문학석사 학위논문

# Exploring the Effects of Tokenizer Extension on Domain-specific Fine-tuning of Language Models

## 언어 모델의 전문 분야 미세 조정을 통한 토크나이저 확장의 효과 연구

2025 년 02 월

서울대학교 대학원

언어학과 언어학전공

김 재 윤

# Exploring the Effects of Tokenizer Extension on Domain-specific Fine-tuning of Language Models

지도 교수 신 효 필

이 논문을 문학석사 학위논문으로 제출함

2025년 02월

서울대학교 대학원

언어학과 언어학전공

김 재 윤

김재윤의 문학석사 학위논문을 인준함

2025년 02월

위 원 장 이 상 아 (인)

부위원장 신 효 필 (인)

위 원 김 문 형 (인)

# Exploring the Effects of Tokenizer Extension on Domain-specific Fine-tuning of Language Models

Advising Professor, Dr. Hyopil Shin

Submitting a master's thesis of Art

February 2025

Graduate School of Humanities
Seoul National University
Linguistics Major

Jaeyoon Kim

Confirming the master's thesis written by
Jaeyoon Kim
February 2025

| | | |
|---|---|---|
| Chair | Sangah Lee | (Seal) |
| Vice Chair | Hyopil Shin | (Seal) |
| Examiner | Munhyong Kim | (Seal) |

# Abstract

Jaeyoon Kim

Department of Linguistics

The Graduate School

Seoul National University

Large language models (LLMs) that undergo extensive pretraining on massive datasets over long periods have become dominant nowadays. Since it is highly challenging for individuals to train such models from scratch, it has become common practice to fine-tune shared pretrained models for specific tasks. However, when the vocabulary distribution of the data used for fine-tuning significantly differs from the vocabulary which the existing tokenizer can process, issues can arise, such as the tokenizer failing to handle the data properly or excessively fragmenting words into overly short tokens.

Extending the tokenizer by adding new vocabulary items can be an effective solution to mitigate these problems, however, there has been little in-depth research on the specific effects of tokenizer extension. Therefore, this study aimed to analyze the effects of tokenizer extension on domain-specific fine-tuning by training small models, using tokenizers extended with medical data and conducting several analyses. The medical domain, characterized by its frequent use of specialized terminology, was anticipated to benefit positively from tokenizer extension.

Experiments were conducted by extending BPE (Byte Pair Encoding)-based methods, including SentencePiece BPE, Byte-level BPE, and WordPiece, which uses a similar algorithm. The results showed that while the tokenizer's compression capability slightly improved, the memory and time required for model training increased. In addition, evaluated with 4-options multiple choice questions from MultiMedQA, models with extended tokenizer showed worse performance than the models with not extended ones. From these results, it could be concluded that tokenizer extension may not be helpful, when it comes to fine-tuning a language model with domain-specific data.

**Keywords :** Tokenizer Extension, Medical Fine-tuning, Language Model, Byte Pair Encoding, WordPiece, SentencePiece BPE, Byte-level BPE, Compression

**Student Number :** 2023-28898

# Table of Contents

# List of Algorithms

# List of Figures

# List of Tables

# 1. Introduction

Since the advent of the Transformer (Vaswani et al., 2017) architecture, which demonstrated high performance in natural language processing, numerous models such as GPT, BERT, and T5 have been developed based on variations of the Transformer. These models are now chosen and applied based on the specific requirements of a given task. However, with the introduction of the scaling laws for neural language models and the emergence of "emergent abilities" in causal language models of sufficient scale, large language models (LLMs) with billions of parameters have come into existence. LLMs of a sufficiently large scale can handle most tasks in a few-shot or zero-shot manner, leading to their widespread use across various tasks. However, training high-performing models requires access to vast amounts of well-curated data, large-scale computing resources, and extensive pretraining periods. Consequently, it is often impractical or highly inefficient for individual users to train their own models from scratch. This has led to the development of platforms for sharing pretrained models and advanced fine-tuning techniques that allow users to effectively adapt pretrained models for specific needs.

One critical consideration when fine-tuning a pretrained model is vocabulary. This becomes particularly important, when fine-tuning a model for a new language or specialized domain. Significant differences between the vocabulary distributions of the pretraining corpus and the fine-tuning corpus can lead to out-of-vocabulary (OOV) issues or overly fragmented tokenization, where words are broken into too many tokens, making it difficult for the model to understand their meaning. To

address this, extending the vocabulary and merge rules of the tokenizer to better capture the words in the fine-tuning corpus can be an effective approach. However, the impact of such extensions has not been thoroughly analyzed in existing studies.

In this research, the main goal was to analyze the effects of tokenizer extension on domain-specific fine-tuning of language models, focusing on its application in the medical domain. The medical field, characterized by an extensive use of specialized terminology, is a domain where such tokenizer extensions could be highly beneficial. For experiment, three models: BERT, MobileLLM, SmolLM2 with extended tokenizers were fine-tuned with medical texts and multiple choice questions. These models adopt BPE-based methods: WordPiece, SentencePiece BPE, and Byte-level BPE as tokenization respectively, which comprise the majority of recent NLP tokenizations.

After training, the models were evaluated with 4-options multiple choice questions from MultiMedQA, and the effects of tokenizer extension on other factors such as compression and training costs were measured. By analyzing the results obtained from the above, this study provides insights toward tokenizer extension and clarify whether it is beneficial to extend tokenizers in case of domain-specific fine-tuning of language models.

# 2. Related Works

## 2.1. Subword Tokenizations

### 2.1.1. Background of Tokenizers

Regardless of their architecture, language models ultimately process numbers. This implies that all language models must first convert text into numerical representations before they can model language. This conversion process involves the following steps in Figure 1:

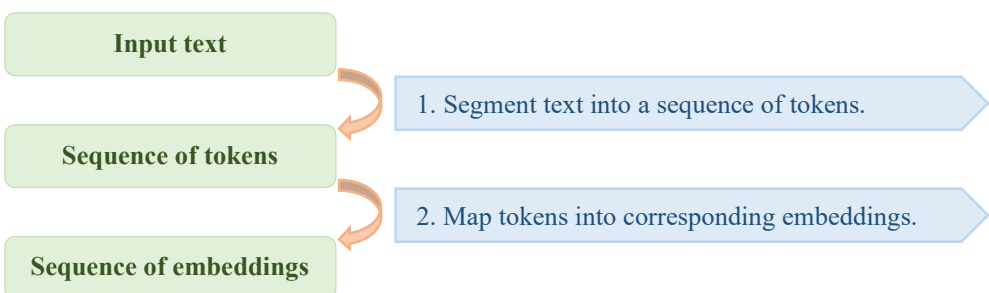


Figure 1. Steps of converting input text into numerical representations

The process of converting a token sequence into stacked embeddings (Step 2) is typically performed in parallel by first encoding the tokens into mapped indices, then transforming these indices into one-hot vectors, and finally multiplying them with embedding look-up table. Breaking this down further, the steps can be outlined as in Figure 2:

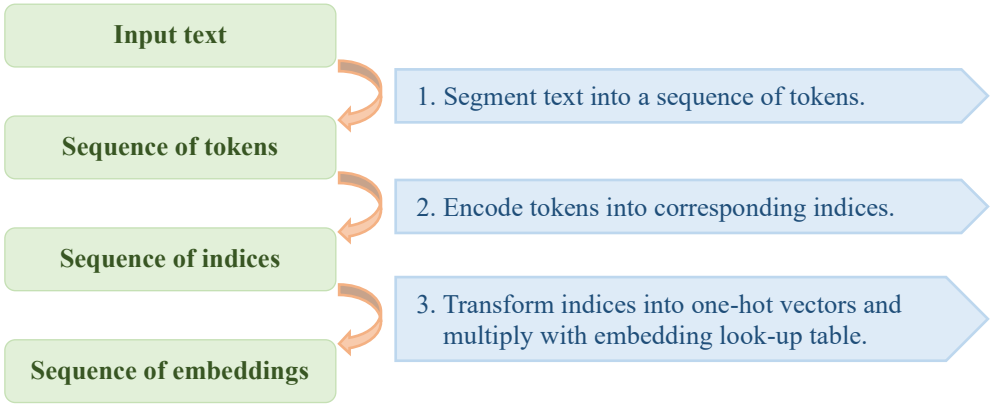


Figure 2. Steps of converting input text into numerical representations (revised)

Since dense vector representations of text—such as Word2Vec (Mikolov et al., 2013), GloVe (Pennington et al., 2014)—were introduced, these steps have become a fixed component of all language models. Among the steps, Step 3 is typically handled directly by the models, as the embedding look-up tables are now commonly trainable or frozen matrices of parameter that is integrated into the models themselves. However, the processes up to Step 2—converting input text into a sequence of indices—require an external module, commonly referred to as a **tokenizer**. Language models acquire their understanding of language by working with the numerical arrays produced by the tokenizer, therefore it is essential to use the same tokenizer that was employed during the model's training, to exploit a language model's ability effectively.

Some tokenizers perform **normalization** and **pre-tokenization** before segmenting and encoding the input text. Normalization involves tasks which are similar to text cleaning, ranging from simple substitutions—such as converting uppercase letters to lowercase, stripping accent symbols, or replacing all whitespace characters with spaces—to more complex processes like applying regular

expressions or Unicode normalization methods such as NFC and NFD. Pre-tokenization is the process of breaking the text into segments based on predefined rules prior to tokenization. This can include splitting text using whitespace or specific characters like '-'(hyphens) or '/'(slashes) as delimiters, isolating characters such as digits and punctuations from other adjacent characters, or separating character sequences from different Unicode categories and subcategories.

A tokenizer then performs tokenization and encoding based on a predefined vocabulary. Through specific algorithms, tokens are registered in the vocabulary alongside their indices, and other components such as merge rules are obtained if they are required for tokenization, depending on the type of the tokenizer. Subsequently, texts can be segmented into tokens that match those in the vocabulary and converted into their mapped indices. The critical issue here is determining what units to use as tokens, when constructing the vocabulary and implementing segmentation.

When segmenting text, one of the most intuitive units would be **word**. Words can be easily segmented using whitespace as a delimiter, aligning with the intuitive goal of tokenization, which is to break down a document or sentence into smaller meaningful units. Early language models often used words as tokens, and it was common to build vocabularies by segmenting a corpus based on whitespace, setting minimum frequency thresholds, and collecting unique items. However, using words as tokens has several limitations: it requires large number of tokens to capture inflected forms of words, and it often leads to out-of-vocabulary (OOV) issues (since the vocabulary is sensitive only to the training corpus), reducing the model's flexibility.

### 2.1.2. Subwords as Tokens

As discussed above, using words as units for tokenization is intuitive but comes with significant limitations. This has led to the development of methods that use other units as tokens. The main issue with word-level tokenization arises from the sheer number of unique words, making it impractical. To address this, breaking text into smaller units than words to reduce diversity among tokens can be an effective solution. When splitting text into smaller units than words, a natural choice might be characters, thus methods using characters as units (Chung et al., 2016) or using both words and characters as units (Luong and Manning, 2016) had been proposed.

However, using characters as units introduces several problems. First, converting each character in the input text into indices results in sequences that are excessively long. This reduces the amount of text that can be captured within a fixed-length index array and increases processing costs for texts with same length. Moreover, individual character tokens convey very little meaning. Considering that the goal of tokenization is to segment input text into smaller meaningful units to help the model process the text, characters are not ideal as units. To sum up, while words and characters are intuitive and convenient for segmenting text, they are not suitable as tokenization units.

As a countermeasure, **subwords**, which are smaller than words but longer than characters, were proposed as tokenization units. Subwords strike a balance by being more flexible than words while containing more information than characters, and diverse approaches to building subword vocabularies and performing segmentation with them have emerged. As models using subword tokenization have demonstrated strong performance, subwords has been established as the standard tokenization unit,

and today, all well-known models rely on subword tokenization such as WordPiece and BPE.

### 2.1.3. Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) is a type of data compression technique that repeatedly replaces the most frequent pair of bytes in a sequence with an unused byte (Gage, 1994). This method can also be applied to character sequences, making it suitable for tokenization. BPE forms the foundation of many tokenizers used in large language models nowadays, however it was introduced to natural language processing even before the rise of transformers. It was first utilized in neural machine translation to address the out-of-vocabulary (OOV) issue by breaking unknown and rare words into subwords (Sennrich et al., 2015).

When used for tokenization, BPE follows a similar process to its original compression method: it learns a vocabulary by iteratively merging the most frequent pairs of characters or subwords. During this process, the merge operations are recorded as merge rules. To tokenize an input text, the text is first converted into a character sequence, and the merge rules are applied to the sequence to generate a sequence of tokens.

### 2.1.4. WordPiece

WordPiece is a tokenization method developed by Google Research, initially designed to address technical challenges in speech recognition systems. To be specific, languages like Korean and Japanese which blend diverse characters within

sentences and use few whitespaces, often result in a high number of homonyms, making text processing difficult. WordPiece was created to ease this by adding word boundary markers to the corpus segmented on spaces, and using a vocabulary trained to maximize the likelihood of character sequences in the corpus (Schuster & Nakajima, 2012). Inspired by the application of BPE, Google later adopted WordPiece for their neural machine translation system (Wu et al., 2016). WordPiece also became the tokenization method of BERT (Devlin et al., 2019), solidifying its position as a major tokenization method.

Initially, Google did not release Python code for training WordPiece models due to dependencies with C##, leaving the details of how pairs were merged to maximize corpus likelihood unclear. There had been several failed attempts to replicate the original WordPiece vocabularies using the same corpora, and HuggingFace eventually succeeded to replicate the vocabulary with its WordPieceTrainer, revealing the specific training algorithm of a WordPiece model. Unlike BPE, which merges pairs based on raw frequency, WordPiece calculates a score for each pair as 'frequency / (product of the frequencies of the two elements)' and merges the pair with the highest score. When it comes to segmentation, WordPiece identifies the longest possible token starting from the beginning of the words. If a word is too long or contains a part which cannot be covered by the vocabulary, a WordPiece model replace the word with an unknown token.

### 2.1.5. SentencePiece BPE

SentencePiece (Kudo & Richardson, 2018) is a pre-tokenization and decoding

method designed to achieve "lossless tokenization". Lossless tokenization ensures that:

1. The token sequence generated from a given text is unique.
2. The text restored from a given token sequence is also unique.

Borrowing the expression from the original paper, this can be summarized as:

$$\text{Decode}(\text{Encode}(\text{Normalize}(\text{text}))) = \text{Normalize}(\text{text})$$

Such a one-to-one correspondence requires that no information is lost during text segmentation. In WordPiece, however, spaces are used as delimiters and prefixes are added to continuing subwords, which results in the loss of space sequences (regardless of their length), introducing ambiguity during decoding. To avoid this, SentencePiece escapes spaces with a meta-symbol “_” (U+2581) and treats it as a single Unicode character equivalent to other characters.

