Sampling from Your Language Model One Byte at a Time

Jonathan Hayase[♡] **Alisa Liu**[♡] **Noah A. Smith**[♡♣] **Sewoong Oh**[♡]
[♡]University of Washington [♣]Allen Institute for AI
jhayase@cs.washington.edu

Abstract

Tokenization is used almost universally by modern language models, enabling efficient text representation using multi-byte or multi-character tokens. However, prior work has shown that tokenization can introduce distortion into the model's generations, an issue known as the *Prompt Boundary Problem (PBP)*. For example, users are often advised not to end their prompts with a space because it prevents the model from including the space as part of the next token. While this heuristic is effective in English, the underlying PBP continues to affect languages such as Chinese as well as code generation, where tokens often do not line up with word and syntactic boundaries. In this work, we present an inference-time method to convert any autoregressive LM with a BPE tokenizer into a character-level or byte-level LM. Our method efficiently solves the PBP and is also able to unify the vocabularies of language models with different tokenizers, allowing one to ensemble LMs with different tokenizers at inference time or transfer the post-training from one model to another using proxy-tuning. We demonstrate in experiments that the ensemble and proxy-tuned models outperform their constituents on downstream evals. [1]

1 Introduction

Tokenization is a crucial component of nearly all modern language models: it allows them to consume and produce arbitrary streams of text using only finite vocabularies. The vast majority of tokenizers in use today, such as those based on Byte-Pair Encoding (BPE) [58] or Unigram [27], feature tokens spanning multiple bytes or characters, allowing them to represent text more efficiently than purely byte-level or character-level tokenization [12, 78, 74].

Users of LMs are generally unaware of the tokenization and expect LMs to operate on strings, consuming a prompt as a string and producing a useful string completion thereof. Tokenized LMs approximate this by *(i)* encoding the text as a sequence of tokens, *(ii)* feeding the resulting sequence to the language model, and *(iii)* decoding the generated token sequence back into text. More precisely, let $\mathrm{prompt} \in \Sigma^*$ be a string of arbitrary length over some alphabet Σ, and let $\mathrm{encode}\colon \Sigma^* \to V^*$ and $\mathrm{decode}\colon V^* \to \Sigma^*$ represent the translation between strings and token sequences over a vocabulary V. To complete the prompt, a typical scheme is to sample from the distribution,

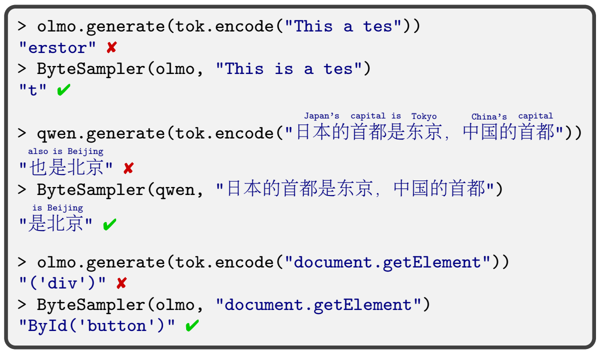
$$\mathsf{P}(t_1, \ldots, t_n \mid [t_1, \ldots, t_k] = \mathrm{encode}(\mathrm{prompt}))\ , \tag{1}$$

where $\mathrm{encode}(\mathrm{prompt})$ is the tokenization of the prompt, which in this example has length k. Note that sampling from this distribution can be done very conveniently by following the three steps above when the model has an autoregressive structure, i.e.,

$$\mathsf{P}(t_{k+1}, \ldots, t_n \mid t_1, \ldots, t_k) \ = \ \prod_{i=k+1}^{n} \mathsf{P}(t_i \mid t_1, \ldots, t_{i-1})\ ,$$

[1]Code is available at `https://github.com/SewoongLab/byte-sampler`.

which is used to sample the completion from $\mathsf{P}(t_{k+1}, \ldots, t_n \mid t_1, \ldots, t_k)$ given the tokenized prompt $[t_1, \ldots, t_k]$. We then return $\mathrm{decode}(t_1, \ldots, t_n)$ to the user. For the most part, this process happens transparently to the user, but under certain circumstances it can introduce distortion to the language model's completions.



```
> olmo.generate(tok.encode("This a tes"))
"erstor" ✗
> ByteSampler(olmo, "This is a tes")
"t" ✓

                               Japan's  capital is  Tokyo      China's  capital
> qwen.generate(tok.encode("日本的首都是东京，中国的首都"))
  also is Beijing
"也是北京" ✗
> ByteSampler(qwen, "日本的首都是东京，中国的首都")
  is Beijing
"是北京" ✓

> olmo.generate(tok.encode("document.getElement"))
"('div')" ✗
> ByteSampler(olmo, "document.getElement")
"ById('button')" ✓
```

Figure 1: `ByteSampler` resolves the prompt boundary problem (exhibited in the output of `generate()`). In this example, `␣test`, 都是, and `.getElementById` are all single tokens in the respective tokenizers.

The Prompt Boundary Problem (PBP). In particular, Eq. (1) introduces distortion whenever the prompt ends on a prefix of what could otherwise be a single token. More concretely, consider LLAMA-3.2-1B and suppose the user's prompt ends with the text `‘‘becau’’` (`["bec" = 17106, "au" = 2933]` as tokens): The user most likely expects the continuation to begin with `‘‘se’’` (325) since "because" is a common word. However during training, the model has only ever seen the word "because" represented as a single token (`11458`) and never as the sequence `[17106, 2933, 325]`. Accordingly, the actual next token LLAMA-3.2-1B predicts is `‘‘z’’` (89) which, while plausible in some scenarios, is an arguably unlikely continuation representing an artifact of tokenization. While this example may seem contrived at first glance, there are many situations where this problem may arise (Fig. 1 shows a few more examples):

1. In languages that do not separate words with whitespace, such as Chinese and Japanese, tokens can span multiple words, so this issue can arise even when the prompt ends with a complete word.

