tokenization

Multi-Word Tokenization (MWT) [5] 2.3 uses different approach by looking outside just a single word. It notices that same groups of words frequently appear together and function as a single concept, like 'machine learning'. Instead of splitting these, MWT treats the entire phrase as one single token.

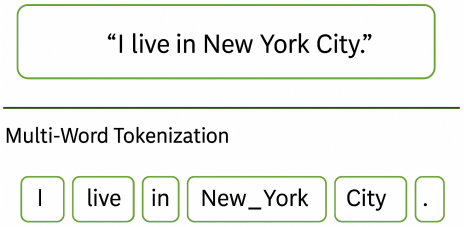


Figure 2.3: Mulit-Word tokenization

This idea has some benefits. Firstly, it leads to shorter sequences of tokens, because multiple words are condensed as one. This reducing in length of sequence can potentially decrease the computational workload. Secondly, it can improve the way meaning is represented, as it keeps words, which meaning depends on each other, together. It works by specifying which group words should be considered as one token.

While moving from word-level to subword tokenization was a major step forward—helping balance vocabulary size and the ability to handle new words—subword methods often still separate words within meaningful multi-word phrases. Multi-word tokenization specifically targets this issue, aiming to create representations that are potentially both more efficient (due to shorter sequences) and richer in meaning by preserving these common phrasal units.

2.2 Tokenization Pipeline Combining Multiple Streams of Preprocessed Text

In addition to using different tokenization methods, there is also a possibility of improvement by text preprocessing. Application of preprocessing to the dataset may make the data cleaner, without noise, and improve the performance of the model.

2.2.1 Preprocessing on Text Classification

This study [6] evaluated all combinations of several basic preprocessing methods (lowercase, stopword removal, punctuation removal, spelling correction, HTML removal, repeated character reduction) for text classification applying

three supervised ML methods: BN (Bayes Networks)[7], SMO (a variant of SVM) [8], and Random Forest (RF) [9] using a bag of word unigrams.

Their key findings were:

- stopword removal significantly improved results for three out of four benchmark datasets.
- lowercase conversion was the single most beneficial step for the dataset where stopword removal failed.
- no single preprocessing method or combination was universally optimal, the best strategy depended heavily on the dataset.

They concluded that systematically testing combinations of preprocessing steps is advisable, as some combinations almost always improved accuracy over the baseline for Bag-of-Words models.

2.2.2 Text Preprocessing in Neural Network Architectures

This work [10] specifically examined the impact of simple preprocessing (tokenizing, lowercase, lemmatizing, multi-word grouping) on neural text classifiers (CNN and CNN+LSTM) using pre-trained Word2vec [11] embeddings.

The findings contrasted somewhat with the previous study results:

- general domain datasets (news, reviews) - simple tokenization often performed as well as or better than lowercase or lemmatization.
- specialized medical domain dataset - lowercase and lemmatization provided significant improvements over the simple setting.
- lemmatization, while traditionally useful for linear models, showed limited general benefit for these neural architectures, possibly because embeddings already capture some lexical relationships.
- word embeddings trained on multi-word-grouped corpora perform well when applied to simple tokenized datasets.

2.2.3 Effective Preprocessing Algorithm for Text Classification

The main idea of the novel preprocessing strategy is that it combines stopword removal and regular filtering with tokenization and lowercase conversion, which can effectively reduce the feature dimension and improve the text feature matrix quality.

This study [12] divided the pipeline into two phases: Tokenization with Normalization and Stop Word Removal with Stemming/Lemmatization. They found tokenization and normalization to be more effective overall. They proposed a combined strategy (stopword removal and regular filtering + tokenization + lowercase conversion) that reduced feature dimensions and

improved matching performance on the 20 Newsgroups dataset compared to state-of-the-art methods. They also supported the idea that the combination of multiple preprocessing strategies can enhance the accuracy of classification.

Chapter 3

Methodology

In this chapter, the dataset, source code, and the methods descriptions will be provided.

3.1 The WikiText-2 Dataset

The WikiText-2 dataset [13] was produced from a subset of English Wikipedia articles. Using Wikipedia as a source offers the advantage of broad topic coverage and relatively clean, structured text.

Minimal pre-processing is applied to the dataset. Meaning, it retains the original capitalization and punctuation, which is crucial for evaluating models on their ability to handle these linguistic features. Extensive normalization steps, such as lowercase or punctuation removal are avoided. A fixed vocabulary is used, and out-of-vocabulary words (words not present in this vocabulary) are mapped to a special <unk> (unknown) token.

WikiText-2 serves primarily as a standard benchmark dataset for the development, evaluation, and comparison of language models. Perplexity, a measure of how well a probability model predicts a sample, is the most commonly reported metric on this dataset.

3.2 nanoGPT

All experiments were performed using Andrej Karpathy's nanoGPT code from the repository [14]. To make it suitable for the puposes of this thesis, some parts of the code were changed or added. The explicitly stated goal of nanoGPT is to be the simplest, smallest, and most easily understandable implementation for training and running inference with GPT-like models.

3.3 Preprocessing Techniques

3.3.1 Lowercase

Text data usually contains different variants of capitalization. Words may appear capitalized at the beginning of sentences, in proper nouns, for emphasis or inconsistently due to stylistic choices or errors. From a computational viewpoint, treating ”Word”, ”word” and ”WORD” as distinct entities significantly increases the vocabulary size and complexity of the data. Converting all text to lowercase normalizes the text and ensures consistency. ”Hello” and ”hello” both become ”hello”. Lowercase helps models recognize that these tokens have the same meaning by treating different cases of the same word equally. This reduction in vocabular variation improves efficiency and may also improve the accuracy of NLP.

Lowercase brings significant benefits, but it also has drawbacks. The main complication involves the potential loss of information. Capitalization can carry important information. For example, distinguish proper nouns or acronyms, and indicate attention. Converting all text to lowercase removes this information. The lowercase represents a balance between simplification and potential information loss.

3.3.2 Punctuation Removal

Punctuation marks (e.g., commas, periods, exclamation points, hyphens) are essential for human readability and grammatical structure but often introduce noise and unnecessary complexity for NLP algorithms. Most of the time, punctuation does not really change what it means in the text. Getting rid of punctuation makes the text data simpler, reduces the vocabulary size (as ”word.” and ”word” are treated the same) and helps standardise the input for later steps like tokenization or embedding generation. This cleaning step helps models focus more on the important linguistic content.

Removing punctuation makes analysis easier and ensures data uniformity, which is crucial for algorithmic performance. However, like lowercase, it can lead to information loss. Punctuation can sometimes carry semantic or stylistic meaning. For example, exclamation points can indicate strong feelings, question marks can show interrogatives, and hyphens can change the meaning of words (e.g., ”state-of-the-art”).

3.3.3 Lemmatization

There are many different words that have the same root meaning in natural language (e.g., ”run”, ”running”, ”ran”). Treating each form as a different token can increse vocabulary size and data sparsity, which might make it harder for a model to recognise the semantic connection between these related words. Lemmatization is a technique used to address this by reducing words to their base or root form, thereby normalising the vocabulary and

grouping semantically related terms. This normalization helps NLP algorithms perform better by reducing dimensionality and making it easier to identify the meanings of the core words.