Since SentencePiece itself is a method which only guarantees lossless tokenization, it has to be combined with other tokenization models such as BPE or Unigram (Kudo, 2018). As one of such combinations, SentencePiece BPE uses SentencePiece to pre-tokenize the corpus, learns the vocabulary and merge rules using BPE, and follows SentencePiece's decoding process. Additionally, SentencePiece supports a strategy called Byte-fallback, which handles unknown tokens by replacing them with byte tokens according to their UTF-8 encoding. SentencePiece BPE adopts this strategy as well, ensuring that tokens not covered by the vocabulary can still be processed.

By combining the strengths of both SentencePiece and BPE, SentencePiece BPE has become one of the most widely used tokenization methods, being utilized in leading LLMs such as Llama(Touvron et al., 2023a), Llama 2(Touvron et al., 2023b) and Gemma(Team et al., 2024a), Gemma 2(Team et al., 2024b).

### 2.1.6. Byte-level BPE

Byte-Level BPE is a variant of BPE, first introduced as the tokenization method for GPT-2 (Radford et al., 2019). The core idea of Byte-Level BPE is to convert all Unicode characters in a corpus into byte sequences based on UTF-8 encoding before applying BPE. In other BPE tokenization methods, such as SentencePiece BPE, the training corpus's unique Unicode characters must first be added to the initial vocabulary to learn the vocabulary and merge rules. However, Byte-Level BPE avoids the issue of an excessively large initial vocabulary. This is because UTF-8 encoding represents characters using only 256 bytes, the initial vocabulary for Byte-Level BPE needs to include only tokens representing these 256 bytes. Additionally, since all Unicode characters are mapped to code points and can be represented using UTF-8 encoding, Byte-Level BPE is inherently immune to the out-of-vocabulary (OOV) problem and does not require counteractive strategies like Byte-fallback.

The 256 byte tokens in Byte-Level BPE correspond to the earliest Unicode characters, excluding control characters. Plus, Unicode characters with code points below 256 are directly represented by the corresponding byte token. Aside from converting text into byte sequences, the training process and segmentation in Byte-Level BPE follow the same steps as other BPE methods. In addition, the original Byte-Level BPE introduced an additional mechanism to prevent suboptimal merges,

by discouraging merges between byte tokens representing different categories of Unicode characters. Thanks to its robust and powerful advantages, Byte-Level BPE has become an unquestionable major tokenization method. It is widely used in leading large language models such as Llama 3 (Dubey et al., 2024), Qwen (Bai et al., 2023), and Qwen 2.5 (Yang et al., 2024).

## 2.2. BPE and Compression

As described earlier, BPE was initially designed as a simple data compression method. While it has since been adopted in natural language processing through its use in neural machine translation and is now a core component of most tokenizers, its mechanism and effects remain closely tied to compression. Consequently, research into the relationship between BPE, compression, and its effects in NLP continues to gain attention. Below, key studies related to BPE and compression are summarized.

Research on BPE's connection to compression can be divided into two camps: those arguing that the effectiveness of BPE as a tokenizer stems from its compression capability and those who dispute this claim. To introduce a study from the former camp, in Gallé (2019), the author hypothesized that models would perform better if the same sentence could be captured with fewer tokens, assuming a fixed budget for vocabulary size. By applying BPE and other dictionary-based compression algorithms to tokenization, the study compared model performance on machine translation tasks. The results showed that models capturing sentences with fewer tokens achieved higher BLEU scores on the test set, leading to the conclusion that

BPE's effectiveness is linked to its compression capability.

Similarly, Goldman et al. (2024) investigated the relationship between BPE tokenizer compression and model performance. Keeping the vocabulary size fixed, they controlled the level of compression by varying the number of documents used to train the tokenizer. Models were then trained on these documents and evaluated on benchmarks such as QQP and MultiNLI, and it was found that models using tokenizers with higher compression performed better. Additional experiments suggested that the degree of compression could serve as an evaluation metric for tokenizers.

On the other hand, Schmidt et al. (2024) presented a contrasting perspective. After varying the degree of compression through vocabulary size and training 64 models under some other controlled conditions, they evaluated model performance on several downstream tasks in a few-shot setting. Their findings showed minimal performance differences based on vocabulary size, while factors such as pre-tokenization, vocabulary construction, and segmentation had a greater impact on performance. The study concluded that elements other than compression played a larger role in determining model performance and, casted doubt on the idea that BPE's effectiveness is derived from its compression ability.

While BPE remains an effective tokenization method, some studies argue that it is not optimal because it fails to account for morphology. In Bostrom & Durett (2020), the authors criticized BPE for its lack of alignment with morphological structure, making it less suitable for language model pretraining. They pretrained models using BPE and Unigram tokenizers on masked language modeling tasks and compared their downstream performance. On evaluation, models using Unigram outperformed those using BPE, leading the authors to conclude that BPE's inability to reflect

morphology makes it suboptimal.

Conversely, Gutierrez-Vasques et al. (2023) offered a different view. By training BPE tokenizers on languages with varying morphological typologies and analyzing the resulting vocabularies, the study suggested that BPE adapts its compression to align with morphological typology. In morphologically rich languages, productive subwords were prioritized, while in less inflected languages, idiosyncratic subwords were learned first. Regarding these results, the authors concluded that BPE generates subwords that characterize each language's structure.

The relationship between BPE and compression, as well as its connection to morphology and effectiveness, remains unclear and controversial. While compression is fundamental to BPE's design, the reasons for its effectiveness may lie elsewhere, highlighting the need for further research. And in this study, among such research questions, the relation between extension of BPE-based tokenizers and improvement on compression is examined.

# 3. Datasets & Models

This section provides detailed information about datasets and models, which will be repeatedly suggested on sections afterwards, and used throughout the entire process of experiment.

## 3.1. Datasets

### 3.1.1. PubMed

PubMed is a free platform for browsing biomedical literature, which is developed and maintained by the U.S. National Library of Medicine (NLM) at the National Institutes of Health (NIH). It provides access to a vast database of citations and abstracts in the fields of biomedicine and life sciences, with the goal of enhancing global and personal health. PubMed was launched in January 1996 providing access to institutional facilities like university libraries, however from June 1997, it became freely available to the public, significantly broadening access to biomedical literature. As of December 2024, PubMed comprises more than 37 million citations from MEDLINE (a bibliographic database hosted by NLM), life science journals, and online books. While it does not include full-text articles, it often provides articles' abstracts, or links to full-text available through publisher websites or PMC.

PubMed also offers various search functionalities, including advanced search options, clinical queries, and a single citation matcher, in addition to tools for data

mining and bulk processing. Plus, since it ensures the entire database to be available for download in XML format, it facilitates large-scale analyses and the development of biomedical language models. While PubMed provides such diverse conveniences, only download service was exploited to collect abstracts from medical articles, for experiments on section 5.

### 3.1.2. PMC (PubMed Central)

PMC (PubMed Central) is a free digital repository which archives open access full-text scholarly articles from biomedical and life sciences journals. It offers direct access to full-text articles unlike PubMed (which only provides abstract, citations, and links to full-text), enhancing the public's ability to discover, read, and build upon biomedical knowledge. It is established by the U.S. National Institutes of Health's National Library of Medicine (NIH/NLM) in 2000, and it contains over 5.2 million articles from approximately 4,000 journals (including some publishers implementing delayed release) nowadays. PMC serves as a comprehensive resource for researchers, healthcare professionals, and the public to access a vast collection of biomedical literature.

PMC utilizes standardized XML formats to ensure the longevity and accessibility of its content. Articles are submitted by publishers in XML or SGML formats and are converted to the NLM Archiving and Interchange DTD for consistency. This process facilitates linking to related data objects and integration with other NCBI databases, providing a robust platform for information retrieval and research. For researchers and developers interested in bulk data access, PMC also offers tools for

bulk download, text mining, and other machine analysis, supporting a wide range of scientific inquiries and applications. Among these services, bulk download was used to collect full-text medical articles for experiments on section 5.

### 3.1.3. MultiMedQA subsets

MultiMedQA (Singhal et al., 2023) is a comprehensive benchmark introduced by Google Research to evaluate the performance of large language models in the medical domain. It combines six existing open question-answering datasets, encompassing professional medical exams, research, and consumer health queries, and was utilized to assess Med-PaLM, a large language model fine-tuned for medical question answering. Among 6 subsets of MultiMedQA, 3 of datasets are composed of 4-options multiple choice questions, and widely adopted for measuring language models' medical knowledge. These 3 subsets are used for fine-tuning and evaluating models on later sections, since they share the number of options.

#### 3.1.3.1. MedQA

MedQA (Jin et al., 2020) is a comprehensive open-domain question-answering dataset tailored for the medical field. It comprises multiple-choice questions designed to assess the professional knowledge and clinical decision-making abilities of physicians, covering a wide range of medical topics. The dataset presents a significant challenge for existing open-domain question-answering systems, as it requires models to retrieve relevant information from extensive medical literature

and perform complex reasoning to arrive at the correct answers. The dataset and baseline source code of MedQA are publicly available for research purposes, supporting the development and evaluation of advanced question-answering models in the medical domain.

The questions of MedQA are sourced from professional medical board examinations across three regions: the United States, Mainland China, and Taiwan, and as result, the dataset includes 12,723 questions in English, 34,251 in simplified Chinese, and 14,123 in traditional Chinese. Among these, questions in English are used only.

#### 3.1.3.2. MedMCQA

MedMCQA (Pal et al., 2022) is a large-scale Multiple-Choice Question Answering (MCQA) dataset specifically designed for the medical domain. It comprises over 194,000 high-quality multiple-choice questions, sourced from two postgraduate entrance exams in medicine, AIIMS(All India Institute of Medical Sciences) PG and NEET(National Eligibility cum Entrance Test) PG, which are conducted by AIIMS and NBE(National Board of Examinations). The questions cover 2,400 healthcare topics across 21 medical subjects, and is structured to test a model's reasoning abilities across a wide range of medical subjects and topics, contributing to advancements in natural language understanding within the medical field. MedMCQA is publicly available for research purposes, with data and code accessible through its GitHub repository.

Each sample of MedMCQA contains question, options, and correct answer, and

most of samples additionally includes a detailed explanation of the correct answer by experts. The train and the validation split were used, while the test split which does not include correct labels was excluded.

#### 3.1.3.3. MMLU Clinical Topics

The MMLU (Massive Multitask Language Understanding) benchmark (Hendrycks et al., 2020) is designed to evaluate the breadth and depth of a language model's knowledge across 57 diverse subjects, categorized into STEM, humanities, social sciences, and others. It comprises approximately 16,000 multiple-choice questions that range from elementary to advanced professional levels, assessing both world knowledge and problem-solving abilities. The data of MMLU are publicly available through online platforms such as GitHub and HuggingFace, and MMLU has become one of the standard benchmarks for evaluating large language models.

MMLU Clinical Topics include 6 subjects from MMLU whose subcategories are either biology or health: Anatomy, Clinical Knowledge, College Biology, College Medicine, Medical Genetics, Professional Medicine. The 6 subjects contain samples with question, 4 options, and correct answer in common, and their test splits were used in later sections.

## 3.2. Models

While the following sections provide details of the models, Table 1 summarizes the configurations.

| Model | BERT Base Cased | MobileLLM 125M | SmolLM2 135M |
|---|---|---|---|
| **Architecture** | Transformer Encoder | Transformer Decoder | Transformer Decoder |
| **# of Layers** | 12 | 30 | 30 |
| **Hidden Dimension** | 768 | 576 | 576 |
| **Feed-Forward Dimension** | 3,072 | 1,536 | 1,536 |
| **# of Attention Heads** | 12 | 9 | 9 |
| **# of Parameters** | 108, 310,272 | 124,635,456 | 134,515,008 |
| **Tokenization** | WordPiece | SentencePiece BPE | Byte-level BPE |
| **Vocabulary Size** | 28,996 | 32,000 | 49,152 |

Table 1: Summary of base model configurations

### 3.2.1. BERT

BERT (Devlin et al., 2019) is a model built using the encoder component of the Transformer architecture. The BERT Base configuration was designed to match the model configuration of OpenAI's GPT-1 (Radford et al., 2018) for comparison purposes, with the following specifications:

- Number of layers: 12
- Hidden dimension: 768
- Feed-Forward hidden dimension: 3072
- Attention heads: 12

Unlike GPT-1, which uses the Transformer decoder and is optimized for generative tasks, BERT specializes in encoding input texts and focuses on natural language understanding. Its pretraining involves two key tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). MLM is a task where certain portion of tokens from the input texts are masked, and the model predicts the original

tokens. The masking strategy is as follows:

- 15% of the words are selected for masking, where:
  - 80% (12% of the total) are replaced with the mask token([MASK]).
  - 10% (1.5% of the total) are replaced with random words.
  - 10% (1.5% of the total) are left unchanged.

NSP involves taking a pair of sentences as input and predicting whether the second sentence naturally follows the first. To support this, BERT introduces two special features: the classification token ([CLS]) and segment embeddings, which indicate whether each token belongs to the first or second sentence in the pair.

BERT uses a WordPiece tokenizer, with the prefix "##" prepended to subwords which are not word-initial tokens. The model was trained in two versions: Cased (case-sensitive) and Uncased (case-insensitive). While various versions of BERT with different configurations have been released since its introduction, BERT Base Cased with vocabulary size of 28,996 was used for experiment on section 5.

### 3.2.2. MobileLLM

MobileLLM (Liu et al., 2024) is a model released by Meta as part of its research into efficient language models for mobile devices. Although versions with more than 1 billion parameters are available, the core focus of the research was on optimizing models with fewer than 1 billion parameters. The smallest version, with 125 million parameters, was used for experiments. Its configuration is as follows:

- Number of layers: 30
- Hidden dimension: 576
- Feed-Forward hidden dimension: 1536
- Attention heads: 9

The study emphasized that in smaller models, architecture has a more significant impact on performance than other factors. As a result, MobileLLM incorporates several architectural techniques, including:

- **Embedding Sharing**: The embedding table is reused as the language model (LM) head.
- **Grouped Query Attention**: Optimizes the exploitation of weights.
- **Layer Sharing**: Weights are shared between adjacent blocks, to scale up the model while not increasing the number of trainable parameters.