2. Any tokenizer that features multi-word tokens, which can bring gains in encoding efficiency [18, 29, 34], suffer from the same problem as Chinese and Japanese.

3. When completing code, it is common to request completions while in the middle of an identifier [23].

4. This issue also occurs when performing constrained generation from language models [55].

In general, the user, unaware of the tokenization, expects samples from the properly conditioned distribution,

$$\mathsf{P}(t_1, \ldots, t_n \mid \text{prompt} \sqsubseteq \mathrm{decode}(t_1, \ldots, t_n)) \, , \tag{2}$$

where $\sqsubseteq$ denotes the prefix relation. However, the token-prefix conditioned distribution of Eq. (1) and the *byte-prefix conditioned distribution* of Eq. (2) can differ substantially (e.g., Figure 1). Eq. (2) transcends the arbitrary token boundary set where the user provided prompt stops, decoupling the prompt boundary from token boundaries, to complete the prompt with the *exact* distribution from the language model. This leads to a fundamental algorithmic question of interest: how do we sample from the byte-prefix conditioned distribution of Eq. (2) exactly and efficiently?

Contributions. We introduce an efficient procedure to condition a BPE tokenizer-based model on an arbitrary *byte-prefix* given only access to the tokenizer and log-probability queries to the model (Section 3). We demonstrate in experiments that this represents an exact solution to the Prompt Boundary Problem presented above (Section 4.2). We show that our method can be used to convert the model into a byte-level language model and that this ability can be used to unify the vocabularies of different models. This enables exact byte-level ensembles of language models with different tokenizers (Section 4.3) and allows one to transfer the post-training of one model onto another model at inference time using proxy-tuning [33] (Section 4.4). We demonstrate in proof-of-concept experiments that language model ensembles and proxy-tuned models constructed with our method are able to outperform their constituent models in downstream evaluations.

2 Background

In this section we give essential background regarding tokenization as well a prior work addressing the Prompt Boundary Problem. We discuss additional related works in Appendix A.

	Exact	Preprocessing	Token evaluations	TE w/ Prefix Caching
Backtracking [55, 13, 2]	No	$O(1)$	$O(1)$	N/A
Prefix Covering [72]	Yes	$2^{O(n)}$	$2^{O(n)}$	$2^{O(n)}$
Back Tokenization [70]	Yes	$2^{O(n)}$	$O(n)$	$O(1)$ (optimal)
Byte-Pair Correction [51]	Yes	$O(n)$	$O(n)$	$O(1)$
ByteSampler (ours)	Yes	$O(1)$	N/A	$O(1)$ (optimal)

Table 1: **Incremental complexity** of various mitigations for the prompt boundary problem: we list the complexity (in both preprocessing time and LM evaluations) when sampling each new character while generating an n character string. Our method has the same complexity as backtracking methods while remaining exact, i.e., matching Eq. (2) in distribution, modulo invalid sequences (see below for discussion). We report both the original LM inference complexity as originally presented, as well as upper bounds using analysis from Section 3.1 when using prefix caching. “(optimal)” indicates that the token evaluations for any input will be the minimum required for exactness.

Byte Pair Encoding. BPE was originally presented as a form of data compression in Gage [16] and was proposed for use in NLP in Sennrich et al. [58]. To tokenize a piece of text with a typical BPE-based tokenizer, the text is first split into chunks, a process called *pretokenization*. These chunks, or *pretokens*, are then tokenized separately using BPE (thus no token may cross the boundary between pretokens). The BPE tokenizer processes each pretoken by first converting the text into a sequence of elements of the tokenizer’s base vocabulary (common choices for base vocabulary are individual characters or bytes under UTF-8 encoding). Next, an ordered list of merges is applied to the sequence to form larger tokens. Each merge specifies a contiguous pair of tokens (which may include products of previous merges), and a new token that represents their concatenation. The merges are applied left-to-right and once all valid merges are applied, the tokenization is complete. We show an example application of these steps in Table 2.

Step	Description	Result
0	Original text	“llama and deepseek.”
1	Pretokenization	`llama`|`␣and`|`␣deepseek`|`.`
2	Convert to base vocabulary	`l,l,a,m,a`|`␣,a,n,d`|`␣,d,e,e,p,s,e,e,k`|`.`
3	Apply merges from merge list	`ll` `ama` `␣and` `␣deep` `seek` `.`

Table 2: Step-by-step execution of an example BPE tokenizer.

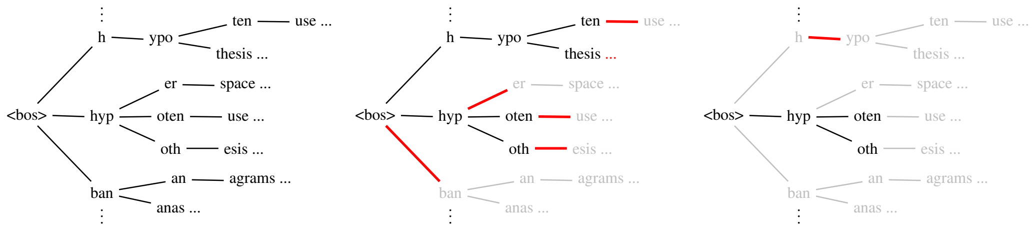
Prompt Boundary Problem. Issues surrounding tokenization have been extensively documented in prior work. The prompt boundary problem was presented for maximum prefix encoding in Phan et al. [51] and for BPE tokenizers in Vieira et al. [72] and Ribeiro [55]. Many methods have been proposed to address the prompt boundary issue. One line of heuristic techniques, including token healing [55] and its generalizations [13, 2] perform “backtracking” by *(i)* removing one or more of the most recent tokens, followed by *(ii)* sampling a continuation of the partial prompt using the language model, constraining the newly generated tokens to match the remaining text.

