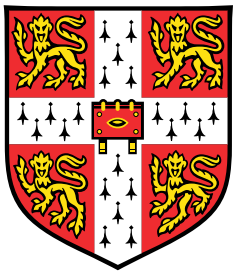
Retrofitting Language Models with Dynamic Tokenisation



Darius Feher

Supervisor: Dr. Ivan Vulić

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy in Machine Learning and Machine Intelligence

St. Edmund's College August 2024

I dedicate this thesis to my supportive family, my late middle school maths teacher who sparked my interest in maths, and my high school maths teacher whose support and inspiration helped me reach this point in my academic journey.

Declaration

I, Darius Feher of St. Edmund's College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. The code used for developing this project has been written in Python, using standard machine learning libraries such as `Pytorch`[1] or `datasets`[2]. However, the following libraries were used or extended for specific functionalities:

1. Training scripts from the `transformers` library from Hugging Face, which were customised to support dynamic tokenisation;[3]
2. The Zero-Shot Tokeniser Transfer (ZeTT) library for converting tokenisers to byte-level and providing utility functions for the hypernetworks;[4]
3. The Parameter-Efficient Fine-Tuning (PEFT) library for merging and training adapters;[5]
4. The FastChat library for evaluating models with a custom MT-Bench script.[6]

The word count for this thesis including tables, figure captions, appendices and footnotes is **14988**.

Darius Feher
August 2024

[1] pytorch.org/
[2] github.com/huggingface/datasets
[3] github.com/huggingface/transformers
[4] github.com/bminixhofer/zett
[5] github.com/huggingface/peft
[6] github.com/lm-sys/FastChat

Acknowledgements

First and foremost, I would like to extend my sincere thanks to my supervisor, Ivan Vulić, as well as to Benjamin Minixhofer. Your guidance, constructive feedback, and support were instrumental in shaping the direction and outcomes of this dissertation. I am deeply grateful for the opportunity to work under your supervision, and I appreciate the time and effort you dedicated to mentoring me throughout this journey. I am looking forward to continuing our work together on this project.

I want to express my sincere gratitude to the Romanian Ministry of Education, whose support, facilitated through the Agency for Credits and Scholarships, made it possible for me to study at the University of Cambridge. Their funding has been crucial in shaping my educational journey.

Finally, I want to express my appreciation to Andrej, Clare, Elijah, and Sedie (in no particular order), who were not only my course mates but have also become my friends. Without you, my experience at Cambridge would not have been nearly as meaningful or memorable.

Abstract

Large Language Models (LLMs) are the backbone of modern Natural Language Processing (NLP) applications. They typically rely on subword tokenisation, breaking text into pieces of words or entire words, for efficient processing. Although this technique proves effective in representing arbitrary sequences of text, it presents several challenges, including difficulty in handling sequences of numbers, spelling errors, and susceptibility to certain types of textual manipulations. Importantly, these subword tokenisers use *static vocabularies*, which are biased towards high-resource languages, resulting in over-segmentation in low-resource languages and inducing unfairness towards these languages. The static nature of the vocabulary limits the model's adaptability to new words or evolving language use, requiring periodic and expensive retraining to update the vocabulary and the embeddings.

To tackle this, we propose a framework for retrofitting language models (LMs) with *dynamic tokenisation*, a mechanism that allows the token boundaries to adapt based on the input text, in LMs pre-trained with subword tokenisation. We repurpose the pre-trained hypernetwork from Minixhofer et al. (2024), trained for transfer to another static tokeniser, to enable dynamic tokenisation by predicting token embeddings for any (newly) encountered token. Implementing dynamic tokenisation at the batch-level for encoders significantly reduces token sequence lengths with minimal impact on performance, thus improving inference speed and ensuring more equitable language representation. In decoders, we apply dynamic tokenisation at the sample-level which enables for instance chat-based adaptation. Using an Approximate Nearest Neighbour index, we achieve fast generation with a one million token vocabulary, demonstrating scalability to even larger, dynamic vocabularies. Overall, our findings show that dynamic tokenisation significantly improves inference speed and promotes fairness across languages, overcoming the limitations of static tokenisation and enabling more equitable and adaptable LMs.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

ANN	Approximate Nearest Neighbours
BPE	Byte-Pair-Encoding
EM	Expectation-Maximisation
FVT	Fast Vocabulary Transfer
HN	Hypernetwork
LP	Longest-Prefix Tokenisation
LoRA	Low-Rank Adaptation
MMLU	Massive Multitask Language Understanding
MT-Bench	Multi-Turn Benchmark
NER	Named Entity Recognition
OOV	Out-of-vocabulary
UNER	Universal Named Entity Recognition
XNLI	Cross-lingual Natural Language Inference

Symbols

$\mathcal{D}$	Dataset
$\mathcal{D}_{\text{batch}}$	Batch in a dataset
d_{model}	Dimension of a model

E_ϕ	Embeddings function
ϕ	Embeddings matrix
T	Tokenisation function
$\mathcal{V}$	Vocabulary

Chapter 1

Introduction

Large Language Models (LLMs) are the backbone of modern Natural Language Processing (NLP) applications, enabling advanced language understanding and generation (Zubiaga, 2024). However, their effectiveness heavily relies on their tokenisers, which are responsible for *tokenising* the input text (Minaee et al., 2024; Minixhofer et al., 2024). This is a fundamental step in NLP, and involves breaking raw text into smaller units called *tokens*, which are part of the tokeniser's *vocabulary*. Since machines can only work with numerical data, tokens are converted into numerical IDs, which are then used to obtain *embeddings* — fixed-size vectors that serve as the model's representation of a token. These embeddings capture the semantic properties of the tokens, enabling machine learning (ML) models to understand and process textual data effectively. This process, from raw text to embeddings, is outlined in Figure 1.1.

Traditional approaches focussed on character-level, byte-level, subword and word tokenisation. **Character-** (Boukkouri et al., 2020; Tay et al., 2021; Clark et al., 2022) **and byte-level** (Xue et al., 2022; Yu et al., 2024) methods have several benefits, such as a small vocabulary (i.e., the set of individual characters or bytes), an increased robustness to noise and capability of handling rare words, which is especially useful for low-resource languages (Xie et al., 2018). Despite these advantages, they suffer from reduced processing speed (i.e., throughput) due to the increased length of sequences or the requirement for sequences to learn how to be effectively pooled (Nawrot et al., 2022; Lee et al., 2017). This impacts especially the training phase (Clark et al., 2022), where large amounts of data need to be processed.

On the other hand, **word tokenisation** methods offer faster processing speeds due to shorter sequences, but they struggle with out-of-vocabulary (OOV) words (e.g., rare or new words) and morphological variations. These OOV words are typically replaced with a generic

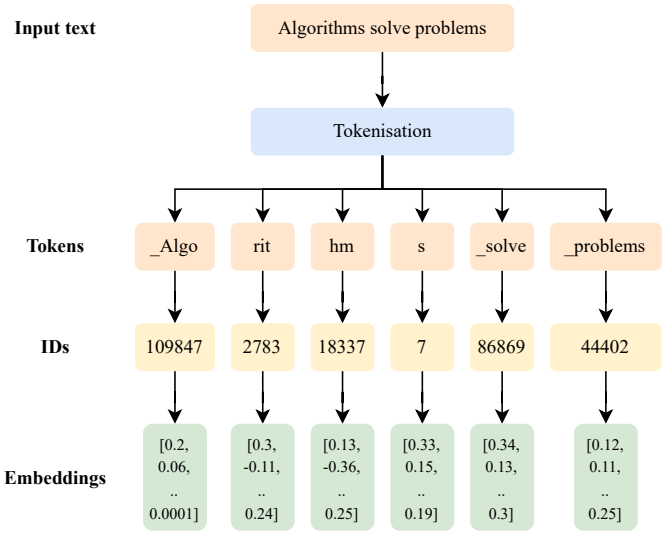


Fig. 1.1 Illustration of how text is tokenised and converted into embeddings using XLM-RoBERTa. Each embedding has a size of 768. An underscore (_) precedes tokens that originally had a space before them, following the tokenisation convention used by this model. By default, a space is added at the beginning of the sentence.

<UNK> token. Additionally, these methods require large vocabularies to cover the variety of words in a language which can be inefficient.

Given the limitations of character-level and word-level tokenisation, a commonly used alternative is **subword tokenisation**. This approach strikes a balance between the two by breaking down text into smaller, more manageable units, such as pieces of word or entire words. The goal is to keep common words as they are while breaking down rare words into smaller, frequently occurring subwords. Techniques like Byte-Pair Encoding (BPE), as described by Sennrich et al. (2015), and WordPiece, developed by Schuster and Nakajima (2012), effectively handle OOV words by breaking them into known subword units, while also maintaining manageable vocabulary sizes and sequence lengths. Figure 1.2 presents an example to illustrate the difference between these three methods, highlighting the contrast in vocabulary size and token sequence length. Although this technique proves effective in representing arbitrary sequences of text (Mielke et al., 2021), it presents several challenges, including difficulty in handling sequences of numbers (Golkar et al., 2023), spelling errors (Sun et al., 2020; Xue et al., 2022), and susceptibility to certain types of textual manipulations (Eger and Benz, 2020; Rust et al., 2022).

The challenges of this rigid tokenisation are further exacerbated in multilingual contexts, where scarce data in certain languages leads to over-segmentation, limiting cross-lingual transfer (Wang et al., 2021). This induces unfair treatment for certain languages, increasing

the inference costs of the language models (LMs) and reducing their performance (Ahia et al., 2023). Additionally, byte-level tokenisers also show a bias towards Latin scripts, disadvantaging non-Latin script languages (Mielke et al., 2021; Petrov et al., 2024). Combined with the fact that multilingual tokenisers do not perform as well as monolingual ones (Rust et al., 2020), these issues underscore the need for a more flexible or dynamic tokenisation, that adapts token boundaries based on the input text, offering a fairer approach across languages.

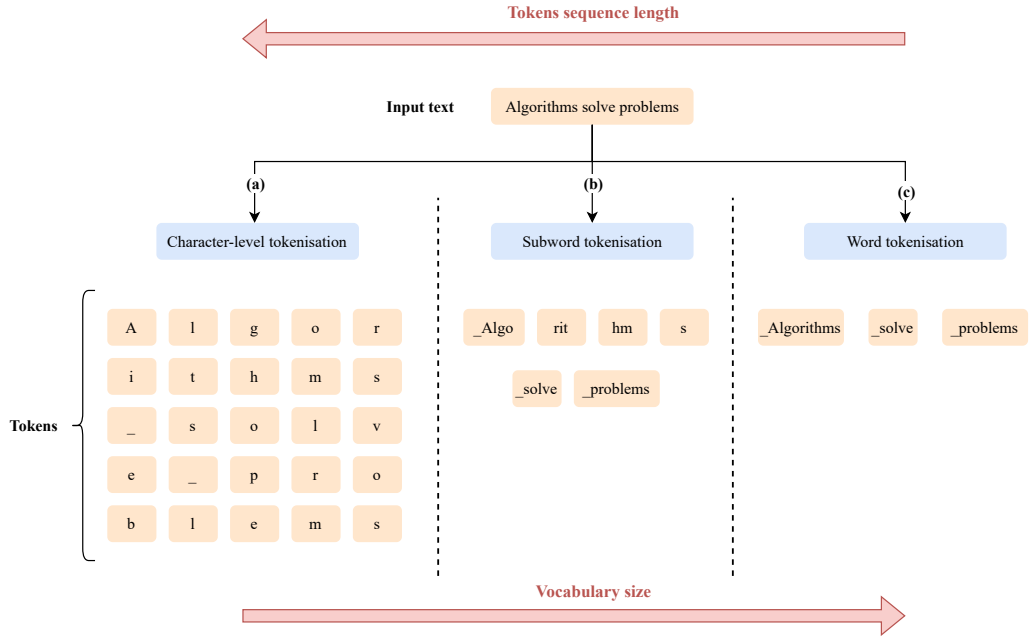


Fig. 1.2 Comparison between existing tokenisations.

Furthermore, all these methods are *static*, meaning they rely on a predefined, fixed vocabulary that does not adapt to new data once the model is trained. This static nature can limit the model's adaptability to new words or evolving language use, requiring periodic and expensive retraining to update the vocabulary and the embeddings. Considering that LMs typically rely on static subword tokenisation and given these limitations, our work explores the potential of retrofitting LMs with *dynamic tokenisation* through the use of hypernetworks. Specifically, this approach allows the token boundaries to adapt based on the input text, in LMs pre-trained with subword tokenisation.

To achieve this objective, we build upon the work of Minixhofer et al. (2024), which addresses a critical limitation in contemporary LMs: the inability to change the tokeniser after training, which can limit their efficiency and effectiveness, especially in multilingual settings or specialised domains like programming. Hence, in their work, they focus on transferring an LM to an arbitrary, but fixed, tokeniser via zero-shot prediction of the embedding parameters. For instance, transferring the XLM-RoBERTa LM to the GPT-2 tokeniser, while preserving

the performance on the downstream tasks. To achieve this, a hypernetwork (HN) — a type of network that generates the parameters for another network (Ha et al., 2016) — is trained on a diverse set of tokenisers to predict the embedding for every token of the new tokeniser. See §2.5 for more details regarding the HN training and architecture.

Unlike Minixhofer et al.'s (2024) approach, which uses an HN to predict the embeddings for the tokens in the arbitrary, but fixed, target tokeniser, our method repurposes the pre-trained HN to predict the embeddings of the tokens obtained by applying our dynamic tokenisation. Additionally, while their approach is limited to a static, fixed vocabulary after transfer, we are adapting our vocabulary to new tokens as they appear in the input text. The key benefit of using an HN is that it requires training only once for a given model, enabling it to predict embeddings for any new token afterwards. Without HNs, implementing dynamic tokenisation would require expensive and impractical retraining of the model's embedding parameters, making real-time adaptation unfeasible. Thus, in our work, the first step is to *(1)* decide on a tokenisation (i.e., a way to break the text into tokens), and then *(2)* using the same pre-trained hypernetwork from Minixhofer et al. (2024), embed the tokens under the current tokenisation technique. The process workflow is highlighted in Figure 1.3.

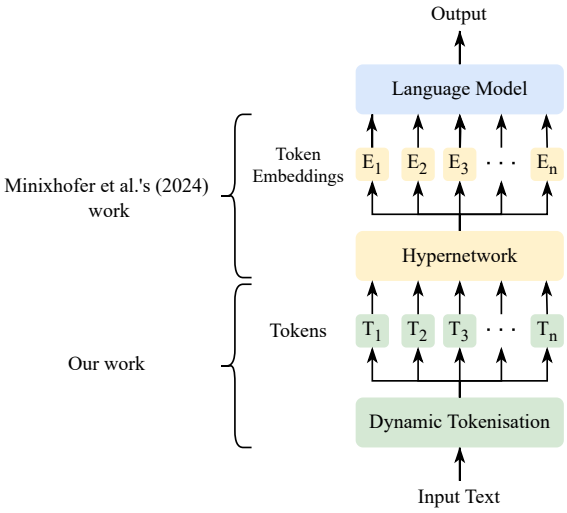


Fig. 1.3 A high-level workflow of our work on retrofitting LMs with dynamic tokenisation, building upon the work of Minixhofer et al. (2024).

The significance of this research lies in several key benefits:

- It has the potential to **increase throughput** by decreasing the tokenised sequences length. This is particularly important because the memory and time complexity of attention layers grow quadratically with respect to sequence length (Vaswani et al., 2017).

- It offers **a more equitable approach across languages**, particularly benefiting low-resource languages by alleviating the issue of over-segmentation and potentially improving performance;
- It enables the transition from the fixed-size vocabulary of static subword tokenisers to effectively **unbounded vocabularies**, enhancing the model's adaptability to new or rare words and evolving language use;

1.1 Contributions

A summary of the most important novel contributions is presented below:

- We developed two methods for retrofitting LMs with dynamic tokenisation: a batch-level method for encoders, and a sample-level technique for decoders (Chapter 3);
- We carefully evaluated these methods and showed that our dynamic tokenisation techniques significantly reduce token sequences with minimal performance degradation, depending on the setting, thus improving inference speed and leading to a fairer representation across languages;
- We demonstrated the effective use of the hypernetwork from Minixhofer et al. (2024) to manage large-scale vocabularies (one million tokens) with an approximate nearest neighbour index in decoder models, overcoming the typical parameter overhead by dynamically retrieving embeddings (Chapter 5).

1.2 Thesis Outline

The thesis is organised as follows:

- Chapter 2 presents the background and related work;
- Chapter 3 introduces the methodology used in this study;
- Chapter 4 details the experimental setup;
- Chapter 5 presents and discusses the results;
- Finally, Chapter 6 outlines the conclusions, impacts, limitations, and future work.

Chapter 2

Background and Related Work

This chapter lays the theoretical foundation required for the discussions and experiments that follow in this thesis. It begins by exploring different subword tokenisation algorithms (§2.1), followed by a discussion on language models (LMs) (§2.2). The chapter then highlights the benefits of using adapters with contemporary LMs. Subsequent sections include a discussion on multilingual models and tokenisers (§2.3), followed by a review of techniques for embedding initialisation (§2.4) and prediction, with a specific focus on the hypernetwork (HN) approach proposed by Minixhofer et al. (2024) (§2.5). The chapter concludes with a related work (§2.6), summary and problem formulation for *dynamic tokenisation*, setting the stage for the subsequent chapters (§2.7).

2.1 Tokenisers

As discussed in Chapter 1, word tokenisers struggle with out-of-vocabulary (OOV) words and character or byte-level tokenisers produce long sequences, among other issues. Recent developments like MegaByte (Yu et al., 2024), Canine (Clark et al., 2022), and SpaceByte (Slagle, 2024) aim to eliminate subword tokenisation by using bytes or characters. However, they still rely on some form of tokenisation, inheriting the associated limitations. Challenges include the need for sequence pooling (Nawrot et al., 2022), reliance on specific linguistic features, and difficulties with languages without space delimiters such as Chinese. This can also introduce biases, especially in low-resource languages where characters may span multiple bytes, leading to longer processing times, decreased performance, and higher computational costs (Ahia et al., 2023).

Subword tokenisers, on the other hand, strike a balance between character-level (or byte) and word-level tokenisers, while introducing their own set of limitations. These include over-segmentation of rare or misspelled words or a bias towards Latin scripts. Despite

these drawbacks, subword tokenisers are, at the moment of writing, the most widely used. Therefore, they will form the focus of the remainder of this section.

Let $\mathcal{V}$ denote a *vocabulary*, and T denote a *tokenisation function*. A tokeniser can then be defined by a tuple consisting of these two components, $(\mathcal{V}, T)$.[1] The vocabulary $\mathcal{V}$ contains the set of tokens, while the tokenisation function T is used to segment the input text into smaller units, which are part of $\mathcal{V}$. Importantly, for a given $\mathcal{V}$, there are multiple ways to encode the same input text into a sequence of tokens (Hofmann et al., 2022), with T determining the specific encoding method. These two components, $\mathcal{V}$ and T, are usually integral to any tokenisation algorithm, with recent work advocating to decouple them, allowing any tokenisation function to be applied to any given vocabulary (Uzan et al., 2024).

2.1.1 Normalisation and Pre-tokenisation

Before tokenising a text, an *optional* pre-processing step takes place, which involves two stages:

- **Normalisation**: Text is cleaned up by removing unnecessary whitespace, lowercasing characters, removing accents, or applying Unicode normalisation steps;[2]
- **Pre-tokenisation**: This involves dividing the text into preliminary units, often based on whitespace and punctuation, using a regular expression (i.e., regex). A widely used pre-tokeniser is available in the `pre-tokenizers` module of the Hugging Face's `tokenizers` package,[3] as well as the Moses pre-tokeniser, formerly known as a tokeniser (Koehn et al., 2007).

The resulting pre-tokens serve as the initial units that the tokenisation function T will further process. To formalise this, we define $T(w) := (t_1, t_2, \ldots, t_n)$ as a tokenisation of the word (i.e., pre-token) w into n subword tokens such that $\forall i, t_i \in \mathcal{V}$, and the concatenation of $t_1, t_2, \ldots, t_n$ reconstructs w. After tokenisation, the model maps the sequence of (discrete) tokens $t_1, \ldots, t_n$ to their corresponding embeddings — continuous representation — through an embedding function $E_\phi : \mathcal{V} \to \mathbb{R}^{d_{\text{model}}}$.[4] These embeddings capture semantic properties of the tokens and enable ML models to process the data effectively. They are parameterised by a matrix ϕ, used as a lookup table assigning a distinct d_{model}-dimensional vector —

[1] For consistency reasons, we adopt the same notation used by Minixhofer et al. (2024).

[2] unicode.org/reports/tr15/

[3] github.com/huggingface/tokenizers/tree/main

[4] For simplicity, the step of converting tokens to their numerical IDs, as illustrated in Figure 1.1, is omitted here.

representing a row of the matrix — to every element in $\mathcal{V}$. The resulting matrix has a size of $|\mathcal{V}| \times d_{\text{model}}$.

Having defined tokenisers and their main components, we will now explore different tokenisation algorithms such as Byte-Pair-Encoding (BPE), WordPiece and UnigramLM.