The idea behind lemmatization is to find the most common form of a word, also known as the 'lemma', by looking at its morphology and context. You usually need to know the word's part of speech, because the lemma can be different based on the context (e.g., the lemma of "better" is "good" as an adjective, but "well" as an adverb).

3.3.4 Replacing Numerical Sequences with a Placeholder Token

Numerical data are common in texts, but can be challenging to work with. There are an infinite number of possible numerical values, which leads to extremely high vocabulary sizes and data sparsity if each number is treated as a unique token. A lot of specific numbers only appear rarely, which makes it hard for models to learn meaningful representations. Additionally, in a lot of NLP tasks, the exact number is not as important as the fact that a number was used in a specific situation. So, replacing the numbers with a placeholder token, like <|num|>, fixes these problems by reducing the number of words needed, dealing with sparsity, and letting the model spot patterns about the presence and position of numbers, rather than their specific values. This means that systems tend to generalise more robustly when they encounter new numeric expressions in practice, since they already understand the contextual role of a number's presence without having seen that exact figure during training.

But of course, this means losing exact numerical detail, so it is important to choose the right technique for each task. A single <|num|> token is great when it is just necessary to match a number, but if the number's importance to the model is key, it will require something more specialised.

3.3.5 Utilizing Special Markers

This preprocessing technique involves adding predefined, distinct tokens to the input text that are not part of the original content. In this work, special markers like <|exclamation|>, <|question|> and <|end|> will be used.

Special Tokens are unique strings that are added to the model's vocabulary. They are inserted at specific positions within the input sequence – commonly at the beginning, end, or to separate different segments of text.

Markers like <|staret|> and <|end|>(or <|eos|>) explicitly define the beginning and end of a sequence. This helps the model understand where the relevant content starts and stops, which is crucial for processing variable-length inputs or in sequence-to-sequence tasks. Markers like <|question|> can give the model specific metadata about the nature or type of the input text. By making the input structure more explicit, markers can help the model learn patterns more effectively. In addition, these markers can help

the model focus on the relevant parts of the input or understand its overall structure.

3.4 Tokenization

3.4.1 The Synergy of Subword and Multi-Word Tokenization

Combining subword and multi-word tokenization is a method of connecting the unique advantages of both techniques, which represent the text in more effective way. The fundamental idea is to use multi-word tokenization to better idetify the relationship between common expressions. Concurrently, subword tokenization manages less frequent terms by breaking them into smaller, known parts. In further chapters, we experiment with the integration of subword and multi-word units and analyze their impact on computational efficiency.

3.4.2 Training Tokenizer Tailored for Specific Dataset

For this work, the BPE subword tokenization algorithm will be used considering that the baseline model for this work was trained using pretrained GPT-2 BPE tokenizer. This will make it suitable to compare these two approach.

Training the Tokenizer on the Original Dataset

Initially, a BPE tokenizer will be created by training it on the untouched training set of the WikiText-2 dataset. The whole point of this process is to train the tokenizer to understand how the language is used in the source material. This tokeniser should perform well when used with a language model that has also been trained or fine-tuned on the original WikiText-2 data. The idea is that the tokenizer will have created subword units and a vocabulary that are perfectly in sync with the text features that the language model has come across.

Training the Tokenizer on Preprocessed Versions of the Dataset

In the next step, experiments will be performed to see what happens when the tokenizer is trained on versions of the WikiText-2 dataset that have been altered using the preprocessing techniques mentioned previously. This will help to understand if changing the data before the tokenizer training is good or bad for how well it can prepare text for the language model.

Chapter 4

Implementation

4.1 Model Sizes

4.1.1 Key Architectural Dimensions for Scaling

For this study, a set of four Transformer-based language models were utilized, systematically scaled in size to investigate the impact of model capacity on performance. These models are referred to by their approximate number of trainable parameters: 1M, 10M, 30M, and 50M. The scaling of these models was achieved by concurrently increasing three fundamental architectural dimensions of the Transformer framework: the embedding dimension, the number of attention heads in the multi-head attention mechanism and the number of layers (Transformer blocks).

The architecture of each Transformer model was defined by the following key dimensions:

- **Embedding Dimension (D):** This parameter specifies the size of the dense vector representations used for each input token. A larger embedding dimension allows the model to encode richer and more nuanced information about the tokens.

- **Number of Attention Heads:** Within the multi-head self-attention mechanism, the model's ability to focus on different parts of the input sequence is distributed across several "attention heads". Each head operates in a subspace of the embedding dimension. Increasing the number of heads allows the model to jointly attend to information from different representational subspaces at different positions, potentially capturing a wider range of syntactic and semantic relationships.

- **Number of Layers(L):** This refers to the depth of the model, determined by the number of Transformer blocks stacked sequentially. Each layer adds further non-linear transformations, enabling deeper models to learn more complex patterns and higher-level abstractions from the input data.

4.1.2 Model Configurations

The specific configurations for each of the models investigated are detailed below, illustrating progression in size and complexity:

- **1M Parameter Model:**
  - Embedding Dimension: 24
  - Number of Attention Heads: 1
  - Number of Layers: 1
- **10M Parameter Model:**
  - Embedding Dimension: 192
  - Number of Attention Heads: 4
  - Number of Layers: 4
- **30M Parameter Model:**
  - Embedding Dimension: 384
  - Number of Attention Heads: 6
  - Number of Layers: 6
- **50M Parameter Model:**
  - Embedding Dimension: 512
  - Number of Attention Heads: 8
  - Number of Layers: 8
- **123M Pretrained Parameter Model:**
  - Embedding Dimension: 768
  - Number of Attention Heads: 12
  - Number of Layers: 12

4.1.3 Other Parameters of the Models

Table 4.1: Model Training Parameters

Parameter	Smaller Model (from Scratch)	Pretrained Model (GPT-2)
Batch size	32	4
Block size	256	1024
Dropout	0.2	0.0
Learning rate	1e-3	6e-4

- Batch Size: This parameter specifies the quantity of training examples (text sequences) the model analyzes before its internal parameters (weights) are adjusted.

  Smaller Model: When a model is learning everything new, a batch size of 32 makes a good balance. It gets enough examples in each iteration to make sensible adjustments, but not so many because there are still memory limitations.

  Pretrained Model: For a big, already smart model like GPT-2, a smaller batch size of 4 was used. This is mainly because these large models take a lot of memory. Smaller batches allow to fine-tune weights on typical hardware, making frequent, small changes to the existing knowledge.

- Block Size (Context Window): This sets the limit on how many words or word pieces (tokens) the model can look at in one iteration as a single piece of text.

  Smaller Model: A window of 256 tokens lets new model see enough text to learn how nearby words relate to each other. For many types of text, especially with shorter sentences or paragraphs, this is plenty to get started without being too computationally heavy.

  Pretrained Model: Models like GPT-2 were originally trained to look at much longer stretches of text (1024 tokens is common). When we fine-tune them, keeping this larger window allows the model to use its built-in ability to understand connections across longer passages.

- Dropout: A regularization technique to prevent overfitting by randomly ignoring some neurons during training.