Exact methods, which preserve the sampling distribution of the original language model as shown in (5), have also been proposed. Vieira et al. [72] gave an exact method which requires exponential time as well as an approximate solution leveraging beam search. Turaga [70] proposed a method that combines backtracking with the exponential time method of Vieira et al. [72], adding a “back tokenization” step that significantly reduces the number of necessary calls to the language model, but still requires exponential preprocessing. Additionally, Phan et al. [51] proposed an exact method which requires only linear time.

Although all of the above methods, except for Backtracking, are "exact," they may produce slightly different sampling distributions. This is because the methods differ in their handling of invalid token sequences, which are sequences that can never be output by the tokenizer, but can still be generated erroneously by the model. For now, we will assume that the model will always produce valid token sequences, in which case all of the exact methods are identical. We discuss this assumption and the differences when it does not hold in more detail in Appendix D.

3 Method

In this section, we present some simple building blocks and use them to construct a procedure for sampling from a tokenizer-based language model one byte at a time. The fundamental structure of the algorithm is based on what we call the Valid Covering Tree, which is the tree of all possible valid token sequences that share a specific byte prefix and do not extend past the end of the prefix by more than one full token. We show the construction of the Valid Covering Tree in Fig. 2.



(a) Initial (infinite) tree (b) Prune by prefix (c) Prune invalid pairs

Figure 2: *Construction of the Valid Covering Tree* for string prefix "hypot": (a) starting with the infinite tree of all possible token sequences (many edges not shown), we prune branches that (b) do not match the given prefix or begin after the prefix ends or (c) contain invalid contiguous pairs of tokens. More example trees are shown in Appendix E.

The tree depicted in Fig. 2b corresponds to the *cover* described in Vieira et al. [72], who remark that it will generally have exponential size in the length of the prefix. In contrast, the Valid Covering Tree, which is a subtree of the one in Fig. 2b, has several properties which will prove useful:

1. **Correctness:** The tree represents exactly the set of valid sequences of tokens with the prompt as a prefix. (See Section 3.1 and Appendix D.)
2. **Compactness:** The tree is composed of a "trunk" of tokens that are fully determined (starting at the root, every node has only one child) plus a finite number of "branching" nodes at the end of the trunk. (The number is bounded by a constant which depends only on the tokenizer, see Section 3.2.)
3. **Convenience:** The tree can be updated to reflect the addition of a new byte using only constant time and space. (See Algorithm 1.)

Additional implementation details and optimizations are presented in Appendix C.

3.1 Pairwise Validation

Recall that a token sequence is *valid* if it is the encoding of some string under the BPE encoder.[2] The correctness of the pairwise pruning depends on the following proposition regarding validity under BPE tokenization.

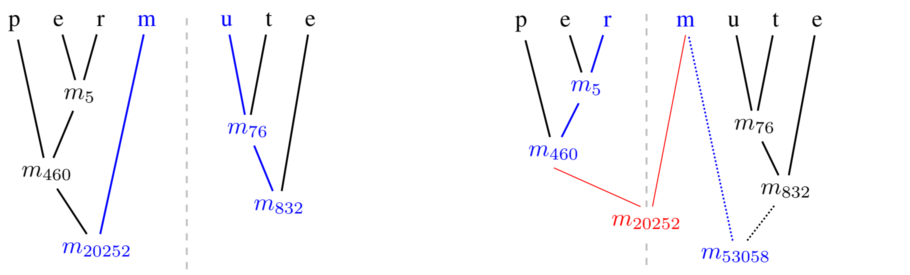
Proposition 3.1. *Let* $(\text{encode}, \text{decode})$ *denote a BPE encoder and decoder pair corresponding to some merge list* M *and vocabulary* V*. We call a token sequence* $T = [t_1, t_2, \ldots, t_n] \in V^n$ *valid if* $\text{encode}(\text{decode}(T)) = T$*. Then* T *is valid if and only if* $[t_i, t_{i+1}]$ *is valid for all* $i \in \{0, \ldots, n-1\}$*.*

To see that this proposition is true, consider two valid token sequences $T_1 = \text{encode}(S_1)$ and $T_2 = \text{encode}(S_2)$. If, while tokenizing the concatenation $S_1 + S_2$, there is no merge applied that

[2]The notion of pairwise validation of token sequences was first used in van Antwerpen and Neubeck [71] as the basis for a streaming algorithm and a fast backtracking-based algorithm for BPE tokenization (without addressing the PBP).

crosses the boundary between S_1 and S_2 then the two strings will "evolve" independently, and we will have $\text{encode}(S_1 + S_2) = T_1 + T_2$ which means $T_1 + T_2$ is valid.

Conversely, if a merge is applied that does cross the boundary, then the final encoding must feature a token crossing the boundary (since no merge can be undone), which means $T_1 + T_2$ cannot be valid since it has no such token. We depict an example of both cases using OpenAI's cl100k tokenizer [48] in Fig. 3.[3]



(a) Valid pair: no merge crossing boundary (b) Invalid pair: merge m_{20252} crosses boundary

Figure 3: Example of valid and invalid token pairs. We show the initial string's bytes and the merges $m_t \in M$ that are applied to the string (in order of t) to tokenize the string. In the invalid case, merge m_{53058} cannot occur because a conflicting merge m_{20252} was applied earlier. The key observation is that we only need to consider the trajectory at the boundary (in blue) to decide if the pair is valid.