Guided by previous researches highlighting the importance of depth in small models, MobileLLM is designed to have a deep-thin (many layers, with narrow widths) structure. Multiple versions of the model were released based on different configurations of embedding sharing and layer sharing, and a version with embedding sharing enabled but layer sharing disabled was adopted for the experiments. MobileLLM uses a SentencePiece BPE tokenizer, and its vocabulary size is 32,000

### 3.2.3. SmolLM2

SmolLM2 (Allal et al., 2024) is a model developed by Hugging Face as an updated version of the SmolLM project, which focused on building small, fast models. Although no official paper has been published, the model and related information are available online through GitHub and HuggingFace. SmolLM2 emphasizes the use of high-quality, large-scale pretraining data, which has also been publicly released for use by others.

Base and Instruct versions are available, for models with 3 different scales: 135M, 360M and 1.7B. 135M Base model, which is used for the experiments on later sections, are built with the following configuration:

- Number of layers: 30
- Hidden dimension: 576
- Feed-Forward hidden dimension: 1536
- Attention heads: 9

The project also supports tools such as summarizers, rewriters, and AI agents, based on the larger 1.7B Instruct model. These tools can be easily accessed through Python code, after a simple installation process. For tokenization, SmolLM2 uses a Byte-level BPE tokenizer with vocabulary size of 49,152.

# 4. Tokenizer Extension

In this section, the methods of extending the original tokenizers of BERT, MobileLLM, SmolLM2 are explained step-by-step. The processes of extending the original tokenizers were conducted in two main stages:

1. **Training Supplement Tokenizers** for tokenization methods (WordPiece, SentencePiece BPE, Byte-level BPE).
2. **Extending the Original Vocabulary and Merge Rules** with the supplement tokenizer's vocabulary and merge rules.

To ensure the extended tokenizers to work flawlessly, vocabulary and merge rules of each supplement tokenizer must be compatible with those of the original tokenizer. Therefore, the steps above should be implemented with thorough consideration of the features of the original tokenizers.

## 4.1. Details on the Original Tokenizers

While the original tokenizers include a number of components, they can be broken down into 3 major parts: 1) **Normalizations & Pre-tokenizations**, 2) **Training Algorithms**, and 3) **Tokenization Algorithms**. The following sections explain the details of each part.

### 4.1.1. Normalizations & Pre-tokenizations

In case of the normalization or pre-tokenization of the original tokenizer being stricter than that of a supplement tokenizer, some of non-overlapping tokens from the supplement tokenizer might not survive the normalization or pre-tokenization. For instance, even if a token spelled ‘The’ from a case-sensitive supplement tokenizer is appended to the original tokenizer’s case-insensitive vocabulary, it will never appear in the results of tokenization since all ‘The’ in the input text will be normalized into ‘the’. As result, the embedding mapped to ‘The’ becomes untrainable, which leads the extension to be meaningless.

To avoid adding untrainable tokens to the original vocabulary, appropriate normalization and pre-tokenization should have been applied when training the supplement tokenizers. The normalization and pre-tokenization details for each tokenizer are as the followings.

#### 4.1.1.1. BERT

BERT's tokenizer is designed to be capable of implementing various normalizations and pre-tokenizations, while all the normalization and pre-tokenization steps can be included or excluded by users. BERT’s tokenizer can apply the following normalization steps:

1. Remove control characters.

   Example: ‘\x7f’ (delete) → ‘’.
2. Replace all whitespace with spaces.

Example: ‘\x0b’ (vertical tabulation) → ‘ ‘.

3. Add spaces around each Chinese character.

   Example: ‘漢字’ → ‘ 漢　字 ’.

4. Convert uppercase letters to lowercase.

   Example: ‘ABc’ → ‘abc’.

5. Remove accent diacritics.

   Example: ‘áèõ’ → ‘aeo’.

Control characters refer to characters which belong to Unicode general category ‘others’, while whitespace includes characters which belong to general category ‘separator, space (Zs)’ or one of the bidirectional categories like ‘B’, ‘S’, or ‘WS’. Horizontal tab (‘\t’), line feed (‘\n’), and carriage return (‘\r’) are treated as whitespace and replaced with spaces rather than being removed. The BERT base cased tokenizer used in the experiments applies only steps 1, 2, and 3.

Next, BERT’s pre-tokenization includes:

1. Split text by spaces as delimiters.

   Example: ‘a text’ → ‘a’, ‘text’.

2. Isolate each punctuation from surrounding characters.

   Example: ‘a.!?b’ → ‘a’, ‘.’, ‘!’, ‘?’, ‘b’.

Here, punctuations are defined as the characters which satisfy at least one of the followings: 1) belong to the ASCII code, but do not not belong to letters or digits, 2) belong to Unicode general category ‘punctuation’. For reference, the list of punctuations in ASCII plane are shown in Table 2.

| Character | Name | Unicode Code Point | Character | Name | Unicode Code Point |
|---|---|---|---|---|---|
| ! | Exclamation Mark | 33 | ; | Semicolon | 59 |
| ” | Quotation Mark | 34 | < | Less-than Sign | 60 |
| # | Number Sign | 35 | = | Equals Sign | 61 |
| $ | Dollar Sign | 36 | > | Greater-than Sign | 62 |
| % | Percent Sign | 37 | ? | Question Mark | 63 |
| & | Ampersand | 38 | @ | Commercial At | 64 |
| ' | Apostrophe | 39 | [ | Left Square Bracket | 91 |
| ( | Left Parenthesis | 40 | \ | Reverse Solidus | 92 |
| ) | Right Parenthesis | 41 | ] | Right Square Bracket | 93 |
| * | Asterisk | 42 | ^ | Circumflex Accent | 94 |
| + | Plus Sign | 43 | _ | Low Line | 95 |
| , | Comma | 44 | ` | Grave Accent | 96 |
| - | Hyphen-minus | 45 | { | Left Curly Bracket | 123 |
| . | Full Stop | 46 \| | Vertical Line | 124 |
| / | Solidus | 47 | } | Right Curly Bracket | 125 |
| : | Colon | 58 | ~ | Title | 126 |

Table 2: Punctuations in ASCII plane

#### 4.1.1.2. MobileLLM

MobileLLM’s tokenizer follows SentencePiece normalization, escaping spaces as meta symbols and appending a meta symbol at the beginning of the text. No pre-tokenization is applied.

#### 4.1.1.3. SmolLM2

SmolLM2’s tokenizer does not perform normalization. Its pre-tokenization isolates each digit (0–9), and converts all characters into one or more byte tokens based on UTF-8 encoding.

Table 3 summarizes the normalization and pre-tokenization processes for the three tokenizers:

| | Normalization | Pre-tokenization |
|---|---|---|
| BERT | - Remove control characters<br>- Replace all whitespaces with spaces<br>- Prepend and append a space to Chinese characters | - Segment texts with spaces as delimiter<br>- Isolate each punctuation |
| MobileLLM | - Replace spaces with meta symbol '▁'<br>- Prepend meta symbol '▁' to texts | - |
| SmolLM2 | - | - Isolate each digit<br>- Decompose Unicode characters into UTF-8 byte tokens |

Table 3: Summary of normalization and pre-tokenizations, implemented by the tokenizers of BERT, MobileLLM, and SmolLM2

When training supplement tokenizers, in addition to the unique processes of each tokenizer, two pre-tokenizations were applied universally: 1) isolating each punctuation, and 2) isolating each digit.

### 4.1.2. Training Algorithms

When incorporating vocabularies from 2 tokenizers, it must be guaranteed that the 2 tokenizers share the algorithm with which they are trained. This is because what components tokenizers obtain during training and how the tokens in their vocabularies look like vary according to training algorithms, which makes a supplement vocabulary trained with a different algorithm to be not compatible with an original tokenizer. To illustrate, a token spelled '##able' (which stands for word-medial or word final suffix 'able') from a vocabulary trained with WordPiece algorithm, will not be properly processed by a SentencePiece BPE tokenizer, which expects the form of 'able' as representation of a token with such role.

As mentioned above, BERT, MobileLLM, and SmolLM2's tokenizer adopted WordPiece, SentencePiece BPE, Byte-level BPE as their tokenizations. The 3

tokenizations share the key training method, but their specific algorithms slightly differ. Detailed illustration of the training algorithms are as follows.

#### 4.1.2.1. WordPiece

A WordPiece tokenizer is trained by iteratively searching and merging the pair with best **merge score**, starting from the processed training corpus decomposed into characters, until any of the stopping criteria is satisfied. The merge score is computed as follows:

$$merge\ score\ = \frac{\text{(frequency of the pair)}}{\text{(frequency of the}1_{st}\text{component)} \times \text{(frequency of the}2_{nd}\text{component)}}$$

For each iteration, the score above is computed for all the bigrams, and the pair with the best score is merged and registered as a new token alongside an index, and the corpus is updated by replacing bigrams match the pair with the token. The training procedure finishes, if the frequency of the pair with best merge score is lower than the predefined minimum frequency. Otherwise, the iteration continues until the number of tokens have been learned reaches the preferred vocabulary size.

In advance of the iterations, special tokens and unique initial unigrams are added to the vocabulary, where initial unigrams contain the corpus decomposed into individual characters. The training algorithm can be briefly summarized into the following steps:

1. Normalize, pre-tokenize the corpus

2. Decompose the corpus into initial unigrams (which are characters).
3. Add special tokens and unique initial unigrams to the vocabulary
4. Compute the merge score of bigrams
5. Merge the pair with best merge score into a token and add it to the vocabulary.
6. Replace all bigrams which match the pair with the merged token.
7. Repeat 4-6 until:
    A. Frequency of the best pair $\geq$ Minimum frequency criterion
    B. Current vocabulary size $\leq$ Preferred vocabulary size

It should be noted that, a prefix which identify continuing subwords is required for this procedure. The prefix is prepended to initial unigrams which are not on word-initial position, and removed from the second component of the pair when a merge occurs. For example, with '##' as the prefix (which is adopted by BERT), 'Wo' and '##rd' are merged into 'Word' while '##wo' and '##rd' are merged into 'word'. The prefix may vary depending on the tokenizer, since any other symbol can be set as the prefix if needed.

Including all the features mentioned above, the whole training procedure of WordPiece can be described as Algorithm 1.

**Algorithm 1** Train a WordPiece tokenizer

**Require:**

- $C$ : Training corpus
- $S$ : Special tokens
- $Vsize$ : Preferred vocabulary size
- $freq_{min}$ : Minimum frequency criterion
- $prefix$ : Prefix for continuing subwords

**Ensure:** Trained vocabulary $V$

**procedure** WordPieceTrainer($C, V_{size}, freq_{min}, prefix, S$)
  $C \leftarrow Normalize(C)$ ▷ Nomalization
  $C \leftarrow Pre\text{-}tokenize(C)$ ▷ Pre-tokenization
  $V \leftarrow$ empty mapping
  $i \leftarrow 0$
  **for** $v \in S$ **do**
    $V[v] \leftarrow i$ ▷ Assign indices to special tokens first
    $i \leftarrow i + 1$
  **end for**
  $C_{uni} \leftarrow$ empty array ▷ Initial unigrams from the corpus
  **for** $chunk \in C$ **do** ▷ $chunk$s refer to units after pre-tokenization
    $chunk_{buff} \leftarrow$ empty array
    **for** $k \leftarrow 0$ **to** len($word$) $- 1$ **do** ▷ Collect unigrams in $chunk$
      $char \leftarrow w[k]$
      **if** $k > 0$ **then** ▷ Prepend $prefix$ to continuing characters
        $char \leftarrow prefix + char$
      **end if**
      $chunk_{buff}$.append($char$)
    **end for**
    $C_{uni}$.append($chunk_{buff}$)
  **end for**
  **for** $v \in$ sorted(unique(concatenate($C_{uni}$))) **do**
    $V[v] \leftarrow i$ ▷ Assign indices to unique initial unigrams
    $i \leftarrow i + 1$
  **end for**
  **while** len($V$) $< Vsize$ **do** ▷ Stop if $Vsize$ is satisfied
    $C_{bi} \leftarrow$ empty array ▷ Bigrams obtained from unigrams
    **for** $chunk \in C_{uni}$ **do**
      **for** $k \leftarrow 0$ **to** len($chunk$) $- 2$ **do** ▷ Collect bigrams in $chunk$
        $pair \leftarrow (chunk[k], chunk[k+1])$
        $C_{bi}$.append($pair$)
      **end for**
    **end for**
    $score_{best} \leftarrow 0$
    **for** $pair \in C_{bi}$ **do** ▷ Find the pair with best merge score
      $score \leftarrow \dfrac{C_{bi}.\text{count}(pair)}{C_{bi}.\text{count}(pair[0]) \times C_{bi}.\text{count}(pair[1])}$
      **if** $score > score_{best}$ **then**
        $score_{best} \leftarrow score$
        $pair_{best} \leftarrow pair$
      **end if**
    **end for**

```
        if C_bi.count(pair_best) < freq_min then
            break                    ▷ Stop if the best pair does not pass freq_min
        end if
        v ← join(pair_best[0], pair_best[1].strip(prefix))          ▷ Merged token
        V[v] ← i                                                  ▷ Assign index to the token
        i ← i + 1                                                 ▷ Move on to the next index
        for all chunk ∈ C_uni do                                  ▷ Update chunks in C_uni
            chunk_buff ← empty array
            k ← 0
            while k < len(chunk) do
                if (chunk[k], chunk[k + 1]) = pair_best then
                    chunk_buff.append(v)    ▷ Replace best pairs with the token
                    k ← k + 2
                else
                    chunk_buff.append(chunk[k])          ▷ Do nothing otherwise
                    k ← k + 1
                end if
            end while
            chunk ← chunk_buff
        end for
    end while
    return V                                              ▷ Output the trained vocabulary
end procedure
```

#### 4.1.2.2. SentencePiece BPE

SentencePiece BPE tokenizers are trained by iteratively searching pairs and merging them, similarly to WordPiece. However, the training algorithm computes naïve frequencies rather than merge score, based on the merged form of pairs rather than pairs themselves. The training procedure breaks if any of the stopping criteria is met, which are same to those of WordPiece.

BPE-based tokenizers require trained **merge rules** for tokenization, since they implement tokenization by applying those rules in order, to the initial unigrams of decomposed input text. Merge rules contain pairs of symbols indicating which bigrams to be merged next, and details about how these are applied are on section 4.1.3. Merge rules can easily be obtained, by simply storing the merges occurred

during the training procedure. Since the training algorithm of SentencePiece BPE computes the frequency of merged forms of pairs, 2 or more pairs which build up the merged form could be learned as merge rules from 1 iteration. The following steps summarizes the training algorithm:

1. Normalize, pre-tokenize the corpus
2. Decompose the corpus into initial unigrams (which are characters).
3. Add special tokens and unique initial unigrams to the vocabulary
4. Compute the frequency of merged forms of bigrams
5. Add the merged form with highest frequency to the vocabulary.
6. Add all bigrams which build up the merged token to the merge rules
7. Replace all bigrams which build up the merged token with the merged token.
8. Repeat 4-7 until:
    A. Frequency of the merged token $\geq$ Minimum frequency criterion
    B. Current vocabulary size $\leq$ Preferred vocabulary size

Unlike WordPiece which adopts prefix for continuing subwords, SentencePiece BPE does not prepend any symbol to tokens on step 2. The training procedure of a SentencePiece BPE tokenizer can be illustrated as Algorithm 2.