  Smaller Model: When a model is learning from scratch, it can easily just memorize the answers. Dropout encourages it to build a more flexible understanding that works on new, unseen text.

  Pretrained Model: The pretrained model does not use dropout. These models have already learned a lot from data that they are less likely to overfit during a relatively short fine-tuning phase. In fact, dropout could sometimes get in the way of adapting their already strong knowledge.

- Learning Rate: The learning rate is like the size of the steps the model takes as it tries to learn new information.

  Smaller Model: This is a fairly standard, moderately sized step. It is big enough for the model to make meaningful progress as it learns everything from the beginning.

  Pretrained Model: The model's existing knowledge is valuable, so it is necessary to make small, careful adjustments to tune it for the new task, rather than making considerable changes that might ruin what is already known.

4.2 Preprocessing Techniques Implementation

4.2.1 Lowercase

The lowercase preprocessing step aims to standardize the text by converting all alphabetic characters to their lowercase equivalents. The implementation directly utilizes Python's built-in string method lower(), which efficiently processes the input text string and returns a new string with all characters converted to lowercase.

```
def lowercase_text(text):
    return text.lower()
```

4.2.2 Punctuation Removal

Punctuation removal is implemented to eliminate punctuation marks from the text, which can reduce noise and simplify the input for the language model. This is achieved using Python's string manipulation capabilities. Specifically, the translate() method is applied to the input string in conjunction with a translation table created by str.maketrans("", "", string.punctuation). The string.punctuation constant provides a collection of standard punctuation characters, and str.maketrans generates a table that maps each of these characters to None, deleting them from the string.

```
def remove_punctuation(text):
    return text.translate(str.maketrans("", "",
            string.punctuation))
```

4.2.3 Lemmatization

Lemmatization is used to reduce words to their base or dictionary form. The implementation leverages the spaCy library. An input text string is first processed by a pre-loaded spaCy language model, which performs linguistic analysis. The lemma_ attribute of each token in the processed document is then extracted, and these lemmas are subsequently joined back together with spaces to form the lemmatized text.

```
import spacy
lemmatizer = spacy.load('en_core_web_sm')

def lemmatize_text(text):
    doc = lemmatizer(text)
    lemmatized_tokens = [token.lemma_ for token in doc]
    lemmatized_text = ' '.join(lemmatized_tokens)
    return lemmatized_text
```

4.2.4 Insertion of Special Markers

The insertion of special markers is a technique used to explicitly annotate sentences with specific characteristics or to delimit them. In this implementation, the NLTK library's sent_tokenize() function first segments the input text into individual sentences. These sentences are then processed: if a sentence strip ends with a question mark (?), it is prepended with the marker <|question|> and appended with <|end|>. Similarly, if it ends with an exclamation mark (!), it is prepended with <|exclamation|> and appended with <|end|>. Sentences not matching these conditions are included without these specific type markers, and all processed sentences are then re-joined with spaces.

```
from nltk.tokenize import sent_tokenize

def insert_special_markers(text):
    sentences = sent_tokenize(text)
    processed_sentences = [
        f"<|question|> {sentence} <|end|>"
        if sentence.strip().endswith("?") else
        f"<|exclamation|> {sentence} <|end|>"
        if sentence.strip().endswith("!") else
        sentence
        for sentence in sentences
    ]
    return " ".join(processed_sentences)
```

4.2.5 Replacing Numbers

Replacing numbers involves substituting all occurrences of numerical digits within the text with a singular, special placeholder token. The implementation utilizes Python's built-in regular expression module. The function re.sub(r'+', '<|num|>', text) is employed, where the pattern r'+' matches any sequence of one or more digits, replacing each match with the string literal <|num|>.

```
import re

def replace_numbers(text):
    return re.sub(r'\d+', '<|num|>', text)
```

4.3 Dual-Stream Architecture Model

The architecture developed for this work is a custom dual-stream model. This architecture is based on Andrej Karpathy's nanoGPT implementation. The main idea behind GPTDualStream is that it can process two different but related input token sequences at the same time by using parallel Transformer

streams. This means that the model can learn and integrate information from both inputs: the original input and preprocessed one.

4.3.1 Architectural Overview of GPTDualStream

The GPTDualStream model is designed to accept a pair of input sequences: an original sequence (denoted as ”orig” in the implementation) and an other sequence (denoted as ”other”), which represents a modified version.

4.3.2 Dual Input Embedding Layers

The model features two separate token embedding layers to create initial vector representations for the two input sequences. The original tokens are processed by standard token embedding layer (self.transformer.wte) inherited from the base GPT class [14]. A new token embedding layer (self.wte2) is introduced specifically for the other tokens. Both self.transformer.wte and self.wte2 are configured with the same vocabulary size (config.vocab_size) and embedding dimension (config.n_embd), ensuring that tokens from both streams are mapped into the same dimensional space.

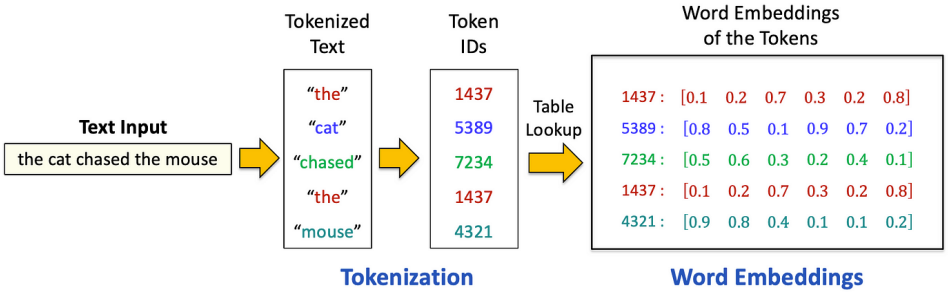


Figure 4.1: Embeddings

4.3.3 Shared Positional Embeddings

A single positional embedding layer (self.transformer.wpe), also inherited from the base GPT class[14], is utilized for both input streams. The same set of positional encodings is added to the token embeddings of both the original sequence (to form x1) and the other sequence (to form x2) before they enter their respective Transformer streams. This shared positional information implies an alignment between the two input sequences. Dropout is applied after adding positional embeddings.