This implies a fast method to test whether a pair of tokens is valid: we inspect the merge trajectory along the boundary between the tokens and check if any conflicting merges would be applied. The worst case merge tree depth is fixed by the tokenizer, so this check can be done in constant time.[4]

3.2 Streaming Tokenization

Given a stream of input bytes, we will use Algorithm 1 to update "branches" of the Valid Covering Tree, while writing the fully determined "trunk" of tokens to an output stream.

Algorithm 1: Streaming BPE tokenization maintaining a tree matching Fig. 2c

```
Input: Branching tree $T$, new byte $b$
Output: stream of fully determined tokens
for every node $N$ that ends one byte before $b$ do
    add all valid next tokens as children of $N$;          // See Fig. 2c
end
Prune branches that do not match $b$;                      // See Fig. 2b
while the root of $T$ has only one child do
    Add the root token to the output stream and make its only child the new root;
end
```

Now we will show that this approach is computationally efficient. To bound the worse-case behavior, we use the observation of Berglund and van der Merwe [3] that each output token can be fully determined using only a constant amount of lookahead (in bytes), where the constant depends only on the tokenizer. This implies that the branching tree T will have bounded depth, since any token that is fully determined will be removed from the tree and written to the output stream. The branching factor of the tree is also bounded by a constant depending on the tokenizer. Thus, the number of edges of T is bounded by a constant, which also means the pruning described in Fig. 2 can be carried out in constant time. For more concrete performance numbers see Section 4.1, where we show that the tree has only 0.72 extra non-leaf nodes on average.

[3]It's worth noting that the analogs of Proposition 3.1 do *not* hold for either Unigram [27] or Wordpiece [57] tokenizers.

[4]We generally expect the depth of the merge trees to scale with the logarithm of the vocabulary size V, although we ignore scaling with respect to the tokenizer's parameters for brevity.

3.3 Language Modeling Using Valid Covering Trees

Now that we can easily compute Valid Covering Trees, we can use them to perform various common language modeling operations.

To **compute the probability of a prefix** under the LM, we sum the cumulative probabilities the LM assigns to the sequences represented by all leaves of the tree.

To **sample a continuation of a prefix**, we compute the probability (as above) of every leaf and sample one of them accordingly. We are then free to continue sampling a continuation from that leaf using normal token-level sampling. This can be used to solve the PBP without paying the cost of sampling one byte at a time.

To **compute the next byte distribution given a prefix**, we group the leaves by the next byte they would entail and sum the probabilities (as above) of the leaves in each group. This can be combined with a sampling rule to generate text one byte at a time. Naturally, this will generate text more slowly than sampling at the token level. We quantify this overhead in Section 4.2.

We use "ByteSampler" to refer to this collection of capabilities for convenience.

4 Experiments

In our experiments, we apply ByteSampler at inference time to off-the-shelf language models. In Section 4.1 we show that our method has less computational overhead compared to other exact methods. Next, in Section 4.2, we show that exact methods perform better than heuristics in character-level language modeling. Finally, we present several applications of our method to enable higher-level functions such as ensembling (Section 4.3) and proxy-tuning (Section 4.4) models with mismatched tokenizers.

4.1 Efficiency

As discussed in Section 2, there are several existing methods which are also "exact." Although each technically corresponds to a different sampling distribution, we do not expect there to be any significant differences between them in practice. Therefore, the main distinguishing factor to consider is the method's computational cost. To estimate the cost in a realistic setting, we sample a random 100 character substring from the OLMo2 pretraining corpus [47] and estimate how many inference tokens (according to the OLMo2 tokenizer) each method requires to calculate the probability of the substring as a text prefix. Note that the substring is sampled uniformly, so it is about 80% likely to end in the middle of a word. We report the average inference cost in tokens, averaged over 10,000 samples, for several methods in Table 3.

Method	Inference Tokens	Overhead vs. BPE
No mitigation (plain BPE)	23.51	0
Prefix Covering [72]	2.12×10^{30}	$+2.12 \times 10^{30}$
Byte-Pair Correction [51]	72.99	+49.47
Byte-Pair Correction with prefix caching	25.61	+2.09
ByteSampler (ours)[5]	**24.24**	**+0.72**

Table 3: **Inference cost of various exact solutions to the prompt boundary problem.** Our method has 65% less overhead than the next best method. Overhead vs. BPE measures the average additional tokens of inference required by the method, compared to plain BPE. Importantly, the overhead is paid for each byte when sampling at the byte level, making low overhead crucial for efficient sampling.

4.2 Character-Level Language Modeling

[5]We believe Back Tokenization [70] should match our method when it comes to required inference tokens. However, its worst-case exponential *preprocessing* time limits its practicality.

[6]For token level prediction, calculated using a conversion rate of 4.518 characters per token.

Prediction unit	Method	Loss per unit	Bits per character[6]
Token	Plain BPE	2.67	0.80
Character	No mitigation (plain BPE)	4.81	6.53
Character	ByteSampler (ours)	**0.60**	**0.81**

Table 4: **Language modeling loss of OLMo2-1B on English text using various methods**. We compare three settings: *(i)* the original token-level cross-entropy loss when predicting the next token; *(ii)* the character-level loss when predicting the next character by directly tokenizing the prompt and calculating the next character distribution; and *(iii)* the character-level loss obtained using ByteSampler to predict the next character. The higher loss per unit for token-level prediction is to be expected, as tokens are harder to predict than bytes. Once the loss is normalized to bits per character, our method and the original model achieve similar results, which demonstrates that our method does not degrade language modeling quality.