**Algorithm 2** Train a SentencePiece BPE tokenizer

**Require:**

- $C$ : Training corpus
- $S$ : Special tokens
- $B$ : Byte tokens in order, for Byte-Fallback method
- $Vsize$ : Preferred vocabulary size
- $freq_{min}$ : Minimum frequency

**Ensure:** Trained vocabulary $V$, merge rules $M$

**procedure** SentencePieceBPETrainer($C, V_{size}, freq_{min}, S, B$)
  $C \leftarrow Normalize(C)$ ▷ Nomalization
  $C \leftarrow Pre\text{-}tokenize(C)$ ▷ Pre-tokenization
  $V \leftarrow$ empty mapping
  $M \leftarrow$ empty array
  $i \leftarrow 0$
  **for** $v \in S$ **do**
    $V[v] \leftarrow i$ ▷ Assign indices to special tokens first
    $i \leftarrow i + 1$
  **end for**
  **for** $v \in B$ **do**
    $V[v] \leftarrow i$ ▷ Assign indices to byte tokens next
    $i \leftarrow i + 1$
  **end for**
  $C_{uni} \leftarrow$ empty array ▷ Initial unigrams from the corpus
  **for** $chunk \in C$ **do** ▷ *chunk*s refer to units after pre-tokenization
    $chunk_{buff} \leftarrow$ empty array
    **for** $k \leftarrow 0$ **to** len($word$) $- 1$ **do** ▷ Collect unigrams in *chunk*
      $char \leftarrow w[k]$
      $chunk_{buff}$.append($char$)
    **end for**
    $C_{uni}$.append($chunk_{buff}$)
  **end for**
  **for** $v \in$ sorted(unique(concatenate($C_{uni}$))) **do**
    $V[v] \leftarrow i$ ▷ Assign indices to unique initial unigrams
    $i \leftarrow i + 1$
  **end for**
  **while** len($V$) $< Vsize$ **do** ▷ Stop if $Vsize$ is satisfied
    $C_{bi} \leftarrow$ empty array ▷ Bigrams obtained from unigrams
    **for** $chunk \in C_{uni}$ **do**
      **for** $k \leftarrow 0$ **to** len($chunk$) $- 2$ **do** ▷ Collect bigrams in *chunk*
        $pair \leftarrow chunk[k] + chunk[k + 1]$
        $C_{bi}$.append($pair$)
      **end for**
    **end for**
    $freq_{best} \leftarrow 0$
    **for** $pair \in C_{bi}$ **do** ▷ Find the pair with highest frequency
      $freq \leftarrow C_{bi}$.count($pair$)
      **if** $freq > freq_{best}$ **then**
        $freq_{best} \leftarrow freq$
        $pair_{best} \leftarrow pair$
      **end if**

```
        end for
        if freq < freq_min then
            break                    ▷ Stop if the best pair does not pass freq_min
        end if
        v ← pair_best                                              ▷ Merged token
        V[v] ← i                                          ▷ Assign index to the token
        i ← i + 1                                         ▷ Move on to the next index
        for all chunk ∈ C_uni do                          ▷ Update chunks in C_uni
            chunk_buff ← empty array
            k ← 0
            while k < len(chunk) do
                if chunk[k] + chunk[k + 1] = pair_best then
                    chunk_buff.append(v)     ▷ Replace best pairs with the token
                    merge ← (chunk[k], chunk[k + 1])
                    if merge ∉ M then                ▷ Append unique merge rules
                        M.append(merge)
                    end if
                    k ← k + 2
                else
                    chunk_buff.append(chunk[k])          ▷ Do nothing otherwise
                    k ← k + 1
                end if
            end while
            chunk ← chunk_buff
        end for
    end while
    return V, M                ▷ Output the trained vocabulary and merge rules
end procedure
```

#### 4.1.2.3. Byte-level BPE

Just as SentencePiece BPE, Byte-level BPE tokenizers are trained by iteratively searching pairs with highest frequency and merging them. The only part where training algorithm of Byte-level BPE is different from SentencePiece BPE is that it computes the frequency of pairs before being merged. Since the pair with highest frequency is selected, only one merge rule is obtained per 1 iteration. Considering the distinction above, the training algorithm of Byte-level BPE can be summarized as follows:

1. Normalize, pre-tokenize the corpus
2. Decompose the corpus into initial unigrams (which are characters).
3. Add special tokens and unique initial unigrams to the vocabulary
4. Compute the frequency of bigrams
5. Merge the pair with highest frequency into a token and add it to the vocabulary.
6. Add the pair with highest frequency to the merge rules
7. Replace all bigrams which match the pair with the merged token.
8. Repeat 4-7 until:
    A. Frequency of the merged token $\geq$ Minimum frequency criterion
    B. Current vocabulary size $\leq$ Preferred vocabulary size

Byte-level BPE does not prepend any prefix to token on step 2, same as SentencePiece BPE. Algorithm 3 illustrates the training procedure of a Byte-level BPE tokenizer.

**Algorithm 3** Train a Byte-level BPE tokenizer

**Require:**

- $C$ : Training corpus
- $S$ : Special tokens
- $B$ : Byte tokens for UTF-8 representation of characters
- $Vsize$ : Preferred vocabulary size
- $freq_{min}$ : Minimum frequency

**Ensure:** Trained vocabulary $V$, merge rules $M$

**procedure** ByteLevelBPETrainer($C, V_{size}, freq_{min}, S, B$)
  $C \leftarrow Normalize(C)$ ▷ Nomalization
  $C \leftarrow Pre\text{-}tokenize(C)$ ▷ Pre-tokenization
  $V \leftarrow$ empty mapping
  $M \leftarrow$ empty array
  $i \leftarrow 0$
  **for** $v \in S$ **do**
    $V[v] \leftarrow i$ ▷ Assign indices to special tokens first
    $i \leftarrow i + 1$
  **end for**
  **for** $v \in B$ **do**
    $V[v] \leftarrow i$ ▷ Assign indices to byte tokens next
    $i \leftarrow i + 1$
  **end for**
  $C_{uni} \leftarrow$ empty array ▷ Initial unigrams from the corpus
  **for** $chunk \in C$ **do** ▷ *chunk*s refer to units after pre-tokenization
    $chunk_{buff} \leftarrow$ empty array
    **for** $k \leftarrow 0$ **to** len($word$) $- 1$ **do** ▷ Collect unigrams in *chunk*
      $char \leftarrow w[k]$
      $chunk_{buff}$.append($char$)
    **end for**
    $C_{uni}$.append($chunk_{buff}$)
  **end for**
  **for** $v \in$ sorted(unique(concatenate($C_{uni}$))) **do**
    $V[v] \leftarrow i$ ▷ Assign indices to unique initial unigrams
    $i \leftarrow i + 1$
  **end for**
  **while** len($V$) $< Vsize$ **do** ▷ Stop if $Vsize$ is satisfied
    $C_{bi} \leftarrow$ empty array ▷ Bigrams obtained from unigrams
    **for** $chunk \in C_{uni}$ **do**
      **for** $k \leftarrow 0$ **to** len($chunk$) $- 2$ **do** ▷ Collect bigrams in *chunk*
        $pair \leftarrow (chunk[k], chunk[k + 1])$
        $C_{bi}$.append($pair$)
      **end for**
    **end for**
    $freq_{best} \leftarrow 0$
    **for** $pair \in C_{bi}$ **do** ▷ Find the pair with highest frequency
      $freq \leftarrow C_{bi}$.count($pair$)
      **if** $freq > freq_{best}$ **then**
        $freq_{best} \leftarrow freq$
        $pair_{best} \leftarrow pair$
      **end if**

```
        end for
        if freq < freq_min then
            break                    ▷ Stop if the best pair does not pass freq_min
        end if
        v ← join(pair_best)                                          ▷ Merged token
        V[v] ← i                                                ▷ Assign index to the token
        merge ← pair_best
        M.append(merge)                                          ▷ Append the merge rule
        i ← i + 1                                               ▷ Move on to the next index
        for all chunk ∈ C_uni do                                 ▷ Update chunks in C_uni
            chunk_buff ← empty array
            k ← 0
            while k < len(chunk) do
                if (chunk[k], chunk[k + 1]) = pair_best then
                    chunk_buff.append(v)    ▷ Replace best pairs with the token
                    k ← k + 2
                else
                    chunk_buff.append(chunk[k])          ▷ Do nothing otherwise
                    k ← k + 1
                end if
            end while
            chunk ← chunk_buff
        end for
    end while
    return V, M              ▷ Output the trained vocabulary and merge rules
end procedure
```

### 4.1.3. Tokenization Algorithms

As well as the training algorithms, tokenization algorithms should be considered when integrating 2 tokenizers' vocabularies and merge rules. This is because an extended tokenizer may fail to capture new tokens from the input text, if the tokenization algorithms of the two tokenizers are inconsistent. The details of the tokenization algorithms of WordPiece, SentencePiece BPE and Byte-level BPE are as follows.

#### 4.1.3.1. WordPiece

The tokenization algorithm of WordPiece captures the longest token from each pre-tokenized unit, by repeatedly trying to match the left part of the unit with tokens in the vocabulary. This key process can be described as below:

1. Put the start and end offset on the leftmost and rightmost characters of the unit.
2. Find a token which matches with the current window
3. If there is no token matching, reduce the end offset by 1
4. If there is a token matching, append the token's index to the output array, replace the start offset with current end offset, and reset the end offset to the end of the unit.
5. Repeat 2-4, until there is no character remaining in the unit.

When the start offset is not on the initial character of the unit, or in other words, a token representing a continuing subword should be captured, the algorithm automatically prepends the predefined prefix which is mentioned on section 4.1.2.1. In addition to the algorithm above, WordPiece tokenization returns the unknown token, regrading a unit as OOV(Out of Vocabulary), on cases below:

- The unit is longer than the predefined maximum length of each unit.
- A part of unit cannot be capture by any of the tokens in the vocabulary

The tokenization algorithm of WordPiece therefore requires the unknown token, unlike other BPE-based tokenizations. Algorithm 4 illustrates the tokenization algorithm of WordPiece.

**Algorithm 4** WordPiece Tokenization

**Require:**

- $V$ : Vocabulary
- $prefix$ : Prefix for continuing subwords
- $chunk_{max}$ : Maximum length of words
- $unk$ : Unknown token
- $text$ : Input text

**Ensure:** Token index array $ids$

**procedure** WordPieceTokenizer($V, prefix, chunk_{max}, unk, text$)
  $text \leftarrow Normalize(text)$ ▷ Normalization
  $text \leftarrow Pre\text{-}tokenize(text)$ ▷ Pre-tokenization
  $ids \leftarrow$ empty array ▷ Array of indices to be returned
  **for** $chunk \in text$ **do** ▷ $chunk$s refer to units after pre-tokenization
    **if** len($chunk$) $> chunk_{max}$ **then**
      $ids$.append($V[unk]$) ▷ Too long chunks are treated as OOV
    **else**
      $start \leftarrow 0$
      $ids_{buff} \leftarrow$ empty array
      **while** $start <$ len($chunk$) **do** ▷ Find the longest matching token
        $end \leftarrow$ len($chunk$)
        $token \leftarrow unk$ ▷ $token$ remains to $unk$ if no match is found
        **while** $start < end$ **do** ▷ Try until a matching token is found
          $token_{buff} \leftarrow chunk[start : end]$
          **if** $start > 0$ **then** ▷ Add $prefix$ to continuing subwords
            $token_{buff} \leftarrow prefix + token_{buff}$
          **end if**
          **if** $token_{buff} \in V$ **then** ▷ Stop if a matching token is found
            $token \leftarrow token_{buff}$
            **break**
          **end if**
          $end \leftarrow end - 1$ ▷ Try shorter one if there is no match
        **end while**
        **if** $token = unk$ **then** ▷ Stop if any $unk$ occurred
          **break**
        **end if**
        $ids_{buff}$.append($V[token]$) ▷ Append the matched token's index
      **end while**
      **if** $token = unk$ **then**
        $ids_{buff} \leftarrow [V[unk]]$ ▷ $chunk \leftarrow unk$ if any $unk$ occurred
      **end if**
      $ids \leftarrow$ concatenate($ids, ids_{buff}$) ▷ Include the indices in the chunk
    **end if**
  **end for**
  **return** $ids$
**end procedure**

#### 4.1.3.2. SentencePiece BPE

Since the procedure of obtaining tokens by merging bigrams is stored as merge rules, a SentencePiece BPE tokenizer only has to apply the merge rules to the input text for tokenization. The pre-tokenized units are decomposed into characters at first, and bigrams matching with each merge rule are replaced with the merged token. After all the rules are applied, tokens are captured from each unit and converted into indices.

As mentioned, SentencePiece BPE utilizes **Byte-fallback** method to process unknown tokens, which disassemble the tokens into characters and convert each character into UTF-8 bytes. Thanks to the method, a SentencePiece BPE tokenizer does not require an unknown token in its vocabulary for tokenization, whose tokenization algorithm can be described as Algorithm 5.