4.3.4 Parallel Transformer Processing Streams

The core of the GPTDualStream architecture consists of two independent stacks of Transformer blocks:

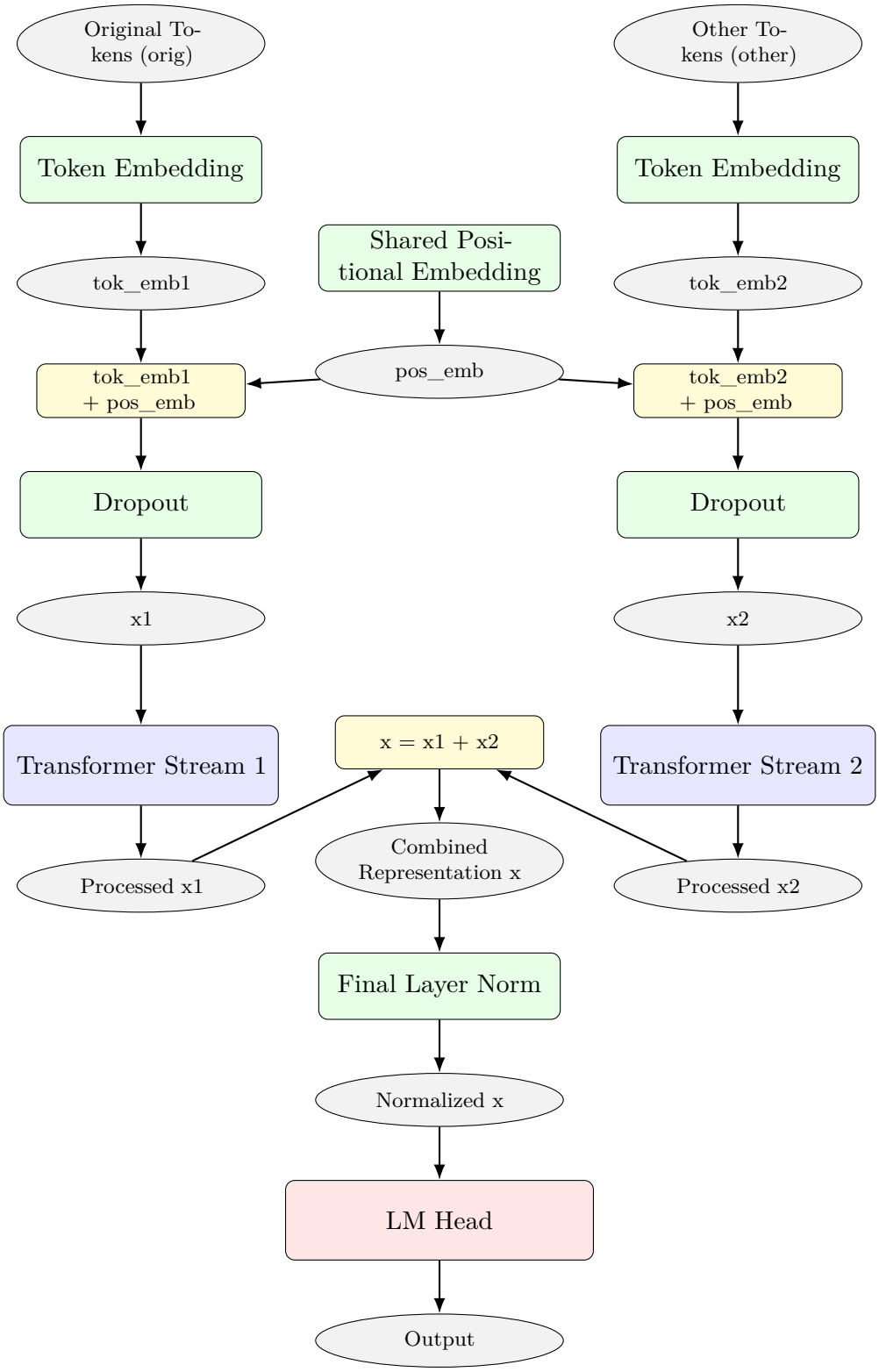


Figure 4.2: Graph of the GPTDualStream architecture.

```
self.transformer_stream1 = nn.ModuleList(
[Block(config) for _ in range(config.n_layer)])
self.transformer_stream2 = nn.ModuleList(
[Block(config) for _ in range(config.n_layer)])
```

Each stream is a ModuleList containing ”number of layers” instances of the Block class (which implements the standard Transformer block with multi-head self-attention and a feed-forward neural network, as defined in nanoGPT). The input x1 (derived from original tokens) is processed exclusively by transformer_stream1. The input x2 (derived from other tokens) is processed exclusively by transformer_stream2. These two streams operate in parallel, meaning that the computations within transformer_stream1 do not directly influence transformer_stream2 at this stage, and their respective Block instances have independent sets of weights.

4.3.5 Stream Merging Mechanism

After the x1 and x2 sequences have been processed through all layers of their respective Transformer streams, their resulting hidden state representations are combined. The merging operation is performed via element-wise addition of the two output tensors: $x = x1 + x2$. This summation creates one combined representation x, that includes contextual information learned separately from the original and other input sequences.

4.3.6 Final Output Projection

The merged representation x is put through a final layer normalization step using self.transformer.ln_f (final layer normalization from the base nanoGPT). Subsequently, the normalized, merged representation is passed to the language modeling head, self.lm_head (also likely from the base GPT class), which projects it into a distribution of logits over the vocabulary. These logits can then be used to calculate a loss function (e.g., cross-entropy) if target tokens are provided, or to generate new tokens during inference.

4.3.7 Relationship to nanoGPT

The GPTDualStream architecture utilizes several components from Karpathy's nanoGPT. Specifically, it inherits or reuses the Block structure (comprising multi-head self-attention and feed-forward layers), the token embedding (wte) and positional embedding (wpe) mechanisms, dropout (self.transformer.drop), the final layer normalization (ln_f), and the language modeling head (lm_head).

The main architectural extension is the implementation of the second token embedding layer (wte2) and the parallel transformer_stream1 and transformer_stream2, along with the strategy for merging their outputs. This extension allows specialized processing of dual inputs while keeping the efficiency of the nanoGPT.

4.4 Create Multi-Word Tokenizer

The Fast Vocabulary Transfer (FVT) and Multi-Word Tokenizer (MWT) are used in the creation of Multi-Word Tokenizer. These modules are part of the open-source repository fast-vocabulary-transfer [15], which provides implementations of techniques presented in the EMNLP 2022 [16] and 2023 [17] papers on vocabulary adaptation and sequence compression.

FVT is designed to adapt a pretrained tokenizer's vocabulary to better suit a specific in-domain dataset. This process involves adjusting the tokenizer's merge rules to align with the statistical properties of the target corpus, resulting in improved tokenization efficiency and model performance on domain-specific tasks.

The implementation begins by loading the raw WikiText-2 dataset using HuggingFace's [18] load_dataset, splitting it into "train", "validation", and "test" subsets. All text lines from each split are concatenated into a single Python list called in_domain_data. This combined list serves as the in-domain data for adapting the base GPT-2 vocabulary. The train_tokenizer method from FastVocabularyTransfer class takes in-domain data, the pretrained tokenizer and the desired vocabulary size (e.g., 50,257 tokens) to adjusts the tokenizer's vocabulary to better represent the in-domain data.

MWT extends the adapted tokenizer by combining frequent multi-word expressions into single tokens, improving the model's ability to capture common phrases.

Instance of MultiWordTokenizer is created by taking the in-domain tokenizer obtained from the FVT process. Multi-Word tokenizer is trained by learn_ngrams method choosing the range of n-gram lengths (e.g., 2 to 4) and the number of top frequent n-grams to include (e.g., 5,000). After that the top-K n-grams are added to the vocabulary of the tokenizer. The resulting vocab_size is 55257.

By following these steps, the tokenizer becomes more adjusted to the specific linguistic patterns of the target dataset, potentially improving the performance and efficiency of the model.