2.1.2 Byte-Pair-Encoding

Initially introduced as a compression algorithm (Gage, 1994), Byte-Pair-Encoding (BPE) was later adapted for subword tokenisation by Sennrich et al. (2015) in the context of neural machine translation. BPE can operate on **character-level**, but replaces unrecognised characters (e.g., some emojis) with the `<UNK>` token, and **byte-level**, which uses a 256-byte vocabulary to represent any character, however, producing longer token sequences.

Training

The algorithm starts with a base vocabulary of individual characters or bytes and learns merge rules by iteratively creating new tokens from the most frequent pair of existing tokens until a desired vocabulary size is reached, as detailed in Algorithm 1. Additionally, the learned merge rules, $\mathcal{M}$, are stored in the order they are created.

Algorithm 1 *Byte-Pair-Encoding Training Algorithm*

Input: Corpus of text *corpus* and desired vocabulary size *targetVocabSize*
Output: Vocabulary $\mathcal{V}$ and merge rules $\mathcal{M}$

procedure TRAINBPE(*corpus*, *targetVocabSize*)
    Initialise *vocabulary*, $\mathcal{V}$, with all unique characters in the corpus
    Initialise $\mathcal{M} \leftarrow \{\}$ ▷ Used to store the merge rules
    Split *corpus* into words (i.e., pre-tokens) and then into characters or bytes
    Compute frequency of each word in the corpus
    while $|\mathcal{V}| <$ *targetVocabSize* **do** ▷ Iteratively learn merge rules until the desired vocabulary size is reached
        pairFreqs ← ComputePairFrequencies(*corpus*)
        bestPair ← GetMostFrequentPair(*pairFreqs*) ▷ E.g., bestPair ← (‘th’, ‘is’)
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{$*GetMergeRule(bestPair)*$\}$ ▷ E.g., $\mathcal{M} \leftarrow \mathcal{M} \cup \{$(‘th’, ‘is’)$\}$
        $\mathcal{V} \leftarrow \mathcal{V} \cup \{$*GetMergedPair(bestPair)*$\}$ ▷ E.g., $\mathcal{V} \leftarrow \mathcal{V} \cup \{$‘this’$\}$
        Apply *bestPair* merge rule to *corpus*
    end while
    return $\mathcal{V}$, $\mathcal{M}$
end procedure

Tokenisation

To tokenise a text, we first normalise and pre-tokenise it, followed by splitting it into characters or bytes. We then apply all the learned merge rules in $\mathcal{M}$, **in the same order as they were learned**, to obtain the corresponding tokens, which are part of $\mathcal{V}$, as detailed in Algorithm 2. The BPE tokenisation process is therefore parametrised by the learned merge table.

Algorithm 2 *Byte-Pair-Encoding Tokenisation Algorithm*

Input: Text *text* to be tokenised and merge rules $\mathcal{M}$
Output: List of tokens from $\mathcal{V}$

procedure TOKENISEBPE(*text*, $\mathcal{M}$)
    Initialise tokens ← {} ▷ List that will hold the final sequence of tokens
    Split *text* into words (i.e., pre-tokens)
    for each *word* in *words* **do**
        Split each *word* into characters or bytes
        for *mergeRule* in $\mathcal{M}$ **do**
            if *mergeRule* exists in the split of the *word* **then**
                Apply *mergeRule* to the split of the *word*
            end if
        end for
        Append processed *word* to *tokens* after applying all merge rules in $\mathcal{M}$
    end for
    return List of tokens ▷ Final list of tokens reflecting the structure of vocabulary $\mathcal{V}$
end procedure

2.1.3 WordPiece

Another popular subword tokeniser is WordPiece, proposed by (Schuster and Nakajima, 2012) for Korean and Japanese text, and also used by models like BERT (Devlin et al., 2018).

Training

The training algorithm of WordPiece is similar to that of BPE, with two main differences:

1. The initial vocabulary consists of all the first characters of each word (i.e., pre-token), while the characters inside a word are **prefixed** with a symbol or character such as '##' used by BERT. For instance, the word `solve` is split into `s ##o ##l ##v ##e`;

2. In each iteration, a merge rule is learned. However, instead of choosing the most frequent pair, WordPiece selects the pair with the **highest score** according to Equation 2.1.

$$\text{score}(p_1, p_2) = \frac{\text{Freq}(p_1, p_2)}{\text{Freq}(p_1) \times \text{Freq}(p_2)}, \tag{2.1}$$

where

$$\text{Freq}(p) = \text{frequency of pair } p \text{ in corpus}$$

Tokenisation

Unlike BPE, which saves both the vocabulary, $\mathcal{V}$, and the merge rules, $\mathcal{M}$, after training, for use in tokenisation, WordPiece saves only the vocabulary $\mathcal{V}$. During tokenisation, WordPiece follows a **per-word left-to-right longest-match-first** strategy (Song et al., 2021). Specifically it selects the longest possible token in $\mathcal{V}$ that is a prefix of the text. This process is repeated to tokenise any remaining parts of the text. The tokenisation process is detailed in Algorithm 3. Unlike BPE, WordPiece's tokenisation is not parameterised, meaning it does not rely on any learned parameters such as a merge table. Instead, it can be applied to any predefined vocabulary.

Algorithm 3 *WordPiece Tokenisation Function*

Input: A word w and vocabulary $\mathcal{V}$
Output: A list of tokens from the vocabulary $\mathcal{V}$

procedure TOKENISEWORDPIECE(w , $\mathcal{V}$)
    Initialise $tokens \leftarrow \{\}$ ▷ Start with an empty list of tokens
    while w is not empty **do** ▷ Continue until all characters are processed
        Find the longest prefix of w that is in $\mathcal{V}$
        Add the longest prefix to $tokens$
        Remove the prefix from w
        if w is not empty **then**
            Prepend "##" to w ▷ Mark the continuation of the word
        end if
    end while
    return $tokens$ ▷ Return the list of tokens
end procedure

2.1.4 UnigramLM

Another commonly used tokenisation algorithm is UnigramLM, introduced by Kudo (2018), which uses a probabilistic approach.

Training

As opposed to the other two algorithms, UnigramLM starts with a large vocabulary $\mathcal{V}$ including all unique characters and the most frequent substrings in the corpus. Alternatively, BPE can be run with a sufficiently large number of merges to obtain the initial $\mathcal{V}$. The model makes an independence assumption between subwords — hence the name — and the probability of a subword sequence $\mathbf{x} = (t_1, \ldots, t_n)$ is given by the individual probabilities:

$$P(\mathbf{x}) = \prod_{i=1}^{n} p(t_i), \quad \forall i, t_i \in \mathcal{V}, \quad \sum_{t \in \mathcal{V}} p(t) = 1 \tag{2.2}$$

The algorithm uses the Expectation-Maximisation (EM) to optimise the probability of each subword, $p(t)$. During each iteration, the EM algorithm maximises the marginal likelihood of the observed data:

$$\mathcal{L} = \sum_{s=1}^{|\mathcal{D}|} \log\left(P\left(X^{(s)}\right)\right) = \sum_{s=1}^{|\mathcal{D}|} \log\left(\sum_{\mathbf{x} \in C_{X^{(s)}}} P(\mathbf{x})\right) \tag{2.3}$$

where $P(\mathbf{x})$ is the probability of a subword sequence $\mathbf{x}$ and $C_{X^{(s)}}$ is the set of all possible decompositions of $X^{(s)}$ in $\mathcal{V}$. The vocabulary is then reduced by iteratively removing subwords that minimally impact the overall likelihood until the desired vocabulary size is achieved. This process is summarised in Algorithm 4.

Algorithm 4 *UnigramLM Training Algorithm*

Input: Corpus of text *corpus* and desired vocabulary size *targetVocabSize*
Output: Vocabulary $\mathcal{V}$ and subword probabilities $p(t)$

procedure TRAINUNIGRAMLM(*corpus*, *targetVocabSize*)
    Initialise seed vocabulary $\mathcal{V}$ with all unique characters and most frequent substrings in *corpus*
    Initialise subword probabilities $p(x)$ for each subword x in $\mathcal{V}$
    while $|\mathcal{V}| >$ *targetVocabSize* **do** ▷ Iteratively reduce vocabulary size
        Use EM algorithm to optimise $p(t)$ given the current $\mathcal{V}$
        Compute $loss_i$ for each subword t_i ▷ Loss indicates the likelihood reduction when t_i is removed
        Sort subwords by $loss_i$
        Retain top $\eta\%$ of subwords ▷ Keep essential subwords, e.g., $\eta = 80$
        Ensure that all single-character subwords are retained to avoid OOV issues
    end while
    return $\mathcal{V}$, $p(t)$ ▷ Returns a vocabulary $\mathcal{V}$ and probabilities for all tokens in $\mathcal{V}$
end procedure

Tokenisation

After training, we obtain a vocabulary $\mathcal{V}$ and $p(t), \forall t \in \mathcal{V}$, allowing us to tokenise a sentence s using UnigramLM parametrisation as follows:

$$\begin{aligned} T(s) :=& \arg\max_{C \in C_s} \left(\log\left(P(C)\right)\right) \\ =& \arg\max_{C \in C_s} \left(\log\left(\prod_{t \in C} p(t)\right)\right) \\ =& \arg\max_{C \in C_s} \sum_{t \in C} \log\left(p(t)\right) \end{aligned} \tag{2.4}$$

where C_s represents all possible decompositions of s in $\mathcal{V}$. However, for efficiency, the Viterbi algorithm (Viterbi, 1967) is used to find the most likely sequence of subwords. Therefore, the UnigramLM tokenisation is parameterised by the probabilities learned for each token $t \in \mathcal{V}$.

Summary of Tokenisers

In this section we reviewed three widely used subword tokenisers algorithms: BPE, WordPiece and UnigramLM. Since no single tokenisation method is perfect for all applications (Mielke et al., 2021), it is crucial to understand the similarities and differences between existing approaches. Table 2.1 provides a concise comparison of these tokenisers.

2.2 Language Models

Language Models are the backbone of modern NLP applications, excelling at tasks involving the understanding or generation of natural language by learning patterns from large corpora of text. The previous section on tokenisers (§2.1) sets the stage for understanding how tokenisation choices impact LMs. Hence, in this section, we explore the fundamentals of language modelling, followed by an overview of transformer architecture, including encoders and decoders.

2.2.1 Language Modelling

Language modelling, the task at the core of LMs, involves learning the structure and patterns within languages by predicting the next word given the previous words in a sequence, serving as a self-supervised objective. This approach is known as causal language modelling (CLM).

Aspect	BPE	WordPiece	UnigramLM
Training Process			
Initial vocabulary	Small, consisting of individual characters or bytes	Small, consisting of individual characters, prefixing mid-word characters with '##'	Large, includes all unique characters and most frequent substrings
Training Goal	Learn merge rules $\mathcal{M}$ and iteratively increase vocabulary $\mathcal{V}$	Learn merge rules $\mathcal{M}$ and iteratively increase vocabulary $\mathcal{V}$	Learn subword probabilities and iteratively reduce vocabulary
Key Difference in Training	Merges most frequent token pairs	Merges token pairs based on a score (Equation 2.1) which favours pairs that occur together more often than would be expected by chance based on their individual frequencies	Uses EM algorithm to maximise likelihood and reduces vocabulary by removing lowest probability tokens
Type of output	$\mathcal{V}$, $\mathcal{M}$	$\mathcal{V}$	$\mathcal{V}$, $p(t), \forall t \in \mathcal{V}$
Tokenisation Process			
Encoding strategy	Applies learned merge rules to text	Longest-match-first search for subwords in vocabulary	Uses Viterbi algorithm to find the most likely subword sequence
Worst-Case time complexity for encoding a word w	$\mathcal{O}(\lvert w\rvert \log(\lvert w\rvert))$, where $\lvert w\rvert$ is the length of the word. This assumes an efficient implementation using priority queue and linked list.	$\mathcal{O}(\lvert w\rvert)$, where $\lvert w\rvert$ is the length of the word	$\mathcal{O}(\lvert w\rvert \cdot \lvert\mathcal{V}\rvert)$, where $\lvert\mathcal{V}\rvert$ is the vocabulary size
Parametrisation	Merge table $\mathcal{M}$	No parametrisation	$p(t) \forall t \in \mathcal{V}$

Table 2.1 Comparison of BPE, WordPiece, and UnigramLM Tokenisation Algorithms.

In a broader sense, language modelling also includes other self-supervised objectives applied to text, such as masked language modelling (MLM) and others. In this study, we focus on CLM and MLM, which are detailed below.

Causal Language Modelling

CLM trains models to predict the next token given previous tokens, useful for text generation tasks (Radford et al., 2019). The joint probability of a token sequence $\mathbf{x} = (t_1, t_2, \ldots, t_n)$ is modelled as:

$$P(\mathbf{x}) = p(t_1) \prod_{i=2}^{n} p(t_i \mid t_1, t_2, \ldots, t_{i-1}) \tag{2.5}$$

Masked Language Modelling

MLM, unlike CLM, masks tokens and trains the model to predict them using bidirectional context, suitable for tasks like classification and question answering (Devlin et al., 2018).

2.2.2 Transformers

The development of the transformer architecture represented a significant advancement in language modelling. It relies on a key component, the *attention mechanism*, which enables efficient parallel processing and effectively captures long-term dependencies (Vaswani et al., 2017). This mechanism allows the model to focus selectively on tokens that are most relevant for understanding context. Specifically, each token from a sequence $\mathbf{x} = (t_1, t_2, \ldots, t_n)$ is mapped to a high-dimensional vector, creating an input matrix X. Before computing the scaled dot-product, X is projected into queries (Q), keys (K), and values (V) using learnable matrices. The scaled dot-product attention is then computed as:

$$\text{Attention}(X) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_{\text{model}}}}\right)V \tag{2.6}$$

An advantage of the transformer architecture is its flexibility to be used as an encoder (left side of Figure 2.1), decoder (right side of Figure 2.1) or encoder-decoder. Given the emphasis of this thesis on encoders and decoders, a brief overview of these components is provided next.

Encoders

The encoders in transformer architecture process an entire sequence at once, using embeddings as input. These embeddings pass through layers of self-attention and feed-forward networks, creating a representation that captures contextual information suitable for tasks like text classification or question answering. Additionally, a common pre-training task for encoders is Masked Language Modelling (MLM) and models like BERT (Devlin et al., 2018) use next sentence prediction (NSP) as an additional task.

Decoders

Decoders are commonly used for autoregressive generation tasks such as text generation, using CLM to generate tokens sequentially. Starting with a special start token (e.g., <s>), decoders use masked self-attention to focus only on previously generated tokens.

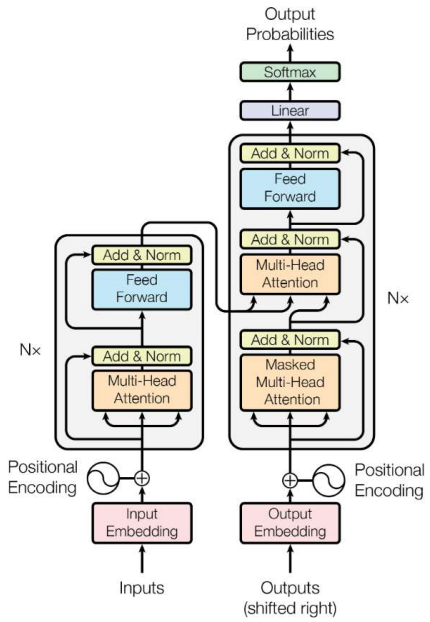


Fig. 2.1 Transformer architecture. Figure sourced from Vaswani et al. (2017).

Importantly, the embeddings may or may not be shared between input and output. From now on, we will denote the entire set of embedding parameters as ϕ, with ϕ_{in} representing the input embeddings and ϕ_{out} the output embeddings, if differentiation is necessary.

To generate a token, the softmax function is applied at each iteration to the output of the dot-product between the final hidden state vector $\mathbf{h}_k$ from the decoder at time step k and the transpose of the output embedding matrix $E_{\phi_{\text{out}}}$. This produces a probability distribution over the vocabulary:

$$\mathbf{p}_k = \text{softmax}(\mathbf{h}_k \cdot E_{\phi_{\text{out}}}^{\top}) \tag{2.7}$$

The resulting probability distribution is used to sample the next token (Ippolito et al., 2019), either by selecting the most likely token or using other strategies like nucleus (Holtzman et al., 2019) or top-k sampling (Fan et al., 2018).

Large Language Models

Large Language Models (LLMs) are transformer-based models that have been scaled up through pre-training on large amounts of text corpora. These models require a large number of parameters, often ranging from hundreds of millions to billions, in order to learn complex patterns and structure in natural language.

Low-Rank Adaptation

Given the large size of these LLMs, it is expensive to fine-tune them through transfer-learning for each downstream task. Low-Rank Adaptation (LoRA) offers a computationally efficient solution by modifying the original model's weight matrices with low-rank updates (Hu et al., 2021). By optimising a small set of parameters, LoRA reduces the risk of overfitting, thus enhancing performance even with limited data, particularly beneficial in the context of low-resource languages.

Formally, LoRA reparametrises the weight matrix $W \in \mathbb{R}^{d\times k}$ by adding low-rank modifications:

$$h = Wx + \alpha BAx \tag{2.8}$$

where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ represent low-rank matrices, with $r \ll \min(d,k)$ and α represents the scaling factor.

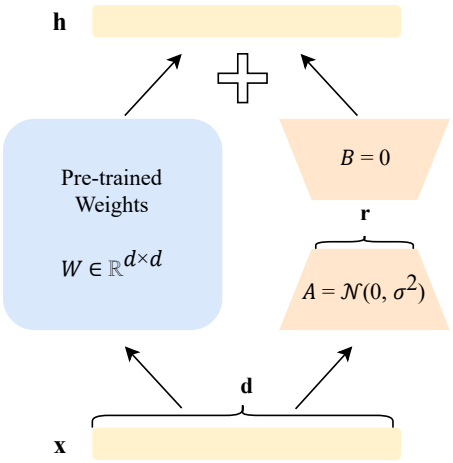


Fig. 2.2 LoRA reparametrisation. Only A and B are trained. Figure reproduced from Hu et al. (2021).

A benefit of LoRA is its capacity to merge adapters, useful for multi-task and transfer learning. Different methods exist for merging: **Linear/Task Arithmetic** (Ilharco et al., 2022) merges adapters with equal ranks using weighted sums; **Concatenation** (Xu et al., 2024; Kong et al., 2024) allows adapters of different ranks to be combined by stacking their matrices; or **TIES** (Yadav et al., 2023) merges adapters by pruning and aligning weights based on majority sign masks.

Importantly, LoRA is just one instance from the parameter-efficient fine-tuning (PEFT) techniques, with others such as prefix tuning and bottleneck adapters also being widely recognised (c.f. Pfeiffer et al., 2023).

2.3 Multilingual Models and Tokenisers

Having presented the details of tokenisation and language models, it is important to address their application and development in a multilingual context. Models such as MBERT — a multilingual version of BERT pre-trained on 104 languages (Devlin et al., 2018), XLM-ROBERTA (Conneau et al., 2020) — pre-trained on 100 languages — or BLOOM — pre-trained on 46 natural languages and 13 programming ones, are trained to learn universal representations across languages. They often use byte-level tokenisers, which not only enable them to handle any language, regardless of the script, but also facilitate knowledge transfer between languages (Yuan et al., 2023).

The performance of these multilingual models heavily depends on their tokeniser, and it is known that these models suffer from "curse of multilinguality," where beyond a certain number of languages, performance decreases on both monolingual and cross-lingual tasks. A major concern is the large vocabulary size required to support multiple languages. For example, XLM-ROBERTA and MBERT use vocabularies of $\approx 250,000$ tokens to cover 100+ languages, averaging ≈ 2500 unique tokens per language. These large vocabularies require a significant allocation of model parameters to embedding layers: between 47% (`base` model) and 71% (`large`) for XLM-ROBERTA, and 52% for MBERT (Chung et al., 2020). In contrast, their monolingual counterparts allocate $14-31\%$ for ROBERTA, and $9-21\%$ for BERT, depending on the model size (i.e., `base` or `large`). Another model, XLM-V, uses a 1M vocabulary that, despite improving performance, takes 93% of the model's parameters and poses challenges with softmax computation and memory requirements (Liang et al., 2023).

This setup raises questions not only about the efficiency of multilingual models, but also their ability to represent all languages equitably. Additionally, research indicates that monolingual tokenisers outperform their multilingual counterparts (Rust et al., 2020). However, transferring to a monolingual tokeniser introduces challenges in generating embeddings for new tokens in $\mathcal{V}_{\text{new}}$. While it is possible to train these embeddings from scratch, this approach can be expensive. An alternative is to initialise new token embeddings through heuristic methods and refine them via continued training (§2.4), or employ advanced techniques that predict embeddings without the need to necessarily continue training (§2.5). The following sections will explore some of these techniques.