**Algorithm 5** SentencePiece BPE Tokenization

**Require:**

- $V$ : Vocabulary
- $M$ : Merge rules
- $B$ : Byte tokens in order, for Byte-Fallback method
- $text$ : Input text

**Ensure:** Token index array $ids$

**procedure** SentencePieceBPETokenizer($V, M, B, text$)
  $text \leftarrow Normalize(text)$ ▷ Normalization
  $text \leftarrow Pre\text{-}tokenize(text)$ ▷ Pre-tokenization
  $ids \leftarrow$ empty array ▷ Array of indices to be returned
  **for** $chunk \in text$ **do** ▷ $chunk$s refer to units after pre-tokenization
    $ids_{buff} \leftarrow$ empty array
    $chunk_{buff} \leftarrow$ empty array
    **for** $k \leftarrow 0$ **to** len($chunk$) $- 1$ **do** ▷ Decompose $chunk$s into characters
      $char \leftarrow chunk[k]$
      $chunk_{buff}$.append($char$)
    **end for**
    $chunk \leftarrow chunk_{buff}$
    **for** $m \in M$ **do** ▷ Update $chunk$s based on merge rules
      $k \leftarrow 0$
      $chunk_{buff} \leftarrow$ empty array
      **while** $k <$ len($chunk$) **do**
        **if** $(chunk[k], chunk[k+1]) = m$ **then**
          $token \leftarrow chunk[k] + chunk[k+1]$
          $chunk_{buff}$.append($token$)
          $k \leftarrow k + 2$
        **else**
          $token \leftarrow chunk[k]$
          $chunk_{buff}$.append($token$)
          $k \leftarrow k + 1$
        **end if**
      **end while**
      $chunk \leftarrow chunk_{buff}$
    **end for**
    **for** $token \in chunk$ **do** ▷ Convert captured tokens into indices
      **if** $token \notin V$ **then** ▷ Apply Byte-fallback to unknown tokens
        **for** $k \leftarrow 0$ **to** len($chunk$) $- 1$ **do**
          $char \leftarrow token[k]$
          $bytes \leftarrow$ UTF-8(char) ▷ Convert characters to UTF-8 bytes
          **for** $b \in bytes$ **do**
            $token_{byte} \leftarrow B[b]$ ▷ Convert bytes to byte tokens
            $ids_{buff}$.append($V[token_{byte}]$)
          **end for**
        **end for**
      **else**
        $ids_{buff}$.append($V[token]$)
      **end if**
    **end for**
    $ids \leftarrow$ concatenate($ids, ids_{buff}$) ▷ Include the indices in the chunk

```
    end for
    return ids
end procedure
```

#### 4.1.3.3. Byte-level BPE

Although the number of merge rules learned from a merge is different, Byte-level BPE tokenizers applies the merge rules for tokenization, similarly to SentencePiece BPE tokenizers. Tokens are captured after applying all the rules to decomposed units, and OOV(Out of Vocabulary) does not occur due Byte-level BPE tokenization's immunity. The tokenization algorithm of Byte-level BPE can be illustrated as Algorithm 6.

**Algorithm 6** Byte-level BPE Tokenization

**Require:**

- $V$ : Vocabulary
- $M$ : Merge rules
- $text$ : Input text

**Ensure:** Token index array $ids$

**procedure** ByteLevelBPETokenizer($V, M, text$)
  $text \leftarrow Normalize(text)$ ▷ Normalization
  $text \leftarrow Pre\text{-}tokenize(text)$ ▷ Pre-tokenization
  $ids \leftarrow$ empty array ▷ Array of indices to be returned
  **for** $chunk \in text$ **do** ▷ $chunk$s refer to units after pre-tokenization
    $ids_{buff} \leftarrow$ empty array
    $chunk_{buff} \leftarrow$ empty array
    **for** $k \leftarrow 0$ **to** len($chunk$) $- 1$ **do** ▷ Decompose $chunk$s into bytes
      $char \leftarrow chunk[k]$
      $chunk_{buff}$.append($char$)
    **end for**
    $chunk \leftarrow chunk_{buff}$
    **for** $m \in M$ **do** ▷ Update $chunk$s based on merge rules
      $k \leftarrow 0$
      $chunk_{buff} \leftarrow$ empty array
      **while** $k <$ len($chunk$) **do**
        **if** $(chunk[k], chunk[k+1]) = m$ **then**
          $token \leftarrow chunk[k] + chunk[k+1]$
          $chunk_{buff}$.append($token$)
          $k \leftarrow k + 2$
        **else**
          $token \leftarrow chunk[k]$
          $chunk_{buff}$.append($token$)
          $k \leftarrow k + 1$
        **end if**
      **end while**
      $chunk \leftarrow chunk_{buff}$
    **end for**
    **for** $token \in chunk$ **do** ▷ Convert captured tokens into indices
      $ids_{buff}$.append($V[token]$)
    **end for**
    $ids \leftarrow$ concatenate($ids, ids_{buff}$) ▷ Include the indices in the chunk
  **end for**
  **return** $ids$
**end procedure**

## 4.2. Training Supplement Tokenizers

As mentioned above, BERT, MobileLLM, and SmolLM2 each utilize different methods, resulting in distinct token structures within their vocabularies. Therefore, separate supplement tokenizers were trained for each tokenization type.

### 4.2.1. Training Corpus

Half of the data used for continual pretraining was selected as the training corpus for the supplement tokenizers. Though it is a standard practice to train a tokenizer using the same corpus as the model, the supplement tokenizers were trained using five different minimum frequency thresholds to examine the effect of the number of tokens extended. The vocabulary size of each supplement tokenizer trained with different thresholds is summarized in Table 4:

| Minimum Frequency | WordPiece | SentencePiece BPE | Byte-level BPE |
|---|---|---|---|
| 1,000 | 80,624 | 83,655 | 78,216 |
| 5,000 | 40,552 | 38,058 | 32,939 |
| 10,000 | 30,807 | 27,458 | 22,327 |
| 50,000 | 18,093 | 13,995 | 8,839 |
| 100,000 | 15,131 | 10,926 | 5,763 |

Table 4: Summary of vocabulary sizes of dummy tokenizers, trained with a half of the model training corpus

During this process, alongside medical terms, non-medical tokens which build the context around the medical terms were naturally included in the vocabulary. This tendency was more obvious on supplement tokenizers trained with high minimum frequency criteria, since medical terms cannot build the context alone and they only

appear in special contexts. In other words, context words were mainly learned as tokens on minimum frequency of 100,000, while medical terms were also learned as tokens on minimum frequency of 1,000.

To evaluate the direct impact of extending tokenizers with only medical terms, additional supplement tokenizers were trained using a list of medical terms as the training corpus. This list was created by crawling headwords from the Merriam-Webster Medical Dictionary, applying the same pre-tokenization shared by supplement tokenizers, and deduplicating the entries. As result, 49,206 medical terms were collected.

Among the collected medical terms, some of terms contained overlapping parts, for instance:

- **hepa**-: **hepa**tic, **hepa**tis, **hepa**titis, **hepa**tectomy, **hepa**ticotomy & etc.
- **jejun**-: **jejun**a, **jejun**itis, **jejun**ogastric, **jejun**oileal, **jejun**ostomy & etc.
- **pelv**-: **pelv**is, **pelv**igraph, **pelv**imeter, **pelv**iscope, **pelv**iolithotomy & etc.

These subwords turned out to be prefixes from Latin language related to certain parts of human body, which are:

- hepa-: a prefix which indicates something related to liver
- jejun-: a prefix which indicates something related to a part of intestine
- pelv-: a prefix which indicates something related to pelvic bone

Likewise, a number of subwords which seem to be derived from Latin were found in the list. Regarding this, to control how subwords were captured, supplement

tokenizers were also trained using four different minimum frequency thresholds. All unique terms were registered as tokens when the minimum frequency was set to 1, while only highly frequent subwords were learned when the minimum frequency was set to 10. The resulting vocabulary sizes are shown in Table 5:

| Minimum Frequency | WordPiece | SentencePiece BPE | Byte-level BPE |
|---|---|---|---|
| 1 | 72,932 | 72,415 | 72,549 |
| 2 | 17,137 | 16,903 | 17,010 |
| 5 | 7,167 | 7,078 | 7,239 |
| 10 | 3,782 | 3,694 | 3,852 |

Table 5: Summary of vocabulary sizes of dummy tokenizers, trained with the list of medical terms crawled from Merriam-Webster Medical Dictionary

## 4.3. Extending the Original Vocabulary and Merge Rules

Non-overlapping tokens and merge rules from the supplement tokenizers were added to the original vocabularies, reflecting the algorithms of WordPiece, SentencePiece BPE, and Byte-level BPE.

### 4.3.1. WordPiece

WordPiece does not use merge rules during tokenization, as it captures the longest possible token from the start of the text. The supplement tokenizer for WordPiece does not learn merge rules either, thus the extension of BERT's tokenizer with WordPiece supplement tokenizer reduces to a simple procedure: appending non-overlapping tokens to the original vocabulary in index order. The extension BERT's vocabulary using the WordPiece supplement tokenizer can be expressed as

Algorithm 7:

**Algorithm 7** Extend BERT tokenizer with WordPiece dummy tokenizer

**Require:**

- $V_D$ : Dummy tokenizer vocabulary
- $V_N$ : New vocabulary(copy of original vocabulary)

$i \leftarrow \text{len}(V_N)$
**for all** $v \in V_D$ **do** ▷ Non-overlapping tokens
  **if** $v \notin V_N$ **then**
    $V_N[v] \leftarrow i$ ▷ Assign index to the token
    $i \leftarrow i + 1$ ▷ Move on to the next index
  **end if**
**end for**

### 4.3.2. SentencePiece BPE

SentencePiece BPE uses merge rules during both training and tokenization. Unlike traditional BPE, SentencePiece BPE incorporates all possible combinations of tokens into the merge rules. Since single-character tokens are registered without merge rules, they must be added to the vocabulary first to correctly derive subsequent merge rules. The process for extending MobileLLM's vocabulary using the SentencePiece BPE supplement tokenizer can be expressed as Algorithm 8:

**Algorithm 8** Extend MobileLLM2 tokenizer with SentencePiece BPE dummy tokenizer

**Require:**

- $V_D$ : Dummy tokenizer vocabulary
- $V_N$ : New vocabulary(copy of original vocabulary)
- $M_D$ : Dummy tokenizer merge rules
- $M_N$ : New merge rules(copy of original merge rules)

$i \leftarrow \text{len}(V_N)$
**for all** $v \in V_D$ **do** ▷ Non-overlapping, single-character tokens
  **if** $\text{len}(v) = 1 \wedge v \notin V_N$ **then**
    $V_N[v] \leftarrow i$ ▷ Assign index to the token
    $i \leftarrow i + 1$ ▷ Move on to the next index
  **end if**
**end for**
**for all** $m \in M_D$ **do** ▷ Non-overlapping, merged tokens
  $v \leftarrow \text{join}(m)$
  **if** $v \notin V_N$ **then**
    **for** $k \leftarrow 1$ **to** $\text{len}(v) - 1$ **do** ▷ Find all possible merges
      **if** $v[:k] \in V_N \wedge v[k:] \in V_N$ **then**
        $M_N$.append$(m)$ ▷ Append the merge rules
      **end if**
    **end for**
    $V_N[v] \leftarrow i$ ▷ Assign index to the token
    $i \leftarrow i + 1$ ▷ Move on to the next index
  **end if**
**end for**

### 4.3.3. Byte-level BPE

Byte-level BPE learns one merge rule per token during training, incorporating the highest-frequency pair. Single-character tokens should be added to the vocabulary first just as SentencePiece BPE, followed by tokens derived from merge rules. The process for extending SmolLM2's vocabulary using the Byte-level BPE supplement tokenizer can be expressed as Algorithm 9:

**Algorithm 9** Extend SmolLM2 tokenizer with Byte-level BPE dummy tokenizer

**Require:**

- $V_D$ : Dummy tokenizer vocabulary
- $V_N$ : New vocabulary(copy of original vocabulary)
- $M_D$ : Dummy tokenizer merge rules
- $M_N$ : New merge rules(copy of original merge rules)

$i \leftarrow \text{len}(V_N)$
**for all** $v \in V_D$ **do** ▷ Non-overlapping, single-character tokens
  **if** $\text{len}(v) = 1 \wedge v \notin V_N$ **then**
    $V_N[v] \leftarrow i$ ▷ Assign index to the token
    $i \leftarrow i + 1$ ▷ Move on to the next index
  **end if**
**end for**
**for all** $m \in M_D$ **do** ▷ Non-overlapping, merged tokens
  $v \leftarrow \text{join}(m)$
  **if** $v \notin V_N$ **then**
    $M_N$.append$(m)$ ▷ Append the merge rule
    $V_N[v] \leftarrow i$ ▷ Assign index to the token
    $i \leftarrow i + 1$ ▷ Move on to the next index
  **end if**
**end for**

Table 6 summarizes the number of tokens newly added to the vocabulary, through the extension algorithms above. The tokenizers extended using supplement tokenizers trained with continual pretraining data are indicated by the prefix ‘Auto’, while the tokenizers extended using supplement tokenizers trained with medical terms are indicated by the prefix ‘Crawled’. The numeric suffixes such as ‘5’ or ‘10,000’ indicate the minimum frequency criteria applied when training supplement tokenizers. For instance, supplement tokenizers trained with medical terms with minimum frequency criterion of 2 are indicated by ‘Crawled-2’

| Model | BERT | MobileLLM | SmolLM2 |
|---|---|---|---|
| **Auto-1000** | 91,705 (+ 62,709) | 98,413 (+ 66,413) | 93,884 (+ 44,732) |
| **Auto-5000** | 56,887 (+27,891) | 57,055 (+ 25, 055) | 60,542 (+ 11,390) |
| **Auto-10000** | 49,247 (+20,251) | 48,369 (+ 16,369) | 54,753 (+ 5,601) |
| **Auto-50000** | 40,954 (+11,958) | 38,999 (+ 6,999) | 50,097 (+ 945) |
| **Auto-100000** | 39,439 (+ 10,443) | 37,411 (+5,411) | 49,644 (+492) |
| **Crawled-1** | 94,776 (+65,780) | 97,829 (+ 65,829) | 109,122 (+59,970) |
| **Crawled-2** | 42,934 (+ 13,938) | 44,684 (+12,684) | 59,984 (10,832) |
| **Crawled-5** | 34,167 (+ 5,171) | 36,397 (+ 4,397) | 52,888 (+3,436) |
| **Crawled-10** | 31,351 (+2,355) | 33,828 (+ 1,828) | 50,396 (+ 1,244) |
| **Original** | 28,996 | 32,000 | 49,152 |

Table 6: The number of tokens extended

# 5. Experiments

## 5.1. Training Tasks

To validate the effectiveness of tokenizer extension, the models with extended tokenizer, as well as the base model with not extended one, were trained in two stages: 1) **Continual Pretraining** and 2) **Multiple Choice Fine-Tuning**. The following sections provide details of the training process.

### 5.1.1. Continual Pretraining

The models were pretrained using the following methods:

- **BERT**'s variations were trained with *masked language modeling*.
- **MobileLLM** and **SmolLM2**'s variations were trained with *causal language modeling*.

For BERT and SmolLM2, the pretrained weights of their language modeling heads were publicly available, so these were loaded and further trained. For MobileLLM, since embedding sharing was used, the language modeling head was not trained separately.

The training data consisted of medical texts collected from **PMC** and **PubMed**. As PMC provides full article data, there was a concern that only the beginning

sections (e.g., abstracts, introductions) of articles might be overly utilized due to the limited context length. To address this, the articles were divided into sections and sampled accordingly. A total of 3.8 billion words were collected, with an average word count per sample of 444.24 words.

### 5.1.2. Multiple Choice Fine-tuning

To train the models to be capable of answering 4-option multiple choice questions, the **MedMCQA train split** (182,822 questions) was formatted using a consistent template for fine-tuning.

- **MobileLLM** and **SmolLM2** were fine-tuned using causally formatted data, including explanations, via *causal language modeling*.
- **BERT** was fine-tuned using data formatted only up to the options and paired with labels, via *text classification*.