4.5 Train BPE tokenizer on WikiText-2

A custom byte-level BPE tokenizer was developed from scratch, specifically for the WikiText-2 dataset. The objective was to achieve a vocabulary size comparable to that of GPT-2, approximately 50,257 subword units.

The initial step involved loading the raw WikiText-2 data. The BPE method was selected for its efficacy in identifying frequent character combinations and combining them into single tokens. The tokenizer was configured to label any unrecognized sequences with the <unk> placeholder.

This approach allows the tokenizer to process text directly at the byte level. The tokenizer was also equipped with a corresponding decoder to ensure the original byte offsets could be recovered, so it would be able to

reconstruct of the original text. The BPE trainer function was to iteratively identify common subword pairs within the training text and merge them into new, distinct tokens until the target vocabulary size of 50,257 was attained. The threshold for including subwords was set at a minimum of two, which effectively filtered out units that happened very rarely. Furthermore, five special tokens were reserved: <s> (start of sequence), <pad> (for padding), </s> (end of sequence), <unk> (unknown).

To ensure compatibility and ease of use with standard NLP frameworks this model was encapsulated within a PreTrainedTokenizerFast object. This involved defining its encoding (text to numerical IDs) and decoding (numerical IDs back to text) procedures.

4.6 Train BPE tokenizer on All Preprocessed Versions of WikiText-2

A key aspect of this process is the extensive preparation of the training data, which involves applying a variety of text preprocessing techniques to the WikiText-2 dataset. This approach aims to create a rich and diverse corpus for the tokenizer to learn from.

The script begins by defining several preprocessing functions, the same that were descrbed prevously in ”Preprocessing Techniques Implementation” section.

The core of the data preparation strategy involves generating numerous variations of the original WikiText-2 text. The script iterates through each data split (’train’, ’validation’, ’test’). For every example within these splits, it systematically applies all possible combinations of the five preprocessing functions defined above. This is achieved by generating every permutation of enabling or disabling each of the preprocessing flags (lower, punct, lemma, markers, nums). For each combination, the original text is transformed by applying the selected preprocessing steps sequentially. If no flags are enabled, the original text is used. All these processed text variants, from all splits and all combinations, are then collected into a single comprehensive list. This list forms the final, expanded training corpus fed to the BPE tokenizer.

Following this data preparation, the script proceeds to train the BPE tokenizer itself. The methodology is similar to the one described in the previous section ”Train BPE tokenizer on WikiText-2”. The key difference is that the tokenizer is trained on the aggregated list of processed texts instead of original dataset. Also, the BpeTrainer is configured with a target vocabulary size of 50,261. This is due to the original vocab_size being 50,257 plus four additional custom markers used in preprocessing: <|question|>, <|exclamation|>, <|end|>, and <|num|>. .

Chapter 5

Results

5.1 Analysis of Model Perplexity Results

This chapter analyzes the perplexity scores of language models with different sizes (1M, 10M, 30M and 50M parameters) when exposed to different text preprocessing techniques. Perplexity is a measure of how well a probability model predicts a sample. A lower perplexity score usually means a better model performance.

Table 5.1: Perplexity on the test set under various preprocessing techniques. **DS** - where DualStream architecture was used.

Model Size	Original	Lowercase **DS**	No Punctuation **DS**	Numbers Token **DS**	Special Markers **DS**	Lemmatize **DS**
1M	422.048	422.639	445.010	435.050	436.927	418.971
10M	192.634	254.849	269.646	268.632	258.053	260.947
30M	180.264	193.976	213.789	196.928	211.785	190.275
50M	180.461	200.065	178.149	198.991	184.700	191.880

5.1.1 Impact of Model Size

Generally, as the model size increases, perplexity tends to decrease. For example, with the ”Original” data, perplexity drops from 422.048 for the 1M model to 192.634 for the 10M model, and further to 180.264 for the 30M model. The 50M model shows a slight increase to 180.461 for the original data, which might indicate that for this specific dataset and preprocessing, the model might be approaching its optimal capacity or experiencing slight overfitting, though this is less common for perplexity to increase with size alone. More consistently, for most preprocessing techniques, increasing model size from 1M to 50M leads to a reduction in perplexity.

5.1.2 Impact of Preprocessing Techniques

Original: This serves as the baseline.

Lowercase: Converting text to lowercase generally increases perplexity compared to the original, especially for smaller models (e.g., 1M: 422.048 vs 422.639; 10M: 192.634 vs 254.849). This suggests that case information is valuable for the model.

Punctuation: Removing punctuation significantly increases perplexity for the 1M, 10M and 30M models (e.g., 10M: 192.634 vs 269.646). However, for the 50M model, removing punctuation results in the lowest perplexity (178.149) among all variations for that size, suggesting that with a sufficiently large model, the absence of punctuation might simplify the language modeling task or that the model learns to handle it better.

Numbers: Removing numbers generally increases perplexity compared to the original, though the effect is less noticeable than removing punctuation for smaller models.

Markers: Removing markers also tends to increase perplexity compared to the original.

Lemmatization: This technique shows mixed results. For the 1M model, it slightly improves perplexity (422.048 vs 418.971). For the 10M and 30M models, it leads to higher perplexity than the original, but for the 50M model, it is slightly higher than original but better than some other preprocessing steps like lowercase.

5.2 Exploring Combinations of Preprocessing Techniques

The initial experiments showed that the 1M parameter models did not lead to any noticeable enhancements, so further analysis will include only 10M, 30M and 50M.

The following combinations of preprocessing techniques will be evaluated:

- **Lowercase + Punctuation Removal:** Combining these two aims to create a cleaner, more uniform text. This could be particularly beneficial if the model is sensitive to case variations and punctuation characters. The expectation is that their combined effect will simplify the input language more effectively than either technique alone, potentially leading to better model learning.

- **Punctuation Removal + Lemmatization:** Combining these two is a strong approach to text normalization. By removing punctuation and then reducing words to their lemma, the text becomes standardized. This could help the model focus on core semantic meanings and improve performance by reducing input variability.

- **Punctuation Removal + Marker Replacement:** Combining the most effective individual techniques at 50M aims to further enhance the input. Removing punctuation after or before replacing markers ensures that the context around these markers is also clean, potentially allowing

the model to better understand the role of these markers within the sentence.

- **Lowercase + Punctuation Removal + Lemmatization:** Applying these three steps together aims to achieve a very simplified text and vocabulary reduction. The assumption is that this extensive preprocessing will minimize input noise and variability, allowing the model to learn more generalizable representations.
- **All together:** This complete cleaning and standardisation will create a highly concentrated dataset where the model can focus on key linguistic patterns.

Table 5.2: Perplexity of preprocessing combinations on the test set. **DS** - where DualStream architecture was used.

Model Size	Original	Lower + Punct **DS**	Punct + Lemma **DS**	Punct + Markers **DS**	Lower + Punct + Lemma **DS**	All **DS**
10M	192.634	254.373	260.421	261.727	243.739	237.756
30M	180.264	188.293	189.724	210.811	188.653	188.801
50M	180.461	187.306	184.470	196.124	182.467	189.576

5.2.1 Overall Trend Across Model Sizes

Consistent with Table 5.1, increasing model size generally leads to lower perplexity across all combinations of preprocessing techniques. For instance, for the "All" combination, perplexity drops from 237.756 for 10M to 188.801 for 30M, and then slightly increases to 189.576 for 50M.

