Retrofitting Large Language Models with Dynamic Tokenization

Anonymous ACL submission

Abstract

Current language models (LMs) use a fixed, *static* subword tokenizer. This default choice typically results in degraded efficiency and language capabilities, especially in languages other than English. To address this issue, we challenge the static design and propose retrofitting LMs with *dynamic tokenization*: a way to dynamically decide on token boundaries based on the input text via a subword-merging algorithm inspired by byte-pair encoding. We merge frequent subword sequences in a batch, then apply a pre-trained embedding-prediction hypernetwork to compute the token embeddings on-the-fly. For encoder-style models (e.g., XLM-R), this on average reduces token sequence lengths by >20% across 14 languages while degrading performance by less than 2%. The same method applied to prefilling and scoring in decoder-style models (e.g., Mistral-7B) results in minimal performance degradation at up to 17% reduction in sequence length. Overall, we find that dynamic tokenization can mitigate the limitations of static tokenization by substantially improving inference speed and promoting fairness across languages, enabling more equitable and adaptable LMs.

1 Introduction

(Large) Language Models (LMs) are the backbone of modern NLP applications, enabling advanced language understanding and generation. However, their effectiveness heavily relies on their tokenizers, which are responsible for *tokenizing* the input (Rust et al., 2021; Fujii et al., 2023; Toraman et al., 2023; Ali et al., 2024; Minaee et al., 2024; Minixhofer et al., 2024). This fundamental step involves breaking raw text into smaller units called *tokens*, which are part of the tokenizer'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.

Language	Original Subword Tokenization	#tokens
English	A sub/stantial im/prove/ment fosters further im/prove/ment/s	12
Swahili	U/bor/esh/aj/i mk/ub/wa una/ku/za u/bor/esh/aj/i za/idi	18
#merges	**Dynamic Tokenization**	**#tokens**
1	A sub/stantial *improve*/ment fosters further *improve*/ment/s	10 (83%)
1	U/*boresh*/aj/i mk/ub/wa una/ku/za u/*boresh*/aj/i za/idi	16 (89%)
2	A sub/stantial *improvement* fosters further *improvement*/s	8 (67%)
2	U/*boreshaj*/i mk/ub/wa una/ku/za u/*boreshaj*/i za/idi	14 (78%)
4	A *substantial improvement* fosters further *improvements*	6 (50%)
11	*Uboreshaji mkubwa unakuza uboreshaji zaidi*	5 (28%)

Table 1: Comparison of static subword vs dynamic tokenization for the same sentences in **English** and **Swahili**. Embeddings for tokens in *blue* are obtained using a hypernetwork (HN) which composes the subword-level embeddings, as highlighted by /. The last row shows the number of merges required to achieve word-level tokenization, serving as a 'compression upper bound' of the proposed approach. The percentages show the fraction of the original token count remaining after merging.

The majority of contemporary LMs rely on *subword tokenizers* (e.g., Devlin et al., 2019; Touvron et al., 2023) that are inherently static, as their vocabularies remain fixed post-training. This rigidity limits the model's adaptability, requiring expensive retraining to update both the vocabulary and embeddings (Dagan et al., 2024). Moreover, subword tokenizers struggle with handling sequences of numbers (Golkar et al., 2023), are sensitive to spelling errors (Sun et al., 2020; Xue et al., 2022) and often suffer from over-segmentation in languages other than English (Wang et al., 2021). This leads to inequitable performance across languages, increasing inference costs, latency, and reducing overall model effectiveness (Ahia et al., 2023). While character- or byte-level tokenization provides a potential solution, it produces long token sequences which brings additional challenges, such as the need for dynamic compute allocation or token pooling to stay compet-

itive to models using subword tokenization in terms of efficiency (Nawrot et al., 2023). These issues underscore the need for a more flexible or dynamic tokenization that adapts token boundaries based on the input text. This is the focus of our work. Specifically, we introduce a way of *retrofitting* existing (subword-based) LMs with dynamic tokenization.

Our proposed dynamic tokenization approach focuses on improving *efficiency* and *cross-lingual fairness* by repurposing a hypernetwork (HN) introduced by Minixhofer et al. (2024) — originally intended for zero-shot transfer across tokenizers — to support dynamic tokenization; see Table 1 for an illustrative example, and later Figure 1. This adaptation uses the HN to dynamically generate token embeddings on-the-fly, substantially reducing token sequence lengths at minimal performance degradation, and also effectively enabling an *unbounded vocabulary* for encoding text.

This approach, as our extensive experiments demonstrate, is highly beneficial for *prefilling* (computing the key-value states of a prompt) and *scoring* (computing the likelihood of a text) with generative models. However, applying it to autoregressive next-token generation is more challenging since softmax normalization over an unbounded vocabulary is intractable. We thus aim to achieve the benefits of dynamic tokenization for autoregressive generation by expanding to a large, but still bounded vocabulary (in practice, 1M tokens); we introduce a highly efficient method to deal with the large vocabulary. It is based on an approximate nearest neighbor index to overcome the parameter overhead and the softmax bottleneck (Yang et al., 2018) by dynamically retrieving tokens.

Contributions. **1)** We propose an approach of retrofitting LMs with dynamic tokenization, achieving a 22.5% reduction in token sequence length on XNLI and a 26.4% reduction on UNER, with minimal performance degradation. This improves inference speed and leads to fairer compute allocation across languages (see Section 5). **2)** We adapt the same method to *prefilling* and *scoring* in decoder-style LLMs, achieving minimal performance degradation at up to 17% sequence length reduction. **3)** Since naïvely applying our method to autoregressive generation is intractable, we further investigate generation with a large but bounded vocabulary of 1M tokens, achieving additional gains in efficiency. Our code is publicly available at `[Anonymous-GitHub-Repository]`.

2 Background and Related Work

Tokenizers: Preliminaries. We follow the tokenizer definition used by Uzan et al. (2024) and Minixhofer et al. (2024). Let $\mathcal{V}$ denote a *vocabulary*, and T a *tokenization function*. A tokenizer is then a tuple consisting of these two components, $(\mathcal{V}, T)$. The vocabulary $\mathcal{V}$ contains the set of tokens, while the tokenization 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. After tokenizing the input text into a sequence of tokens, each token is then mapped to a continuous vector representation using the embedding function $E_\phi : \mathcal{V} \rightarrow \mathbb{R}^{d_{\text{model}}}$, parameterized by a matrix ϕ. This matrix serves as a lookup table, assigning each token a unique d_{model}-dimensional vector.

Static Tokenizers. Existing tokenizers implement character-, byte-, subword- and word-level tokenization. Character- (El Boukkouri et al., 2020; Tay et al., 2022; Clark et al., 2022) and byte-level (Xue et al., 2022; Yu et al., 2023) methods offer advantages such as small vocabularies and increased robustness to noise, helping in handling rare words and low-resource languages. However, they suffer from reduced processing speed due to longer token sequences or required sequence pooling, impacting training and inference efficiency (Clark et al., 2022; Nawrot et al., 2023). Furthermore, byte-level tokenizers are biased against non-Latin scripts (Limisiewicz et al., 2024).

Word tokenization methods provide faster processing with shorter token sequences, but struggle with out-of-vocabulary (OOV) words and require large vocabularies. A commonly used 'middle ground' is thus subword tokenization, which breaks down the text into smaller, more manageable units, such as pieces of words or entire words. Techniques like Byte-Pair Encoding (BPE; Sennrich et al., 2016), WordPiece (Schuster and Nakajima, 2012) and UnigramLM (Kudo, 2018), handle OOV words by breaking them into known subword units, while also maintaining manageable vocabulary sizes and sequence lengths. Crucially, all these methods are *static*, relying on a predefined vocabulary $\mathcal{V}$ that does not adapt to new data post-training, limiting adaptability to new words or evolving language. This is problematic especially in multilingual contexts, leading to over-segmentation, re-

duced performance, and increased inference costs in languages other than English (Ahia et al., 2023). These issues highlight the need for dynamic tokenization to potentially achieve higher efficiency and more equitable performance across languages.