Examples of MedMCQA questions formatted as templates for each case were provided on Table 7.

| | |
|---|---|
| **Options Only** | **Instruction**<br>The following is a question about medical knowledge. Please choose the right answer.<br><br>**Question**<br>Which vitamin is supplied from only animal source:<br><br>**Options**<br>A) Vitamin C<br>B) Vitamin B7<br>C) Vitamin B12<br>D) Vitamin D |
| **Explanation Added** | **Instruction**<br>The following is a question about medical knowledge. Please choose the right answer.<br><br>**Question**<br>Which vitamin is supplied from only animal source:<br><br>**Options**<br>A) Vitamin C<br>B) Vitamin B7<br>C) Vitamin B12<br>D) Vitamin D<br><br>**Answer**<br>C) Vitamin B12. Ans. (c) Vitamin B12 Ref: Harrison's 19th ed. P 640* Vitamin B12 (Cobalamin) is synthesized solely by microorganisms.* In humans, the only source for humans is food of animal origin, e.g., meat, fish, and dairy products.* Vegetables, fruits, and other foods of nonanimal origin doesn't contain Vitamin B12 .* Daily requirements of vitamin Bp is about 1-3 pg. Body stores are of the order of 2-3 mg, sufficient for 3-4 years if supplies are completely cut off. |

Table 7: An example of formatted MedMCQA Question

### 5.1.3. Module Extensions

With the extension of the tokenizer, the modules of the models with extended tokenizer associated with the vocabulary size—such as the embedding lookup table and the language modeling heads—were also extended and initialized randomly.

- For the language modeling heads, additional parameters were initialized randomly based on predefined methods for each model.
- New embeddings were initialized using a multivariate normal distribution which shared the mean and covariance of the original embeddings.

- Since BERT was not pretrained for text classification, a linear classifier head was added during the multiple choice fine-tuning stage, and it was initialized in the same way as the language modeling heads.

A summary of the initialization methods for the modules in each model was provided on Table 8.

| Model | BERT | MobileLLM | SmolLM2 |
|---|---|---|---|
| **Distribution** | Normal | Normal | Normal |
| **Mean** | 0.0 | 0.0 | 0.0 |
| **Std.** | 1/50 | 1/50 | 1/24 |
| **Embeddings** | Multivariate normal distribution with mean and covariance of original embeddings | | |

Table 8: Predefined random initialization methods of extended modules

The number of additional parameters introduced by the module extensions, as well as the total number of parameters in each model after these changes, were detailed on Table 9.

| Model | BERT Base Cased Masked LM | BERT Base Cased Text Classification | MobileLLM 125M Causal LM | SmolLM2 135M Causal LM |
|---|---|---|---|---|
| **Auto-1000** | 156,564,025 (+ 48,223,221) | 156,472,322 (+ 48,160,512) | 162,889,344 (+ 38,253,888) | 160,280,640 (+ 25,765,632) |
| **Auto-5000** | 129,788,983 (+ 21,448,179) | 129,732,098 (+ 21,420,288) | 139,067,136 (+ 14,431,680) | 141,075,648 (+ 6,560,640) |
| **Auto-10000** | 123,913,823 (+ 15,573,019) | 123,864,578 (+ 15,552,768) | 134,064,000 (+ 9,428,544) | 137,741,184 (+ 3,226,176) |
| **Auto-50000** | 117,536,506 (+ 9,195,702) | 117,495,554 (+ 9,183,744) | 128,666,880 (+ 4,031,424) | 135,059,328 (+ 544,320) |
| **Auto-100000** | 116,371,471 (+ 8,030,667) | 116,332,034 (+ 8,020,224) | 127,752,192 (+ 3,116,736) | 134,798,400 (+ 283,392) |
| **Crawled-1** | 158,925,624 (+ 50,584,820) | 158,830,850 (+ 50,519,040) | 162,552,960 (+ 37,917,504) | 169,057,728 (+ 34,542,720) |
| **Crawled-2** | 119,059,126 (+ 10,718,322) | 119,016,194 (+ 10,704,384) | 131,941,440 (+ 7,305,984) | 140,754,240 (+ 6,239,232) |
| **Crawled-5** | 112,317,303 (+ 3,976,499) | 112,283,138 (+ 3,971,328) | 127,168,128 (+ 2,532,672) | 136,494,144 (+ 1,979,136) |
| **Crawled-10** | 110,151,799 (+ 1,810,995) | 110,120,450 (+ 1,808,640) | 125,688,384 (+ 1,052,928) | 135,231,552 (+ 716,544) |
| **Baseline** | 108,340,804 | 108,311,810 | 124,635,456 | 134,515,008 |

Table 9: Summary of number of parameters newly added to models

## 5.2. Training Configurations

### 5.2.1. Hyperparameters

The hyperparameters used during training are as follows:

- **Training Epochs**
  - Continual Pretraining : 1
  - Multiple Choice Fine-tuning : 5
- **Batch Size** : 512
- **Maximum Input Length**
  - BERT : 512
  - MobileLLM : 2,048
  - SmolLM2 : 2,048
- **Random Seed** : 42

Sequences exceeding the maximum input length were truncated, and all samples were padded to the length of the longest sequence within each batch. Although SmolLM2 is capable of handling up to 8,192 tokens at once, the maximum input length was set to 2,048 to align with MobileLLM, considering the distribution of the number of words in the training data.

### 5.2.2. Optimization & Learning Rate Scheduling

AdamW was used as the optimizer with the following configurations:

- **Weight decay** : 0.0

- **Adam $\beta_1$** : 0.9
- **Adam $\beta_2$** : 0.999
- **Adam $\epsilon$** : $1e-8$

Learning rate scheduling was set as follows:

- 1% of total steps as warmup steps (increasing from 0 to the peak)
- Peak learning rate determined through the learning rate search
- Cosine decay from the peak learning rate to 1/100

To identify the optimal peak learning rate, experiments were conducted with the following values: $5 \times 10^{-3}$, $1 \times 10^{-3}$, $5 \times 10^{-4}$, $1 \times 10^{-4}$, $5 \times 10^{-5}$, $1 \times 10^{-5}$, and $5 \times 10^{-6}$. The base models (with unexpanded tokenizer) were trained for 100 steps, using same learning rate scheduling with each learning rate as the peak value, and the losses were computed on 512 unseen samples (1 batch) from training data. The results of this search are summarized on Table 10.

| Model / Task | Continual Pretraining | Multiple Choice Fine-tuning |
|---|---|---|
| BERT Base Cased | $5 \times 10^{-4}$ | $1 \times 10^{-4}$ |
| MobileLLM 125M | $5 \times 10^{-4}$ | $5 \times 10^{-4}$ |
| SmolLM2 135M | $1 \times 10^{-3}$ | $1 \times 10^{-3}$ |

Table 10: Summary of learning rates selected by learning rate search

### 5.2.3. Device Settings

All training processes were conducted on a single NVIDIA A100 80GB GPU. To

accommodate limited RAM, gradients were accumulated before being passed to the optimizer, ensuring the training batch size (model input batch size × gradient accumulation steps) consistently to be 512. Brain Float 16 (bfloat16) precision was adopted, for all computations performed throughout the entire training as well as the representation of the model.

# 6. Analyses

## 6.1. Evaluation on MultiMedQA Subset

After completing the training process, the models were evaluated on three subsets of MultiMedQA: MedQA, MedMCQA, and MMLU clinical topics. For all benchmarks except MedMCQA, the test splits were used for evaluation. Since MedMCQA's test split is publicly available but not labeled, its validation split was used instead. For measuring performances, **macro accuracy** was used. The number of questions in the subjects are as follows:

- MedQA: 1,273
- MedMCQA: 4,183
- MMLU Anatomy: 135
- MMLU Clinical Knowledge: 265
- MMLU College Biology: 144
- MMLU College Medicine: 173
- MMLU Medical Genetics: 100
- MMLU Professional Medicine: 272
- Total 6,545

Tables 12, 13, and 14 present the evaluation results for variations of **BERT**, **MobileLLM**, and **SmolLM2**, respectively. In these tables:

- **Blue blocks** indicate the performance of the base models.
- **Orange blocks** represent the highest performance in specific subjects.

- **Green blocks** highlight cases where models with extended tokenizers outperformed their base models.
- **Purple blocks** in Tables 12 and 13 represent the performance of untrained MobileLLM and SmolLM2 models.

The following tendencies are observed from the results:

- Variations of **BERT** trained with extended tokenizers tended to perform worse than the base model in certain subjects, while they usually outperformed the base model in the other subjects. There was little difference in performance between the two groups of models that used different corpora for tokenizer extension.
- For **MobileLLM**, models whose tokenizers were extended with continual pretraining data performed worse than the base model. However, models whose tokenizers were expanded with medical terms frequently outperformed the base model. Notably, in all cases where a model with an extended tokenizer achieved the highest subject-level performance, the tokenizer had been extended using medical terms.
- For **SmolLM2**, models with an extended tokenizer outperformed the base model in many subjects.
- Across all three models, the **highest average accuracy** was consistently achieved by models with tokenizers extended using **medical terms**. However, base models still achieved the highest performance in at least one subject in each case.
- No specific tokenizer extension method showed consistent superiority

across all conditions.

Besides the evaluation on macro accuracy, the number of questions where the models provided right answer (which is close to micro accuracy) were counted. The results are as in Table 11:

| | BERT Base Cased | MobileLLM 125M | SmolLM2 135M |
|---|---|---|---|
| **Auto-1000** | 1991 | 1908 | 2103 |
| **Auto-5000** | 1982 | 1798 | **2136** |
| **Auto-10000** | **2062** | 1746 | 2087 |
| **Auto-50000** | 1923 | 1824 | 2106 |
| **Auto-100000** | **2032** | 1667 | 2046 |
| **Crawled-1** | **2066** | **2229** | 2090 |
| **Crawled-2** | 1924 | 2223 | 2062 |
| **Crawled-5** | **2079** | 2174 | **2135** |
| **Crawled-10** | 1997 | 2216 | 2105 |
| **No extension** | **1998** | **2225** | **2124** |

Table 11: Overview of the number of the correct answers chosen by models

While some of the extended models showed better performance than the base model, most of them failed to outperform the base model as well as in macro average.

In summary, according to the evaluation results, tokenizer extension did not improve model performances in most of cases. While some of the variation models outperformed the base model, most of them showed poor performance. Thus, also considering that no extension method consistently helped the model to achieve better performance, it could be concluded that tokenizer extension may have negative effects on model performances.

| Benchmark | MedQA | MedMCQA | MMLU Anatomy | MMLU Clinical Knowledge | MMLU College Biology | MMLU College Medicine | MMLU Medical Genetics | MMLU Professional Medicine | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Auto-1000** | 25.53% | 32.44% | 23.70% | 31.70% | 24.31% | 27.75% | 35.00% | 27.57% | 28.50% |
| **Auto-5000** | 26.00% | 31.87% | 27.41% | 30.94% | 22.92% | 26.59% | 24.00% | 35.29% | 28.13% |
| **Auto-10000** | 25.84% | 33.68% | 31.11% | 29.81% | 22.92% | 30.06% | 36.00% | 30.15% | 29.95% |
| **Auto-50000** | 22.55% | 31.48% | 28.15% | 28.68% | 24.31% | 28.90% | 26.00% | 34.56% | 28.08% |
| **Auto-100000** | 26.32% | 32.68% | 25.93% | 30.94% | 28.47% | 24.28% | 35.00% | 34.93% | 29.82% |
| **Crawled-1** | 28.36% | 32.66% | 27.41% | 31.70% | 26.39% | 28.32% | 33.00% | 36.03% | 30.48% |
| **Crawled-2** | 23.88% | 31.39% | 20.00% | 29.43% | 25.00% | 27.17% | 31.00% | 32.35% | 27.53% |
| **Crawled-5** | 25.77% | 33.97% | 23.70% | 31.32% | 30.56% | 27.17% | 33.00% | 33.46% | 29.87% |
| **Crawled-10** | 24.43% | 32.66% | 24.44% | 31.32% | 25.69% | 24.86% | 30.00% | 34.56% | 28.50% |
| **No Extension** | 23.96% | 32.39% | 28.89% | 32.45% | 28.47% | 28.32% | 37.00% | 31.62% | 30.39% |

Table 12: Overview of performances on MultiMedQA subsets,
variations of BERT Base Cased

| Benchmark | MedQA | MedMCQA | MMLU Anatomy | MMLU Clinical Knowledge | MMLU College Biology | MMLU College Medicine | MMLU Medical Genetics | MMLU Professional Medicine | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Auto-1000** | 27.73% | 31.51% | 22.96% | 20.75% | 25.00% | 20.23% | 28.00% | 19.12% | 24.41% |
| **Auto-5000** | 25.84% | 28.59% | 22.96% | 27.55% | 22.22% | 24.86% | 29.00% | 23.90% | 25.62% |
| **Auto-10000** | 26.87% | 27.47% | 25.19% | 27.17% | 20.83% | 23.12% | 29.00% | 18.38% | 24.75% |
| **Auto-50000** | 27.81% | 28.81% | 22.22% | 27.17% | 24.31% | 20.23% | 30.00% | 23.16% | 25.46% |
| **Auto-100000** | 24.82% | 26.23% | 23.70% | 26.79% | 24.31% | 20.23% | 25.00% | 20.59% | 23.96% |
| **Crawled-1** | 29.38% | 36.48% | 34.81% | 29.43% | 24.31% | 20.81% | 35.00% | 36.03% | 30.78% |
| **Crawled-2** | 27.89% | 35.74% | 37.78% | 35.09% | 26.39% | 29.48% | 38.00% | 37.50% | 33.48% |
| **Crawled-5** | 27.73% | 35.36% | 27.41% | 37.36% | 29.86% | 22.54% | 33.00% | 33.46% | 30.84% |
| **Crawled-10** | 29.69% | 35.98% | 34.07% | 31.70% | 30.56% | 27.75% | 35.00% | 27.94% | 31.59% |
| **No Extension** | 28.04% | 36.36% | 31.11% | 35.09% | 21.53% | 27.17% | 39.00% | 34.93% | 31.65% |
| **Baseline** | 27.65% | 31.99% | 18.52% | 21.51% | 25.69% | 20.81% | 30.00% | 18.38% | 24.32% |