2.4 Embedding Initialisation

To transfer an LM to a new tokeniser, different embedding initialisation techniques exist. Let us denote the source tokeniser as $(\mathcal{V}_a, T_a)$ with embedding parameters ϕ_a, and the target tokeniser as $(\mathcal{V}_b, T_b)$ with embeddings ϕ_b. One such approach, Fast Vocabulary Transfer (FVT), initialises each new token t_{new} for all $t_{\text{new}} \in (\mathcal{V}_b \setminus \mathcal{V}_a)$ based on the embeddings of tokens from the source tokeniser that t_{new} decomposes into under T_a (Gee et al., 2024):

$$E_{\phi_b}(t_{\text{new}}) = \frac{1}{|T_a(t_{\text{new}})|} \sum_{t \in T_a(t_{\text{new}})} E_{\phi_a}(t) \tag{2.9}$$

Other approaches (Minixhofer et al., 2022; Liu et al., 2024; Tran, 2020) require auxiliary embeddings, $E_{\phi_{\text{aux}}} : \mathcal{V}_{\text{aux}} \to \mathbb{R}^{d_{\text{aux}}}$, where $\mathcal{V}_{\text{aux}}$ significantly overlaps with $\mathcal{V}_a$ and $\mathcal{V}_b$, i.e., $|\mathcal{V}_{\text{aux}} \cap \mathcal{V}_a| \not\ll |\mathcal{V}_a|$ and $|\mathcal{V}_{\text{aux}} \cap \mathcal{V}_b| \not\ll |\mathcal{V}_b|$. Using $E_{\phi_{\text{aux}}}$, tokens from both $\mathcal{V}_a$ and $\mathcal{V}_b$ are embedded into a unified semantic space. Embeddings for the target vocabulary ϕ_b are then initialised as a weighted average of the source embeddings ϕ_a, where the weights are assigned based on tokens' similarity in the auxiliary embedding space $E_{\phi_{\text{aux}}}$.

A similar method, FOCUS (Dobler and De Melo, 2023), uses an auxiliary embedding space, $E_{\phi_{\text{aux}}}$ — with the requirement that $|\mathcal{V}_{\text{aux}} \cap \mathcal{V}_b| \not\ll |\mathcal{V}_b|$ — to initialise the embeddings of new tokens $t_{\text{new}} \in \mathcal{V}_b \setminus \mathcal{V}_a$. This is done by computing a weighted combination of the embeddings of overlapping tokens from $\mathcal{V}_a \cap \mathcal{V}_b$:

$$E_{\phi_b}(t_{\text{new}}) = \sum_{t \in (\mathcal{V}_a \cap \mathcal{V}_b)} w(t, t_{\text{new}}) \cdot E_{\phi_a}(t) \tag{2.10}$$

where $w(t, t_{\text{new}})$ are weights computed based on the similarity of tokens in the auxiliary embedding space $E_{\phi_{\text{aux}}}$, using cosine similarity and sparsemax (Martins and Astudillo, 2016). Additionally, FOCUS and FVT directly copy the embeddings for overlapping tokens, $E_{\phi_b}(t) = E_{\phi_a}(t)$ for all $t \in \mathcal{V}_a \cap \mathcal{V}_b$. However, recent studies show that copying the overlapping tokens' embeddings is not necessarily the optimal initialisation in the new tokeniser (Minixhofer et al., 2024).

2.5 Embedding Prediction Using Hypernetworks

Instead of relying on heuristics, more advanced methods use neural networks to predict or generate the weights for the (new) token embeddings. This shift from heuristic initialisation to generative models allows for flexible adaptation to new tokenisers, as it uses learned patterns and relationships across the entire vocabulary. Previous work employed neural

networks to predict the embeddings of rare (Schick and Schütze, 2019) or OOV (Pinter et al., 2017) words in traditional word models, an approach later adapted by Schick and Schutze (2020) for BERT models.

A hypernetwork is a specialised neural network that generates the weights (i.e., parameters) for another network (Ha et al., 2016). This concept underpins the previous works, thus, we can view them as embedding prediction hypernetworks. However, the HN proposed by Minixhofer et al. (2024), which is relevant to this thesis, offers a significant advancement. Unlike previous methods that were limited to extending existing tokenisers, this HN allows for the transfer of an LM to any arbitrary, but fixed tokeniser. Additionally, it is objective-agnostic, meaning it can be applied to encoder, decoder, or encoder-decoder models.

This HN is trained by sampling a diverse set of UnigramLM tokenisers, enhancing its ability to generalise well to other tokenisers. The primary objective during training is to find the optimal parameters θ for the hypernetwork H, allowing H_θ to map a given tokeniser $(\mathcal{V}_b, T_b)$ to corresponding embeddings parameters ϕ_b for a given pre-trained LM (c.f. §3.1 from Minixhofer et al., (2024), for more details about HN training). Figure 2.3 illustrates the information flow during this training process.

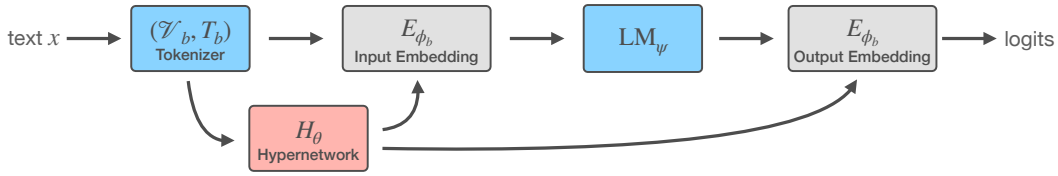


Fig. 2.3 Hypernetwork predicts input $E_{\phi_{b_{\text{in}}}}$ and output $E_{\phi_{b_{\text{out}}}}$ embeddings based on the target tokeniser $(\mathcal{V}_b, T_b)$. Figure sourced from Minixhofer et al. (2024).

Additionally, for each sampled tokeniser, tokens t from $\mathcal{V}_b$ are decomposed using the source tokenisation function T_a. The resulting embeddings, $E_{\phi_a}(T_a(t))$, are fed to the *Hypernetwork Language Model (HLM)* (c.f. §3.2 from Minixhofer et al., (2024), for more details about HLM architecture). The HLM's role is to learn how to compose these embeddings into one, $E_{\phi_b(t)}$. The process is illustrated in Figure 2.4. The trained hypernetwork H can then be used to generate embeddings for any new token t.

2.6 Dynamic Tokenisation Related Work

Previous adjacent works on both encoder and decoder LMs focussed on adapting tokenisers to specific domains or languages by expanding the vocabulary to include relevant tokens (Sachidananda et al., 2021; Balachandran, 2023; Cui et al., 2023; Fujii et al., 2024). New token

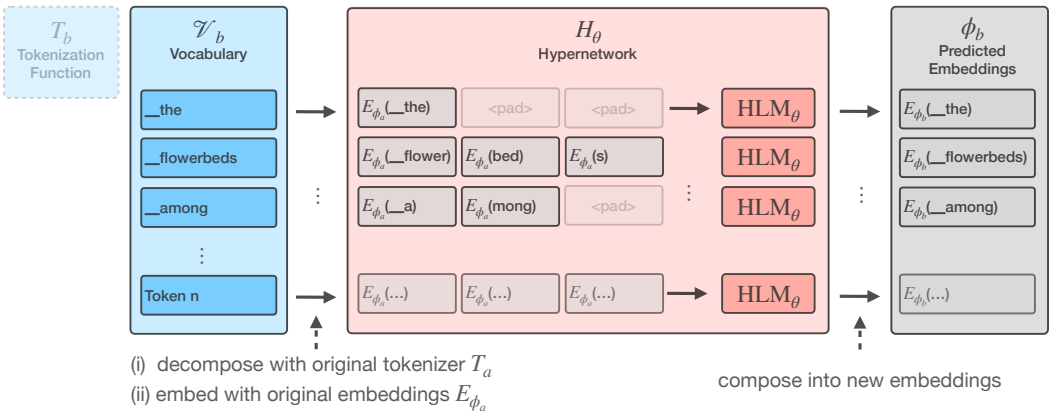


Fig. 2.4 Hypernetwork architecture consisting of an HLM which learns to compose embeddings for each $t \in \mathcal{V}_b$ (target tokeniser) under the original tokenisation T_a into a new embedding, $E_{\phi_b}(t)$, amortising over the target tokenisation function T_b. Figure sourced from Minixhofer et al. (2024).

embeddings are typically obtained via heuristics (Sachidananda et al., 2021), but they often require additional fine-tuning to achieve optimal performance. As the vocabulary size increases, the embedding matrix grows, sometimes accounting for up to 93% of the model's parameters (Liang et al., 2023), limiting the number of new tokens that can be effectively added. While this approach makes tokenisation more adaptable, the need to fine-tune embeddings for each domain or language highlights the requirement for a more dynamic solution.

In contrast, the Copy-Generator (CoG) method by Lan et al. (2023), redefines text generation by copying and pasting text segments (i.e., phrases or words) from an existing large text collection, conditioned on the phrases generated previously. These phrases can be considered as the "vocabulary" for generation, enabling the model to generate multiple tokens at once, instead of relying only on a predefined set of tokens. Moreover, CoG offers training-free adaptation to new knowledge sources or domain-specific text collections. The only requirement for CoG is the offline training of an index, which can include up to a few billion encoded phrases, and a phrase encoder based on a multi-layer perceptron. Another approach similar to CoG is Nearest Neighbor Speculative Decoding or NEST (Li et al., 2024). It selects spans based on a confidence score derived from the retrieval process, and then uses speculative decoding to decide whether to accept the entire span or only parts of it, enabling multiple tokens prediction. However, NEST often generates factual errors due to its reliance on the quality of passage and token retrieval, and it struggles with in-context examples. Similarly, CoG faces limitations in generation diversity and maintains similar latency to Transformers, as finding phrases is computationally expensive with such a large index.

Unlike these methods, which consider multi-token generation, our approach focuses only on the next token generation, as the HN is pre-trained specifically to generate embeddings for tokens rather than entire sentences.

2.7 Summary

Previous sections provided an overview of different subword tokenisation algorithms such as BPE, WordPiece or UnigramLM, and their integration into LMs in both monolingual and multilingual contexts. This highlighted a limitation stemming from LMs' use of static vocabularies and tokenisation functions. In monolingual models, these static elements often lead to a lack of adaptability to evolving language use or domain-specific texts. In multilingual settings, the challenges are exacerbated, as the requirement to handle many languages results in inefficient parameters allocation. Static vocabularies in such contexts struggle to offer fair linguistic coverage, often favouring high-resource languages. This not only reduces the model's effectiveness across diverse languages, but also induces unfair treatment for low-resource ones. These issues underscore the need for a more flexible tokenisation that can update the vocabulary and tokenisation function in real-time, ensuring a fairer language representation and enhancing the adaptability of language models in diverse linguistic environments.

Problem Formulation

Dynamic tokenisation can be defined as the process where the vocabulary $\mathcal{V}$ and tokenisation function T are continuously updated or adapted based on the input text's characteristics. This contrasts with static tokenisation, where $\mathcal{V}$ and T remain unchanged post-training. Dynamic tokenisation allows real-time updates to $\mathcal{V}$ and T, during *inference* or *fine-tuning*, to better adapt to different languages or domain-specific terms.

More formally, let the initial tokeniser be $(\mathcal{V}_{\text{init}}, T_{\text{init}})$. As the LM operates with new text data $\mathcal{D}$, the tokenisation function T_{init} is updated to T_{new}, which aims to provide a more compact representation for data $\mathcal{D}$. The update process can be represented by the function $\mathcal{U}$:

$$T_{\text{new}}(\mathcal{D}) = \mathcal{U}(T_{\text{init}}(\mathcal{D})) \tag{2.11}$$

The following chapter provides details regarding our methodology for dynamic tokenisation.

Chapter 3

Methodology

This chapter presents the methodology we used for retrofitting language models (LMs) with *dynamic tokenisation*. It is organised into two main sections: the first describes the methodology for encoder LMs (§3.1), and the second covers decoder models (§3.2). The architectural differences between encoders and decoders (§2.2.2) guided the separate methodologies: encoders are typically used for text understanding, and decoders for text generation, allowing us to apply dynamic tokenisation at various levels within the system's architecture.

3.1 Dynamic Tokenisation with Encoders LMs

As discussed in the previous chapters, dynamic tokenisation changes the traditional static encoding process by adaptively adjusting token boundaries based on the input text (see Figure 1.3). To retrofit or adapt an LM pre-trained with subword tokenisation to dynamic tokenisation, we need to perform two steps: *(1)* decide on a tokenisation; and *(2)* obtain the token embeddings.

3.1.1 Decide on a Dynamic Tokenisation

Let $\mathcal{D}$ represent the input data to be tokenised. For the first step, we introduce the update function $\mathcal{U}$, which dynamically maps the tokens obtained under the initial tokenisation scheme, T_{init}, to tokens derived from a new tokenisation function, T_{new}. This aims to reduce over-segmentation of the subwords, thus resulting in a more compact representation for $\mathcal{D}$, where $|T_{\text{init}}(\mathcal{D})| > |T_{\text{new}}(\mathcal{D})|$.

Given that during fine-tuning or inference phases, the LM operates at the *batch-level*, $\mathcal{U}$ is specifically applied at this level on $\mathcal{D}_{\text{batch}}$. This approach is important as it allows $\mathcal{U}$ to dynamically adapt the tokenisation to the unique linguistic features in each batch. The goal

of $\mathcal{U}$ is to provide a tokenisation that reduces over-segmentation for rare or out-of-vocabulary (OOV) words while being more equitable across different languages.

To achieve this, we define $\mathcal{U}\left(T_{\text{init}}(\mathcal{D}_{\text{batch}})\right)$ in a similar way to the BPE approach. Specifically, we merge tokens in $T_{\text{init}}(\mathcal{D}_{\text{batch}})$ using a process comparable to the one used in BPE training (Algorithm 1). However, unlike the traditional BPE that identifies the best pair of tokens to merge from a fixed corpus, our method adapts BPE to work dynamically within each batch. More precisely, for each batch, $\mathcal{D}_{\text{batch}}$, tokenised under the initial tokenisation $T_{\text{init}}\left(\mathcal{D}_{\text{batch}}\right)$, we initialise the vocabulary $\mathcal{V}_{\text{new}}$ with all the unique subword tokens in $T_{\text{init}}\left(\mathcal{D}_{\text{batch}}\right)$ — this is different than traditional BPE which starts with a character- or byte-level vocabulary. We then perform a predetermined number of merge operations, m, to iteratively combine the most frequent adjacent tokens within the batch.

We formally define the update function $\mathcal{U}$ as:

$$\mathcal{U} : \left(T_{\text{init}}\left(\mathcal{D}\right), m\right) \rightarrow T_{\text{new}}\left(\mathcal{D}\right) \tag{3.1}$$

where m represents the number of merge operations to perform.

In our approach, the "training" data that we use for the BPE-inspired merging process, $T_{\text{init}}\left(\mathcal{D}_{\text{batch}}\right)$, is the same as the data we aim to tokenise, and, by applying each merge sequentially, we are implicitly tokenising the batch data under the new tokenisation, $T_{\text{new}}\left(\mathcal{D}_{\text{batch}}\right)$. The implication of this is that we can combine both the training and tokenisation algorithms of the original BPE into a single algorithm. In addition to this, since the new tokenisation is applied only to this specific batch data $\mathcal{D}_{\text{batch}}$, and not used for any other texts, we can avoid storing and computing the new merge rules $\mathcal{M}_{\text{new}}$ as well as the new vocabulary $\mathcal{V}_{\text{new}}$. This simplified version of the process is outlined in Algorithm 5.

Algorithm 5 *Dynamic Tokenisation Algorithm*

Input: Tokenised batch data *tokenisedBatch* under initial tokenisation, $T_{\text{init}}\left(\mathcal{D}_{\text{batch}}\right)$; number of merges m
Output: Tokenised batch data under a new, dynamically learned tokenisation, $T_{\text{new}}\left(\mathcal{D}_{\text{batch}}\right)$,

procedure APPLYDYNAMICTOKENISATION(*tokenisedBatch*, m)
    for $i \leftarrow 1$ **to** m **do** ▷ Perform m merge operations based on batch data
        pairFreqs ← ComputePairFrequencies(*tokenisedBatch*)
        bestPair ← GetMostFrequentPair(*pairFreqs*) ▷ Identify the most frequent adjacent pair
        Apply *bestPair* merge rule to *tokenisedBatch* ▷ This modifies the batch in place
    end for
    return *tokenisedBatch* ▷ This represents the initial batch data $\mathcal{D}_{\text{batch}}$ under the new tokenisation, $T_{\text{new}}\left(\mathcal{D}_{\text{batch}}\right)$
end procedure

Importantly, in our approach, we start with a tokenisation at subword-level — the tokens from the initial tokenisation T_{init}. This **subword-level** is our starting point, or the **lower-bound** for the new tokenisation, T_{new}, obtained when $m = 0$. On the other hand, we consider the **word-level** (or pre-tokens) as the **upper-bound** for T_{new}. In other words, we do not want to merge two adjacent tokens which are part of different words such as `_the` and `_computer`. Instead, we want to merge subwords like `_computer` and `s` into `_computers`. In these examples, '_' precedes tokens that represent the beginning of new words, maintaining clear word boundaries.

Table 3.1 illustrates a more concrete example of how dynamic tokenisation is applied on a batch of size 4, after one merge (i.e., $m = 1$).

Sample ID in $\mathcal{D}_{\text{batch}}$	Initial Tokenisation (T_{init})	After 1 merge (T_{new} after m=1)
1	['_Under', **'tak'**, **'ing'**, '_task', 's']	['_Under', **'taking'**, '_task', 's']
2	['_Breath', **'tak'**, **'ing'**, '_views']	['_Breath', **'taking'**, '_views']
3	['_Over', **'tak'**, **'ing'**, '_the', '_car']	['_Over', **'taking'**, '_the', '_car']
4	['_Algo', 'rit', 'hm', 's', '_solve', '_problems']	['_Algo', 'rit', 'hm', 's', '_solve', '_problems']

Table 3.1 Batch-level dynamic tokenisation showing initial tokenised text and the result after applying dynamic tokenisation with one merge ($m = 1$). The token pair `('tak', 'ing')` is the most frequent and has been merged. The tokens are obtained using XLM-RoBERTa tokeniser.

Finally, the choice of using a BPE-inspired merging approach is motivated by our specific scenario: we want to start with a vocabulary consisting of the subword tokens in the batch and expand it with new tokens. This rules out UnigramLM due to its requirement for a large initial vocabulary (§2.1.4), that is subsequently reduced by removing tokens. While WordPiece is a feasible option, with the only difference being its scoring mechanism (shown in Equation 2.1), both methods implicitly tokenise the $\mathcal{D}_{\text{batch}}$, thus removing any concerns regarding tokenisation time complexity (see Table 2.1). Given that no tokenisation method is perfect (Mielke et al., 2021), we opted for BPE in this study and suggest exploring WordPiece as a potential future experiment.

3.1.2 Obtain Token Embeddings

The second step of our approach is to obtain the embeddings for all the tokens in the batch. These embeddings are required to convert the discrete tokenised text into continuous

vector representations, enabling the pre-trained network to perform effective and efficient computations.

After mapping the tokens from the initial tokenisation to a more compact tokenisation, $T_{\text{init}}(\mathcal{D}_{\text{batch}}) \rightarrow T_{\text{new}}(\mathcal{D}_{\text{batch}})$, we need to obtain the embeddings for all tokens $t \in T_{\text{new}}(\mathcal{D}_{\text{batch}})$. While Minixhofer et al. (2024) showed that the HN can be used to transfer an LM to a fixed tokeniser, our work repurposes it as a means to achieve dynamic tokenisation. This adaptation allows the HN to generate the embedding parameters for any token, regardless of its presence in the $\mathcal{V}_{\text{init}}$. Although we could use the original embeddings, $E_{\phi_{\text{init}}}$, for tokens that remain unchanged during dynamic tokenisation (since they are part of $\mathcal{V}_{\text{init}}$), Minixhofer et al. (2024) showed that copying these tokens' embeddings is not necessarily the optimal initialisation in the new tokeniser.

Therefore, for each $t \in T_{\text{new}}(\mathcal{D}_{\text{batch}})$, we apply the hypernetwork H_θ to obtain its embedding:

$$E_{\phi_{\text{new}}}(t) = H_\theta(t), \quad \forall t \in T_{\text{new}}(\mathcal{D}_{\text{batch}}) \tag{3.2}$$

where $E_{\phi_{\text{new}}}(t)$ is the embedding for token t.

We can also view this process as transferring the LM to a new tokeniser $(\mathcal{V}_{\text{new}}, T_{\text{new}})$ for each batch, dynamically adjusting token boundaries based on the specific data within that batch. Figure 3.1 illustrates the information flow for dynamic tokenisation applied to encoders.