Vocabulary Expansion. Previous work on adaptive tokenization focused on expanding vocabularies with domain- or language-specific tokens. However, this greatly increases the size of the embedding matrix — sometimes accounting for up to 93% of model parameters (Liang et al., 2023) — which limits how many new tokens can be effectively added and results in inefficient parameter allocation. New token embeddings are typically initialized with heuristics (Minixhofer et al., 2022; Gee et al., 2022; Liu et al., 2024; Gee et al., 2022) and require additional training for optimal performance, restricting real-time adaptation. We use Fast Vocabulary Transfer (FVT; Gee et al., 2022) as a baseline heuristic. FVT generates embeddings for a new token by tokenizing it with the original tokenizer and averaging the embeddings of its subword tokens. Alternative multi-token generation techniques like Copy-Generator (Lan et al., 2023) and Nearest Neighbor Speculative Decoding (Li et al., 2024) use token databases and nearest neighbor retrieval, but face challenges with factual accuracy and computational efficiency. In contrast, our pre-trained HN efficiently generates individual token embeddings removing the need for fine-tuning across domains, addressing both the parameter overhead of vocabulary expansion and the computational requirements of multi-token generation.

Token Embedding Prediction. Instead of relying on heuristics to initialize the embeddings of new tokens, more advanced methods predict them using neural networks. This includes using 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 Schütze (2020) for BERT (Devlin et al., 2019). However, these methods are limited to expanding the existing tokenizers rather than enabling transfer to an entirely different tokenizer. In contrast, Zero-Shot Tokenizer Transfer (ZeTT; Minixhofer et al., 2024) enables transferring LMs to any arbitrary, but *fixed/static* tokenizer. This extends beyond only enabling vocabulary extension to full transfer to a completely new tokenizer while preserving the LM's performance to a large extent in most cases by using a hypernetwork to predict the token embeddings.

3 Methodology

Problem Formulation. *Dynamic tokenization* changes the traditional static encoding process by adaptively adjusting token boundaries based on the input text, continuously updating the vocabulary $\mathcal{V}$ and the tokenization function T. This contrasts with the static tokenization, where $\mathcal{V}$ and T remain fixed post-training. More formally, let the initial tokenizer be $(\mathcal{V}_{\text{init}}, T_{\text{init}})$. As the LM operates with new text data $\mathcal{D}$, the tokenization function T_{init} is updated to T_{new}. The update process can be represented by the function $\mathcal{U}$:

$$T_{\text{new}}(\mathcal{D}) = \mathcal{U}(T_{\text{init}}(\mathcal{D})) \tag{1}$$

To retrofit an LM pre-trained with subword tokenization to dynamic tokenization, two steps are required: *(1)* deciding on a tokenization T_{new}; and *(2)* obtaining the token embeddings. This approach can be applied to any case where the (subword-level) token sequence is known in advance.

3.1 Dynamic Tokenization via BPE-Style Compression

Deciding on a Dynamic Tokenization. Let $\mathcal{D}$ represent the input data to be tokenized. The first step in dynamic tokenization involves updating the initial tokenization T_{init} to a new function T_{new}, using the update function $\mathcal{U}$. Since our focus is on efficiency, this update aims to minimize over-segmentation in the input data $\mathcal{D}$, resulting in a more compact representation for $\mathcal{D}$ (i.e., $|T_{\text{new}}(\mathcal{D})| \leq |T_{\text{init}}(\mathcal{D})|$).

Importantly, given that LMs operate at *batch-level*, $\mathcal{U}$ is specifically applied at this level on $\mathcal{D}_{\text{batch}}$. This allows $\mathcal{U}$ to dynamically adapt the tokenization to the unique linguistic features in each batch.

To define $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}))$, we take inspiration from BPE (Sennrich et al., 2016). Specifically, for each batch $\mathcal{D}_{\text{batch}}$ tokenized under the initial scheme T_{init}, we begin with a *batch-specific vocabulary* $\mathcal{V}_{\text{new}}$ comprised of all unique subword tokens present in $T_{\text{init}}(\mathcal{D}_{\text{batch}})$. We then perform a fixed number of merge operations, m, combining the most frequent adjacent tokens within the batch, continuously refining the tokenization to better compress $\mathcal{D}_{\text{batch}}$.

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

$$\begin{aligned} &\mathcal{U} : (T_{\text{init}}(\mathcal{D}), m) \rightarrow T_{\text{new}}(\mathcal{D}) \\ &\text{with} \quad |T_{\text{new}}(\mathcal{D})| \leq |T_{\text{init}}(\mathcal{D})|, \end{aligned} \tag{2}$$

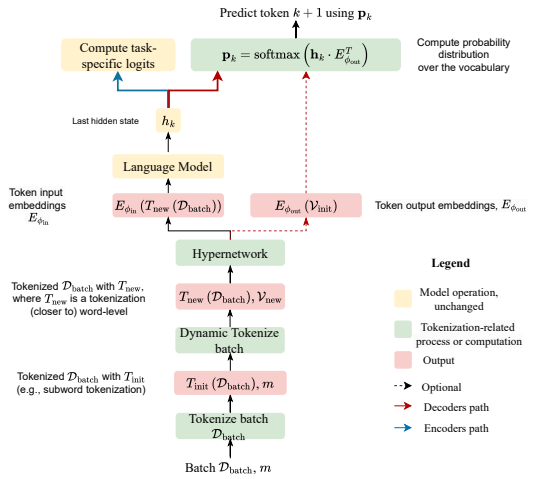


Figure 1: Dynamic tokenization applied to encoders and decoders LMs.

where m represents the number of merge operations to perform. Since the BPE-style merging process is applied to $T_{\text{init}}(\mathcal{D}_{\text{batch}})$ — the data we wish to tokenize — we implicitly tokenize this batch under the new tokenization scheme $T_{\text{new}}(\mathcal{D}_{\text{batch}})$ by sequentially applying merge operations. This allows us to simplify the training and tokenization processes of traditional BPE into a single, unified algorithm, outlined in Appendix A.[1]

Subword-level tokenization represents the starting point or the *lower-bound* for the new tokenization, T_{new}, obtained when $m = 0$, and equivalent with T_{init}. On the other hand, we consider word-level, or, more precisely, pre-token-level,[2] as the *upper-bound* for T_{new}. In other words, we constrain the merging process to never merge adjacent tokens which are part of different words.

Obtaining Token Embeddings. After mapping the tokens from the initial tokenization to a more compact tokenization, $T_{\text{init}}(\mathcal{D}_{\text{batch}}) \rightarrow T_{\text{new}}(\mathcal{D}_{\text{batch}})$, we need to obtain the embeddings for all tokens $t \in T_{\text{new}}(\mathcal{D}_{\text{batch}})$. To achieve this, we repurpose the HN trained by Minixhofer et al. (2024). While the HN was originally intended to transfer an LM to a fixed, static tokenizer, we observe that it can also be used to achieve dynamic tokenization: since the HN *amortizes* over the tokenization function (i.e., embedding predictions for every token are independent of each other), it does not require a static (or even bounded) vocabulary. Therefore, for each $t \in T_{\text{new}}(\mathcal{D}_{\text{batch}})$, we apply the hypernetwork H_θ to obtain its embedding:

$$E_{\phi_{\text{new}}}(t) = H_\theta(t), \quad \forall t \in T_{\text{new}}(\mathcal{D}_{\text{batch}}) \tag{3}$$

where $E_{\phi_{\text{new}}}(t)$ is the embedding for token t and ϕ_{new} is the matrix corresponding to the batch-specific vocabulary, $\mathcal{V}_{\text{new}}$. This process can alternatively also be viewed as transferring the LM to a new tokenizer $(\mathcal{V}_{\text{new}}, T_{\text{new}})$ for each batch, dynamically adjusting token boundaries based on the specific data within that batch. Recall that it is applicable to any case where the token sequence is known in advance, i.e., any use-case of encoder-style LMs, as well as prefilling and scoring of generative (decoder-style) LMs, as shown in Figure 1.

4 Experimental Setup

Models. We use XLM-R (Conneau et al., 2020) as the representative multilingual encoder-style LM. To test our method on a decoder-style model LM, we use both the base and instruct versions of Mistral-7B (Jiang et al., 2023). This choice is partially due to the fact that the two established models come with pre-trained HNs.