Table 13: Overview of performances on MultiMedQA subsets, variations of MobileLLM 125M

| Benchmark | MedQA | MedMCQA | MMLU Anatomy | MMLU Clinical Knowledge | MMLU College Biology | MMLU College Medicine | MMLU Medical Genetics | MMLU Professional Medicine | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Auto-1000** | 26.08% | 34.45% | 29.63% | 32.83% | 22.92% | 24.28% | 38.00% | 33.09% | 30.16% |
| **Auto-5000** | 26.55% | 35.07% | 28.89% | 33.21% | 22.22% | 25.43% | 33.00% | 34.93% | 29.91% |
| **Auto-10000** | 27.10% | 33.88% | 22.96% | 33.21% | 25.69% | 27.75% | 34.00% | 31.99% | 29.57% |
| **Auto-50000** | 25.61% | 34.50% | 29.63% | 33.96% | 22.92% | 28.32% | 37.00% | 32.35% | 30.54% |
| **Auto-100000** | 24.12% | 33.71% | 28.89% | 32.83% | 31.94% | 24.86% | 30.00% | 30.88% | 29.65% |
| **Crawled-1** | 26.71% | 34.09% | 30.37% | 33.21% | 30.56% | 20.23% | 34.00% | 30.15% | 29.91% |
| **Crawled-2** | 26.63% | 33.47% | 25.19% | 32.08% | 23.61% | 22.54% | 43.00% | 32.35% | 29.86% |
| **Crawled-5** | 26.39% | 34.97% | 30.37% | 35.47% | 27.08% | 27.75% | 34.00% | 29.41% | 30.68% |
| **Crawled-10** | 26.39% | 34.57% | 31.85% | 34.34% | 27.78% | 22.54% | 38.00% | 26.47% | 30.24% |
| **No Extension** | 27.49% | 34.64% | 28.89% | 32.08% | 30.56% | 22.54% | 35.00% | 30.51% | 30.21% |
| **Baseline** | 27.49% | 29.98% | 23.70% | 23.02% | 29.86% | 22.54% | 26.00% | 18.38% | 25.12% |

Table 14: Overview of performances on MultiMedQA subsets, variations of SmolLM2 135M

## 6.2. Measure on Compression

The compression abilities of tokenizers were measured, to estimate their efficiency. The measurement was done with a batch (512) of samples from the continual pretraining, in the following ways:

- The average number of tokens to which each tokenizer segments the batch of samples was measured. (Table 15)
- The average length of the texts captured by each tokenizer with 2,048 tokens were measured. (Table 16)

The batch of samples include relatively long samples (Avg. 35,043 words), to ensure that none of the tokenizers segment any of the samples into a sequence shorter than 2,048 tokens. The results are shown on Table 15 and 16.

| Model | BERT | MobileLLM | SmolLM2 |
|---|---|---|---|
| **Auto-1000** | 50796.04 (84.37%) | 59219.74 (84.30%) | 59334.67 (95.31%) |
| **Auto-5000** | 53047.58 (88.11%) | 61829.26 (88.00%) | 60743.11 (97.57%) |
| **Auto-10000** | 54166.84 (89.97%) | 63031.42 (89.72%) | 61230.94 (98.36%) |
| **Auto-50000** | 56873.87 (94.46%) | 66022.09 (93.97%) | 61996.99 (99.59%) |
| **Auto-100000** | 58007.60 (96.35%) | 67386.34 (95.91%) | 62125.66 (99.79%) |
| **Crawled-1** | 54205.07 (90.03%) | 64524.85 (91.84%) | 61181.67 (98.28%) |
| **Crawled-2** | 56779.98 (94.31%) | 67643.05 (96.28%) | 61856.12 (99.36%) |
| **Crawled-5** | 57863.62 (96.11%) | 68567.18 (97.59%) | 62050.02 (99.67%) |
| **Crawled-10** | 58542.42 (97.24%) | 69132.15 (98.40%) | 62154.71 (99.84%) |
| **No** | 60207.04 (100%) | 70257.11 (100%) | 62254.73 (100%) |

Table 15: Average number of tokens in the sample batch, tokenized by extended tokenizers

| Model | BERT | MobileLLM | SmolLM2 |
|---|---|---|---|
| **Auto-1000** | 10052.71 (117.90%) | 8849.59 (119.73%) | 8877.70 (104.45%) |
| **Auto-5000** | 9691.38 (113.66%) | 8500.62 (115.01%) | 8699.93 (102.36%) |
| **Auto-10000** | 9503.46 (111.45%) | 8328.49 (112.68%) | 8631.39 (101.55%) |
| **Auto-50000** | 9045.33 (106.08%) | 7904.12 (106.94%) | 8525.04 (101.55%) |
| **Auto-100000** | 8864.21 (103.96%) | 7728.57 (104.56%) | 8510.92 (100.14%) |
| **Crawled-1** | 9465.69 (111.01%) | 8088.73 (109.44%) | 8638.88 (101.64%) |
| **Crawled-2** | 9030.18 (105.90%) | 7687.54 (104.01%) | 8550.32 (100.60%) |
| **Crawled-5** | 8871.82 (104.05%) | 7576.90 (102.51%) | 8525.61 (100.31%) |
| **Crawled-10** | 8771.72 (102,87%) | 7513.79 (101.66%) | 8512.00 (100.15%) |
| **No** | 8526.79 (100%) | 7391.17 (100%) | 8499.33 (100%) |

Table 16: Average number of characters captured by extended tokenizers, with 2,048 tokens, on sample batch

The extended tokenizer demonstrated the ability to capture up to approximately 15.7% fewer tokens for texts of the same length, and up to approximately 19.73% more characters for token sequences of the same length. The degrees of compression reinforced per extended token were:

- **The number of tokens** needed to capture texts was reduced by **0.00095%** on average.
- **The number of characters** captured by fixed number (2,048 here) of tokens was increased by **0.00085%** on average.

While the improvement in compression ability is not entirely proportional to the number of added tokens, it is evident that the extended tokenizers exhibit enhanced compression capabilities. However, the extent of compression increased per extended token is very small, and significant improvement of compression may require plenty of extended tokens.

Meanwhile, it should be noted that since the vocabulary size of the extended tokenizer varies depending on the original tokenizer, the degree of improvement in compression also differed accordingly.

## 6.3. Training Costs

To evaluate the efficiency of tokenizer extension, the following training costs from the training process were measured:

- **Train Runtime**: The total time required for the continual pretraining stage.
- **Maximum Memory Usage**: The peak GPU memory required during training, measured at each stage for input batch sizes (512 / gradient accumulation steps).

### 6.3.1. Train Runtime

The train runtimes for continual pretraining are summarized in Figure 3. For all three models, while the results were not perfectly consistent, the additional training time compared to the base model was generally proportional to the amount of vocabulary and merge rules added to the original tokenizer. The amount of time for continually pretraining the models was increased by **0.0025% per 1 extended token** on average.

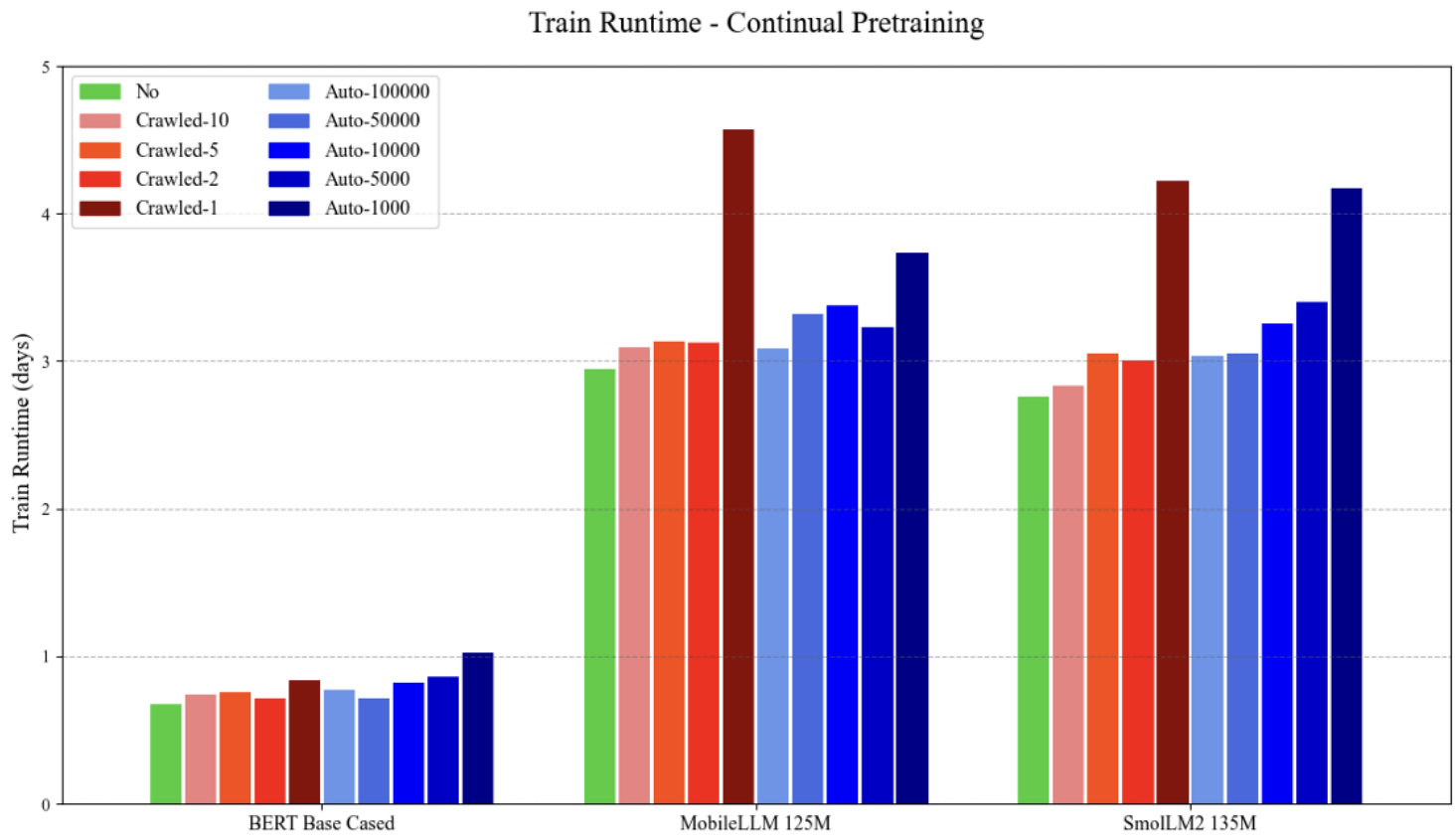


Figure 3: Train runtime for continual pretraining stage

### 6.3.2. Maximum Memory Usage

The maximum memory usage is detailed in Figures 4–9. Bars that reach the ceiling of the chart indicate out-of-memory (OOM), which means the amount of memory required for training exceeded the available device memory (80 GiB), with respect to the input batch size. Observations are as follows:

- **Continual Pretraining**: During the continual pretraining stage, similar to the train runtime, the memory required to train the model increased proportionally with the amount of vocabulary and merge rules added to the original tokenizer.
- **Multiple Choice Fine-tuning**: A similar trend was observed during the multiple-choice fine-tuning stage, though in the case of BERT, differences between tokenizer extensions were negligible.

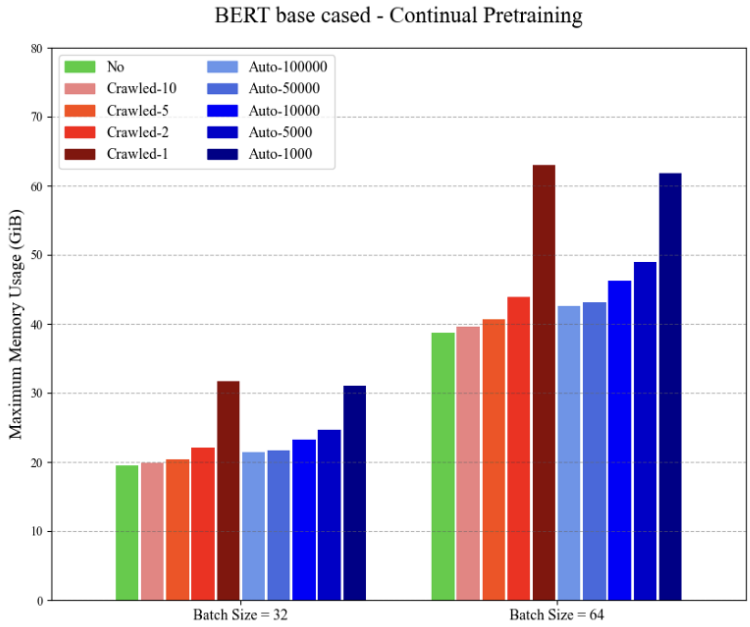


Figure 4: Maximum memory usage during continual pretraining stage of BERT Base Cased

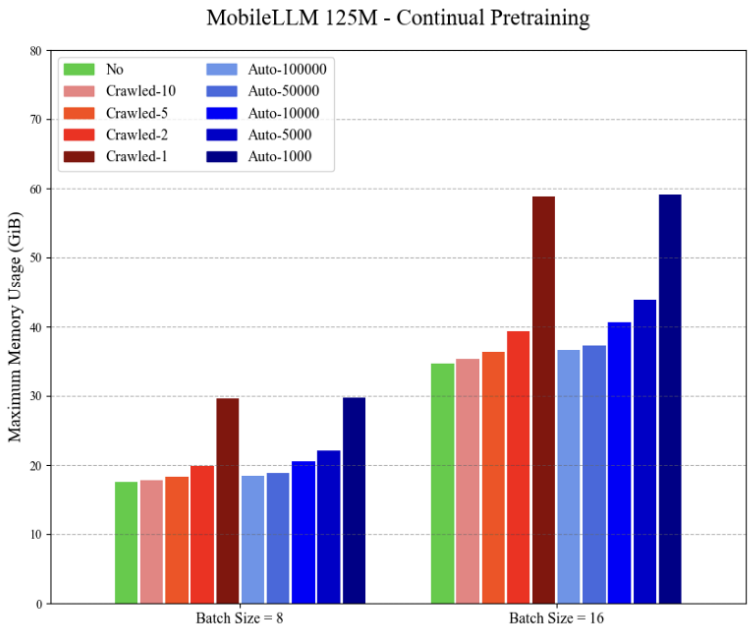


Figure 5: Maximum memory usage during continual pretraining stage of MobileLLM 125M