5.2.2 Comparison with Individual Techniques

Original: Remains the baseline for comparison.

Lowercase + Punctuation: This combination results in higher perplexity than the "Original" for all tested model sizes (e.g., 10M: 192.634 vs 254.373).

Punctuation + Lemmatization: This combination also generally leads to higher perplexity than the "Original".

Punctuation + Markers: This combination shows a significant increase in perplexity compared to the "Original", especially for the 10M and 30M models.

Lowercase + Punctuation + Lemmatization: This combination results in perplexity values higher than the "Original".

All: Applying all preprocessing steps generally leads to higher perplexity than using the original data, except for the 50M model where c (punctuation removal alone) performed best. For the 10M model, 'All' (237.756) is better than many individual or simpler combinations like "Lower + Punct" (254.373) or "Punct + Lemma" (260.421), but still worse than "Original".

5.3 Multi-Word Tokenizer

This table assesses the impact of using a Multi-Word Tokenizer (MWT) compared to the original tokenization, across different model sizes.

Table 5.3: Perplexity of MultiWord Tokenizer on the test set. **DS** - where DualStream architecture was used.

Model Size	Original	Original + MWT **DS**	MWT
1M	422.048	480.188	925.794
10M	192.634	292.744	340.966
30M	180.264	240.578	366.556
50M	180.461	225.094	334.353

5.3.1 Effect of Model Size

Similar to previous tables, increasing model size reduces perplexity for all tokenization strategies. For the original tokenizer, perplexity decreases from 422.048 (1M) to 180.461 (50M). The same trend is observed for 'Original+MWT' and 'MWT'.

5.3.2 Effect of Multi-Word Tokenizer

Original: Standard tokenizer performance.

Original + MWT: This refers to using the original WikiText-2 data tokenized with pretrined gpt2 tokenizer as input to one stream of the DualStream model and same dataset tokenized using Multi-Word Tokenizer trained on WikiText-2 dataset as the second stream. This approach consistently results in higher perplexity compared to the original tokenizer (a) across all model sizes (e.g., 10M: 192.634 vs 292.7438).

MWT: This refers to dataset being tokenized using Multi-Word Tokenizer trained on WikiText-2 dataset as the second stream. This approach yields the highest perplexity values among the three options across all model sizes (e.g., 10M: 192.634 vs 340.9660).

The results suggest that for this specific task and dataset, the Multi-Word Tokenizer, as implemented in this work's code, performs worse than the original tokenizer. This could happen due to various reasons, such as the MWT not aligning well with the language patterns in the data or creating a vocabulary that is harder for the model to learn.

5.4 Fine-tuning Pretrained GPT-2 Model

Earlier tables 5.1 5.2 5.3 with models trained from scratch (1M to 50M parameters) generally showed that larger models tend to have lower perplexity.

Testing a 123M parameter model is a natural progression to observe if this trend continues and to what extent.

Pretrained models like GPT-2 have learned great amounts of linguistic knowledge from large, diverse datasets. The hypothesis is that fine-tuning such a model, even a relatively small one like a 123M variant, should significantly outperform models trained from scratch on a smaller dataset like WikiText-2. This is a common and effective strategy in NLP.

This pretrained model has a standard configuration (n_layer=12, n_head=12, n_embd=768) 4.1.2, often referred to as GPT-2 "small". It represents a significant step up from the 50M models in previous tables, allowing an investigation into the benefits of both increased size and pretraining.

This section focuses on understanding how a 123M parameter pretrained GPT-2 model responds to different preprocessing techniques during fine-tuning 5.4 and the impact of using a custom-trained BPE tokenizer 5.5.

This table shows the perplexity of a 123M parameter pretrained GPT-2 model fine-tuned on data with different preprocessing techniques. The perplexity values here are significantly lower than in the previous tables, which is expected due to the use of a much larger pretrained model.

5.4.1 Models Fine-tuned on WikiText-2 Tokenized by Pretrained GPT-2 Tokenizer

Table 5.4: Perplexity of pretrained GPT-2 models on the test set: Original was finetuned on plain WikiText-2, all others are combinations of preprocessing techniques. **DS** - where DualStream architecture was used.

Model Size	Original	Punct **DS**	Punct + Lemma **DS**	Lower + Punct + Lemma **DS**	All **DS**
123M	21.075	25.123	22.956	22.964	23.167

Key results and insights:

- "Original" (21.075) Performs Best: Fine-tuning the 123M pretrained GPT-2 on the plain, unprocessed WikiText-2 yields the lowest perplexity. This is a very significant result. It suggests that the pretrained model is capable of handling the raw linguistic features present in the original data (case, punctuation, full word forms) and, in fact, benefits from them.

- Preprocessing Hurts Performance: Every preprocessing variation applied before fine-tuning resulted in higher perplexity compared to using the original data.

- Removing Punctuation (25.123) caused the largest drop in performance.

- Combining Punctuation removal with Lemmatization (22.956) was slightly better than just removing punctuation but still worse than ”Original”.
- Adding Lowercase (”Lower + Punct + Lemma”: 22.964) offered no improvement over ”Punct + Lemma”.
- Applying ”All” changes (23.167) also resulted in poor performance.

5.4.2 Explaining Poor Performance

Pretrained models like GPT-2 learn intricate patterns from massive text corpora, including the significance of capitalization, the role of punctuation in structuring sentences and conveying meaning, and the subtle differences between word forms. Removing these features is essentially discarding information that the model has learned to use. The model, when fine-tuned, tries to adapt to this ”simplified” input but performs less optimally because the input is less rich.

Mismatch with Pretraining: The model was pretrained on text that included all these linguistic features. While fine-tuning adapts it, forcing it to work with data that drastically differs from its pretraining diet can be suboptimal.

Comparing to earlier table 5.1, for the 50M model trained from scratch, removing punctuation surprisingly yielded a slightly better perplexity (178.149) than the original (180.461). This suggests that for a model learning from scratch on a limited dataset, reducing the complexity of the input might sometimes help. However, for the 123M pretrained model, which already has a strong linguistic foundation and higher capacity, this simplification becomes a obstacle. The pretrained model is ”smart enough” to use the extra information, and removing it is detrimental.

5.4.3 Models Fine-tuned on WikiText-2 Tokenized by Custom BPE Tokenizer

Table 5.5: Perplexity of pretrained models using custom trained BPE tokenizer (compared to pretrained GPT-2) on the test set: BPE - tokenizer trained on WikiText-2 original dataset, BPE-all - trained on all combinations of preprocessed dataset WikiText-2. : **DS** - where DualStream architecture was used.

Model Size	Original using GPT-2	BPE without preproc	BPE-all without preproc	BPE-all Punct+Lemma **DS**
123M	21.075	15.601	25.892	23.467

Key results and insights:

- ”Original using GPT-2” (21.075): This is the baseline from previous table 5.4 – the pretrained GPT-2 model fine-tuned on original WikiText-2 using its own standard GPT-2 tokenizer.