Datasets. For our XLM-R experiments, we use two datasets: Cross-lingual Natural Language Inference (XNLI; Conneau et al., 2018), and Universal Named Entity Recognition (UNER; Mayhew et al., 2024). These datasets quantify the effect of dynamic tokenization across a total of 14 languages, with XNLI focusing on *sentence-level* and UNER on *token-level* understanding.[3] For Mistral-7B experiments, we use the following evaluation benchmarks: the "lite" version of Global-MMLU (Singh et al., 2024) in English, French, German, Spanish and Portuguese, and the English Multi-Turn Benchmark (MT-Bench; Chiang and Lee, 2023).

Embeddings. We compare the performance of the model using (i) the original embeddings, (ii) FVT embeddings[4] (see Section 2) and (iii) HN-generated embeddings.

[1] Additionally, since the new tokenization is only applied to the specific batch data $\mathcal{D}_{\text{batch}}$ and not reused for other text, there is no need to store or compute new merge rules $\mathcal{M}_{\text{new}}$ or maintain an expanded vocabulary $\mathcal{V}_{\text{new}}$.

[2] Pre-tokens are preliminary units often equivalent to words (Mielke et al., 2021). For simplicity of terminology, we use *pre-tokens* and *words* interchangeably, but note that 'pre-token' is the more precise term.

[3] For XNLI, we evaluate on 13 different languages: 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.

[4] We use FVT since it achieves comparable performance to FOCUS (Dobler and de Melo, 2023) while being substantially faster than FOCUS (Minixhofer et al., 2024), which is crucial for our dynamic setup.

Hyperparameters. Appendix B details the hyperparameter settings used in our experiments.

4.1 Experiments with Encoder Models

We train a LoRA adapter (Hu et al., 2022) for both *task* — natural language inference for XNLI and named entity recognition for UNER — and *dynamic tokenization* adaptation. The adapter jointly learns to adapt to the task and operate with coarser token granularities. We perform two experiments: *(1)* training an adapter with a fixed number of merges m and *(2)* training an adapter with m sampled from a Uniform distribution.[5]

(1) Predetermined Number of Merges. Here, we train an adapter with dynamic tokenization that reduces sequence length by a fixed percentage of the maximum possible reduction. We set this percentage to 50% for XNLI and 75% for UNER. Note that 100% reduction corresponds to the difference between the initial sequence length — obtained when tokenizing with T_{init} — and the sequence length obtained with word-level tokenization (i.e., the number of words in the sequence). We apply the function $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$ for each batch to meet the specific reduction percentage in sequence length. This is necessary since the correlation between the number of merges m and the sequence length reduction varies across languages and datasets; for instance, 140 merges achieve 100% relative sequence reduction on English XNLI, whereas 250 merges are required for the same reduction on Turkish XNLI.

(2) Sampling from a Uniform Distribution. In the second approach, instead of using a fixed number of merges m and applying the function $\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m)$ with the same m across the training batch, we introduce stochasticity into the tokenization process. Specifically, we explore the impact of sampling different numbers of merges from a Uniform distribution. By training the adapter with tokenizations sampled from this distribution, we hypothesize that the model will learn to be more robust to the type of dynamic tokenization used (i.e., the value of m). We sample a tokenizer per batch (i.e., a fixed m) rather than a tokenizer for each sample in the batch due to the high computational requirements of the latter. The tokenization function applied during training is then:

$$\mathcal{U}(T_{\text{init}}(\mathcal{D}_{\text{batch}}), m), \quad m \sim \mathrm{U}(0, m_{\text{max}}) \qquad (4)$$

where m_{max} is determined by $\mathcal{D}$ and represents the merge level yielding word-level tokenization.

4.2 Experiments with Decoder Models

Unlike XLM-R, we do not train the Mistral-7B decoder model as we evaluate our method out-of-the-box on a pretrained checkpoint. We apply dynamic tokenization with a fixed number of merges m to the input batch $\mathcal{D}_{\text{batch}}$. We evaluate performance trends across all sequence reduction percentages, 0% to 100%, where again 100% corresponds to the reduction achieved with word-level tokenization. For prefilling, we compute the key-value states of the dynamically tokenized input sequence. For scoring, our goal is to compute the conditional probability $p(\text{suffix}|\text{prefix})$. Our key insight enabling dynamic tokenization is that we do not need to compute $p(\text{prefix})$ to compute the normalized conditional probability of some suffix relative to other suffixes. This means we can dynamically tokenize the prefix, then use the original tokenization to tokenize (and evaluate the probability of) the suffix. In practice, e.g., for MMLU this means processing each input prompt using dynamic tokenization only keeping the last hidden state $\mathbf{h}$, and compute a probability distribution over the four answer choices using $\mathbf{h}$ with the embeddings of the answer choices `A`, `B`, `C` and `D`. This setup also allows evaluating the quality of the HN output embeddings by comparing the performance when the suffix sequence is embedded with the original embeddings versus the HN-generated embeddings.

5 Results and Discussion

5.1 Encoder Models

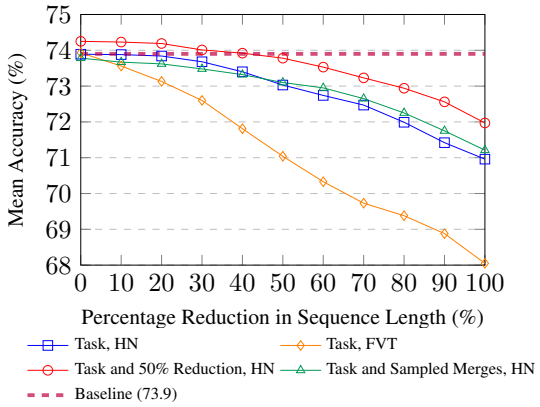
Evaluation results for XLM-R with task adapters and joint task and tokenization adapters, using different tokenization and embedding strategies are summarized in Table 2 for XNLI and Table 3 for UNER. Figure 2a illustrates the average performance across languages in XNLI with different sequence length reductions and adapters, while Figure 2b focuses on English-only results. Corresponding results for UNER are shown in Figure 3.

Dynamic tokenization substantially reduces sequence lengths. Applying dynamic tokenization with an adapter jointly trained for the task and a

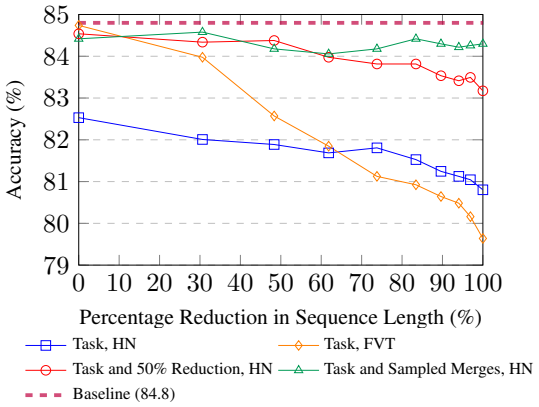
[5] Our preliminary experiments included selecting m from a Gaussian distribution. However, this yielded suboptimal results. Additionally, we investigated disentangling task adaptation from tokenization adaptation, but this also led to suboptimal results; future work could re-investigate whether disentangling the task and the tokenization is possible.