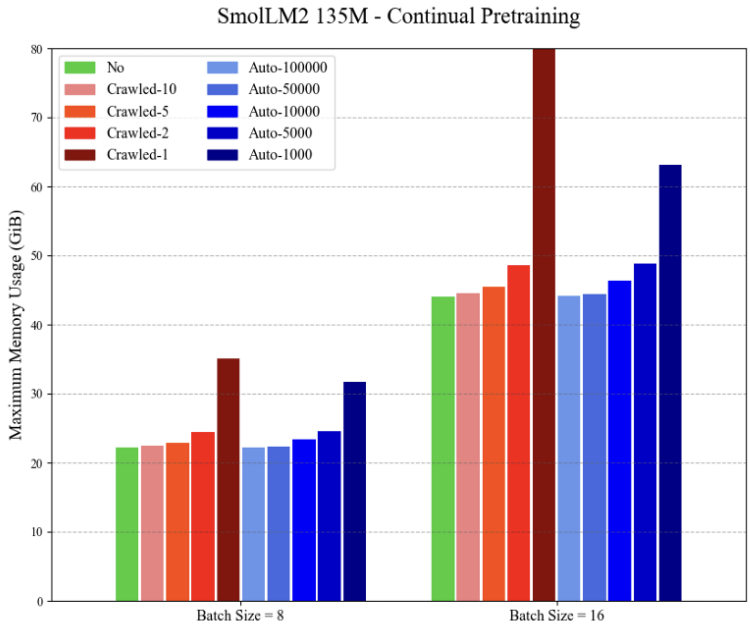


Figure 6: Maximum memory usage during continual pretraining stage of SmolLM2 135M

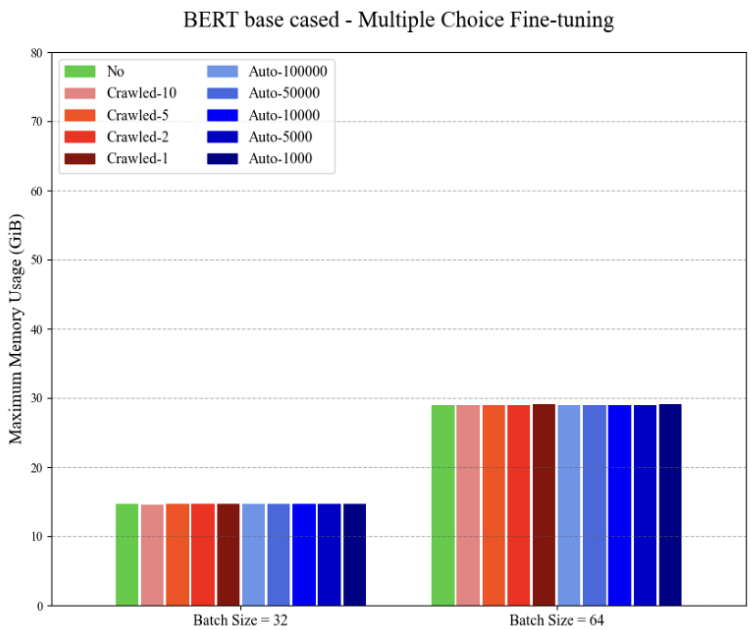


Figure 7: Maximum memory usage during multiple choice fine-tuning stage of BERT Base Cased

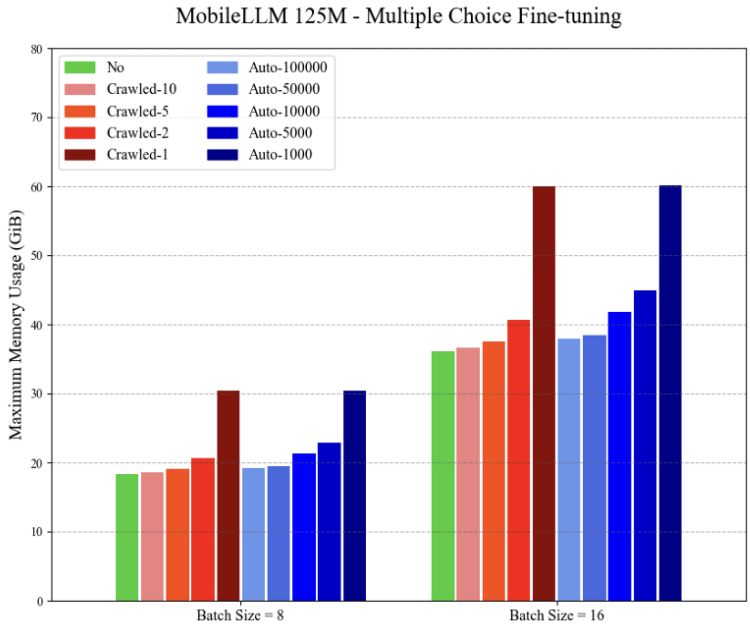


Figure 8: Maximum memory usage during multiple choice fine-tuning stage of MobileLLM 125M

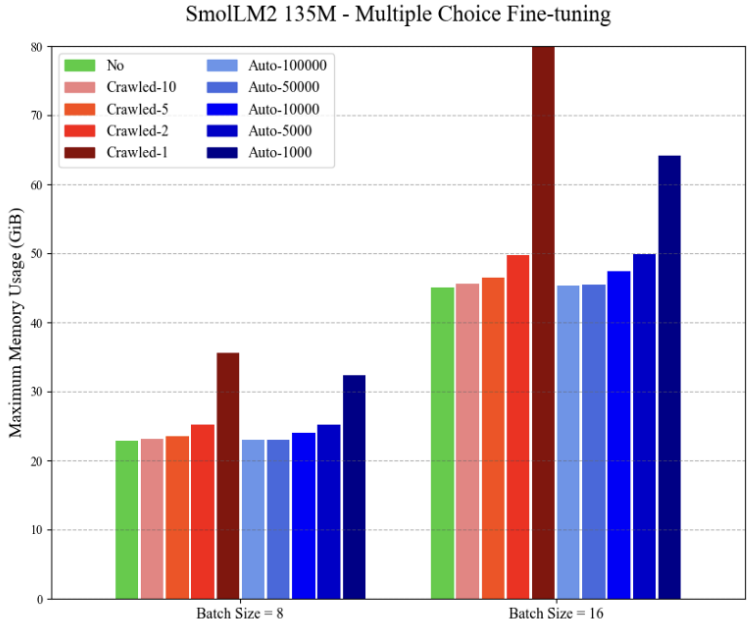


Figure 9: Maximum memory usage during multiple choice fine-tuning stage of SmolLM2 135M

Overall, adding tokens to the tokenizer generally increases the training time and GPU memory usage proportionally to the number of added tokens. This highlights that excessive tokenizer extensions can lead to substantial training costs, warranting careful consideration.

# 7. Conclusion

In this study, the effects of extending a tokenizer using medical text data or list of medical terms are analyzed. The medical field is characterized by the frequent use of specialized terminology, and tokenizers trained on general data may encounter challenges such as out-of-vocabulary (OOV) issues or segmenting text into excessively long sequences when processing medical texts. While extending a tokenizer prior to training models on such domain-specific data may offer advantages, there has been limited in-depth research on this topic. This study aimed to fill that gap, and through experiments and analyses, the following findings were uncovered:

- **Negative Effect on Performance**: Evaluation results showed that models trained with an extended tokenizer mostly performed worse than the base model, besides the performance inconsistency according to extension methods. In other words, extending the tokenizer does not guarantee improved performance or even possibly hinders model's performance, regardless of the extension method.
- **Improved Compression**: Extending the tokenizer enhances its compression capability, but the extent of enhancement was small. In addition, the number of added tokens varies depending on the vocabulary distribution of the original tokenizer, which affects the degree of improvement in compression. Therefore, it is necessary to take the original tokenizer's vocabulary into

account when performing an extension.

- **Increased Training Costs**: Since the modules related to vocabulary size must also be extended to accommodate the larger tokenizer, extending a tokenizer inevitably increases training costs. For this reason, it is essential to consider the device's memory capacity and the available time for experiments, and only extend the tokenizer by a reasonable number of tokens.

In summary, extending a tokenizer for domain-specific fine-tuning seems to have negative impact on model performance. While it helps language models to capture longer texts with shorter sequence of tokens, the benefit remains insignificant unless thousands of tokens are extended, and excessive extension of tokenizer leads to large increase in training costs. Therefore, it could be concluded that tokenizer extension is may not be helpful, with respect to domain-specific fine-tuning of language models.

# Bibliography


Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. B., Tunstall, L., Piqueres, A., Marafioti, A., Zakka C., Werra, L., & Wolf, T. (2024). SmolLM2 - with great data, comes great performance.

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., ... & Zhu, T. (2023). Qwen technical report. *ArXiv, abs/2309.16609.*

Bostrom, K., & Durrett, G. (2020). Byte Pair Encoding is Suboptimal for Language Model Pretraining. *Findings*.

Chung, J., Cho, K., & Bengio, Y. (2016). A Character-level Decoder without Explicit Segmentation for Neural Machine Translation. *ArXiv, abs/1603.06147*.

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *North American Chapter of the Association for Computational Linguistics*.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., ... & Ganapathy, R. (2024). The llama 3 herd of models. *ArXiv, abs/2407.21783*.

Gage, P. (1994). A new algorithm for data compression. *The C Users Journal archive, 12*, 23-38.

Gallé, M. (2019). Investigating the Effectiveness of BPE: The Power of Shorter Sequences. *Conference on Empirical Methods in Natural Language Processing*.

Goldman, O., Caciularu, A., Eyal, M., Cao, K., Szpektor, I., & Tsarfaty, R. (2024). Unpacking Tokenization: Evaluating Text Compression and its Correlation with Model Performance. *Annual Meeting of the Association for Computational*

*Linguistics*.

Gutierrez-Vasques, X., Bentz, C., & Samardžić, T. (2023). Languages through the looking glass of bpe compression. *Computational Linguistics, 49*(4), 943-1001.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D.X., & Steinhardt, J. (2020). Measuring Massive Multitask Language Understanding. *ArXiv, abs/2009.03300*.

Jin, D., Pan, E., Oufattole, N., Weng, W. H., Fang, H., & Szolovits, P. (2021). What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences*, *11*(14), 6421.

Kudo, T. (2018). Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. *ArXiv, abs/1804.10959*.

Kudo, T., & Richardson, J. (2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. *Conference on Empirical Methods in Natural Language Processing*.

Liu, Z., Zhao, C., Iandola, F.N., Lai, C., Tian, Y., Fedorov, I., Xiong, Y., Chang, E., Shi, Y., Krishnamoorthi, R., Lai, L., & Chandra, V. (2024). MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases. *ArXiv, abs/2402.14905*.

Luong, M., & Manning, C.D. (2016). Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models. *ArXiv, abs/1604.00788*.

Merriam-Webster. (2016). Merriam-Webster medical dictionary. Merriam Webster.

Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*.

Pal, A., Umapathi, L. K., & Sankarasubbu, M. (2022, April). Medmcqa: A large-scale

multi-subject multi-choice dataset for medical domain question answering. In *Conference on health, inference, and learning* (pp. 248-260). PMLR.

Pennington, J., Socher, R., & Manning, C.D. (2014). GloVe: Global Vectors for Word Representation. *Conference on Empirical Methods in Natural Language Processing*.

PubMed. https://pubmed.ncbi.nlm.nih.gov/

PubMed Central. https://pmc.ncbi.nlm.nih.gov/

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, *1*(8), 9.

Schmidt, C.W., Reddy, V., Zhang, H., Alameddine, A., Uzan, O., Pinter, Y., & Tanner, C. (2024). Tokenization Is More Than Compression. *Conference on Empirical Methods in Natural Language Processing*.

Schuster, M., & Nakajima, K. (2012, March). Japanese and korean voice search. In 2012 IEEE international conference on acoustics, speech and signal processing (ICASSP) (pp. 5149-5152). IEEE.

Sennrich, R., Haddow, B., & Birch, A. (2015). Neural Machine Translation of Rare Words with Subword Units. *ArXiv, abs/1508.07909*.

Singhal, K., Azizi, S., Tu, T., Mahdavi, S. S., Wei, J., Chung, H. W., ... & Natarajan, V. (2023). Large language models encode clinical knowledge. *Nature*, *620*(7972), 172-180.

Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., ... & Kenealy, K. (2024). Gemma: Open models based on gemini research and technology. *ArXiv, abs/2403.08295*.

Team, G., Riviere, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., ... & Garg, S. (2024). Gemma 2: Improving open language models at a practical size. *ArXiv, abs/2408.01108*.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M. A., Lacroix, T., ... & Lample, G. (2023). Llama: Open and efficient foundation language models. *ArXiv, abs/2302.13971*.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., ... & Scialom, T. (2023). Llama 2: Open foundation and fine-tuned chat models. *ArXiv, abs/2307.09288*.

Vaswani, A., Shazeer, N.M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., & Polosukhin, I. (2017). Attention is All you Need. *Neural Information Processing Systems*.

Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., … & Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *ArXiv, abs/1609.08144*.

Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., ... & Qiu, Z. (2024). Qwen2. 5 Technical Report. *ArXiv, abs/2412.15115*.

# 국문 초록

최근 들어 대량의 데이터로 장시간 사전 훈련된 LLM들이 유행하고 있으며, 개인이 이러한 모델을 처음부터 훈련하여 사용하기란 대단히 어려우므로 공유된 모델을 미세 조정(fine-tuning)하여 사용하는 것이 일반적이게 되었다. 그런데 미세 조정에 활용하려는 데이터의 어휘 분포가 기존의 토크나이저(tokenizer)가 처리할 수 있는 토큰 목록에서 크게 벗어날 경우, 토크나이저가 이를 처리하지 못하거나 너무 잘게 분절하는 문제가 발생한다. 토크나이저에 새로운 어휘들을 추가하는 토크나이저 확장(extension)이 이러한 문제를 완화하는 좋은 해결책이 될 수 있으나, 토크나이저 확장이 어떤 효과를 불러오는지에 대한 면밀한 연구는 아직 이루어진 바가 없다.

본 연구에서는 따라서 의학 데이터를 통해 확장한 토크나이저를 사용해 소형 모델들을 훈련하고, 몇 가지 분석을 통해 전문 분야에 대한 미세 조정에서의 토크나이저 확장의 효과를 확인하고자 하였다. 의학 분야는 다양한 전문용어가 빈번히 사용되는 분야로, 토크나이저 확장으로부터 긍정적인 효과를 얻을 수 있을 것으로 예상하였다. 그러나 BPE(Byte Pair Encoding) 기반의 SentencePiece BPE, Byte-level BPE 및 비슷한 알고리즘을 사용하는 WordPiece를 확장하여 실험을 수행한 결과, 토크나이저의 압축(compression) 능력이 소폭 향상된 반면 모델 훈련에 필요한 메모리와 시간이 증가하였다. 또한 MultiMedQA의 4지선다형 문제들로 모델들

을 평가한 결과, 토크나이저를 확장한 대부분의 모델의 성능이 확장하지 않은 모델의 성능보다 낮았다. 이러한 결과들로 미루어 볼 때, 전문 분야에 대해 언어 모델을 미세 조정할 때의 토크나이저 확장이 유리하지는 않은 것으로 생각된다.