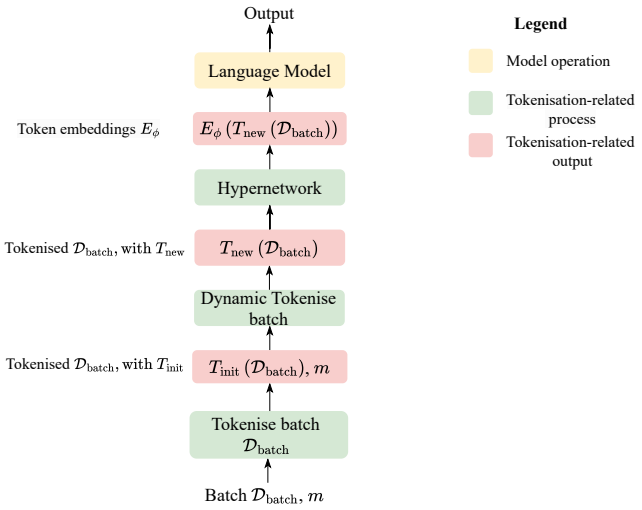


Fig. 3.1 Dynamic tokenisation applied to encoder LMs.

The use of the HN from Minixhofer et al. (2024) is motivated by their results, which show that the performance is approximately preserved without further training the embedding layer.

Besides that, the HN can be applied to obtain embeddings for any token, regardless of its presence in the initial vocabulary $\mathcal{V}_{\text{init}}$, making it feasible to apply dynamic tokenisation in real-time. Other methods for initialising token embeddings, such as FOCUS (Dobler and De Melo, 2023), WECHSEL (Minixhofer et al., 2022), OFA (Liu et al., 2024), RAMEN (Tran, 2020), or FVT (Gee et al., 2024), require additional training of the embedding layer, making them unsuitable for our application, especially during inference where real-time processing is required. In addition to this, the approach used by Schick and Schutze (2020), while using an HN to predict embeddings, it only works for words already in the vocabulary, rather than handling OOV tokens, a key requirement for our approach.

3.2 Dynamic Tokenisation with Decoders LMs

Theoretically, the same dynamic tokenisation procedure used for encoder LMs can be applied to decoders. This involves expanding the initial vocabulary $\mathcal{V}_{\text{init}}$ with the new tokens obtained when encoding the input with dynamic tokenisation, resulting in $\mathcal{V}_{\text{init}} \cup \mathcal{V}_{\text{new}}$, used to encode the data and generate the next token. We then compute a probability distribution over the tokens in $\mathcal{V}_{\text{init}} \cup \mathcal{V}_{\text{new}}$, following the procedure outlined in §2.2.2. Figure 3.2 illustrates how dynamic tokenisation *could* be applied to decoders.

However, while encoders typically operate at batch-level, decoders are mostly used for autoregressive tasks like text generation or completion (e.g., chatbots), and therefore **do not** generally operate at the batch-level. Based on this, we introduce a method for decoders that operates without requiring data in batches, unlike the approach used for encoders.

There are several applications of decoder LMs that could benefit from dynamic tokenisation:

1. In tasks where the LM retrieves documents such as **Retrieval-Augmented Generation (RAG)**, dynamic tokenisation can adapt to the specific content of the retrieved documents and the prompt. This is especially useful in domains with technical terms that a static tokeniser might over-segment (Ding et al., 2024);

2. **Chatbot Applications**: Particularly useful in dynamic environments like customer service, dynamic tokenisation can adjust to the evolving topics of discussion and the unique vocabularies of different users. This adaptability is also advantageous in multilingual contexts to prevent over-segmentation (Rainey et al., 2023);

3. In tasks such as **summarisation**, dynamic tokenisation can help adapt token boundaries based on the input document to be summarised, which is beneficial for handling domain-specific terms effectively.

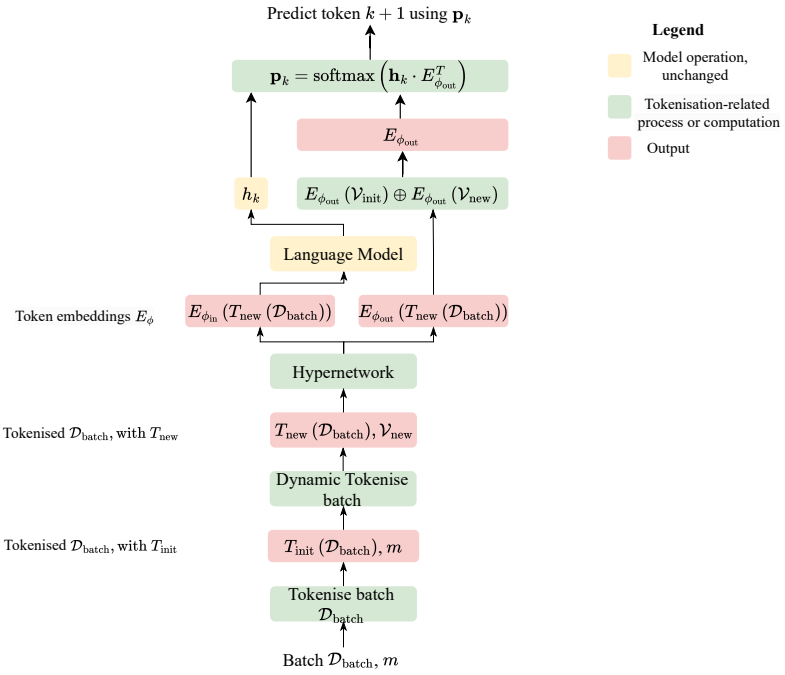


Fig. 3.2 Dynamic tokenisation approach that can *theoretically* be applied to decoder LMs to predict token $k+1$. $\oplus$ represents concatenation.

While dynamic tokenisation (as applied to encoders) could theoretically enhance decoder performance in these applications by adapting to specific input data, its effectiveness is bounded by the vocabulary used. Specifically, it restricts the LM to predict and generate new tokens only from $\mathcal{V}_{\text{init}} \cup \mathcal{V}_{\text{new}}$, where $\mathcal{V}_{\text{new}}$ is determined based on the input data. To overcome this limitation, we propose expanding the vocabulary to a larger scale.

In our method, we use an approach that, while similar to CoG (Lan et al., 2023) and NEST (Li et al., 2024) in its training-free adaptation to a domain or language, differs by using a large vocabulary of tokens rather than a phrase table. The need for a phrase table, typically on the order of billions of phrases, is significantly reduced when using a token vocabulary. We *significantly* expand the initial vocabulary, resulting in $\mathcal{V}_{\text{large}}$, aiming to include more specialised terms and word variations in English. This expansion increases the granularity of the tokens from subword-level, closer to word-level, and aligns with our goal for encoders LMs. Consequently, this results in shorter token sequences, improving the inference speed. While this expanded vocabulary is statics in nature, it offers similar advantages to a dynamic one, and more importantly, the integration of the LM together with such a large vocabulary is possible by using token embeddings generated by the HN from Minixhofer et al. (2024). Unlike traditional methods, these embeddings do not necessarily require additional training. More details on this can be found in §4.5.2. Importantly, although the current vocabulary is

static, it sets the foundation for future adaptations that could include dynamic vocabulary adjustments, which are not explored in this study.

Therefore, our approach to adapt a decoder LM pre-trained with subword tokenisation to a more dynamic tokenisation, requires a few steps: *(1)* expand the vocabulary to a large size; *(2)* decide on a tokenisation; *(3)* obtain token embeddings and index construction.

3.2.1 Vocabulary Expansion

In the first step, we aim to expand the initial vocabulary of a decoder LM, $\mathcal{V}_{\text{init}}$, to a significantly larger vocabulary, $\mathcal{V}_{\text{large}}$. To achieve this, we can apply one of the widely used subword tokenisers such as BPE, WordPiece or UnigramLM on a large corpus to obtain a vocabulary of $|\mathcal{V}_{\text{large}}| - |\mathcal{V}_{\text{init}}|$ tokens.[1] In our method, we use BPE algorithm to find $\mathcal{V}_{\text{new}}$ and $\mathcal{M}_{\text{new}}$. We then obtain $\mathcal{V}_{\text{large}} = \mathcal{V}_{\text{init}} \cup \mathcal{V}_{\text{new}}$.

3.2.2 Decide on a Tokenisation Function

The next step is to decide on a tokenisation function to use with the new vocabulary $\mathcal{V}_{\text{large}}$. Although we have obtained the merge rules $\mathcal{M}_{\text{new}}$, these rules are only valid for encoding text from $\mathcal{V}_{\text{new}}$. The source tokeniser, $(\mathcal{V}_{\text{init}}, T_{\text{init}})$, used by the LM could be BPE, WordPiece, UnigramLM, or another type, making it challenging to merge it with the new one $(\mathcal{V}_{\text{new}}, T_{\text{new}})$. Even if both tokenisers use BPE, combining their merge tables, $\mathcal{M}_{\text{init}}$ and $\mathcal{M}_{\text{new}}$, is challenging because the rules in each table are stored in the order they were learned. There is no guarantee that merging these tables, even with more complex methods than simple concatenation (e.g., frequency-based merging) can encode all the tokens in $\mathcal{V}_{\text{large}}$. Additionally, even if the two tokenisers are trained on the same corpus, small modifications in normalisation or pre-tokenisation steps can result in the two merge tables not being able to be combined without risking the failure to encode all tokens in $\mathcal{V}_{\text{large}}$. Table 3.2 shows an example where merging two BPE tokenisers results in conflicts.

These challenges highlight the complexity involved in merging tokenisers and the need for a tokenisation function that facilitates merging. To address this, we use a **Longest-Prefix** (LP) matching tokenisation function, denoted T_{LP}, similar to the default method used by WordPiece when the continuation prefix is set to `blank` (i.e., no character, see §2.1.3). This was motivated by the work of Uzan et al. (2024), who empirically showed that LP greedy tokenisation performs surprisingly well compared to other tokenisation functions.

[1] In practice, the SentencePiece package (Kudo and Richardson, 2018) is often used, particularly for low-resource languages, because it works without the need for pre-tokenised input

	Tokeniser 1	**Tokeniser 2**	**Merged Tokeniser**
Initial Vocabulary	a, b, c, d, e	a, b, c, d, e	a, b, c, d, e
Merge Tables			
Rule 1	‘a’, ‘b’ → ‘ab’	‘a’, ‘d’ → ‘ad’	‘a’, ‘b’ → ‘ab’
Rule 2	‘ab’, ‘c’ → ‘abc’	‘ad’, ‘e’ → ‘ade’	‘ab’, ‘c’ → ‘abc’
Rule 3	‘d’, ‘e’ → ‘de’	‘b’, ‘c’ → ‘bc’	‘d’, ‘e’ → ‘de’
Rule 4	-	-	‘a’, ‘d’ → ‘ad’
Rule 5	-	-	‘ad’, ‘e’ → ‘ade’
Rule 6	-	-	‘b’, ‘c’ → ‘bc’
New Vocabulary	a, b, c, d, e, ab, abc, de	a, b, c, d, e, ad, ade, bc	a, b, c, d, e, ab, abc, de, ad, ade, bc
Example tokenise: ‘ade’			
Step 1	[‘a’, ‘d’, ‘e’]	[‘a’, ‘d’, ‘e’]	[‘a’, ‘d’, ‘e’]
Step 2	-	[‘ad’, ‘e’]	[‘a’, ‘de’]
Step 3	-	[‘ade’]	-

Table 3.2 Example illustrating how combining merge rules from two BPE tokenisers results in conflicts when tokenising “ade”.

3.2.3 Obtain Token Embeddings and Index Construction

As it was the case for encoder LMs, we use the same hypernetwork (pre-trained on the decoder LM) from Minixhofer et al. (2024) to obtain token embeddings for all the tokens $t \in \mathcal{V}_{\text{large}}$, using Equation 3.2, with $\mathcal{V}_{\text{large}}$.

However, for decoders, the flow for generating the next token is as follows:

Step 1: Input Tokenisation Tokenise the textual prompt $\mathbf{x}$: $T_{\text{LP}}(\mathbf{x})$;

Step 2: Input Embeddings Obtain the input embeddings for each token in $T_{\text{LP}}(\mathbf{x})$ using the hypernetwork: $E_{\phi_{\text{in}}}(T_{\text{LP}}(\mathbf{x}))$;

Step 3: LM Processing Forward the tokenised input to the LM;

Step 4: Output Embeddings Obtain the output embeddings for each token in the vocabulary $\mathcal{V}_{\text{large}}$: $E_{\phi_{\text{out}}}(\mathcal{V}_{\text{large}})$;

Step 5: Compute Probability Distribution Compute the probability distribution over the vocabulary $\mathcal{V}_{\text{large}}$ using the output embeddings and the last hidden state $\mathbf{h}$:

$$\mathbf{p} = \text{softmax}\left(\mathbf{h} \cdot E_{\phi_{\text{out}}}(\mathcal{V}_{\text{large}})^{\top}\right).$$

Step 6: Sample Next Token Sample the next token from the probability distribution $\mathbf{p}$.

The computational bottleneck with this process is in Step 5, where the dot product is calculated before applying softmax. Specifically, this step requires calculating the dot product between the last hidden state vector $\mathbf{h}$, which has a shape of $(1, d_{\text{model}})$, and the transposed output embeddings matrix $E_{\phi_{\text{out}}}\left(\mathcal{V}_{\text{large}}\right)^{\top}$, which has a shape of $(d_{\text{model}}, |\mathcal{V}_{\text{large}}|)$. This operation is computationally expensive due to the large size of the vocabulary $\mathcal{V}_{\text{large}}$, as we know that $|\mathcal{V}_{\text{large}}|$ is significantly larger than $|\mathcal{V}_{\text{init}}|$.

To address this issue, we use an index that supports approximate nearest neighbours (ANN) search, similar to the methods employed by CoG (Lan et al., 2023) and NEST (Li et al., 2024). This involves training an index on the output embeddings of all tokens in the vocabulary, $E_{\phi_{\text{out}}}\left(\mathcal{V}_{\text{large}}\right)$, a process that can be performed offline since the tokens in $\mathcal{V}_{\text{large}}$ are known in advance.

The index then enables efficient and effective nearest neighbour searches. We can use the last hidden state $\mathbf{h}$ as a query to find the k closest tokens. This modifies Step 5 by computing the dot product only between $\mathbf{h}$ and the embeddings of the top k closest tokens, denoted as $E_{\phi_{\text{out}}}(\mathcal{I}_k(\mathbf{h}))$, where $\mathcal{I}_k(\mathbf{h})$ represents the k nearest tokens, and k is typically a low value.

Using an ANN significantly reduces computational costs because, *during generation*, the hypernetwork only needs to generate embeddings for the k nearest tokens rather than for all tokens in $|\mathcal{V}_{\text{large}}|$. Note that the hypernetwork generates embeddings for all tokens in $\mathcal{V}_{\text{large}}$ only during index training, and this is not needed anymore during generation.

By using an ANN index with the hypernetwork, we maintain the same number of parameters in the LM. Thus, our approach avoids the issue seen in other models with large vocabularies — especially in multilingual contexts — where the embedding layer can account for up to 93% of the total model parameters (Liang et al., 2023). In our method, the LM keeps its original embedding matrix for the initial vocabulary, while the hypernetwork dynamically generates the required token embeddings in real-time. These embeddings can be either output embeddings determined by the index $\mathcal{I}$ or input embeddings for the predicted next tokens. Thus, instead of having an embedding matrix of size $|\mathcal{V}_{\text{large}}| \times d_{\text{model}}$, we have the initial embedding matrix of size $|\mathcal{V}_{\text{init}}| \times d_{\text{model}}$. This setup supports the use of very large vocabularies with the LM and facilitates adaptation to specific languages or domains without continued training of the LM. The only requirement are: a language- or domain-specific vocabulary and an index $\mathcal{I}$ based on this vocabulary. Figure 3.3 illustrates the updated flow with the ANN.

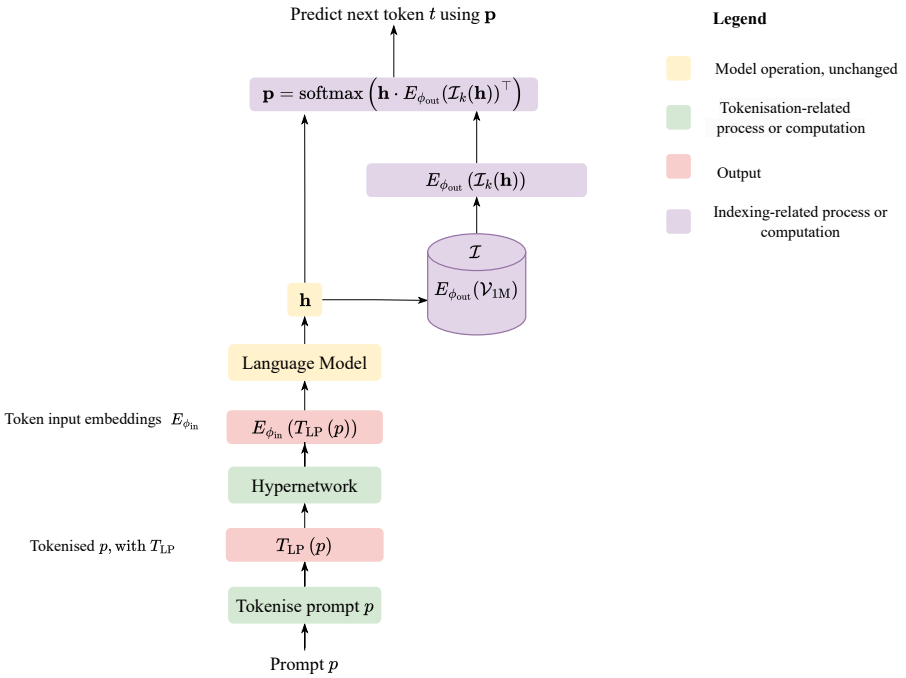


Fig. 3.3 Dynamic tokenisation with expanded vocabulary $\mathcal{V}_{\text{large}}$ and ANN index $\mathcal{I}$ applied to decoder LMs.

3.3 Summary

In this chapter, we have detailed the methodology for retrofitting LMs with dynamic tokenisation. For encoder LMs, we introduced a batch-level dynamic tokenisation method, applying the function $\mathcal{U}$ to perform a BPE-inspired merging of subword tokens within each batch, reducing over-segmentation and improving efficiency. In our approach, we repurposed the hypernetwork from Minixhofer et al. (2024) for dynamic tokenisation, allowing us to generate the embeddings for all the tokens in the batch. In contrast, our approach for decoder LMs operates at the **sample-level**, expanding the initial vocabulary $\mathcal{V}_{\text{init}}$ to a significantly larger vocabulary $\mathcal{V}_{\text{large}}$ and using LP tokenisation. To optimise the next token generation, we integrated an index that supports ANN search, significantly reducing the computational associated with generating the next token with $\mathcal{V}_{\text{large}}$. The hypernetwork from Minixhofer et al. (2024) was used to generate both input $E_{\phi_{\text{in}}}$ and output $E_{\phi_{\text{out}}}$ embeddings, during tokenisation and generation. This approach maintains the model's parameter allocation while enabling it to handle very large vocabularies and adapt to specific languages or domains without requiring continued training for the LM. The adaptation is possible by dynamically updating the index based on domain-specific terms or the course of the conversation. While this is not explored in our work, it represents the ultimate goal of dynamic tokenisation.

Chapter 4

Experimental Setup

This chapter presents the experimental setup used in this project including the models (§4.1), benchmarks (§4.2, §4.3, §4.4), experiments on encoders (§4.5.1) and decoders (§4.5.2), and evaluation metrics (§4.6).

4.1 Models

We selected XLM-RoBERTa (Conneau et al., 2020) (XLM-R) as a multilingual encoder-style LM and MISTRAL-7B as the decoder-style LM specifically because pre-trained hypernetworks (HNs) are available for these models.[1] We use the base version of XLM-R and the first version (i.e., `v0.1`) of MISTRAL-7B (Jiang et al., 2023) (both base and instruct models) to match the versions used for hypernetwork training, ensuring compatibility in our experiments.

4.2 Benchmarks for Encoder LMs Experiments

For our encoder LMs experiments we use two datasets: Cross-lingual Natural Language Inference (XNLI) and Universal Named Entity Recognition (UNER). The decision to use these datasets was motivated by their multilingual nature, with XNLI focusing on *sentence-level* understanding and UNER on *token-level*.

Cross-lingual Natural Language Inference

The XNLI dataset (Conneau et al., 2018) is a multilingual extension of the Multi-Genre Natural Language Inference (MultiNLI) dataset (Williams et al., 2017), designed to evaluate

[1] Available at: github.com/bminixhofer/zett

cross-lingual sentence understanding. In this task, models are trained in a source language and then evaluated in different target languages. The XNLI dataset includes 5010 test and 2490 development sentence pairs (i.e., premises and hypotheses), covering 15 languages, including Swahili and Urdu, two lower-resource languages.

Universal Named Entity Recognition

The UNER (Mayhew et al., 2023) is another multilingual dataset designed to evaluate and improve performance of Named Entity Recognition (NER) task. It covers 13 genealogically and typologically diverse languages, using three coarse-grained entity types: person (PER), organisation (ORG) and location (LOC).

4.3 Benchmarks for Decoder LMs Experiments

For the decoder LMs experiments, we use two popular benchmarks: Massive Multitask Language Understanding (MMLU) and Multi-Turn Benchmark (MT-Bench).