Tokenization & Embeddings	Adapter	Accuracy per Language (%)													
		ar	bg	de	el	en	es	fr	hi	ru	sw	tr	ur	vi	Avg.
(1) original	task	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	task	**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	task	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	task	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
(5) word, HN	task + $m_{50\%}$	**67.8**	**74.2**	74.3	**72.4**	83.2	**78.3**	75.7	**66.6**	**72.9**	**61.3**	**67.5**	**66.4**	**75.0**	**72.0**
(6) word, HN	task + m_{sampled}	66.5	74.1	74.5	71.6	**84.3**	77.0	**75.9**	64.9	72.7	58.8	66.5	65.1	73.7	71.2
$\Delta_{\text{Acc. (\%)}}$ (1), (5)		-3.8	-2.3	-2.6	-2.7	-1.6	0.3	-2.8	-2.1	-2.0	-1.9	-4.9	1.0	1.1	-1.9
$\Delta_{\text{Length (\%)}}$ original (1, 2), word (3-6)		-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 2: Accuracy on **XNLI** `validation` with different adapters, tokenizations and embeddings. $\Delta_{\text{Acc. (\%)}}$ is the absolute accuracy change between word-level tokenization with the optimal adapter and HN embeddings (5) and the baseline (1) which uses original tokenization and embeddings. $\Delta_{\text{Length. (\%)}}$ represents the average decrease in token sequence length of word-level tokenization over the original. Boldface indicates the best result for a language when using *original* subword-level tokenization or *word-level* tokenization.



(a) Mean accuracies for XNLI across 13 languages.



(b) Accuracies for different adapters on **English**.

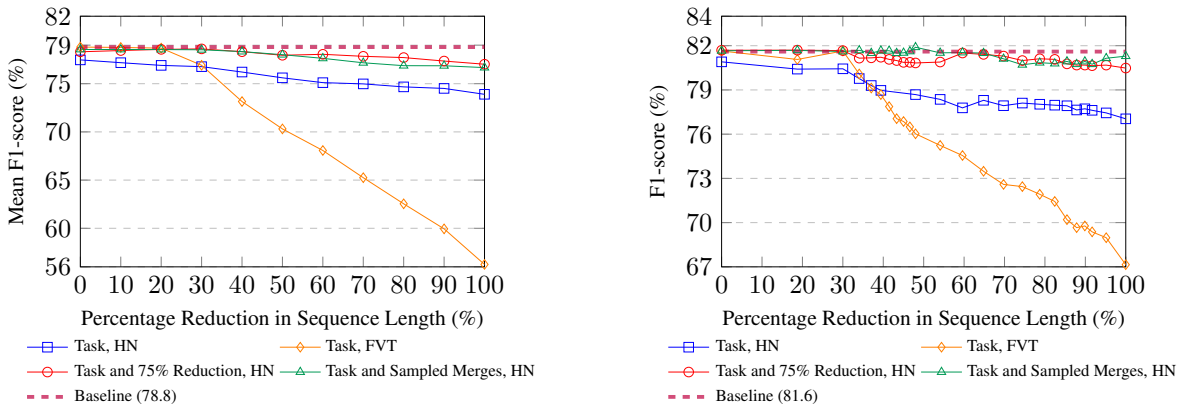
Figure 2: Accuracies on **XNLI** with different adapters as a function of sequence length reduction (%). Adapter names follow the format: *Adapter type, Embeddings used*. In this and subsequent figures, a 0% reduction refers to sequence length obtained with the original subword-level tokenization, while 100% indicates 'upper bound' word-level tokenization. Intermediate percentages show proportional reductions between these two extremes.

Tokenization & Embeddings	Adapter	Language F1-score (%)					
		en_ewt	de_pud	pt_bosque	pt_pud	ru_pud	Avg.
(1) original	task	**81.6**	78.0	**82.3**	**82.9**	**69.0**	**78.8**
(2) original, HN	task	80.9	**78.3**	80.8	82.3	68.4	78.1
(3) word, HN	task	77.0	75.8	77.6	77.3	65.5	74.6
(4) word, FVT	task	67.2	57.0	58.0	58.4	40.7	56.3
(5) word, HN	task + $m_{75\%}$	80.5	75.0	**80.5**	**81.3**	**67.9**	**77.0**
(6) word, HN	task + m_{sampled}	**81.3**	**76.3**	78.5	80.2	67.1	76.7
$\Delta_{\text{F1-score (\%)}}$ (1), (5)		-1.1	-3.0	-1.8	-1.6	-1.1	-1.7
$\Delta_{\text{Length (\%)}}$ original (1, 2), word (3-6)		-17.6	-30.5	-24.1	-24.2	-35.8	-26.4

Table 3: F1-score on **UNER** with different adapters, tokenizations and embeddings. The results reported are on the `validation` split for `ewt` and `bosque` datasets, and `test` split for pud due to the availability.

specified number of merges reduces token sequence length by an average of 22.5% on the XNLI dataset (Table 2), with an average accuracy decrease of 1.9% compared to the original tokenization and embeddings. On the UNER dataset (Table 3), this approach achieves a 26.4% reduction in sequence length, with only a 1.7% decrease in F1-score.

Sampling the tokenization granularity improves in-domain performance. Comparing our two types of adapters, we find that, for both datasets, the adapter trained with m sampled from $\mathrm{U}(0, m_{\max})$ on average outperforms the fixed-merge adapter on English (i.e., in-domain). This adapter yields better results across all sequence length reductions



(a) Mean F1-scores for UNER across 5 languages.

(b) F1-scores for different UNER adapters on **English**.

Figure 3: F1-scores on **UNER** with different adapters as a function of sequence length reduction (%).

and nearly closes the gap with the baseline performance with the original tokenization and embeddings. For XNLI, with word-level tokenization, it obtains an accuracy of 84.3%, compared to 83.2% with the fixed-merge adapter and 84.8% with the baseline (Figure 2b). These results align with the UNER results where the adapter trained with sampled merges achieves an F1-score of 81.3% on word-level tokenization, compared with the fixed-merge adapter at 80.5% and close to the baseline of 81.6% (Figure 3b), confirming that the model benefits from a balanced exposure to different tokenization granularities on in-domain tasks.

Cross-lingual performance. Unlike our in-domain results, the fixed-merges adapter consistently shows stronger cross-lingual transferability than its counterpart trained with sampled merges. On XNLI, it obtains an average accuracy of 72.0% compared to 71.2% (Table 2). Similarly, for UNER, it achieves 77.0% compared to 76.7% (Table 3).

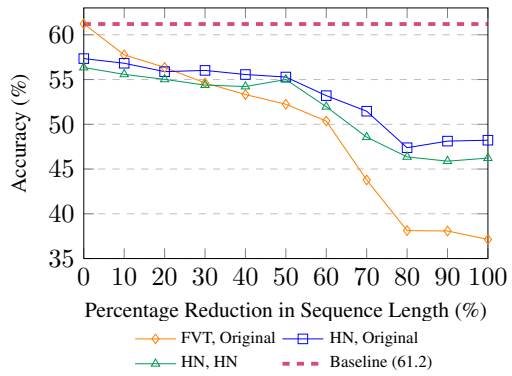
Heuristic embeddings $\ll$ HN-generated embeddings $<$ original embeddings. Using original tokenization with HN embeddings (Setting 2 in Table 2) shows comparable results to original embeddings (Setting 1), in both English and cross-lingual contexts, highlighting the quality of HN embeddings. However, there is a noticeable gap between subword- and word-level HN embeddings in Settings 1 and 3, as the model was not previously exposed to HN embeddings. FVT embeddings, by contrast, show a prominent performance drop: for instance, FVT achieves an average accuracy of 68.2% on XNLI, compared to 71.1% (Table 2) with word-level HN embeddings, and scores 56.3% F1 on UNER, substantially lower than the 74.6% achieved with HN embeddings (Table 3). This suggests that HN embeddings more effectively capture the semantic nuances required for tasks like NER, where accurate token representation is important.

5.2 Decoder Models

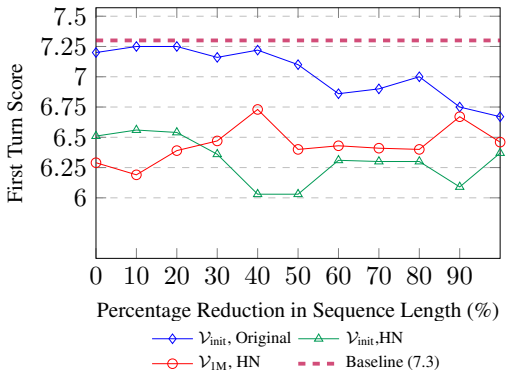
Figure 4 presents performance trends for different granularities obtained using dynamic tokenization for scoring and prefilling (see Section 4.2).