- ”BPE without preproc” (15.601) Performs Best by a Large Margin: This is a key finding. BPE tokenizer was custom-trained only on the original, unprocessed WikiText-2 training data. A 123M parameter model (same GPT-2 architecture but initialized with this new tokenizer’s vocabulary and then fine-tuned on the original WikiText-2) achieved a perplexity of 15.601. This is a substantial improvement over using the generic GPT-2 tokenizer (21.075).

  This strongly suggests that a tokenizer tailored specifically to the target dataset’s vocabulary and statistical properties can be significantly more effective than a generic pretrained tokenizer, even if the model architecture is the same (fine-tuned after re-initializing the embedding layer for the new tokenizer). The custom tokenizer creates subword units that are more optimal for representing WikiText-2, leading to more efficient learning and representation.

- ”BPE-all without preproc” (25.892) Performs Poorly: In this case, the custom BPE tokenizer was trained on a corpus containing all combinations of preprocessed WikiText-2 data. When this tokenizer was used with a model fine-tuned on the original, unprocessed WikiText-2, the perplexity was very high (25.892), even worse than using the standard GPT-2 tokenizer.

  Training a tokenizer on a ”cluttered” or overly diverse corpus (containing many different preprocessing versions) can make it suboptimal for any single, clean version of the data. The tokenizer might learn subword units that are settles across different preprocessing styles, but not ideal for the original data. It creates a vocabulary that is not well-aligned with the actual fine-tuning data.

- ”BPE-all Punct+Lemma” (23.467): The same ”BPE-all” tokenizer (trained on all preprocessing versions) was used, but the model was fine-tuned on WikiText-2 data that had punctuation removed and was lemmatized. The perplexity (23.467) is better than ”BPE-all without preproc” (25.892) but still much worse than ”BPE without preproc” (15.601) and also worse than ”Original using gpt2” (21.075).

  There is a slight improvement when the fine-tuning data (Punct+Lemma) somewhat aligns with some of the data the ”BPE-all” tokenizer saw during its training. However, the tokenizer is still not optimal because it was also trained on many other versions of the data. This again highlights that the tokenizer should ideally be trained on data that closely matches the data the model will be trained/fine-tuned on.

5.4.4 Explaining Good Performance

Tokenizer-Data Alignment: The ”BPE without preproc” (15.601) wins because the tokenizer is perfectly tailored to the characteristics of the original WikiText-2 data, and the model is then fine-tuned on this same data. This harmonious collaboration is a highly effective tool.

A custom BPE tokenizer trained on the original dataset likely generates a more efficient and representative set of subword tokens for that specific dataset compared to the generic GPT-2 tokenizer (which was trained on a much broader, different corpus) or the ”BPE-all” tokenizer (which was trained on a noisy, mixed corpus). Better subword units can lead to shorter sequence lengths on average for common phrases and better handling of domain-specific terms.

MWT 5.3 performed poorly compared to the baseline tokenizer for models trained from scratch. 5.5 shows that custom tokenization itself is not a guaranteed win. The quality of the custom tokenizer and the data it is trained on are paramount. The ”BPE without preproc” is a good custom tokenizer for this task, whereas the MWT 5.3 and the ”BPE-all” tokenizer 5.5 were not well-suited.

5.5 Training Time

This section provides an analysis of the training times recorded for various model configurations, as presented in table 5.6. The data reveals clear trends regarding the impact of model size and the introduction of the DualStream (DS) architecture on the computational resources required for training.

Table 5.6: Training Times for Different Language Models: **DS** - where DualStream architecture was used

Model Configuration	Training Time
1M	18m
1M **DS**	18m
10M	27m
10M **DS**	31m
30M	46m
30M **DS**	55m
50M	1h4m
50M **DS**	1h24m
Pretrained 123M	1h3m
Pretrained 123M **DS**	1h40m

5.5.1 Impact of Model Size on Training Time

A consistent observation across both standard and DualStream models is that training time generally increases with the number of model parameters. This increase is expected, as models with more parameters involve a greater number of computations (matrix multiplications, non-linear activations, etc.) for each forward and backward pass during training. Consequently, processing the same amount of data takes longer.

5.5.2 Impact of the DualStream Architecture

The introduction of the DualStream architecture consistently led to longer training times compared to standard models of equivalent parameter size, with one notable exception at the 1M scale.

The increased training time for DualStream models is inherent to their design. The DS architecture processes two input sequences through two separate parallel Transformer streams before merging their representations. This effectively nearly doubles the number of Transformer block computations for the stream-specific components, adds an embedding layer for the second stream, and includes the merging operation. As model size increases, this additional computational load becomes more pronounced, leading to a larger percentage increase in training time.

Chapter 6

Conclusion

This thesis set out to explore how a language model's size, different ways of preparing text (preprocessing), and different methods for breaking text into pieces (tokenization) affect its performance. The main focus was on how well these models could predict text from the WikiText-2 dataset, measured by a score called perplexity. The study looked at models built from scratch with different numbers of settings (parameters) and also at a larger, pre-trained GPT-2 model. This gave a broad view of how these different elements work together.

6.0.1 Summary

For models that learned everything from scratch, making them bigger (from 1 million to 30 million parameters) generally made them better at predicting text (lower perplexity) when they were given the original, untouched data. Interestingly, the model with 50 million parameters did not do quite as well on the original data. This might mean that for this particular dataset and model design, just making the model bigger without pre-training eventually stops helping as much.

Most of the time, when common text cleaning steps—like making everything lowercase, removing punctuation, changing numbers to a special token, adding special markers, or simplifying words to their basic forms were applied, the models actually performed worse than when they used the original text. This was especially true for the smaller models. It seems these cleaning steps often throw away useful bits of language information. There was one standout case: the 50 million parameter model did best when only punctuation was removed. This hints that bigger models might sometimes find certain simplifications helpful if they can handle the changed input well. Combining several cleaning steps usually did not lead to better results and often made things worse.

The custom-built Multi-Word Tokenizer did not perform as well as the standard tokenizer for any of the models trained from scratch. This could mean that the way this MWT was built or trained in this study was not the best fit for the WikiText-2 data or the task of predicting text.

When a pre-trained GPT-2 model (with 123 million parameters) was fine-tuned, it got much better perplexity scores than any of the models trained

from scratch. This really shows how much of a head start these models get from their initial large-scale training. Importantly, when this pre-trained model was fine-tuned, every single one of the text cleaning methods made it perform worse than when it was fine-tuned on the original WikiText-2 data. This strongly suggests that advanced pre-trained models are good at using all the subtle details in raw text, and heavy cleaning can actually hurt by removing useful information.

A really interesting result came from using custom BPE tokenizers. A BPE tokenizer that was trained only on the original, untouched WikiText-2 training data made a big difference: when used to fine-tune the 123 million parameter pre-trained model, the perplexity score improved a lot (from 21.075 down to 15.601), doing much better than the standard GPT-2 tokenizer. On the other hand, a custom BPE tokenizer trained on a mix of all the different preprocessed versions of WikiText-2 did poorly. This was true even if the fine-tuning data matched one of the cleaning styles it was trained on. This really shows how important it is for the tokenizer's training data to match the kind of data the main language model will actually see.