In this section, we will focus on converting off-the-shelf language models into character-level language models.[7] We then evaluate the character-level prediction performance using the standard cross-entropy loss as well as next-character prediction accuracy in two languages: English in Section 4.2.1 and Chinese in Section 4.2.2.

4.2.1 OLMo2 for English Text

In this setting, we sample a document randomly from the OLMo2 pretraining corpus [47] and choose a random prefix of the document of length at most 1000 characters. We then compute the next-character distribution according to OLMo2-1B [65] using various methods. To allow comparison with the original token-based model, we also truncate the prefix to the nearest token boundary and perform next-token prediction with the original model. We can compare the character-level and token-level losses via bits per character [41], which normalizes the loss to account for the fact that tokens are more difficult to predict due to their greater information content. We report the average loss of the predictions over 100,000 such documents in Table 4.

From the results in Table 4, we can clearly see the effect of the prompt boundary problem: naively predicting the next character by directly applying the tokenizer to an arbitrary string prefix as in Eq. (1) leads to poor performance ("no mitigation" in Table 4). In contrast, ByteSampler nearly matches the performance of the original token-based model ("plain BPE") in bits per character, as expected for exact methods.

For backtracking methods, it is not easy to compute the probability of any particular next character. This prevents us from calculating the cross-entropy loss as in Table 4. For our experiments, we compare to the Token Alignment method of Athiwaratkun et al. [2], which is the most advanced of the proposed backtracking methods and also includes token healing as a special case. We use it to directly predict the next character by sampling greedily and report the average accuracy over 100,000 samples in Table 5.

Interestingly, we find that too much backtracking hurts the performance of the Token Alignment method. We believe this is because the sampling step often segments the remainder of the prompt in a non-standard way, which may harm the performance of the model.

4.2.2 Qwen3 for Chinese Text

Since Chinese writing does not use whitespace, ending the prompt with a complete word does not generally provide a reliable token boundary. This makes it more difficult to heuristically avoid the PBP. Similar to Section 4.2.1, we sample a random prefix of length at most 500 characters of a random document from the Chinese subset of the MADLAD-400 dataset [28]. We then compute the distribution of next characters according to Qwen3-1.7B-Base [66] using various methods and report the average cross-entropy loss over 100,000 documents in Table 6.

[7]We choose character-level modeling for this section, even though our method supports byte-level predictions, because some related methods can only operate on character strings.

[8]For token level prediction, calculated using a conversion rate of 1.415 characters per token.

Method	Next character accuracy	Overhead vs. BPE
No mitigation (plain BPE)	29.490	0
1 Token Backtracking (Token Healing)	71.634	**+0.43**
2 Token Backtracking (Token Alignment)	76.281	+0.53
4 Token Backtracking (Token Alignment)	75.407	+1.08
ByteSampler (ours)	**81.560**	+1.72

Table 5: **Next character prediction accuracy of OLMo2-1B on English text using various methods**. We compare three settings *(i)* directly tokenizing the prompt and greedily sampling until the first character of the completion is determined; *(ii)* using backtracking with Token Alignment (of which Token Healing is a special case) to predict the next character; and *(iii)* using ByteSampler to predict the next character. Overhead vs. BPE measures the average additional tokens of inference required by the method, compared to *(i)*.

Prediction unit	Method	Loss per unit	Bits per Character[8]
Token	Plain BPE	3.43	3.29
Character	No mitigation (plain BPE)	3.79	5.16
Character	ByteSampler (ours)	**2.38**	**3.23**

Table 6: **Language modeling loss of Qwen3-1.7B-Base on Chinese text using various methods**. We use the same settings and metrics as Table 4. Similarly to our English results, ByteSampler achieves a similar normalized language modeling loss (in bits per character) to the original model which can only perform next token prediction.

Once again, the naive method fails while our method achieves similar normalized loss to the original token-level model. We also report next character prediction accuracy to allow comparison with backtracking methods. Note that Chinese has much higher entropy at the character level so the average accuracies are proportionally lower.

4.3 Byte-Level Ensemble

Another application enabled by byte-level sampling is the ensembling of language models with different tokenizers. In general, when vocabularies between LMs are the same, their next-token probability or logit distribution can be combined via arithmetic into a single distribution, but this cannot be done directly when the vocabularies differ. Several works have proposed methods to combine LM predictions despite mismatching vocabularies [25, 38, 35, 75], but these may introduce bias into the sampling distribution. Our method makes the direct ensemble possible by converting models with BPE tokenizers into a byte-wise models, thus unifying their vocabularies.

In our experiment, we consider an ensemble of three small base language models: Qwen3-1.7B [66], OLMo2-1B [47, 65], and Llama3.2-1B [64]. We combine the predictions by computing the average $\boldsymbol{p}_{\text{ensemble}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{p}_i$ where $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ are the next-byte probability distributions for each model. We evaluate the models on a suite of seven tasks and report the results in Table 8.

4.4 Byte-Level Proxy-Tuning

In addition to additive ensembles over probabilities, the *logit*-level predictions of multiple LMs can be combined via arithmetic, with individual LMs acting as "experts" (if their predictions are combined additively) or "anti-experts" (if subtractively) [32, 31, 60, 20, 11, 59]. In particular, this form of ensembling can be used to achieve the *effect* of tuning a large pretrained LM without accessing model weights. To see how this can be done, note that clearly for logit vectors

$$\boldsymbol{\ell}_{\text{tuned}} = \boldsymbol{\ell}_{\text{base}} + (\boldsymbol{\ell}_{\text{tuned}} - \boldsymbol{\ell}_{\text{base}}).$$

[9]Chinese typically uses three bytes for each character when encoded using UTF-8.