Dynamic tokenization improves prefilling and scoring efficiency. In zero-shot evaluation of dynamic tokenization across different granularities, we observe that using the original *output embeddings* consistently outperform the HN-generated output embeddings, highlighting that the degradation in performance is primarily due to the type of output embeddings used rather than the input embeddings (Figure 4). For both MMLU and MT-Bench, reducing the sequence length by 40-50% (relative to the word-level) — corresponding to a 17% average reduction for MMLU and $\approx 6\%$ for MT-Bench (in absolute terms) — and using the original output embeddings (from $\mathcal{V}_{\text{init}}$) yields the smallest gap to the baseline, unlike FVT, whose performance quickly degrades above 30% reduction. Overall, we can use this insight to compress the key-value cache with minimal performance degradation by exclusively changing the input embeddings i.e., without *any* changes to the pre-trained model.

Expanding the vocabulary allows achieving some of the benefits of dynamic tokenization for autoregressive generation. Our dynamic tokenization approach works for LM *scoring* (i.e., computing the conditional probability of a text) and *prefilling*, as we know the sequence in advance. However, this is not the case for autoregressive generation. To address this, we propose a method that



(a) MMLU, average accuracies across 5 languages. Average sequence length at 0%: 905.9 and 100%: 598.1



(b) MT-Bench, English. Average sequence length at 0%: 804.4 and 100%: 692.0

Figure 4: Performance of dynamic tokenization applied to decoders for scoring and prefilling, evaluated across various vocabularies and embeddings in a zero-shot setting. For MMLU, the *baseline* is the accuracy with the original tokenization and embeddings, while for MT-Bench, it is the first-turn score with the original setup. Labels follow the format: a) MMLU: Embeddings used for the dynamic tokenized input and output embeddings used for scoring; b) MT-Bench: Vocabulary $\mathcal{V}_{\text{gen}}$ and embeddings used for generation. $\mathcal{V}_{1\text{M}}$ expands $\mathcal{V}_{\text{init}}$ to 1M tokens.

expands the initial vocabulary, $\mathcal{V}_{\text{init}}$, from ≈32k tokens to 1M tokens, $\mathcal{V}_{1\text{M}}$, moving token granularity closer to word-level while maintaining a large bounded vocabulary. In a nutshell, our approach involves three steps: *(1)* expanding $\mathcal{V}_{\text{init}}$ to $\mathcal{V}_{1\text{M}}$ by applying BPE on a large corpus; *(2)* using longest-prefix (LP) tokenization — instead of the previous dynamic tokenization — to overcome the challenges involved in merging two tokenizers (i.e., the initial tokenizer and the one obtained for $\mathcal{V}_{1\text{M}}$); and *(3)* constructing an approximate nearest neighbor index to efficiently retrieve token embeddings using an HN. This approach is described in full technical detail in Appendix C and Tables 9 and 10 summarize the results with different settings.

Using this expanded vocabulary we achieve shorter token sequences — similar to word-level tokenization — at the expense of degrading the performance by 5.9% for MMLU (English) and 0.9 points on MT-Bench (Table 9 and Table 10). Additionally, retrieving the embeddings for tokens in $\mathcal{V}_{1\text{M}}$ on-the-fly using an HN allows us to maintain the original model's parameter count. The current performance gap could potentially be minimized through n-shot tokenizer transfer, as demonstrated by Minixhofer et al. (2024), but we leave this exploration beyond zero-shot setups to future research.

Throughput analysis. Table 4 shows that dynamic tokenization reduces the main model's FLOPs (e.g., 14.4T to 9.7T in French, 67.4% of the baseline) while the hypernetwork's FLOPs remain below 3% of the total. The gains align with sequence reduction, yielding near-linear throughput improvements with negligible overhead from the hypernetwork. Appendix E shows the complete set of results.

Lng	FLOPs	Sequence Reduction		
		0%	50%	100%
en	Model	10.1T	9.4T	8.5T
	Hypernet	169.3B (1.7%)	191.0B (2.0%)	199.8B (2.2%)
fr	Model	14.4T	12.2T	9.7T
	Hypernet	91.8B (0.6%)	163.5B (1.3%)	238.5B (2.4%)

Table 4: *FLOPs per sample* with dynamic tokenization on multilingual MMLU. Percentages in parentheses denote the fraction of hypernetwork FLOPs out of total.

6 Conclusion

We proposed a novel dynamic tokenization method for (large) language models, using a hypernetwork to dynamically adapt token boundaries based on the input data, which efficiently generates token embeddings on-the-fly. We then demonstrated its usefulness both on encoder- and decoder-style models. As some main findings, we highlight that for encoder models (e.g., XLM-R), our approach substantially reduces token sequence lengths by $> 20\%$ on average over 14 languages, with less than 2% loss in accuracy. When applied to decoder-style models (e.g., Mistral-7B) for prefilling and scoring, our method yields minimal performance degradation with up to 6% reduction in (absolute) sequence length on English. Overall, these results demonstrate that dynamic tokenization can mitigate some of the limitations of static tokenizers, particularly in multilingual settings, improving inference efficiency and promoting fairness across languages.

Limitations

One limitation of our study includes the computational overhead associated with *(1)* generating a new vocabulary for each batch, which increases with the number of merges m; and *(2)* on-the-fly generation of token embeddings using an HN (c.f., Minixhofer et al., 2024). To amortize the latter, we implemented an HN embeddings cache, particularly motivated by the frequent repetition of certain tokens (e.g., "the"; c.f., Appendix D). Additionally, the success of on-the-fly token embeddings relies on the accuracy and robustness of the hypernetwork. Any limitations in the hypernetwork's training or design might directly impact the quality of the token embeddings and, consequently, the overall model performance, but research into a more sophisticated design and training of the hypernetworks goes beyond the scope of this work.

For the main experiments we have opted for representative and established encoder and decoder models for which the pre-trained hypernetworks are already available (Minixhofer et al., 2024) - obtaining similar (or improved, as mentioned above) hypernetworks for other LMs would make our dynamic tokenization approach applicable to other LMs in future work.

Another limitation is that our current dynamic tokenization approach primarily works in settings where the full sequence is known in advance (e.g., scoring and prefilling in decoder models). For autoregressive generation, we offered a first solution which relies on a large but static, bounded vocabulary to achieve some of the benefits of dynamic tokenization (e.g., token sequence length compression). Closing this gap by integrating "true" dynamic tokenization into autoregressive generation remains an open challenge for future research.

Finally, our dynamic tokenization approach operates at the batch-level, chosen to gain a broader context of co-occurring tokens across multiple token sequences; this helps in identifying effective merges. However, future research might explore sample-level dynamic tokenization approaches, particularly given that decoders (e.g., chatbots) often operate at sample-level.

References

Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David Mortensen, Noah Smith, and Yulia Tsvetkov. 2023. Do all languages cost the same? tokenization in the era of commercial language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9904–9923, Singapore. Association for Computational Linguistics.

Mehdi Ali, Michael Fromm, Klaudia Thellmann, Richard Rutmann, Max Lübbering, Johannes Leveling, Katrin Klug, Jan Ebert, Niclas Doll, Jasper Buschhoff, Charvi Jain, Alexander Weber, Lena Jurkschat, Hammam Abdelwahab, Chelsea John, Pedro Ortiz Suarez, Malte Ostendorff, Samuel Weinbach, Rafet Sifa, Stefan Kesselheim, and Nicolas Flores-Herr. 2024. Tokenizer choice for LLM training: Negligible or crucial? In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 3907–3924, Mexico City, Mexico. Association for Computational Linguistics.

Cheng-Han Chiang and Hung-yi Lee. 2023. Can large language models be an alternative to human evaluations? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15607–15631, Toronto, Canada. Association for Computational Linguistics.

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

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard 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*, pages 8440–8451, Online. Association for Computational Linguistics.

Alexis Conneau, Ruty Rinott, Guillaume Lample, Adina Williams, Samuel Bowman, Holger Schwenk, and Veselin Stoyanov. 2018. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2475–2485, Brussels, Belgium. Association for Computational Linguistics.

Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. 2024. Getting the most out of your tokenizer for pre-training and domain adaptation. *ArXiv preprint*, abs/2402.01035.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Konstantin Dobler 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*, pages 13440–13454, Singapore. Association for Computational Linguistics.

Hicham El Boukkouri, Olivier Ferret, Thomas Lavergne, Hiroshi Noji, Pierre Zweigenbaum, and Jun'ichi Tsujii. 2020. CharacterBERT: Reconciling ELMo and BERT for word-level open-vocabulary representations from characters. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6903–6915, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Takuro Fujii, Koki Shibata, Atsuki Yamaguchi, Terufumi Morishita, and Yasuhiro Sogawa. 2023. How do different tokenizers perform on downstream tasks in scriptio continua languages?: A case study in Japanese. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 39–49, Toronto, Canada. Association for Computational Linguistics.

Leonidas Gee, Andrea Zugarini, Leonardo Rigutini, and Paolo Torroni. 2022. Fast vocabulary transfer for language model compression. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 409–416, Abu Dhabi, UAE. Association for Computational Linguistics.

Siavash Golkar, 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. *ArXiv preprint*, abs/2310.02989.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3887–3896. PMLR.

Valentin Hofmann, Hinrich Schuetze, and Janet Pierrehumbert. 2022. An embarrassingly simple method to mitigate undesirable properties of pretrained language model tokenizers. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 385–393, Dublin, Ireland. Association for Computational Linguistics.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Albert Q Jiang, 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. *ArXiv preprint*, abs/2310.06825.

Taku Kudo. 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)*, pages 66–75.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Sneha Kudugunta, Isaac Caswell, Biao Zhang, Xavier Garcia, Derrick Xin, Aditya Kusupati, Romi Stella, Ankur Bapna, and Orhan Firat. 2023. MADLAD-400: A multilingual and document-level large audited dataset. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Tian Lan, Deng Cai, Yan Wang, Heyan Huang, and Xian-Ling Mao. 2023. Copy is all you need. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Minghan Li, 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. *ArXiv preprint*, abs/2405.19325.

Davis Liang, 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 *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13142–13152, Singapore. Association for Computational Linguistics.

Tomasz Limisiewicz, Terra Blevins, Hila Gonen, Orevaoghene Ahia, and Luke Zettlemoyer. 2024. MYTE: Morphology-driven byte encoding for better and fairer multilingual language modeling. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15059–15076, Bangkok, Thailand. Association for Computational Linguistics.

Yihong Liu, Peiqin Lin, Mingyang Wang, and Hinrich Schuetze. 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, pages 1067–1097, Mexico City, Mexico. Association for Computational Linguistics.

Stephen Mayhew, Terra Blevins, Shuheng Liu, Marek Suppa, Hila Gonen, Joseph Marvin Imperial, Börje Karlsson, Peiqin Lin, Nikola Ljubešić, Lester James Miranda, Barbara Plank, Arij Riabi, and Yuval Pinter. 2024. Universal NER: A gold-standard multilingual named entity recognition benchmark. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4322–4337, Mexico City, Mexico. Association for Computational Linguistics.

Sabrina J Mielke, 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. *ArXiv preprint*, abs/2112.10508.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. *ArXiv preprint*, abs/2402.06196.

Benjamin Minixhofer, 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*, pages 3992–4006, Seattle, United States. Association for Computational Linguistics.

Benjamin Minixhofer, Edoardo Maria Ponti, and Ivan Vulić. 2024. Zero-shot tokenizer transfer. *ArXiv preprint*, abs/2405.07883.

Piotr Nawrot, Jan Chorowski, Adrian Lancucki, and Edoardo Maria Ponti. 2023. Efficient transformers with dynamic token pooling. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6403–6417, Toronto, Canada. Association for Computational Linguistics.

Yuval Pinter, Robert Guthrie, and Jacob Eisenstein. 2017. Mimicking word embeddings using subword RNNs. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 102–112, Copenhagen, Denmark. Association for Computational Linguistics.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. 2021. How good is your tokenizer? on the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3118–3135, Online. Association for Computational Linguistics.

Timo Schick and Hinrich Schütze. 2019. Attentive mimicking: Better word embeddings by attending to informative contexts. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 489–494, Minneapolis, Minnesota. Association for Computational Linguistics.

Timo Schick and Hinrich Schütze. 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*, pages 3996–4007, Online. Association for Computational Linguistics.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Shivalika Singh, Angelika Romanou, Clémentine Fourrier, David I Adelani, Jian Gang Ngui, Daniel Vila-Suero, Peerat Limkonchotiwat, Kelly Marchisio, Wei Qi Leong, Yosephine Susanto, et al. 2024. Global mmlu: Understanding and addressing cultural and linguistic biases in multilingual evaluation. *arXiv preprint arXiv:2412.03304*.

Lichao Sun, 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. *ArXiv preprint*, abs/2003.04985.

Yi Tay, Vinh Q. Tran, Sebastian Ruder, Jai Prakash Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler. 2022. Charformer: Fast character transformers via gradient-based subword tokenization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Cagri Toraman, Eyup Halit Yilmaz, Furkan Şahinuç, and Oguzhan Ozcelik. 2023. Impact of tokenization on language models: An analysis for turkish. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 22(4):1–21.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *ArXiv preprint*, abs/2307.09288.

Omri Uzan, Craig W Schmidt, Chris Tanner, and Yuval Pinter. 2024. Greed is all you need: An evaluation of tokenizer inference methods. *ArXiv preprint*, abs/2403.01289.

Xinyi Wang, Sebastian Ruder, and Graham Neubig. 2021. Multi-view subword regularization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 473–482, Online. Association for Computational Linguistics.

Frank F. Xu, Uri Alon, and Graham Neubig. 2023. Why do nearest neighbor language models work? In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 38325–38341. PMLR.

Linting Xue, 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. *Transactions of the Association for Computational Linguistics*, 10:291–306.

Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. 2018. Breaking the softmax bottleneck: A high-rank RNN language model. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Lili Yu, Daniel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. 2023. MEGABYTE: predicting million-byte sequences with multiscale transformers. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

A Dynamic Tokenization Algorithm for Encoder LMs

Algorithm 1 shows the simplified version of the byte-pair encoding inspired merging process we use to dynamic tokenize (i.e., compress) a sequence of tokens.

B Reproducibility Details

A summary of the hyperparameters we used for training and evaluation is shown in Table 5 and Table 6. For decoders, specifically for MMLU, we used the template shown in Figure 5 with a maximum sequence length of 8192. Additionally, both MMLU and MT-Bench were run with `bfloat16` precision and a batch size of 1 due to computational constraints. For MT-Bench specifically, we set the maximum number of tokens to be generated to 1024. Finally, all the experiments were conducted using an NVIDIA GeForce RTX 4090 GPU with 24 GB of VRAM, powered by the NVIDIA driver version 525.105.17 and CUDA version 12.0.

C Vocabulary Expansion for Dynamic Autoregressive Generation

Dynamic tokenization can not be applied in the same way described in Section 3.1 for autoregressive generation since it results in an unbounded vocabulary. However, to still achieve some of the benefits of dynamic tokenization, we introduce a method that expands $\mathcal{V}_{\text{init}}$ to a large (but bounded) size for improved inference efficiency in token generation.

Our approach, similar to CoG (Lan et al., 2023) and NEST (Li et al., 2024) in its training-free domain adaptation, uses a large token vocabulary instead of a phrase table, reducing the need for billions of phrases. We significantly expand the initial vocabulary to $\mathcal{V}_{\text{large}}$, aiming to include more specialized terms and word variations in English. This moves token granularity from subword-level closer to word-level, improving efficiency of the generation process. This vocabulary, while static, integrates with the LM using HN-generated embeddings from Minixhofer et al. (2024), providing a similar flexibility to a dynamic approach, without the need for training the embeddings. Although this approach does not currently include dynamic updates of $\mathcal{V}_{\text{large}}$, it sets the foundation for future dynamic vocabulary adjustments.

Our proposed approach for decoder LMs requires three steps: *(1)* expanding the vocabulary to a large size; *(2)* deciding on a tokenization; *(3)* constructing an approximate nearest neighbor (ANN) index and populating it with token embeddings.