Massive Multitask Language Understanding (MMLU)

The MMLU benchmark (Hendrycks et al., 2020) is designed to evaluate a model's multitask accuracy. It covers 57 different subjects, including humanities, STEM, social sciences, and more specialised topics such as law, medicine, and computer science. Each subject contains multiple-choice questions, with a total of over 14,000 questions aimed at assessing the model's world knowledge and problem-solving ability.

Multi-Turn Benchmark (MT-Bench)

MT-Bench is designed to evaluate LLMs in multi-turn conversational and instruction-following scenarios. It consists of 80 multi-turn questions across eight categories. It is particularly useful for assessing a model's performance in open-ended tasks where user interaction and instruction-following are important. To assess the performance of the model, LLM-as-a-Judge is used, which reduces the need for human involvement by using stronger LLMs as judges (Chiang and Lee, 2023). In our work, we use corrected version of MT-Bench.[2]

[2]Corrections taken from github.com/InflectionAI/Inflection-Benchmarks

4.4 Training Data

The corpus used for both encoder and decoder LMs is MADLAD-400 (Kudugunta et al., 2024), derived from CommonCrawl and covering 419 languages with 5 trillion tokens in the noisy split and 2.8 trillion tokens in the clean partition. In particular, we use the clean English subset for our experiments, which provides high-quality data. The details of how this dataset is used in our project are discussed below.

4.5 Experiments

Since the methodology for encoders differs from that of decoders, we perform different experiments for each, which we outline below. Appendix B presents a summary of the hyperparameters we used for training different adapters and evaluating decoders LMs.

4.5.1 Encoder LMs

The experiments we perform for encoders aim to evaluate the effectiveness of dynamic tokenisation as well as its impact on the sequence length, which in turn affects inference speed. We conduct all the experiments on the XNLI and UNER datasets. The adapters are trained on English, and their performance is evaluated in a cross-lingual setting. For XNLI, we evaluate on 13 different languages, as in the work of Minixhofer et al. (2024): Arabic, Bulgarian, German, Greek, English, Spanish, French, Hindi, Russian, Swahili, Turkish, Urdu, Vietnamese. Similarly, for UNER, we train our adapters on English, "en_ewt" training split, and evaluate on 4 languages: English, German, Portuguese and Russian.[3]

1) Training a task adapter with original subword tokenisation

Given that LMs — in our case XLM-R — are typically trained on large corpora for general language understanding, task adaptation is required to achieve strong performance on downstream tasks like XNLI or UNER. Therefore, in our first experiment, we train a task-specific LoRA adapter (e.g., on XNLI or UNER) with the original tokenisation (and embeddings), and then evaluate it with different tokenisations:

Original tokenisation with original embeddings This setup evaluates the performance with the same tokenisation settings used during the adapter's training;

[3]The limitation to these languages is due to the HN's training constraints, which has not been pre-trained on the other 9 languages available in UNER.

Original tokenisation with hypernetwork embeddings We apply the update function with $m = 0, \mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$, keeping the original tokenisation (i.e., no merges) but replacing the embeddings with those generated by the hypernetwork. T_{init} represents the initial subword tokenisation;

Word tokenisation with hypernetwork embeddings This is equivalent to applying the update function $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$ with $m \to \infty$ set to obtain word-level (pre-token) granularity. In practice, we pre-tokenise the text using a pre-tokeniser to define word boundaries, instead of specifying m;

Interpolation between subword- and word-level By adjusting m in $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$, we evaluate different token granularities between subword-level (original tokenisation, lower-bound) and word-level (upper-bound). This allows us to evaluate how varying degrees of token granularity impact model performance;

Word tokenisation with Fast Vocabulary Transfer embeddings As a **baseline**, we compare word-level tokenisation using Fast Vocabulary Transfer embeddings (Equation 2.9) to HN embeddings.

In our setup, the batch size not only determines the number of examples processed at the same time, but also influences which subwords are merged by $\mathcal{U}$ to create new tokens. We set the batch size to 32 for all experiments.

2) Joint task and dynamic tokenisation adapter training

Unlike the first experiment which trained an adapter with the original tokenisation, in this experiment we train an adapter on XNLI or UNER with our proposed dynamic tokenisation.The motivation behind this is to assess whether exposing the model to HN embeddings, rather than just the original ones, leads to performance improvements. Importantly, we keep the embedding layer parameters frozen when training the adapter. We explore two approaches to applying dynamic tokenisation during training.

In the ***first approach***, we use a pre-determined number of merges m and apply the function $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$ for each batch during training. However, since the correlation between m and sequence compression varies across languages and datasets (e.g., 140 merges for 100% sequence reduction in XNLI English, 160 for UNER), we use percentage reduction as a metric instead. We train adapters with dynamic tokenisation that reduces sequence length by 0%, 25%, 50%, 75%, and 100% of the maximum possible reduction (in English), where 100% corresponds to the difference between the initial sequence length — with subword-level tokenisation, T_{init} — and the sequence length achieved with word-level tokenisation. For

instance, if the average initial sequence length is 100 and word-level tokenisation reduces it to 80, a 25% reduction corresponds to $\frac{25}{100} \cdot (100 - 80) = 5$ tokens. We hypothesise that training the adapter with dynamic tokenisation will improve previous results by exposing the model to specific token granularities and hypernetwork-generated embeddings.

In the ***second approach***, instead of using a fixed number of merges m and applying the function $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$ consistently across the training batch, we introduce stochasticity into the tokenisation process. Specifically, we explore the impact of sampling different tokenisers from various distributions, such as Uniform, Gaussian, Student's t and Cauchy during training:

- **Uniform Distribution**: We sample the number of merges m uniformly from the range $[0, m_{\text{max}}]$:

$$m \sim \mathrm{U}(0, m_{\text{max}}) \tag{4.1}$$

  m_{max} is determined based on the dataset $\mathcal{D}$ and represents the merge level which yields word-level tokenisation.

- **Gaussian (Normal) Distribution**: We sample m from a Gaussian distribution with mean μ and variance σ^2:

$$m \sim \mathcal{N}(\mu, \sigma^2) \tag{4.2}$$

- **Generalised Student's t-Distribution**: We sample m from a Student's t-distribution with location parameter μ, scale parameter γ and degree of freedom ν:

$$m \sim \mathcal{T}(\mu, \gamma, \nu) \tag{4.3}$$

- **Cauchy Distribution**: We sample m from a Cauchy distribution with location parameter μ and scale parameter γ. This is a special case of $\mathcal{T}$, where $\nu = 1$, yielding heavier tails than $\mathcal{T}$ with $\nu > 1$:

$$m \sim \text{Cauchy}(\mu, \gamma) \tag{4.4}$$

The motivation for choosing these distributions is based on their properties. The uniform distribution, U, has a constant probability density across all possible tokenisation granularities. This ensures that the model is uniformly exposed to every merge level m. On the other side, the Gaussian distribution concentrates its probability mass around the mean μ, allowing for deviations controlled by σ^2. The Student's t-distribution, $\mathcal{T}$, has heavier tails than the Gaussian, increasing the likelihood of sampling more extreme tokenisers. Finally, the Cauchy distribution, a special case of $\mathcal{T}$ with $\nu = 1$, allows for even heavier tails.

In our experiments, we sample a tokeniser per batch rather than a tokeniser for each sample in the batch due to the high computational requirements of the latter. By training the adapter with tokenisations sampled from these distributions, we hypothesise that the model will learn to be more robust to the type of dynamic tokenisation used (i.e., the value of m).

The tokenisation function applied during training is then:

$$\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m), \quad m \sim \mathcal{P} \tag{4.5}$$

where $\mathcal{P}$ represents the distribution from which m is sampled.

We evaluate the performance of XLM-R across all the merge levels between subword- and word-level, using the original tokenisation and embeddings from previous experiment as the baseline.

3) Disentangling task adaptation from tokenisation adaptation

In the previous experiment, we explored training adapters for joint task and dynamic tokenisation adaptation. However, a more modular approach is to disentangle these two and have two adapters: one for task-specific adaptation (e.g., XNLI or UNER) and another trained to adapt to different dynamic tokenisation granularities (e.g., different merge levels). This disentanglement allows for greater flexibility, as the task-specific adapter can be reused with different tokenisation adapters, and the other way round.

Therefore, we train two adapters: one optimised for the target task using the original tokenisation and embeddings, and a tokenisation adapter optimised for dynamic tokenisation independently of the task. To achieve this, we train the tokenisation adapter with the masked language modelling (MLM) objective, using the clean English subset of the MADLAD-400 corpus to ensure that it learns generalisable tokenisation granularities.

We experiment with training the tokenisation adapter in two scenarios: *(1)* using word-level tokenisation to allow specialisation with this granularity, and *(2)* sampling tokenisers from either Uniform or Gaussian distributions to enhance robustness across different tokenisation merge levels, based on insights from previous experiments. As for previous experiments, we use the HN to obtain the embeddings.

We then explore different methods to merge the two adapters, including linear/task arithmetic, concatenation and the TIES method (§2.2.2) to integrate the capabilities of both adapters. Finally, we evaluate the performance of the model with the merged adapter on word-level tokenisation with HN embeddings, using the original tokenisation and embeddings as a baseline.

4.5.2 Decoder LMs

As described in our methodology for decoders (§3.2), we expand the initial vocabulary of MISTRAL-7B, $\mathcal{V}_{\text{init}}$, to a significantly larger vocabulary, $\mathcal{V}_{\text{large}}$. Considering that the Oxford English Dictionary contains around 500,000 English words,[4] we expand the initial vocabulary $\mathcal{V}_{\text{init}}$ to one million (i.e., 1M) entries. We denote this vocabulary as $\mathcal{V}_{\text{1M}}$. This doubled size aims to include more specialised terms and word variations not included in the Oxford dictionary. Furthermore, the size of this vocabulary is significantly larger than those used in previous works on vocabulary expansion, allowing us to increase the granularity of the tokens from subword-level, closer to word-level. We obtain this larger vocabulary by training a BPE tokeniser on the clean English subset of MADLAD-400 (§4.4), setting the size of the desired vocabulary to $\text{1M} - |\mathcal{V}_{\text{init}}|$.

Following the expansion of the vocabulary, we construct an index that supports approximate nearest neighbour (ANN) search. Specifically, we compare two types of indices: ScaNN (Guo et al., 2020) and Faiss (Douze et al., 2024), as they perform well on ANN benchmarks.[5]

The experiments we perform with this new tokeniser and index are on MMLU and MT-Bench.

1) MMLU

This benchmark focusses on multitask understanding rather than generation. Therefore, for each prompt p in the dataset, we encode it using the new tokeniser $(\mathcal{V}_{\text{1M}}, T_{\text{LP}})$, generate the input embeddings, $E_{\phi_{\text{in}}}(T_{\text{LP}}(p))$, and feed these embeddings to the LM. We then apply softmax to obtain the probabilities and select the token with the highest probability among the four possible answers: A, B, C or D. We only compare the probabilities of these four tokens, disregarding other tokens in $\mathcal{V}_{\text{1M}}$, as this is the standard original evaluation for MMLU.

We evaluate MISTRAL-7B on MMLU under three different settings: 0-shot, 5-shot with in-context examples selected from random domains, and 5-shot with in-context examples selected from the same domain as the test prompt. The model's performance is compared with different tokenisation methods to assess the impact of the tokenisation function and vocabulary used:

Original tokenisation, vocabulary and embeddings The original tokeniser $(\mathcal{V}_{\text{init}}, T_{\text{init}})$ and its corresponding embeddings.

[4] Available at: oed.com/information/about-the-oed

[5] Benchmarking results: ann-benchmarks.com

Original tokenisation and vocabulary with HN embeddings The original tokeniser $(\mathcal{V}_{\text{init}}, T_{\text{init}})$ with hypernetwork embeddings $H_\theta(\mathcal{V}_{\text{init}}, T_{\text{init}})$.

LP tokenisation, original vocabulary, HN embeddings The longest prefix (LP) tokenisation function function with the original vocabulary $(\mathcal{V}_{\text{init}}, T_{\text{LP}})$ and hypernetwork embeddings $H_\theta(\mathcal{V}_{\text{init}}, T_{\text{LP}})$.

LP tokenisation, 1M vocabulary, HN embeddings The LP tokenisation function with the expanded 1M vocabulary $(\mathcal{V}_{\text{1M}}, T_{\text{LP}})$ and corresponding hypernetwork embeddings $H_\theta(\mathcal{V}_{\text{1M}}, T_{\text{LP}})$.

The specific prompt templates used in these evaluations are provided in Appendix A.

2) MT-Bench

MT-Bench assesses the model's text generation capabilities given a prompt and since the base model, MISTRAL-7B, is not optimised to follow instructions, we instead use its instruct-tuned variant, MISTRAL-7B-INSTRUCT. To obtain the token embeddings, we use the same HN pre-trained for the base model, as previous work by Minixhofer et al. (2024) has shown that an HN pre-trained for a base model can be applied to its fine-tuned versions. Additionally, during the generation process, we use the index to retrieve the top 10 closest token embeddings, following the process illustrated in Figure 3.3.

We evaluate the model using the same four settings as in MMLU, and we also explore different decoding strategies, including greedy decoding, temperature scaling (Ippolito et al., 2019), top-k sampling (Fan et al., 2018), and repetition penalty (Holtzman et al., 2019). For evaluation, we use GPT-3.5-TURBO-1106 as the judge.

4.6 Evaluation Metrics

Throughout our experiments, we use standard evaluation metrics to assess model performance. In the encoder experiments, we use accuracy to evaluate the XNLI results, and F1-score for UNER. Additionally, for the adapter trained on MLM, we use accuracy to assess its ability to predict masked tokens. For the decoder experiments, we also use accuracy as the primary metric for MMLU, while for MT-Bench we use a score between 0 and 10, with 10 representing the highest possible score. Across all experiments, except MT-Bench, we compute the average token sequence length, as we are interested to see the impact of dynamic tokenisation on sequence length. For index comparison, we use Recall@10 to evaluate retrieval effectiveness.

4.7 Summary

In this chapter, we outlined the experimental setup used in this project, including the models, datasets, benchmarks, experiments and evaluation metrics. We detailed the experiments conducted on encoder models using the multilingual datasets XNLI and UNER, which included *(1)* task-specific adapter training, *(3)* joint task and dynamic tokenisation adapter training, as well as *(3)* the disentangling of task and tokenisation adaptations (§4.5.1). Finally, we detailed the experiments on decoder models, which involved expanding the vocabulary to 1M entries and constructing an index, evaluated using the MMLU and MT-Bench benchmarks (§4.5.2).

Chapter 5

Results and Discussion

In this chapter, we present and analyse the results from different experiments on dynamic tokenisation applied to both encoders (§5.1) and decoders (§5.2), as well as some other results regarding the approximate nearest neighbour indices evaluation and (§5.3.1) caching the hypernetwork (HN) embeddings (§5.3.2).

5.1 Encoder LMs

In this section, we present and analyse the results obtained on the three main experiments on encoder LMs (§4.5.1).

5.1.1 Task Adapter Trained with Original Subword Tokenisation and Embeddings

Table 5.1 and 5.2 show the performance of XLM-R-BASE with different tokenisation and embedding configurations on the XNLI and UNER tasks (§4.2).

Impact of tokenisation method. The experiments demonstrate a trade-off between token granularity, controlled by the number of merges in dynamic tokenisation, $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$, and model performance. Specifically, word-level tokenisation ($m \to \infty$) leads to significant token sequence reductions at the cost of performance. For XNLI, sequence length decreases by 22.5% on average, while accuracy decreases by 2.8%. Similarly, for UNER, sequence length is reduced by 26.4% on average, with a corresponding F1-score decrease of 4.2%. The reductions are relative to the baseline, which uses the original subword tokenisation and embeddings.[1] This novel application of XLM-R to word-level tokenisation addresses

[1]This chapter uses the terms "original tokenisation" and "subword tokenisation" interchangeably, as well as "word-level" and "pre-token-level" tokenisation.

Tokenisation & Embeddings	Language Accuracies (%)													
	ar	**bg**	**de**	**el**	**en**	**es**	**fr**	**hi**	**ru**	**sw**	**tr**	**ur**	**vi**	**Avg.**
(1) original	71.6	**76.5**	**76.9**	75.1	**84.8**	78.0	**78.5**	68.7	74.9	**63.2**	**72.4**	65.4	**73.9**	73.9
(2) original, HN	**71.8**	**76.5**	76.7	**75.7**	84.1	**79.0**	78.2	**69.6**	**75.7**	61.7	72.1	**65.9**	73.7	**74.0**
(3) word, HN	67.1	72.8	74.9	71.5	82.5	77.1	75.6	66.2	72.0	59.2	67.4	64.9	73.4	71.1
(4) word, FVT	64.5	68.9	70.8	68.3	79.7	74.2	71.0	65.2	68.6	54.8	63.3	63.8	73.6	68.2
$\Delta_{\text{Acc. (\%)}}$ (1), (3)	-4.5	-3.7	-2.0	-3.6	-2.3	-0.9	-2.9	-2.5	-2.9	-4.0	-5.0	-0.5	-0.5	-2.8
$\Delta_{\text{Length (\%)}}$ (1), (3)	-31.4	-25.1	-22.8	-33.2	-14.7	-17.3	-17.3	-21.8	-28.2	-28.4	-29.4	-17.5	-5.9	-22.5

Table 5.1 Accuracy on XNLI `validation` split when using LoRA trained on XNLI with original subword tokenisation and embeddings. $\Delta_{\text{Acc. (\%)}}$ represents the absolute change in accuracy between word-level tokenisation with HN embeddings (3) and the baseline (1) which uses original tokenisation and embeddings. $\Delta_{\text{Length. (\%)}}$ represents the average decrease in token sequence length of the word-level tokenisation over the original tokenisation. FVT denotes the embeddings obtained using Fast Vocabulary Transfer (§2.4). Boldface indicates the best result for a language.

Tokenisation & Embeddings	Language F1-score (%)					
	en_ewt	**de_pud**	**pt_bosque**	**pt_pud**	**ru_pud**	**Avg.**
(1) original	**81.6**	78.0	**82.3**	**82.9**	**69.0**	**78.8**
(2) original, HN	80.9	**78.3**	80.8	82.3	68.4	78.1
(3) word, HN	77.0	75.8	77.6	77.3	65.5	74.6
(4) word, FVT	67.2	57.0	58.0	58.4	40.7	56.3
$\Delta_{\text{Acc. (\%)}}$ (1), (3)	-4.6	-2.2	-4.7	-5.6	-3.5	-4.2
$\Delta_{\text{Length (\%)}}$ (1), (3)	-17.6	-30.5	-24.1	-24.2	-35.8	-26.4

Table 5.2 F1-score on UNER when using LoRA adapter trained on UNER with original subword tokenisation and embeddings. The results reported are on the `validation` split for `ewt` and `bosque` datasets, and `test` split for `pud` due to the availability.

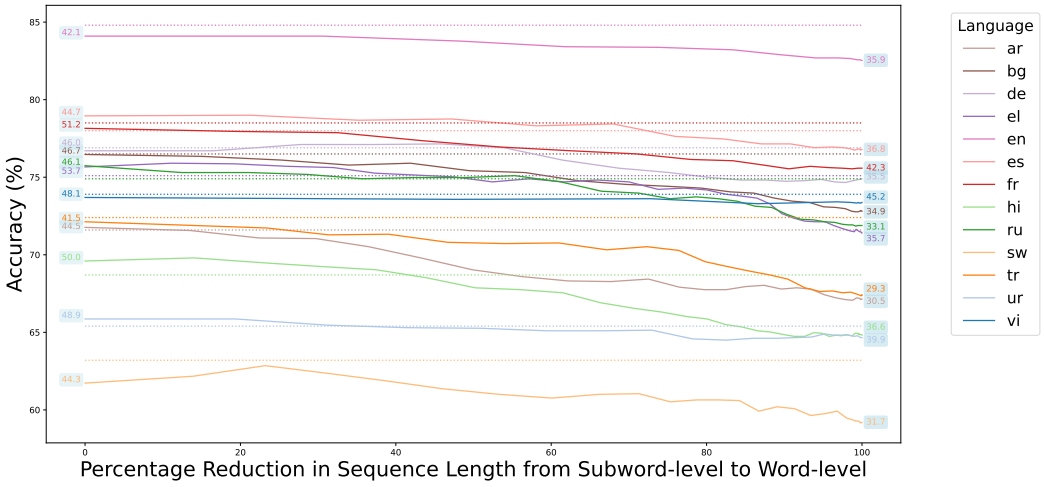
significant challenges such as language imbalance typically seen in subword tokenisers. By achieving relatively minimal losses in accuracy and F1-score, the approach highlights a promising direction towards more equitable language representation (Mielke et al., 2021; Petrov et al., 2024).

Token embeddings comparison. Using original tokenisation with HN embeddings (setting 2) shows comparable results to original embeddings (setting 1), both in English and cross-lingual contexts, highlighting the quality of HN embeddings. However, a noticeable performance gap exists between subword- and word-level HN embeddings (settings 1 and 3). Importantly, FVT embeddings show a significant decrease in performance compared to HN embeddings. For instance, FVT achieves an average accuracy of 68.2% compared to 71.1% with word-level HN embeddings for XNLI, and even more pronounced differences for UNER, with FVT scoring 56.3% F1 compared to 74.6% with HN embeddings. This suggests that HN embeddings better capture the required semantic nuances for tasks like NER, where accurate token representation is important, significantly outperforming FVT.