Method	Next character accuracy	Overhead vs. BPE
No mitigation (plain BPE)	32.8	0
1 Token Backtracking (Token Healing)	49.2	+1.82
2 Token Backtracking (Token Alignment)	49.6	+2.98
4 Token Backtracking (Token Alignment)	49.0	+5.30
ByteSampler (ours)	**52.7**	**+1.60**

Table 7: **Next character prediction accuracy of QWEN3-1.7B-BASE on Chinese text using various methods**. We use the same settings and metrics as Table 5. Similar to our English language results, ByteSampler achieves the best prediction accuracy, but unlike in English, ByteSampler also requires the least overhead of all methods. This highlights that languages with multi-byte characters [9] can behave differently than ones which typically use a single byte for each character.

Task	QWEN3	OLMO2	LLAMA3.2	← Average	Ensemble
Arithmetic [5]	0.974	0.838	0.831	0.881	**0.978**
DROP [15]	0.470	0.409	0.299	0.393	**0.479**
Jeopardy [68]	0.274	0.327	0.264	0.288	**0.347**
LAMBADA [50]	0.727	0.628	0.510	0.622	**0.755**
SQuAD [54]	**0.845**	0.802	0.694	0.780	0.836
TriviaQA [24]	0.389	**0.535**	0.443	0.456	0.526
WikidataQA [4]	0.689	0.643	0.658	0.663	**0.719**

Table 8: **Byte-level ensemble results.** We report the performance (accuracy) of a byte-level ensemble of three models on downstream evals, along with the individual performance of each model. We see that the ensemble is competitive with the best individual model on each task and consistently outperforms the average performance across the three models. We give more details regarding the evaluation in Appendix B.2.

The idea of *proxy-tuning* [33] is to approximate the term $\boldsymbol{\ell}_{\text{tuned}} - \boldsymbol{\ell}_{\text{base}}$ using the difference between a pair of tuned and base proxy models $\boldsymbol{\ell}_{\text{expert}} - \boldsymbol{\ell}_{\text{anti-expert}}$. In our experiments, we proxy-tune a strong base model, LLAMA-3.1-8B, using OLMO2-1B-INSTRUCT and OLMO2-1B as the expert and anti-expert, respectively, which together represent a strong post-training recipe [47, 30].

Shown in Table 9, we find that the proxy-tuned LLAMA 3.1 [63] model consistently outperforms the base model alone as well as the small tuned expert. This highlights a practical application of ByteSampler to "apply" post-training to base models without actually training them, thus disentangling the quality of the base model from that of the post-training recipe.

Task	Metric	LLAMA3.1	OLMO2 INST.	LLAMA3.1 (Proxy Tuned)
AlpacaEval 2	LC winrate	0.88	**33.5**	**33.5**
GSM8K	5 ICE, CoT, EM	55.3	51.9	**76.6**
MMLU	0 ICE, CoT, MC	27.8	35.2	**59.5**

Table 9: **Proxy tuning results.** We report performance on downstream evaluations when proxy-tuning LLAMA3.1-8B using OLMO2-1B-INSTRUCT as the expert and OLMO2-1B as the anti-expert. We see that the proxy tuned model gains the instruction-following capability (AlpacaEval 2) and chain-of-thought capabilities (GSM8K, MMLU) of OLMO2-1B-INSTRUCT while also benefiting from its larger size, allowing it to surpass the expert's individual performance. For details regarding the evaluation, see Appendix B.3.

5 Conclusion

In this work, we introduced ByteSampler, an algorithm that eliminates the Prompt Boundary Problem by converting any BPE tokenizer-based language model into a byte-level model while preserving its generative distribution at the text-level. Interesting extensions of this method include automatic support for arbitrary pretokenizers (discussed in Appendix C.3), generalization to other tokenization schemes (such as Unigram [27], Wordpiece [57], and other variants of BPE [53, 10]), and speculative-decoding at the byte-level.

Beyond correcting sampling artifacts at the prompt-boundary—which is useful in its own right in many situations—the ability to unify vocabularies at inference time enables many forms of model composition, including ensembles of (and post-training transfer between) models with different tokenizers. Other applications of this technology include *(i)* byte-level knowledge distillation to transfer skills more effectively between models with different tokenizers, *(ii)* rapid post-training research leveraging the fact that a post-training recipe (represented by a pair of proxy-tuning experts) can be applied to any number of models without additional training, *(iii)* routing dynamically between models [81] during generation without requiring matching tokenizers, and potentially *(iv)* more convenient LM-powered compression of byte streams.

In general, whenever (mismatching) tokenizers represent an obstacle or inconvenience, our method has the potential to completely bypass it at the cost of (minimally) increased inference compute. We hope that this will prove useful to LM researchers and users alike.

Acknowledgments

We would like to thank Hao Xu and Ke Hayase for helping us brainstorm a good example of the PBP in Chinese. JH and AL are supported by the NSF Graduate Research Fellowship Program. This work was partially funded by NSF 2113530, 2112471, and 2229876, and Microsoft Grant for Customer Experience Innovation.

References

[1] O. Ahia, S. Kumar, H. Gonen, V. Hofmann, T. Limisiewicz, Y. Tsvetkov, and N. A. Smith. MAGNET: Improving the multilingual fairness of language models with adaptive gradient-based tokenization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL `https://openreview.net/forum?id=1e3MOwHSIX`.