Expanding the Vocabulary. 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 tokenizers such as BPE, WordPiece or UnigramLM on a large corpus to obtain a vocabulary of $|\mathcal{V}_{\text{large}}| - |\mathcal{V}_{\text{init}}|$ tokens.[6] In our method, we use BPE algorithm to find $\mathcal{V}_{\text{new}}$ and $\mathcal{M}_{\text{new}}$. We then obtain $\mathcal{V}_{\text{large}} = \mathcal{V}_{\text{init}} \cup \mathcal{V}_{\text{new}}$.

Deciding on a Tokenization Function. Although we have obtained the new merge rules $\mathcal{M}_{\text{new}}$ specific to $\mathcal{V}_{\text{new}}$, integrating these with the source tokenizer's existing rules, $\mathcal{M}_{\text{init}}$, is challenging. This is because the original and new merge rules, even if both derived from BPE, were learned independently and are stored sequentially. An example illustrating the challenges associated with merging $\mathcal{M}_{\text{new}}$ and $\mathcal{M}_{\text{init}}$ is presented in Table 7.

These challenges highlight the complexity involved in merging tokenizers and the need for a tokenization function that facilitates merging. To address this, we use a Longest-Prefix (LP) tokenization function,[7] denoted T_{LP}, similar to the default method used by WordPiece when the continuation prefix is set to `blank` (i.e., no character).

Obtaining Token Embeddings and Index Construction. Similar to the previous experiments, we use an HN 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 with the expanded vocabulary. The next token is generated as follows:

Step 1: Input Tokenization. Tokenize 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 tokenized input to the LM;

Step 4: Output Embeddings. Obtain the output embeddings for each token in the vocabulary

[6] 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-tokenized input

[7] Uzan et al. (2024) show that LP greedy tokenization performs on par or better than other tokenization functions.

Algorithm 1 *Dynamic Tokenization for Encoders*

Input: Tokenized batch data *tokenizedBatch* under initial tokenization, T_{init} ($\mathcal{D}_{\text{batch}}$); number of merges m
Output: Tokenized batch data under a new, dynamically learned tokenization, T_{new} ($\mathcal{D}_{\text{batch}}$)

procedure APPLYDYNAMICTOKENIZATION(*tokenizedBatch*, m)
  for $i \leftarrow 1$ **to** m **do**
    pairFreqs $\leftarrow$ ComputePairFreqs(*tokenizedBatch*)
    bestPair $\leftarrow$ GetMostFrequentPair(*pairFreqs*)
    Apply *bestPair* merge rule to *tokenizedBatch*
  end for
  return *tokenizedBatch* ▷ $\mathcal{D}_{\text{batch}}$ as T_{new} ($\mathcal{D}_{\text{batch}}$)
end procedure

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

Table 5: Summary of random seeds used across different experiments.

Hyperparameter	**XNLI**: Task Adapter with Original Subword Tokenization	**XNLI**: Joint Task and Dynamic Tokenization Adapter	**UNER**: Task Adapter with Original Subword Tokenization & Joint Task and Dynamic Tokenization Adapter
Matrix Rank r	32	128	256
Scaling Factor α	64	256	512
Dropout	0.3		
Epochs	10	{10, 15}	15
Learning Rate	3×10^{-4}	1×10^{-4}	3×10^{-4}
Batch Size	32		
Optimiser	AdamW		
Optimiser Parameters	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
Scheduler	Linear, no warmup steps		
Max Sequence Length	128		
Weight Decay	0		
Precision	bfloat16		

Table 6: Summary of hyperparameters used for LoRA training.

$\mathcal{V}_{\text{large}}$: $E_{\phi_{\text{out}}}(\mathcal{V}_{\text{large}})$;

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 of this approach lies in Step 5, involving a costly dot product calculation between the last hidden state $\mathbf{h}$ and the output embeddings transposed matrix $E_{\phi_{\text{out}}}(\mathcal{V}_{\text{large}})^\top$, due to the large vocabulary size. To mitigate this,

Prompt Template: 5-shot from the same domain as the current question

⟨QUESTION_1⟩
⟨ANSWERS_1⟩
Answer: ⟨ANSWER_1⟩
.
.
.
⟨QUESTION_5⟩
⟨ANSWERS_5⟩
Answer: ⟨ANSWER_5⟩

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

Figure 5: 5-shot prompt template used for MMLU evaluation

	Tokenizer 1	**Tokenizer 2**	**Merged tokenizer**
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 tokenize: 'ade'		
Step 1	['a', 'd', 'e']	['a', 'd', 'e']	['a', 'd', 'e']
Step 2	-	['ad', 'e']	['a', 'de']
Step 3	-	['ade']	-

Table 7: Example illustrating how combining merge rules from two BPE tokenizers results in conflicts when tokenizing "ade".

we implement an ANN index, $\mathcal{I}$, allowing us to use $\mathbf{h}$ to efficiently retrieve the k closest tokens, denoted as $\mathcal{I}_k(\mathbf{h})$. This significantly reduces computational overhead by focusing on the "closest" tokens during generation, therefore maintaining the LM's parameter count and avoiding excessive scaling of the embedding matrix — usually seen in multilingual models. This facilitates using large vocabularies without retraining the embedding layer. Figure 6 illustrates the flow for applying dynamic tokenization to decoders with the ANN.

Experimental Setup. In our experiments, we decide to set $\mathcal{V}_{\text{large}}$ to one million entries which we denote $\mathcal{V}_{1\text{M}}$. To obtain this vocabulary $\mathcal{V}_{1\text{M}}$, we train a BPE tokenizer on the clean English subset of the MADLAD-400 corpus (Kudugunta et al., 2023). Being around twice as large as the Oxford English Dictionary,[8] we expect this vocabulary to contain most English words along other common fragments of text, allowing us to decrease the granularity of the tokens close to word-level on average. This vocabulary is significantly larger than those used in previous works on vocabulary expansion and ≈ 32 times larger than $\mathcal{V}_{\text{init}}$. Following the expansion of the vocabulary, we construct a ScaNN (Guo et al., 2020) index to enable approximate nearest neighbour (ANN) search over HN embeddings. We chose ScaNN due to its good performance on ann-benchmarks.[9] We use this index to retrieve the top 10 closest token embeddings, following the process

[8] oed.com
[9] ann-benchmarks.com

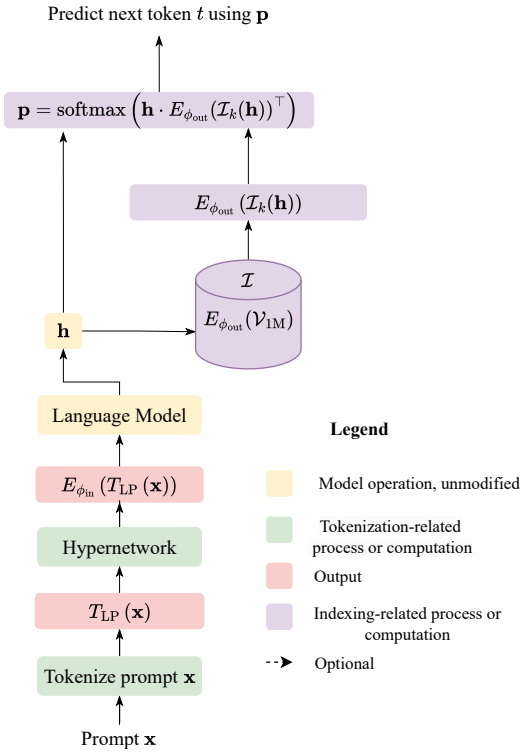


Figure 6: Dynamic tokenization with expanded vocabulary $\mathcal{V}_{\text{large}}$ and ANN index $\mathcal{I}$ applied to decoder LMs.

illustrated in Figure 6.

Table 8 includes the hyperparameters used for ScaNN index training and inference.

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 8: Configuration details for the ScaNN index.