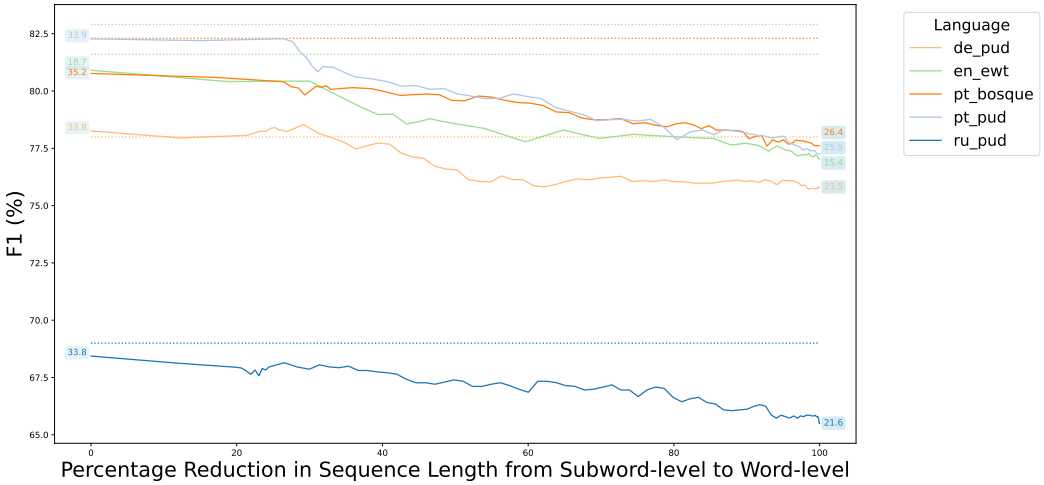
Interpolations between subword- and word-level. Figure 5.1a and 5.1b show the performance trends when applying dynamic tokenisation with different merge levels m. The dotted lines represent the performance upper bound achieved using original tokenisation and embeddings. To account for differences in the merge levels required to achieve word-level tokenisation across languages, the results are presented as relative reduction in sequence length (detailed in §4.5.1). In the XNLI task, most languages maintain a relatively stable accuracy until around 70% sequence reduction, after which a decline occurs. Conversely, in the UNER task, significant performance drops begin at a much lower threshold of around 25% sequence reduction, indicating the sensitivity of token-level tasks — compared to sentence-level — to changes in token granularity. In both tasks, as sequence reduction increases, the gap between the upper bound and dynamic tokenisation results widens. In XNLI, languages like German, Spanish, Swahili, and Urdu show slight performance improvements with a moderate sequence reduction of $15-25\%$, compared to no reduction and HN embeddings.

Language family and sequence reduction. Highly analytic languages like Vietnamese, characterised by a low morpheme-per-word ratio, show smaller reductions. In contrast, synthetic languages, including both agglutinative and fusional types, show significantly greater sequence reductions. This is because these languages tend to synthesise multiple grammatical or conceptual elements into single words using affixes. All the languages (except Vietnamese) in our study are predominantly synthetic — with a few having some analytic characteristics. Additionally, the original vocabulary of XLM-R, contains $\approx 250,000$ subwords from multiple languages, biased towards high-resource languages such as English (Ahia et al., 2023). This bias affects over-segmentation rates differently across languages, leading to less

over-segmentation in languages like English, therefore impacting the sequence reductions when transitioning from subword- to word-level tokenisations. Further experiments could investigate the correlation between the initial language-specific subword distribution in the vocabulary and word-level sequence reductions.



(a) XNLI Accuracies



(b) UNER F1-scores

Fig. 5.1 XNLI accuracy and UNER F1-score trends as the token granularity, controlled by m, shifts from subword-level to word-level across different languages. The continuous lines represent interpolations between these granularities, while the dotted lines indicate the upper boundary of accuracy obtained with the original tokenisation and embeddings. The annotations on the left and right show the average token sequence length with 0% reduction (subword-level) and with 100% reduction (word-level). The evaluation was performed using the task adapter trained on English with original subword tokenisation and embeddings.

5.1.2 Joint Task and Dynamic Tokenisation Adaptation

We now address the performance gap observed between dynamic tokenisation (across all merge levels) with HN embeddings and the original subword tokenisation with original embeddings.

a) Pre-determined Dynamic Tokenisation Level

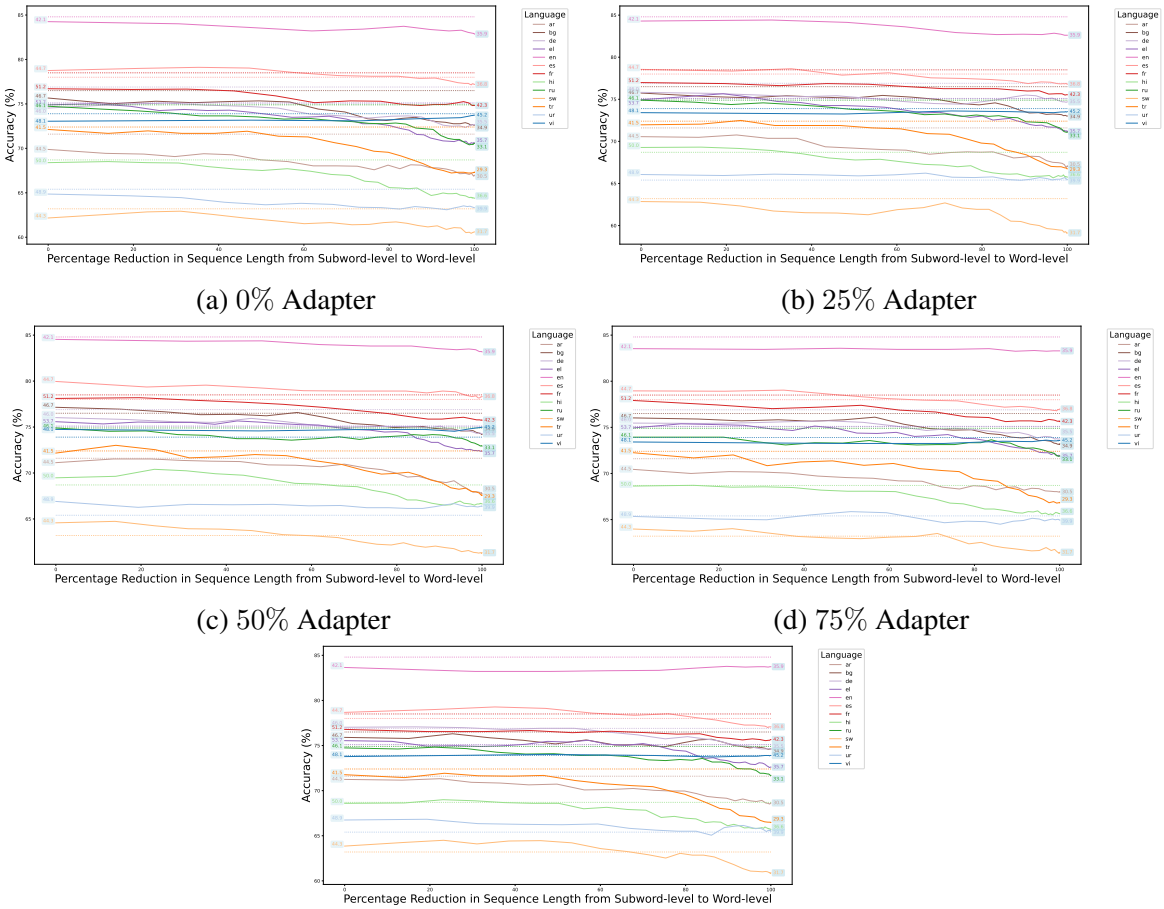
Figure 5.2 and 5.4 illustrate the results obtained by training an adapter on the English split of XNLI/UNER with dynamic tokenisation, applying a pre-determined number of merges. For instance, in the XNLI task, 100% reduction corresponds to 140 merges, $75\% \rightarrow 40$, $50\% \rightarrow 20$, $25\% \rightarrow 10$, and $0\% \rightarrow 0$. Table 5.3 shows the results of these trained adapters when evaluated on word-level tokenisation

	Evaluated on Tokenisation & Embeddings	Adapter	XNLI Accuracy (%)	UNER F1-score (%)
			en	en_ewt
(1)	original, original	original	**84.8**	**81.6**
(2)	word, HN	original	82.5	77.0
(3)	word, HN	0%	82.9	78.7
(4)	word, HN	25%	82.6	79.2
(5)	word, HN	50%	83.2	79.7
(6)	word, HN	75%	83.3	80.5
(7)	word, HN	100%	<u>83.7</u>	<u>80.8</u>

Table 5.3 Performance on XNLI/UNER at word-level using adapter trained on XNLI/UNER with dynamic tokenisation and a pre-determined sequence reduction. Boldface indicates the best results, while the top results with sampled tokenisers for XNLI and UNER are <u>underlined</u>. The rows in grey represent the baselines obtained with the adapter trained on XNLI/UNER with original tokenisation and embeddings.

Training an adapter with dynamic tokenisation minimises the performance gap. For XNLI, the adapter trained with a 50% sequence reduction is the most effective in reducing the performance gap on English, while also transferring well across languages, outperforming the adapter from the previous section (as shown in Figure 5.3). However, some gap remains, particularly as the sequence reduction approaches 100% (i.e., word-level tokenisation). Similarly, for UNER, the 75% sequence reduction adapter is the most effective at closing the gap on English and performs well across languages. However, as for XNLI, some gap remains as sequence reduction approaches 100%. Additionally, the 100% reduction adapter performs comparably but slightly lower results overall. These results demonstrate that training the adapter, while keeping the HN embeddings frozen (i.e., without updating the embedding layer parameters) leads to improved performance across all languages.

Significant improvements for low-resource languages like Urdu and Swahili. The results with the 50% adapter (Figure 5.2c) show that for Urdu, the performance gap is not only closed but 'reversed' across all merge levels. Similarly, for Swahili, the gap is fully closed with up to 70% sequence reduction and 'reversed' with up to 60% reduction. This results to substantial gains in overall sequence reduction, enhancing throughput and advancing equity among language by minimising over-segmentation (Ahia et al., 2023).



(a) 0% Adapter

(b) 25% Adapter

(c) 50% Adapter

(d) 75% Adapter

(e) 100% Adapter

Fig. 5.2 XNLI accuracy trends as the token granularity shifts from subword-level to word-level across different languages. Dotted lines indicate the upper boundary of accuracy obtained with the original tokenisation and embeddings. The evaluation was performed using the task adapter trained on English with dynamic tokenisation, where m was set to achieve **X% relative sequence reduction** on English, and HN embeddings.

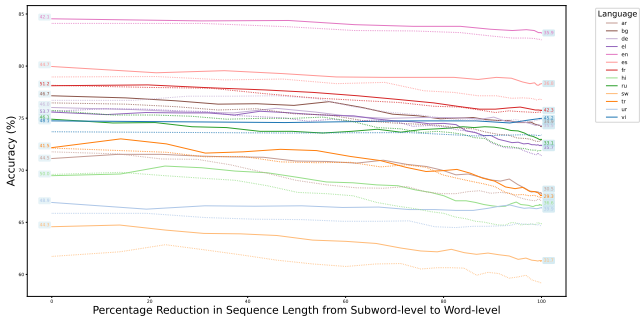
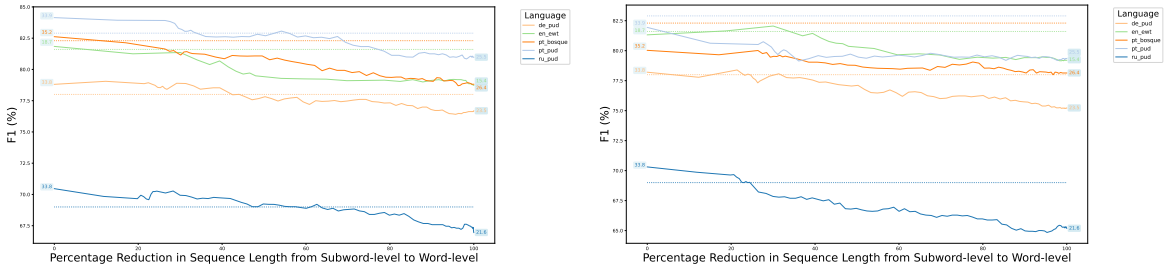


Fig. 5.3 XNLI: Adapter trained with 50% (**continuous line**) compared with the initial adapter trained with subword tokenisation and original embeddings (**dotted line**).



(a) 0% Adapter

(b) 25% Adapter

(c) 50% Adapter

(d) 75% Adapter

(e) 100% Adapter

Fig. 5.4 UNER F1-score trends as the token granularity shifts from subword-level to word-level across different languages. The evaluation was performed using the task adapter trained on English with dynamic tokenisation, where m was set to achieve **X% relative sequence reduction** on English, and HN embeddings.

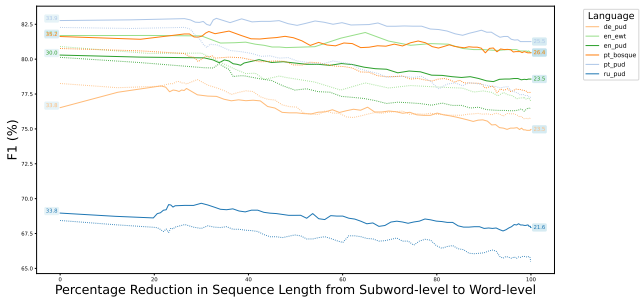


Fig. 5.5 UNER: Adapter trained with 75% (**continuous line**) compared with the initial adapter trained with subword tokenisation and original embeddings (**dotted line**).

b) Sampling Merge Levels

Table 5.4 presents the results of adapters trained using tokenisers sampled from different distributions, evaluated at word-level. For XNLI, the Uniform distribution is the only one that achieves accuracy close to the original (1), with all distributions showing improvement over (2). In contrast, for UNER, the Uniform, Student-t, and Cauchy distributions yield the same result, all significantly outperforming (2). Consequently, Figure 5.6 shows the accuracy trends for the Uniform distribution for XNLI, while Figure 5.7 displays the F1-scores across the three distributions for UNER.

	Tokenisation & Embeddings	Distribution $\mathcal{P}$	XNLI Accuracy (%)	UNER F1-score (%)
			en	en_ewt
(1)	original	-	**84.8**	**81.6**
(2)	word, HN	-	82.5	77.0
(3)	word, HN	Uniform	84.4	80.8
(4)	word, HN	Gaussian	83.7	80.8
(5)	word, HN	Student-t	83.1	79.9
(6)	word, HN	Cauchy	83.1	80.8

Table 5.4 Performance on XNLI/UNER at word-level using adapter trained on XNLI/UNER with dynamic tokenisation and tokenisers sampled from $\mathcal{P}$. The rows in grey represent the baselines obtained with the adapter trained on XNLI/UNER with original tokenisation and embeddings.

Training an adapter with sampled merge levels significantly enhances robustness and performance in English, regardless of the tokenisation strategy. Using a Uniform distribution to sample token granularities almost completely closes the performance gap on English, as seen in the XNLI task where accuracy remains stable across all merge levels (Figure 5.6). Similarly, for UNER, the Uniform distribution consistently outperforms other distributions like Gaussian and Student-t on English (Figure 5.7). This suggests that the

model benefits from a balanced exposure to different tokenisation granularities, enhancing its generalisability.

Sampling merge levels yields better performance than using a fixed number of merges on English. This is evidenced by the consistent or improved accuracy on XNLI compared to the 50% adapter (Figure 5.8) and F1-scores on UNER compared to the 75% adapter (Figure 5.9). Importantly, we obtain these improvements without updating the embedding layer during adapter training, which is an advantage over previous methods, which often rely on initialising embeddings through heuristics such as FVT (Gee et al., 2024) or FOCUS (Dobler and De Melo, 2023), and typically require continued training of these embeddings to optimise performance (Sachidananda et al., 2021; Chung et al., 2020).

Sampling merge levels may negatively impact cross-lingual transfer. While the model remains robust to tokenisation on the source language (i.e., English), we observe a performance decline on XNLI when evaluating on other languages such as Swahili, Urdu, Spanish, or Greek compared to the 50% adapter. For UNER, this issue is observed only for Portuguese. These findings imply that while sampling merge levels can be advantageous for English, it may hurt performance in cross-lingual contexts. Further investigation is needed to determine whether the observed decline is specific to the sampling strategy itself, or if it is influenced by language-specific characteristics, the nature of the task, or other underlying factors.

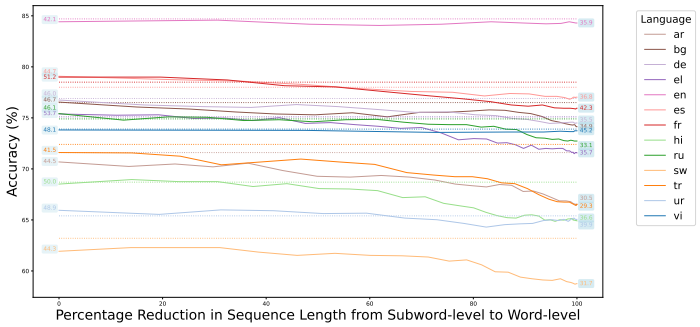
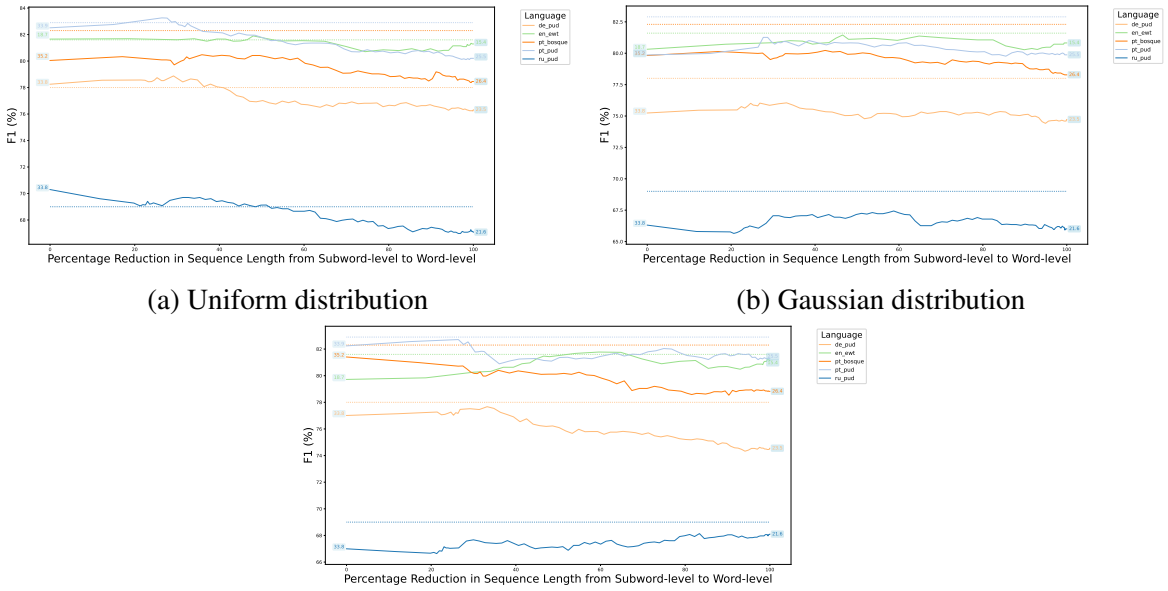


Fig. 5.6 XNLI accuracy across different merge levels. Results obtained using the adapter trained with tokenisers sampled from a **Uniform** distribution and HN embeddings.

5.1.3 Disentangling Task Adaptation from Tokenisation Adaptation

Table 5.5 presents the accuracies achieved for masked token prediction after training a LoRA adapter on the masked language modeling (MLM) task. Given that sampling merge levels from a Cauchy distribution yields the highest accuracy, we test the integration of this adapter with a task adapter trained on XNLI/UNER using original tokenisation and embeddings. The most relevant results of this merging are detailed in Table 5.6.



(a) Uniform distribution

(b) Gaussian distribution

(c) Student-t distribution

Fig. 5.7 F1-score accuracy across different merge levels. Results obtained using the adapter trained on UNER with tokenisers sampled from a different distributions and HN embeddings.