[2] B. Athiwaratkun, S. Wang, M. Shang, Y. Tian, Z. Wang, S. K. Gonugondla, S. K. Gouda, R. Kwiatkowski, R. Nallapati, P. Bhatia, and B. Xiang. Token alignment via character matching for subword completion. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 15725–15738, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.929. URL `https://aclanthology.org/2024.findings-acl.929`.

[3] M. Berglund and B. van der Merwe. Formalizing bpe tokenization. In *13th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2023, 18-19 September, 2023, Famagusta, Cyprus*, pages 16–27. Open Publishing Association, 2023.

[4] BIG-bench. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL `https://openreview.net/forum?id=uyTL5Bvosj`.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pages 1877–1901, 2020.

[6] K. Cao and L. Rimell. You should evaluate your language model on marginal likelihood over tokenisations. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2104–2114, 2021.

[7] Y. Chen, K. Marchisio, R. Raileanu, D. Adelani, P. Stenetorp, S. Riedel, and M. Artetxe. Improving language plasticity via pretraining with active forgetting. In *Advances in Neural Information Processing Systems*. NeurIPS, 2023.

[8] Z. Chen, J. Li, P. Chen, Z. Li, K. Sun, Y. Luo, Q. Mao, D. Yang, H. Sun, and P. S. Yu. Harnessing multiple large language models: A survey on llm ensemble. *arXiv preprint arXiv:2502.18036*, 2025.

[9] N. Chirkova, G. Kruszewski, J. Rozen, and M. Dymetman. Should you marginalize over possible tokenizations? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1–12, 2023.

[10] P. Chizhov, C. Arnett, E. Korotkova, and I. P. Yamshchikov. Bpe gets picky: Efficient vocabulary refinement during tokenizer training. *arXiv preprint arXiv:2409.04599*, 2024.

[11] Y.-S. Chuang, Y. Xie, H. Luo, Y. Kim, J. R. Glass, and P. He. Dola: Decoding by contrasting layers improves factuality in large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[12] J. H. Clark, D. Garrette, I. Turc, and J. Wieting. Canine: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 2022. doi: 10.1162/tacl_a_00448. URL `https://aclanthology.org/2022.tacl-1.5`.

[13] G. Dagan, G. Synnaeve, and B. Rozière. Getting the most out of your tokenizer for pre-training and domain adaptation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[14] K. Dobler and G. De Melo. 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, 2023.

[15] D. Dua, Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In J. Burstein, C. Doran, and T. Solorio, editors, *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 2368–2378, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1246. URL `https://aclanthology.org/N19-1246`.

[16] P. Gage. A new algorithm for data compression. *The C Users Journal archive*, 12:23–38, 1994. URL `https://api.semanticscholar.org/CorpusID:59804030`.

[17] L. Gee, A. Zugarini, L. Rigutini, P. Torroni, et al. 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. Association for Computational Linguistics (ACL), 2022.

[18] L. Gee, L. Rigutini, M. Ernandes, and A. Zugarini. Multi-word tokenization for sequence compression. In M. Wang and I. Zitouni, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 612–621, Singapore, Dec. 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-industry.58. URL `https://aclanthology.org/2023.emnlp-industry.58`.

[19] R. L. Geh, H. Zhang, K. Ahmed, B. Wang, and G. V. d. Broeck. Where is the signal in tokenization space? *arXiv preprint arXiv:2408.08541*, 2024.

[20] A. Gera, R. Friedman, O. Arviv, C. Gunasekara, B. Sznajder, N. Slonim, and E. Shnarch. The benefits of bad advice: Autocontrastive decoding across model layers. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10406–10420, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.580. URL `https://aclanthology.org/2023.acl-long.580`.

[21] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

[22] Y. Huang, X. Feng, B. Li, Y. Xiang, H. Wang, T. Liu, and B. Qin. Ensemble learning for heterogeneous large language models with deep parallel collaboration. *Advances in Neural Information Processing Systems*, 37:119838–119860, 2024.

[23] J. Jackson. Character prefix conditioning, 2025. URL `https://www.cursor.com/blog/cpc`.

[24] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In R. Barzilay and M.-Y. Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1147. URL `https://aclanthology.org/P17-1147`.

[25] J. Kasai, K. Sakaguchi, R. Le Bras, H. Peng, X. Lu, D. Radev, Y. Choi, and N. A. Smith. Twist decoding: Diverse generators guide each other. In Y. Goldberg, Z. Kozareva, and Y. Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4909–4923, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.326. URL `https://aclanthology.org/2022.emnlp-main.326`.

[26] T. Kudo. 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, 2018.

[27] T. Kudo and J. Richardson. 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, 2018.

[28] S. Kudugunta, I. Caswell, B. Zhang, X. Garcia, D. Xin, A. Kusupati, R. Stella, A. Bapna, and O. Firat. Madlad-400: A multilingual and document-level large audited dataset. *Advances in Neural Information Processing Systems*, 36:67284–67296, 2023.

[29] D. Kumar and A. Thawani. BPE beyond word boundary: How NOT to use multi word expressions in neural machine translation. In S. Tafreshi, J. Sedoc, A. Rogers, A. Drozd, A. Rumshisky, and A. Akula, editors, *Proceedings of the Third Workshop on Insights from Negative Results in NLP*, pages 172–179, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.insights-1.24. URL `https://aclanthology.org/2022.insights-1.24`.