6.0.2 Limitations of This Study

While this study offers some useful insights, it is important to note its limitations. The experiments were mostly done on the WikiText-2 dataset. The findings might be different for other types of datasets, other languages, or different topics. The Multi-Word Tokenizer was evaluated based on one specific way of building it, and other MWT methods might give different results.

6.0.3 Further work

Based on what was found and the limitations mentioned, here are some ideas for future research:

- Improving Multi-Word Tokenization: It might be beneficial to look more into why the MWT did not perform well. This could mean trying out different MWT algorithms, changing its training settings, or using different base tokenizers to start it off.
- Smart Preprocessing for Custom Tokenizers: Even though training a tokenizer on all cleaned-up versions of the data did not work well, future studies could try training custom BPE tokenizers on carefully chosen cleaned-up versions of a dataset. This might be useful if a particular cleaning step seems promising for a certain model.
- Testing on More Data and Languages: It would be valuable to repeat the key experiments on different kinds of datasets (e.g., different topics, sizes, and writing styles) and in other languages. This would help see how broadly these conclusions about text cleaning and custom tokenizers apply.

- Looking Deeper into Dual-Stream Models: Since models that process two streams of information have potential, future work could look for other ways to use this kind of setup. Maybe the second stream could get different kinds of helpful information, or perhaps the two streams could be combined in more advanced ways than just adding their outputs, especially when using the very effective custom BPE tokenizer.

By exploring these areas, future research can help get an even better handle on how data representation, model design, and language model performance all connect, ultimately leading to better and more efficient NLP systems.

Appendix A

Bibliography

[1] Cole Stryker and Jim Holdsworth. What is NLP (natural language processing)? Accessed: 2025-04-06.

[2] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units, 2016. arXiv:1508.07909.

[3] Mike Schuster and Kaisuke Nakajima. Japanese and Korean voice search, 2012.

[4] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing, 2018. arXiv:1808.06226.

[5] Leonidas Gee, Leonardo Rigutini, Marco Ernandes, and Andrea Zugarini. Multi-word Tokenization for Sequence Compression, 2024. arXiv:2402.09949.

[6] Yaakov HaCohen-Kerner, Daniel Miller, and Yair Yigal. The influence of preprocessing on text classification using a bag-of-words representation, 2020.

[7] David Heckerman. A Tutorial on Learning With Bayesian Networks, 2022. arXiv:2002.00269.

[8] S. S. Keerthi, K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO Algorithm for SVM Classifier Design, 2001.

[9] Leo Breiman. Random Forests, 2001.

[10] Jose Camacho-Collados and Mohammad Taher Pilehvar. On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis, 2018. arXiv:1707.01780.

[11] Tomas Mikolov, Kai Chen, et al. Efficient Estimation of Word Representations in Vector Space, 2013. arXiv:1301.3781.

[12] Lijie Zhu and Difan Luo. A Novel Efficient and Effective Preprocessing Algorithm for Text Classification, 2023.

[13] Stephen Merity, Caiming Xiong, et al. Pointer Sentinel Mixture Models, 2016. arXiv:1609.07843.

[14] Andrej Karpathy. nanoGPT. Accessed: 2025-04-06.

[15] Leonidas Gee, Andrea Zugarini, Leonardo Rigutini, Marco Ernandes, and Paolo Torroni. Fast Vocabulary Transfer Multi-word Tokenization. Accessed: 2025-05-06.

[16] Leonidas Gee, Andrea Zugarini, Leonardo Rigutini, and Paolo Torroni. Fast Vocabulary Transfer for Language Model Compression. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 409–416, Abu Dhabi, UAE, December 2022. Association for Computational Linguistics.

[17] Leonidas Gee, Leonardo Rigutini, Marco Ernandes, and Andrea Zugarini. Multi-word Tokenization for Sequence Compression. In Mingxuan Wang and Imed Zitouni, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 612–621, Singapore, December 2023. Association for Computational Linguistics.

[18] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M Rush. Huggingface's transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.

Appendix B

List of Attachements

This section details the structure and contents of the supplementary materials for this thesis, which include configuration files, evaluation scripts, tokenization utilities, and training scripts.

Directory Structure and File Descriptions

- `configs/` – This folder contains the configuration files used for training the various models.
  - `train_dual_stream_pretrained.py` – Configuration to train the dual-stream architecture (processing original and preprocessed data streams) using a pretrained GPT-2 model.
  - `train_dual_stream_scratch.py` – Configuration to train the dual-stream architecture (processing original and preprocessed data streams) from scratch.
  - `train_one_stream_scratch.py` – Configuration to train the single-stream architecture from scratch.
- `evaluate/` – This folder contains scripts for evaluating the performance of the trained models.
  - `evaluate_model_dual_stream.py` – Script to evaluate the dual-stream architecture model.
  - `evaluate_model_one_stream.py` – Script to evaluate the single-stream architecture model.
- `tokenization/` – This folder contains scripts related to creating tokenizers and preprocessing data using these tokenizers.
  - `create_bpe_all_tokenizer.py` – Script to train a Byte Pair Encoding (BPE) tokenizer on the WikiText-2 dataset using all types of preprocessed data (e.g., lowercase, punctuation removal).
  - `create_mwt_prepare_data.py` – Script to train a Multi-Word Tokenizer (MWT), save it, and then use it to process the dataset.

- `prepare_data_gpt2.py` – Script to prepare the original dataset using the pretrained GPT-2 tokenizer (originally from nanoGPT).
- `create_bpe_prepare_data.py` – Script to train a BPE tokenizer on the original, untouched WikiText-2 dataset, save it, and then use it to process the dataset.
- `prepare_data_bpe_all.py` – Script to process data using the `bpe_all` tokenizer (trained on all preprocessed data).
- `prepare_data_gpt2_preproc.py` – Script to prepare preprocessed data (applying lowercase, punctuation removal, etc.) using the pretrained GPT-2 tokenizer.

- `train/` – This folder contains the primary scripts for training the different model architectures and configurations.
  - `train_dual_stream_gpt2_preproc.py` – Script to train the dual-stream architecture on both original and preprocessed data from scratch using the pretrained GPT-2 tokenizer.
  - `train_one_stream_mwt.py` – Script to train the single-stream architecture from scratch using the Multi-Word Tokenizer (MWT).
  - `train_dual_stream_pretrained_gpt2.py` – Script to train the dual-stream architecture on both original and preprocessed data using a pretrained GPT-2 model and its tokenizer.
  - `train_one_stream_pretrained_bpe_all.py` – Script to train the single-stream architecture using a pretrained GPT-2 model and the BPE tokenizer trained on all types of preprocessed data.
  - `train_one_stream_gpt2.py` – Script to train the single-stream architecture from scratch using the pretrained GPT-2 tokenizer (originally from nanoGPT).

- `model.py` – This Python file contains the class definitions for the language models, including the base GPT model (originally from nanoGPT) and the custom `GPTDualStream` architecture.

- `configurator.py` – This Python file is responsible for reading and parsing the configuration files located in the `configs/` directory to set up training parameters (originally from nanoGPT).