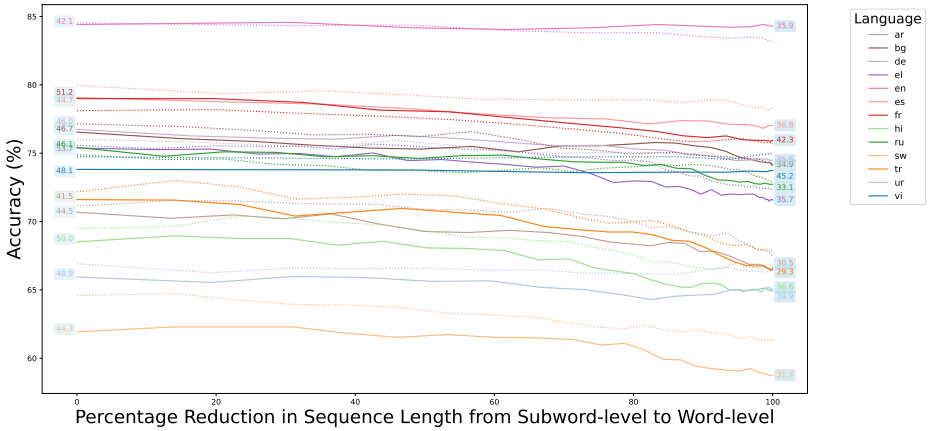


Fig. 5.8 XNLI: Comparison between adapter trained with 50% sequence reduction (**dotted line**) and the adapter trained by sampling tokenisers from a Uniform distribution (**continuous line**).

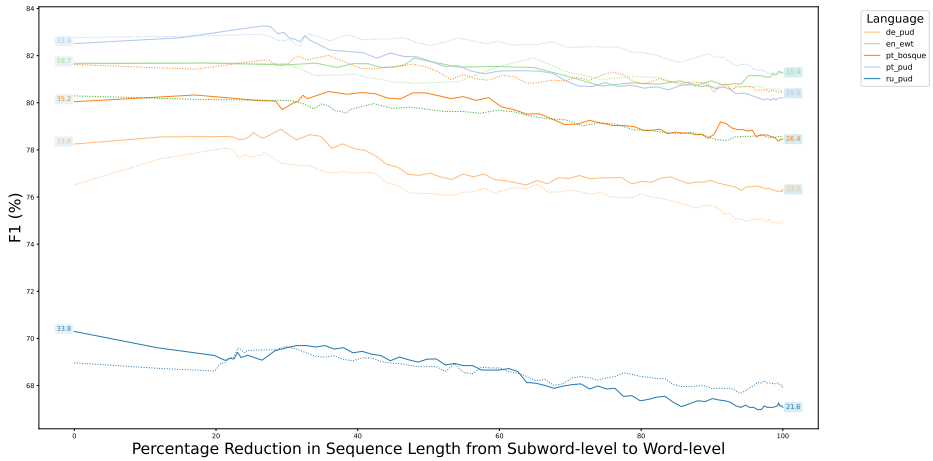


Fig. 5.9 UNER: Comparison between adapter trained with 75% sequence reduction (**dotted line**) and the adapter trained by sampling tokenisers from a Uniform distribution (**continuous line**).

Tokenisation	**Accuracy (%)**
original	60.1
Word	61.8
Uniform	62.0
Gaussian	61.9
Student-t	61.9
Cauchy	**62.4**

Table 5.5 Tokenisation adapters trained on MADLAD-400. Accuracy is computed on word-level for each tokenisation method.

Adapter 1 (Task)	**Adapter 2 (Tokenisation)**	**Merging method**	**Weight 1**	**Weight 2**	**Accuracy or F1-score (%)**
XNLI-original	-	-	-	-	84.8
UNER-original	-	-	-	-	81.6
XNLI	Cauchy	Linear	0.8	0.15	**82.61**
XNLI	Word	Linear	0.9	0.15	82.49
XNLI	Cauchy	Concatenation	0.9	0.5	82.53
XNLI	Cauchy	TIES	1	1	80.09
UNER	Cauchy	Linear	0.9	0.2	76.73
UNER	Cauchy	Concatenation	0.8	0.3	**76.94**
UNER	Cauchy	TIES	1	1	76.62

Table 5.6 Results obtained when merging the task adapter with a tokenisation adapter. The rows in grey represent the baselines obtained with the adapter trained on XNLI/UNER with original tokenisation and embeddings.

Challenges in merging task and tokenisation adapters. For XNLI, the highest achieved accuracy is 82.61%, only marginally better than the 82.5% achieved at word-level without a tokenisation adapter and still below the 84.8% — the upper bound obtained in §5.1.1. For UNER, the merged adapter achieves 76.9%, slightly below the 77.0% achieved without a tokenisation adapter and significantly lower than the upper bound of 81.6%. Future work could explore more extensive hyperparameter tuning, other merging methods, and especially increasing the training data beyond 380M tokens — a limit initially set by computational constraints. Successfully disentangling task and tokenisation adaptation would facilitate the integration of multiple task-specific models, removing the need for retraining a model to achieve optimal performance with dynamic tokenisation.

5.2 Decoder LMs

In this section, we present and analyse the results obtained on the experiments on decoder LMs (§4.5.2).

5.2.1 MMLU

Table 5.7 shows the results obtained with MISTRAL-7B on MMLU on four main settings.

	Tokenisation	Embeddings	Vocab. Size	# Shots	Shots Selection	Accuracy (%)	$\Delta_{\text{Length. (\%)}}$
(1)	original	original	32k	0	-	**60.1**	0
(2)	original	original	32k	5	Random	**60.1**	0
(3)	original	original	32k	5	Same Domain	**61.8**	0
(4)	original	HN	32k	0	-	55.8	0
(5)	original	HN	32k	5	Random	56.9	0
(6)	original	HN	32k	5	Same Domain	58.8	0
(7)	LP	HN	32k	0	-	54.9	-0.7
(8)	LP	HN	32k	5	Random	55.9	-0.8
(9)	LP	HN	32k	5	Same Domain	57.8	-1
(10)	LP	HN	1M	0	-	51.8	-13.6
(11)	LP	HN	1M	5	Random	54.0	-13.4
(12)	LP	HN	1M	5	Same Domain	55.9	-13.6

Table 5.7 Performance of MISTRAL-7B on the MMLU task under different settings. $\Delta_{\text{Length. (\%)}}$ represents the average decrease in token sequence length over the original tokenisation.

Impact of expanding the vocabulary to $\mathcal{V}_{1M}$. When MISTRAL-7B is evaluated with Longest-prefix (LP) tokenisation and HN embeddings using the expanded 1M vocabulary, $\mathcal{V}_{\text{1M}}$, we note a decrease in performance of 1.9% in the 5-shot same domain setting, as well as a reduction in sequence length by 13.6% (settings 12 vs. 9). This reduction is similar to those observed in encoder LMs (14.7% for XNLI, word-level, Table 5.1), suggesting that using $\mathcal{V}_{\text{1M}}$ approaches a **word-level granularity**. As seen with encoder models, using word-level tokenisation results in performance degradation, which is likely due to the generated HN embeddings. Additionally, the reduction in sequence length by 13.6% indicates two key aspects: *(1)* similar to what has been noted with encoders, the initial MISTRAL-7B's vocabulary is biased towards English, which suggests that even at a subword level, the text may not be over-segmented; and *(2)* the large size of $\mathcal{V}_{\text{1M}}$ might be excessive, resulting in many unused tokens. This aligns with the Zipf ceiling effect, as detailed by Liang et al. (2023), where a limited number of tokens are enough to cover 99% of the content, indicating that $\mathcal{V}_{\text{1M}}$ likely includes many tokens from the long tail of the Zipfian distribution. Liang et al. (2023) suggest that a smaller vocabulary could improve performance since tokens from the long tail often have "problematic embeddings," as they are trained on significantly less data. While in our case embeddings are generated rather than trained, a similar issue may arise for such rare tokens. Investigating the quality of HN embeddings for these long tail tokens could provide insights into optimal vocabulary sizes and help refine our dynamic tokenisation method to exclude potentially problematic tokens.

Comparing original and HN embeddings. Unlike the encoder models where the HN embeddings closely matched the performance to original embeddings when using the original tokenisation (Table 5.1), in the decoder setting, we see a noticeable difference. With the initial 32k vocabulary ($\mathcal{V}_{\text{init}}$), the performance using HN embeddings declines by 3% compared to original embeddings (settings 6 vs. 3). This difference highlights potential challenges in adapting HN embeddings for decoders, also noted by Minixhofer et al. (2024) when transferring an LM to a different tokeniser. Although further training of these embeddings could potentially close or minimise this performance gap, the continuous adaptation to new tokens — and embeddings — required by dynamic tokenisation makes such retraining impractical. Therefore, future work could focus on improving the quality of HN embeddings or making changes to decoder architecture that are more flexible or capable of accommodating new tokens effectively.

Impact of LP tokenisation. Using LP tokenisation with the same 32k vocabulary, HN embeddings and in a 5-shot setting from the same domain results in a marginal decrease in performance and sequence length, about 1% compared to the original tokenisation (settings 9 vs. 6). This slight reduction shows the effectiveness of MISTRAL-7B with the greedy LP

tokenisation, maintaining performance nearly at the same level to the model's original BPE tokenisation.

5.2.2 MT-Bench

Table 5.8 outlines the results obtained on MT-Bench, which focusses on assessing the generation capabilities of the model.

	Tokenisation	Embeddings	Vocab. Size	Next token search	Repetition Penalty	Min. Prob.	Sample from top 10?	Avg Score
(1)	Original	Original	32k	exhaustive	-	-	✗	**7.54**
(2)	Original	Original	32k	exhaustive	1.1	-	✗	7.51
(3)	Original	Original	32k	exhaustive	1.1	0.05	✓	7.46
(4)	Original	HN	32k	exhaustive	-	-	✗	6.84
(5)	Original	HN	32k	exhaustive	1.1	-	✗	**7.33**
(6)	Original	HN	32k	exhaustive	1.1	0.1	✓	7.10
(7)	LP	HN	32k	exhaustive	-	-	✗	6.50
(8)	LP	HN	32k	exhaustive	1.1	-	✗	**6.92**
(9)	LP	HN	32k	exhaustive	1.1	0.05	✓	**6.92**
(10)	LP	HN	1M	ScaNN index	-	-	✗	6.26
(11)	LP	HN	1M	ScaNN index	1.1	-	✗	**6.73**
(12)	LP	HN	1M	ScaNN index	1.1	0.05	✓	6.64
(13)	LP	HN	1M	exhaustive	-	-	✗	5.24
(14)	LP	HN	1M	exhaustive	1.1	-	✗	5.54
(15)	LP	HN	1M	exhaustive	1.1	0.05	✓	**6.53**
(16)	LP	HN	1M	exhaustive	1.1	0.02	✓	6.52

Table 5.8 Performance of MISTRAL-7B-INSTRUCT on MT-Bench under different settings.

Consistency with MMLU result patterns. Similar to the MMLU results, original embeddings generally outperform HN embeddings (settings 1 vs. 4). Changing the tokenisation from original to LP with HN embeddings slightly lowers performance (settings 4 vs. 7). A similar decline is observed using $\mathcal{V}_{1M}$ compared to $\mathcal{V}_{init}$ (settings 7 vs. 10).

Token repetition issue. An issue observed in settings with HN embeddings is token repetition, particularly for prompts from domains requiring creativity (e.g., writing). To address this, we introduced a repetition penalty and top-k sampling with minimum probability threshold. This significantly improved model performance, almost closing the gap between the setting using original and HN embeddings with $\mathcal{V}_{init}$ (settings 1 vs. 5).

The problem of token repetition is more amplified when using $\mathcal{V}_{1M}$ and exhaustive search, mostly because the generation is not limited to selecting from the top-k tokens and the token space is about 31 times larger. Using repetition penalty and top-k sampling significantly

improved performance. Additionally, the gap is only 0.19 when comparing $\mathcal{V}_{1M}$ and $\mathcal{V}_{init}$ with LP tokenisation (settings 11 and 8).

Finally, the repetition issue may also stem from the MISTRAL-7B model itself, as multiple reports highlighted similar problems occurring during generation.[2]

ANN vs Exhaustive Search. Contrary to our expectations, the results indicate that using an ANN index outperforms exhaustive search (setting 11 vs 15). This result aligns with findings from other studies, such as those by Xu et al. (2023), who suggest that the slight "inaccuracies" introduced by the ANN index search adds as a level of noise or variability, which acts like a regularisation technique.

Problematic tokens? After analysing the results generated by our model with $\mathcal{V}_{1M}$, we identified some tokens that usually result in an infinite cycle of repetition — up to the maximum length. Examples include `PN`, `mt`, `mss`, `nss`, `Massacre`, `triphosphate`, and some proper nouns like `Hilbert`. This issue may stem from the fact that embeddings for these tokens are suboptimal, as they fall within the long tail of the Zipfian distribution. This observation also highlights a disadvantage of such a large vocabulary, as it includes some tokens that are not needed.

End of sequence issue. We noticed that the model sometimes attempts to end a sequence, but fails to generate the end-of-sequence token (i.e., `</s>`). Instead, it repeatedly generates tokens that resembles `</s>`, such as `</s/></s/><s/></s/></s/></s/></s/></s/>`. This may be linked to how the HN was trained, specifically the method of packing samples without adjusting the attention mask, which could be problematic (Krell et al., 2021).

5.3 Other Results

This section outlines the results of evaluating different indices and the improvements gained from implementing an HN-specific cache.

5.3.1 Verifying the Quality of the Index

To find the optimal hyperparameters and identify the most suitable index for our use case, we first create a dataset of 1000 texts or prompts, each consisting of 128 tokens, extracted from the clean English subset of MADLAD-400 (§4.4). We then train an index using the 1M output embeddings generated by the HN: $H_\theta\left(\mathcal{V}_{1M}, T_{LP}\right)$.

We evaluate the index quality using Recall@10. Each prompt p from the dataset is tokenised using $(\mathcal{V}_{1M}, T_{LP})$ to generate input embeddings with the HN, $E_{\phi_{in}}(T_{LP}(p))$. These

[2] huggingface.co/mistralai/Mistral-7B-v0.1/discussions/29

embeddings are fed into the model to obtain the last hidden state $\mathbf{h}$. We then check if the token predicted using an exhaustive search, Equation 5.1, appears within the top 10 tokens retrieved by the index, $\mathcal{I}_{10}(\mathbf{h})$.

$$\text{Next token} = \arg\max_i \left[\text{softmax}\left(\mathbf{h} \cdot E_{\phi_{\text{out}}}(\mathcal{V}_{1\text{M}})^\top\right)\right]_i \tag{5.1}$$

where i maps to token_i in $\mathcal{V}_{1\text{M}}$.

Table B.3 and B.2 from Appendix B show the hyperparameters we used for our indices. Based on the performance results on our prompt dataset, as detailed in Table 5.9, we selected the ScaNN index from row (5) for its optimal Recall-to-speed ratio.

	Index	**Nprobe**	**Leaves to search**	**Pre-reorder Neighbours**	**Recall@10 (%)**	**Seconds/ Query**
(1)	Faiss	10	-	-	83.3	0.22
(2)	Faiss	15	-	-	88.2	0.33
(3)	Faiss	20	-	-	90.6	0.44
(4)	ScaNN	-	500	200	74.0	**0.02**
(5)	ScaNN	-	1000	2000	**93.6**	0.03

Table 5.9 Performance comparison between Faiss and ScaNN indices across different configurations.

5.3.2 Hypernetwork Embeddings Caching

In all our experiments, we implemented a Least-Recently-Used (LRU) cache for storing HN embeddings to enhance efficiency and reduce overhead. This approach was particularly motivated by the frequent repetition of certain tokens across batches in encoder experiments. Common words like “the” or “and” appear in nearly every utterance, making it practical to cache their embeddings rather than regenerate them for each batch.

The benefits are clear, shown in Figure 5.10: using an HN-specific LRU cache significantly reduces the number of tokens requiring embeddings per batch.

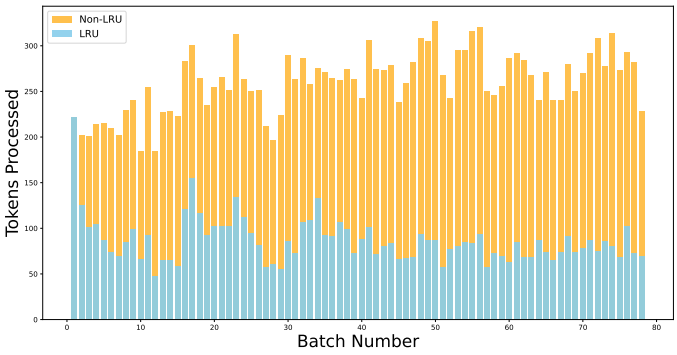


Fig. 5.10 Tokens processed by the hypernetwork using an HN-specific LRU cache versus processing all unique tokens without caching. Results obtained on the `validation` subset of XNLI English.

5.4 Summary

In this section, we presented and discussed the results obtained by applying dynamic tokenisation to encoder and decoder LMs.

The key findings are:

- Using a hypernetwork with word-level tokenisation, without any additional training, yields a sequence length reduction of 22.5% on XNLI (26.2% on UNER) on average while preserving accuracy to 2.8% on XNLI (4.2% UNER) (§5.1.1). This significantly improves LM throughput;
- We showed that the accuracy gap can be fully closed for English, across all merge levels m, by training a task adapter (on XNLI/UNER) using tokenisation levels sampled from a Uniform distribution — with HN embeddings (§5.1.2);
- Using a large vocabulary such as $\mathcal{V}_{1\text{M}}$ yields similar sequence reduction on English as word-level, reducing sequence length by 13.6% on the MMLU task, while performance decreases by only 1.9%;
- For MT-Bench, we showed that generating tokens from $\mathcal{V}_{1\text{M}}$ using LP tokenisation and HN embeddings results in a score of 6.73, producing tokens closer to word-level, improving inference speed at a slight performance decrease of 0.19 compared to using a 32k vocabulary with LP tokenisation. Additionally, using an ANN index enables better performance and faster speed compared to exhaustive search when using a large vocabulary (§5.2.2);

Chapter 6

Conclusion

This thesis has investigated different ways of retrofitting language models with dynamic tokenisation, adapting token boundaries dynamically based on the input. To do so, we repurposed the hypernetwork (HN) from Minixhofer et al. (2024) for dynamic tokenisation. We used the HN for all experiments to generate token embeddings on-the-fly, with minimal computational overhead (c.f. §5 from Minixhofer et al., 2024 for more details). First, we applied dynamic tokenisation to **encoder LMs** at the batch-level (§3.1). This approach achieved up to 35.8% sequence reduction with word-level boundaries for some languages (Table 5.2), maintaining the average performance within a $3-4\%$ margin. Subsequent training with a LoRA adapter using our dynamic tokenisation and HN embeddings, closed the performance gap almost completely on English across all merge levels (§5.1.2). For decoder LMs, we implemented dynamic tokenisation at sample-level (§3.2), expanding the English vocabulary to 1M tokens and using Longest-Prefix (LP) tokenisation. This setup significantly reduced sequence lengths, thus improving inference speeds, as seen on MMLU (§5.2.1). In generation tasks, we used an ANN index and achieved fast generation with a $\mathcal{V}_{1\text{M}}$, demonstrating scalability to even larger, dynamic vocabularies (§5.2.2).

Impact

Retrofitting LMs with dynamic tokenisation has significant impact:

1. It reduces sequence lengths across languages, leading to fairer representation and significantly improved computational efficiency associated with inference;

2. It allows the model to adapt to new domain- or language-specific vocabulary without retraining the embeddings layer, aligning with continual learning paradigms (Wu et al.,

2024). Effectively, it results in an unbounded vocabulary which is determined based on the input data;

3. Successfully integrating a 1M token vocabulary with an ANN index in a zero-shot setting demonstrates the potential of large-scale vocabulary adaptation for decoder LMs with minimal computational overhead. This approach removes the need to store or retrain all embeddings, addressing a key issue in previous models where embeddings took the majority of model's parameter allocation (Chung et al., 2020).

Limitations and Future Directions

While this study introduced dynamic tokenisation in LMs, there are some limitations and several promising directions for future research:

- Dynamic tokenisation is currently not applicable during the training phase of LMs because it relies on an HN, which requires a pre-trained LM to learn to predict embeddings for any new token. Future research should explore integrating dynamic tokenisation, using a different method, into the initial LM training to enhance model adaptability;

- The disentanglement of task and tokenisation adaptation did not match the performance of the joint approach (§5.1.3). Future efforts could potentially explore alternative merging methods, and longer training (more than 380M tokens). Successfully disentangling them would facilitate the integration of task-specific models with different dynamic tokenisation levels without the need for fine-tuning;

- For decoder LMs experiments, we noted a token repetition issue which can be associated either with the HN embeddings or the MISTRAL-7B (§5.2.2). Future work should investigate the root cause of this issue and improve the HN if the issue is related to the embeddings. Additionally, further experiments with other LLMs would be beneficial to ensure generalisability of the results;

- Our method for decoder LMs expands the initial vocabulary but remains static during inference, limiting the model's adaptability. Future work should focus on enabling real-time vocabulary adjustments to better handle evolving language use or domain-specific terms, which is the ultimate goal of dynamic tokenisation;

References

Ahia, Orevaoghene, Sachin Kumar, Hila Gonen, Jungo Kasai, David R Mortensen, Noah A Smith, and Yulia Tsvetkov (2023). "Do all languages cost the same? tokenization in the era of commercial language models". In: *arXiv preprint arXiv:2305.13707*. DOI: 10.18653/v1/2023.emnlp-main.614.

Balachandran, Abhinand (2023). "Tamil-Llama: A New Tamil Language Model Based on Llama 2". In: *arXiv preprint arXiv:2311.05845*. DOI: 10.48550/arXiv.2311.05845.