Importantly, when evaluating our approach on MT-Bench, we use both conversation turns rather than just the first turn, as was done in prefilling (see Section 5.2). This adjustment is made because the current approach with $\mathcal{V}_{\text{1M}}$ focuses on achieving only (closer to) word-level tokenization, without varying the percentage of sequence length reductions. As a result, determining the number of merges is unnecessary and is in fact difficult to approximate when using two turns, since the second turn contains the model's response to the first turn within its prompt.

Further findings. When MISTRAL-7B is evaluated using LP tokenization and HN embeddings with $\mathcal{V}_{\text{1M}}$ on MT-Bench, we note a decrease in scores of 0.24 — compared with the same setting but with $\mathcal{V}_{\text{init}}$.

Similar to the previous experimental results, the coarser granularity decreases performance, likely due to the quality of the generated HN embeddings. However, the gap between HN and original embeddings is more significant here than in encoders or when applying dynamic tokenization for scoring or prefilling, which could potentially be minimized through n-shot tokenizer transfer, as demonstrated by Minixhofer et al. (2024). Additionally, we observe a general trend where performance declines with the use of HN embeddings, worsens further with LP tokenization, and decreases even more with $\mathcal{V}_{\text{large}}$, although with the benefit of reduced token sequence length.

Token repetition penalty improves generations quality. Qualitatively inspecting the MT-Bench generated answers, we observed a token repetition issue in settings with HN embeddings, 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, across all settings using HN embeddings. The token repetition issue may also stem from the MISTRAL-7B model itself, as multiple reports highlighted similar problems occurring during generation.[10]

ANN vs Exhaustive Search. Contrary to our expectations, the results indicate that using an ANN index outperforms exhaustive search (setting 9 and 10). 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 a level of noise or variability, which acts like a regularization technique.

D 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

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

	Tokenization	Embeddings	Vocab. Size	$\Delta_{\text{Length. (\%)}}$	Accuracy (%)
(1)	original	original	32k	0	**61.8**
(2)	original	HN	32k	0	58.8
(3)	LP	HN	32k	-1	57.8
(4)	LP	HN	1M	-13.6	55.9

Table 9: Performance of MISTRAL-7B on the MMLU English task under different settings. $\Delta_{\text{Length. (\%)}}$ represents the average decrease in token sequence length for the prompt over the original tokenization. Evaluation was performed under a 5-shot setting with each shot chosen from same domain as the question prompt.

	Tokenization	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	0.05	✓	7.46
(3)	Original	HN	32k	exhaustive	-	-	✗	6.84
(4)	Original	HN	32k	exhaustive	1.1	0.1	✓	7.10
(5)	LP	HN	32k	exhaustive	-	-	✗	6.50
(6)	LP	HN	32k	exhaustive	1.1	0.05	✓	**6.92**
(7)	LP	HN	1M	ScaNN index	-	-	✗	6.26
(8)	LP	HN	1M	ScaNN index	1.1	0.05	✓	**6.64**
(9)	LP	HN	1M	exhaustive	-	-	✗	5.24
(10)	LP	HN	1M	exhaustive	1.1	0.05	✓	**6.53**

Table 10: Performance of MISTRAL-7B-INSTRUCT on MT-Bench English under different settings.

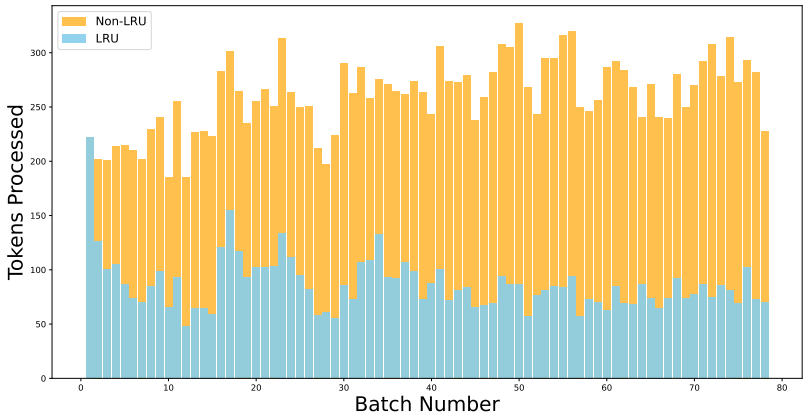


Figure 7: Tokens processed by the hypernetwork using an HN-specific LRU cache versus processing all unique tokens without caching. Results obtained on the `validation` subset of XNLI English.

regenerate them for each batch.

E Throughput analysis results

Table 4 shows the FLOPs per sample for both the main model and hypernetwork with different sequence reduction percentages on multilingual MMLU. We report results for English, French, German, Spanish, and Portuguese. As shown, the model's FLOPs decrease almost linearly with sequence length, while the hypernetwork overhead remains negligible — typically under 3.1% of total FLOPs. Additionally, the overhead from the dynamic tokenization algorithm is minimal and can be offloaded alongside other data loading logic.

Lng	FLOPs	Sequence Reduction / FLOPs										
		0%	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
en	Model	10.1T	10.0T	9.9T	9.7T	9.5T	9.4T	9.2T	9.1T	8.8T	8.7T	8.5T
	Hypernet	169.3B	171.3B	175.0B	180.2B	184.7B	191.0B	196.8B	198.7B	201.5B	198.0B	199.8B
	HN FLOPs / total	1.7%	1.7%	1.7%	1.8%	1.9%	2.0%	2.0%	2.1%	2.2%	2.2%	2.2%
	Seq. Length	682.2	672.8	667.6	655.5	640.6	631.7	619.4	614.4	598.1	586.7	578.4
de	Model	15.0T	14.3T	13.8T	13.2T	12.4T	11.8T	11.1T	10.6T	9.7T	9.0T	8.4T
	Hypernet	82.9B	94.6B	107.8B	128.9B	149.3B	171.1B	194.7B	212.2B	235.5B	251.3B	261.4B
	HN FLOPs / total	0.5%	0.7%	0.8%	1.0%	1.2%	1.4%	1.7%	1.9%	2.2%	2.3%	3.0%
	Seq. Length	1015.0	963.3	937.7	882.0	825.9	782.9	748.2	711.7	668.1	608.2	571.0
es	Model	14.2T	13.6T	13.3T	12.8T	12.2T	11.7T	11.2T	10.7T	10.1T	9.5T	9.0T
	Hypernet	85.2B	92.0B	102.1B	120.4B	140.5B	157.7B	179.1B	200.5B	222.9B	230.6B	238.1B
	HN FLOPs / total	0.6%	0.7%	0.8%	1.0%	1.1%	1.3%	1.6%	1.8%	2.1%	2.4%	2.6%
	Seq. Length	956.3	928.1	893.0	851.0	809.2	779.0	743.7	716.6	677.1	641.1	612.5
fr	Model	14.4T	14.0T	13.7T	13.2T	12.6T	12.2T	11.7T	11.3T	10.7T	10.2T	9.7T
	Hypernet	91.8B	99.4B	109.6B	128.4B	146.6B	163.5B	183.1B	200.1B	223.2B	230.3B	238.5B
	HN FLOPs / total	0.6%	0.7%	0.8%	1.0%	1.2%	1.3%	1.5%	1.7%	2.0%	2.2%	2.4%
	Seq. Length	956.7	927.9	915.3	870.7	830.9	804.6	772.6	745.9	709.1	677.8	651.9
pt	Model	14.2T	13.6T	13.2T	12.7T	12.0T	11.5T	10.9T	10.5T	9.8T	9.1T	8.6T
	Hypernet	84.0B	89.2B	101.0B	118.5B	131.7B	154.0B	175.7B	196.7B	218.6B	229.5B	237.9B
	HN FLOPs / total	0.6%	0.7%	0.8%	1.0%	1.1%	1.3%	1.6%	1.8%	2.1%	2.3%	2.7%
	Seq. Length	952.3	929.7	888.2	850.4	814.5	766.0	728.9	698.3	655.0	615.3	584.8

Table 11: *FLOPs per sample* estimates for the model and hypernetwork when applying dynamic tokenization with different sequence reductions on multilingual MMLU. The *HN FLOPs / total* row shows the fraction of hypernetwork FLOPs out of total FLOPs. *Seq. Length* represents the average number of tokens per sample.