Boukkouri, Hicham El, Olivier Ferret, Thomas Lavergne, Hiroshi Noji, Pierre Zweigenbaum, and Junichi Tsujii (2020). "CharacterBERT: Reconciling ELMo and BERT for word-level open-vocabulary representations from characters". In: *arXiv preprint arXiv:2010.10392*. DOI: 10.48550/arXiv.2010.10392.

Chiang, Cheng-Han and Hung-yi Lee (2023). "Can Large Language Models Be an Alternative to Human Evaluations?" In: *arXiv preprint arXiv:2305.01937*. DOI: 10.48550/arXiv.2305.01937.

Chung, Hyung Won, Thibault Fevry, Henry Tsai, Melvin Johnson, and Sebastian Ruder (2020). "Rethinking embedding coupling in pre-trained language models". In: *arXiv preprint arXiv:2010.12821*. DOI: 10.48550/arXiv.2010.12821.

Clark, Jonathan H, Dan Garrette, Iulia Turc, and John Wieting (2022). "Canine: Pre-training an efficient tokenization-free encoder for language representation". In: *Transactions of the Association for Computational Linguistics* 10, pp. 73–91. DOI: 10.1162/tacl_a_00448.

Conneau, Alexis, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Édouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov (2020). "Unsupervised Cross-lingual Representation Learning at Scale". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8440–8451. DOI: 10.18653/v1/2020.acl-main.747.

Conneau, Alexis, Guillaume Lample, Ruty Rinott, Adina Williams, Samuel R Bowman, Holger Schwenk, and Veselin Stoyanov (2018). "XNLI: Evaluating cross-lingual sentence representations". In: *arXiv preprint arXiv:1809.05053*. DOI: 10.48550/arXiv.1809.05053.

Cui, Yiming, Ziqing Yang, and Xin Yao (2023). "Efficient and effective text encoding for chinese Llama and Alpaca". In: *arXiv preprint arXiv:2304.08177*. DOI: 10.48550/arXiv.2304.08177.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805*. DOI: 10.48550/arXiv.1810.04805.

Ding, Yujuan, Wenqi Fan, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li (2024). "A survey on rag meets llms: Towards retrieval-augmented large language models". In: *arXiv preprint arXiv:2405.06211*. DOI: 10.48550/arXiv.2405.06211.

Dobler, Konstantin and Gerard De Melo (2023). "FOCUS: Effective Embedding Initialization for Monolingual Specialization of Multilingual Models". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13440–13454. DOI: 10.18653/v1/2023.emnlp-main.829.

Douze, Matthijs, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou (2024). "The faiss library". In: *arXiv preprint arXiv:2401.08281*. DOI: 10.48550/arXiv.2401.08281.

Eger, Steffen and Yannik Benz (2020). "From hero to zéroe: A benchmark of low-level adversarial attacks". In: *Proceedings of the 1st conference of the Asia-Pacific chapter of the association for computational linguistics and the 10th international joint conference on natural language processing*, pp. 786–803.

Fan, Angela, Mike Lewis, and Yann Dauphin (2018). "Hierarchical neural story generation". In: *arXiv preprint arXiv:1805.04833*. DOI: 10.48550/arXiv.1805.04833.

Fujii, Kazuki, Taishi Nakamura, Mengsay Loem, Hiroki Iida, Masanari Ohi, Kakeru Hattori, Hirai Shota, Sakae Mizuki, Rio Yokota, and Naoaki Okazaki (2024). "Continual Pre-Training for Cross-Lingual LLM Adaptation: Enhancing Japanese Language Capabilities". In: *arXiv preprint arXiv:2404.17790*. DOI: 10.48550/arXiv.2404.17790.

Gage, Philip (1994). "A new algorithm for data compression". In: *The C Users Journal* 12.2, pp. 23–38.

Gee, Leonidas, Andrea Zugarini, Leonardo Rigutini, and Paolo Torroni (2024). "Fast vocabulary transfer for language model compression". In: *arXiv preprint arXiv:2402.09977*. DOI: 10.48550/arXiv.2402.09977.

Golkar, Siavash, Mariel Pettee, Michael Eickenberg, Alberto Bietti, Miles Cranmer, Geraud Krawezik, Francois Lanusse, Michael McCabe, Ruben Ohana, Liam Parker, Bruno Régaldo-Saint Blancard, Tiberiu Tesileanu, Kyunghyun Cho, and Shirley Ho (2023). "xVal: A continuous number encoding for large language models". In: *arXiv preprint arXiv:2310.02989*. DOI: 10.48550/arXiv.2310.02989.

Guo, Ruiqi, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar (2020). "Accelerating large-scale inference with anisotropic vector quantization". In: *International Conference on Machine Learning*. PMLR, pp. 3887–3896.

Ha, David, Andrew Dai, and Quoc V Le (2016). "Hypernetworks". In: *arXiv preprint arXiv:1609.09106*. DOI: 10.48550/arXiv.1609.09106.

Hendrycks, Dan, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt (2020). "Measuring massive multitask language understanding". In: *arXiv preprint arXiv:2009.03300*. DOI: 10.48550/arXiv.2009.03300.

Hofmann, Valentin, Hinrich Schuetze, and Janet B Pierrehumbert (2022). "An embarrassingly simple method to mitigate undesirable properties of pretrained language model tokenizers". In: Association for Computational Linguistics. DOI: 10.18653/v1/2022.acl-short.43.

Holtzman, Ari, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi (2019). "The curious case of neural text degeneration". In: *arXiv preprint arXiv:1904.09751*. DOI: 10.48550/arXiv.1904.09751.

Hu, Edward J, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen (2021). "LoRA: Low-rank adaptation of large language models". In: *arXiv preprint arXiv:2106.09685*. DOI: 10.48550/arXiv.2106.09685.

Ilharco, Gabriel, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi (2022). "Editing models with task arithmetic". In: *arXiv preprint arXiv:2212.04089*. DOI: 10.48550/arXiv.2212.04089.

Ippolito, Daphne, Reno Kriz, João Sedoc, Maria Kustikova, and Chris Callison-Burch (2019). "Comparison of Diverse Decoding Methods from Conditional Language Models". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 3752–3762. DOI: 10.18653/v1/P19-1365.

Jiang, Albert Q, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. (2023). "Mistral 7B". In: *arXiv preprint arXiv:2310.06825*. DOI: 10.48550/arXiv.2310.06825.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. (2007). "Moses: Open source toolkit for statistical machine translation". In: *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*. Association for Computational Linguistics, pp. 177–180.

Kong, Rui, Qiyang Li, Xinyu Fang, Qingtian Feng, Qingfeng He, Yazhu Dong, Weijun Wang, Yuanchun Li, Linghe Kong, and Yunxin Liu (2024). "LoRA-Switch: Boosting the Efficiency of Dynamic LLM Adapters via System-Algorithm Co-design". In: *arXiv preprint arXiv:2405.17741*. DOI: 10.48550/arXiv.2405.17741.

Krell, Mario Michael, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon (2021). "Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance". In: *arXiv preprint arXiv:2107.02027*. DOI: 10.48550/arXiv.2107.02027.

Kudo, Taku (2018). "Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 66–75. DOI: 10.18653/v1/P18-1007.

Kudo, Taku and John Richardson (2018). "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing". In: *arXiv preprint arXiv:1808.06226*. DOI: 10.48550/arXiv.1808.06226.

Kudugunta, Sneha, Isaac Caswell, Biao Zhang, Xavier Garcia, Derrick Xin, Aditya Kusupati, Romi Stella, Ankur Bapna, and Orhan Firat (2024). "Madlad-400: A multilingual and document-level large audited dataset". In: *Advances in Neural Information Processing Systems* 36.

Lan, Tian, Deng Cai, Yan Wang, Heyan Huang, and Xian-Ling Mao (2023). "Copy Is All You Need". In: *arXiv preprint arXiv:2307.06962*. DOI: 10.48550/arXiv.2307.06962. eprint: 2307.06962.

Lee, Jason, Kyunghyun Cho, and Thomas Hofmann (2017). "Fully character-level neural machine translation without explicit segmentation". In: *Transactions of the Association for Computational Linguistics* 5, pp. 365–378. DOI: 10.1162/tacl_a_00067.

Li, Minghan, Xilun Chen, Ari Holtzman, Beidi Chen, Jimmy Lin, Wen-tau Yih, and Xi Victoria Lin (2024). "Nearest Neighbor Speculative Decoding for LLM Generation and Attribution". In: *arXiv preprint arXiv:2405.19325*. DOI: 10.48550/arXiv.2405.19325.

Liang, Davis, Hila Gonen, Yuning Mao, Rui Hou, Naman Goyal, Marjan Ghazvininejad, Luke Zettlemoyer, and Madian Khabsa (2023). "XLM-V: Overcoming the vocabulary bottleneck in multilingual masked language models". In: *arXiv preprint arXiv:2301.10472*. DOI: 10.48550/arXiv.2301.10472.

Liu, Yihong, Peiqin Lin, Mingyang Wang, and Hinrich Schütze (2024). "OFA: A Framework of Initializing Unseen Subword Embeddings for Efficient Large-scale Multilingual

Continued Pretraining". In: *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 1067–1097. DOI: 10.18653/v1/2024.findings-naacl.68.

Martins, Andre and Ramon Astudillo (2016). "From softmax to sparsemax: A sparse model of attention and multi-label classification". In: *International conference on machine learning*. PMLR, pp. 1614–1623.

Mayhew, Stephen, Terra Blevins, Shuheng Liu, Marek Šuppa, Hila Gonen, Joseph Marvin Imperial, Börje F Karlsson, Peiqin Lin, Nikola Ljubešić, LJ Miranda, et al. (2023). "Universal NER: A Gold-Standard Multilingual Named Entity Recognition Benchmark". In: *arXiv preprint arXiv:2311.09122*. DOI: 10.48550/arXiv.2311.09122.

Mielke, Sabrina J, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y Lee, Benoît Sagot, et al. (2021). "Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP". In: *arXiv preprint arXiv:2112.10508*. DOI: 10.48550/arXiv.2112.10508.

Minaee, Shervin, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao (2024). "Large Language Models: A Survey". In: *arXiv preprint arXiv:2402.06196*. DOI: 10.48550/arXiv.2402.06196.

Minixhofer, Benjamin, Fabian Paischer, and Navid Rekabsaz (2022). "WECHSEL: Effective initialization of subword embeddings for cross-lingual transfer of monolingual language models". In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3992–4006. DOI: 10.18653/v1/2022.naacl-main.293.

Minixhofer, Benjamin, Edoardo Maria Ponti, and Ivan Vulić (2024). "Zero-Shot Tokenizer Transfer". In: *arXiv preprint arXiv:2405.07883*. DOI: 10.48550/arXiv.2405.07883.

Nawrot, Piotr, Jan Chorowski, Adrian Łańcucki, and Edoardo M Ponti (2022). "Efficient transformers with dynamic token pooling". In: *arXiv preprint arXiv:2211.09761*. DOI: 10.18653/v1/2023.acl-long.353.

Petrov, Aleksandar, Emanuele La Malfa, Philip Torr, and Adel Bibi (2024). "Language model tokenizers introduce unfairness between languages". In: *Advances in Neural Information Processing Systems* 36.

Pfeiffer, Jonas, Sebastian Ruder, Ivan Vulić, and Edoardo Maria Ponti (2023). "Modular deep learning". In: *arXiv preprint arXiv:2302.11529*. DOI: 10.48550/arXiv.2302.11529.

Pinter, Yuval, Robert Guthrie, and Jacob Eisenstein (2017). "Mimicking word embeddings using subword RNNs". In: *arXiv preprint arXiv:1707.06961*. DOI: 10.48550/arXiv.1707.06961.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. (2019). "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8, p. 9.

Rainey, Joshua P, Brenna E Blackburn, Chance L McCutcheon, Courtney M Kenyon, Kevin J Campbell, Lucas A Anderson, and Jeremy M Gililland (2023). "A multilingual chatbot can effectively engage arthroplasty patients who have limited English proficiency". In: *The Journal of Arthroplasty* 38.7, S78–S83. DOI: 10.1016/j.arth.2023.04.014.

Rust, Phillip, Jonas F Lotz, Emanuele Bugliarello, Elizabeth Salesky, Miryam de Lhoneux, and Desmond Elliott (2022). "Language modelling with pixels". In: *arXiv preprint arXiv:2207.06991*. DOI: 10.48550/arXiv.2207.06991.

Rust, Phillip, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych (2020). "How good is your tokenizer? on the monolingual performance of multilingual language models". In: *arXiv preprint arXiv:2012.15613*. DOI: 10.18653/v1/2021.acl-long.243.

Sachidananda, Vin, Jason S Kessler, and Yi-An Lai (2021). "Efficient Domain Adaptation of Language Models via Adaptive Tokenization". In: *arXiv preprint arXiv:2109.07460*. DOI: 10.48550/arXiv.2109.07460.

Schick, Timo and Hinrich Schutze (2020). "BERTRAM: Improved Word Embeddings Have Big Impact on Contextualized Model Performance". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 3996–4007. DOI: 10.48550/arXiv.1910.07181.

Schick, Timo and Hinrich Schütze (2019). "Attentive mimicking: Better word embeddings by attending to informative contexts". In: *arXiv preprint arXiv:1904.01617*. DOI: 10.48550/arXiv.1904.01617.

Schuster, Mike and Kaisuke Nakajima (2012). "Japanese and korean voice search". In: *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, pp. 5149–5152. DOI: 10.1109/ICASSP.2012.6289079.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2015). "Neural machine translation of rare words with subword units". In: *arXiv preprint arXiv:1508.07909*. DOI: 10.48550/arXiv.1508.07909.

Slagle, Kevin (2024). "SpaceByte: Towards Deleting Tokenization from Large Language Modeling". In: *arXiv preprint arXiv:2404.14408*. DOI: 10.48550/arXiv.2404.14408.

Song, Xinying, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou (2021). "Fast WordPiece Tokenization". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 2089–2103. DOI: 10.18653/v1/2021.emnlp-main.160.

Sun, Lichao, Kazuma Hashimoto, Wenpeng Yin, Akari Asai, Jia Li, Philip Yu, and Caiming Xiong (2020). "Adv-bert: Bert is not robust on misspellings! generating nature adversarial samples on bert". In: *arXiv preprint arXiv:2003.04985*. DOI: 10.48550/arXiv.2003.04985.

Tay, Yi, Vinh Q Tran, Sebastian Ruder, Jai Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler (2021). "Charformer: Fast character transformers via gradient-based subword tokenization". In: *arXiv preprint arXiv:2106.12672*. DOI: 10.48550/arXiv.2106.12672.

Tran, Ke (2020). "From english to foreign languages: Transferring pre-trained language models". In: *arXiv preprint arXiv:2002.07306*. DOI: 10.48550/arXiv.2002.07306.

Uzan, Omri, Craig W Schmidt, Chris Tanner, and Yuval Pinter (2024). "Greed is all you need: An evaluation of tokenizer inference methods". In: *arXiv preprint arXiv:2403.01289*. DOI: 10.48550/arXiv.2403.01289.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Viterbi, Andrew (1967). "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". In: *IEEE transactions on Information Theory* 13.2, pp. 260–269. DOI: 10.1109/TIT.1967.1054010.

Wang, Xinyi, Sebastian Ruder, and Graham Neubig (2021). "Multi-view subword regularization". In: *arXiv preprint arXiv:2103.08490*. DOI: 10.48550/arXiv.2103.08490.

Williams, Adina, Nikita Nangia, and Samuel R Bowman (2017). "A broad-coverage challenge corpus for sentence understanding through inference". In: *arXiv preprint arXiv:1704.05426*. DOI: 10.48550/arXiv.1704.05426.

Wu, Tongtong, Linhao Luo, Yuan-Fang Li, Shirui Pan, Thuy-Trang Vu, and Gholamreza Haffari (2024). "Continual Learning for Large Language Models: A survey". In: *arXiv preprint arXiv:2402.01364*. DOI: 10.48550/arXiv.2402.01364.

Xie, Jiateng, Zhilin Yang, Graham Neubig, Noah A Smith, and Jaime Carbonell (2018). "Neural cross-lingual named entity recognition with minimal resources". In: *arXiv preprint arXiv:1808.09861*. DOI: 10.48550/arXiv.1808.09861.

Xu, Frank F, Uri Alon, and Graham Neubig (2023). "Why do nearest neighbor language models work?" In: *International Conference on Machine Learning*. PMLR, pp. 38325–38341.

Xu, Jingwei, Junyu Lai, and Yunpeng Huang (2024). "MeteoRA: Multiple-tasks Embedded LoRA for Large Language Models". In: *arXiv preprint arXiv:2405.13053*. DOI: 10.48550/arXiv.2405.13053.

Xue, Linting, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel (2022). "Byt5: Towards a token-free future with pre-trained byte-to-byte models". In: *Transactions of the Association for Computational Linguistics* 10, pp. 291–306. DOI: 10.1162/tacl_a_00461.

Yadav, Prateek, Derek Tam, Leshem Choshen, Colin Raffel, and Mohit Bansal (2023). "Resolving interference when merging models". In: *arXiv preprint arXiv:2306.01708* 1. DOI: 10.48550/arXiv.2306.01708.

Yu, Lili, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis (2024). "Megabyte: Predicting million-byte sequences with multiscale transformers". In: *Advances in Neural Information Processing Systems* 36.

Yuan, Fei, Shuai Yuan, Zhiyong Wu, and Lei Li (2023). "How Vocabulary Sharing Facilitates Multilingualism in LLaMA?" In: *CS Department, Carnegie Mellon University*.

Zubiaga, Arkaitz (2024). *Natural language processing in the era of large language models*. DOI: 10.3389/frai.2023.1350306.

Appendix A

Prompt templates for MMLU

This appendix contains the prompt templates we used for MMLU evaluation (§4.5.2).

Prompt Template: 0-shot

"The following are multiple choice questions (with answers) about ⟨TOPIC⟩.
⟨QUESTION⟩
⟨ANSWERS⟩
Answer:"

Prompt Template: 5-shot from Random Domain

"The following are multiple choice questions (with answers).
This question is about ⟨TOPIC_1⟩ and refers to the following information.
⟨QUESTION_1⟩
⟨ANSWERS_1⟩
Answer: ⟨ANSWER_1⟩
.
.
.
This question is about ⟨TOPIC_5⟩ and refers to the following information.
⟨QUESTION_5⟩
⟨ANSWERS_5⟩
Answer: ⟨ANSWER_5⟩

This question is about ⟨TOPIC⟩ and refers to the following information.
⟨QUESTION⟩
⟨ANSWERS⟩
Answer: ”

Prompt Template: 5-shot from the Same Domain as the Current Question

“The following are multiple choice questions (with answers) about ⟨TOPIC⟩.
This question refers to the following information.
⟨QUESTION_1⟩
⟨ANSWERS_1⟩
Answer: ⟨ANSWER_1⟩
.
.
.
This question refers to the following information.
⟨QUESTION_5⟩
⟨ANSWERS_5⟩
Answer: ⟨ANSWER_5⟩

This question refers to the following information.
⟨QUESTION⟩
⟨ANSWERS⟩
Answer:”

Appendix B

Reproducibility Details

A summary for the hyperparameters we used for training and evaluation in this project is shown in Tables B.4, B.1, and B.2 to B.3.

Experiment	**Python's `random`**	`torch`	`numpy.random`
Encoder experiments	42	42	42
Decoder experiments on MMLU	0	1234	1234
Decoder experiments on MT-Bench	1234	1234	1234

Table B.1 Summary of random seeds used across different experiments.

Attribute	**Value**
Dimension	4096
Nlist	50
Nprobe	30
Quantizer	IndexFlatL2
Index	IndexIVFFlat
Metric	Inner Product

Table B.2 Configuration details for the Faiss index.

Attribute	**Value**
Num. neighbours	200
Num. leaves	2000
Num. leaves to search	250
Training Sample Size	1,000,000
Dim. per block	3
Anisotropic quantization	0.2
Reorder	200
Metric	Dot product

Table B.3 Configuration details for the ScaNN index.

Hyperparameter	**1) Task Adapter with Original Subword Tokenisation, XNLI**	**2) Joint Task and Dynamic Tokenisation Adapter, XNLI**	**1)Task Adapter with Original Subword Tokenisation & 2) Joint Task and Dynamic Tokenisation Adapter, UNER**	**3) Disentangling Task Adaptation from Tokenisation Adaptation**
Matrix Rank r	32	128	256	256
Scaling Factor α	64	256	512	512
Dropout	0.3	0.3	0.3	0.3
Epochs	10	{10, 15}	15	3
Learning Rate	3×10^{-4}	1×10^{-4}	3×10^{-4}	1×10^{-4}
Batch Size	32	32	32	32
Optimiser	AdamW	AdamW	AdamW	AdamW
Optimiser Parameters	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
Scheduler	Linear, no warmup steps	Linear, no warmup steps	Linear, no warmup steps	Linear, no warmup steps
Max Sequence Length	128	128	128	512
Token Masking Probability	N/A	N/A	N/A	15%

Table B.4 Summary of hyperparameters for LoRA training.