[30] N. Lambert, J. Morrison, V. Pyatkin, S. Huang, H. Ivison, F. Brahman, L. J. V. Miranda, A. Liu, N. Dziri, S. Lyu, Y. Gu, S. Malik, V. Graf, J. D. Hwang, J. Yang, R. L. Bras, O. Tafjord, C. Wilhelm, L. Soldaini, N. A. Smith, Y. Wang, P. Dasigi, and H. Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL `https://arxiv.org/abs/2411.15124`.

[31] X. L. Li, A. Holtzman, D. Fried, P. Liang, J. Eisner, T. Hashimoto, L. Zettlemoyer, and M. Lewis. Contrastive decoding: Open-ended text generation as optimization. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12286–12312, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.687. URL `https://aclanthology.org/2023.acl-long.687`.

[32] A. Liu, M. Sap, X. Lu, S. Swayamdipta, C. Bhagavatula, N. A. Smith, and Y. Choi. DExperts: Decoding-time controlled text generation with experts and anti-experts. In C. Zong, F. Xia, W. Li, and R. Navigli, editors, *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 6691–6706, Online, Aug. 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.522. URL `https://aclanthology.org/2021.acl-long.522`.

[33] A. Liu, X. Han, Y. Wang, Y. Tsvetkov, Y. Choi, and N. A. Smith. Tuning language models by proxy. In *First Conference on Language Modeling*, 2024.

[34] A. Liu, J. Hayase, V. Hofmann, S. Oh, N. A. Smith, and Y. Choi. SuperBPE: Space travel for language models. *arXiv preprint arXiv:2503.13423*, 2025. URL `https://arxiv.org/abs/2503.13423`.

[35] C. Liu, X. Quan, Y. Pan, L. Lin, W. Wu, and X. Chen. Cool-fusion: Fuse large language models without training. *arXiv preprint arXiv:2407.19807*, 2024.

[36] Y. Liu, P. Lin, M. Wang, and H. Schütze. 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, 2024.

[37] S. Lundberg. The art of prompt design: Prompt boundaries and token healing, 2023. URL `https://medium.com/towards-data-science/the-art-of-prompt-design-prompt-boundaries-and-token-healing-3b2448b0be38`.

[38] B. Lv, C. Tang, Y. Zhang, X. Liu, Y. Yu, and P. Luo. Specfuse: Ensembling large language models via next-segment prediction. *arXiv preprint arXiv:2412.07380*, 2024.

[39] K. Marchisio, P. Lewis, Y. Chen, and M. Artetxe. Mini-model adaptation: Efficiently extending pretrained models to new languages via aligned shallow training. In *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

[40] C. Mavromatis, P. Karypis, and G. Karypis. Pack of llms: Model fusion at test-time via perplexity optimization. In *First Conference on Language Modeling*, 2024.

[41] S. J. Mielke. Can you compare perplexity across different segmentations?, Apr 2019. URL `https://sjmielke.com/comparing-perplexities.htm`.

[42] B. Minixhofer, F. Paischer, and N. Rekabsaz. 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, 2022.

[43] B. Minixhofer, E. Ponti, and I. Vulić. Zero-shot tokenizer transfer. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[44] B. Minixhofer, I. Vulić, and E. M. Ponti. Universal cross-tokenizer distillation via approximate likelihood matching. *arXiv preprint arXiv:2503.20083*, 2025.

[45] P. Nawrot, J. Chorowski, A. Lancucki, and E. M. Ponti. Efficient transformers with dynamic token pooling. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6403–6417, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.353. URL `https://aclanthology.org/2023.acl-long.353`.

[46] B.-D. Oh and W. Schuler. Leading whitespaces of language models' subword vocabulary pose a confound for calculating word probabilities. In Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3464–3472, Miami, Florida, USA, Nov. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.202. URL `https://aclanthology.org/2024.emnlp-main.202`.

[47] T. OLMo, P. Walsh, L. Soldaini, D. Groeneveld, K. Lo, S. Arora, A. Bhagia, Y. Gu, S. Huang, M. Jordan, N. Lambert, D. Schwenk, O. Tafjord, T. Anderson, D. Atkinson, F. Brahman, C. Clark, P. Dasigi, N. Dziri, M. Guerquin, H. Ivison, P. W. Koh, J. Liu, S. Malik, W. Merrill, L. J. V. Miranda, J. Morrison, T. Murray, C. Nam, V. Pyatkin, A. Rangapur, M. Schmitz, S. Skjonsberg, D. Wadden, C. Wilhelm, M. Wilson, L. Zettlemoyer, A. Farhadi, N. A. Smith, and H. Hajishirzi. 2 olmo 2 furious, 2024. URL `https://arxiv.org/abs/2501.00656`.

[48] OpenAI. Openai platform documentation, 2023. URL `https://platform.openai.com/docs`. Accessed: 2025/05/10.

[49] A. Pagnoni, R. Pasunuru, P. Rodriguez, J. Nguyen, B. Muller, M. Li, C. Zhou, L. Yu, J. Weston, L. Zettlemoyer, G. Ghosh, M. Lewis, A. Holtzman, and S. Iyer. Byte latent transformer: Patches scale better than tokens, 2024. URL `https://arxiv.org/abs/2412.09871`.

[50] D. Paperno, G. Kruszewski, A. Lazaridou, N. Q. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In K. Erk and N. A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, Berlin, Germany, Aug. 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144. URL `https://aclanthology.org/P16-1144`.

[51] B. Phan, M. Havasi, M. Muckley, and K. Ullrich. Understanding and mitigating tokenization bias in language models, 2024. URL `https://arxiv.org/abs/2406.16829`.

[52] T. Pimentel and C. Meister. How to compute the probability of a word. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18358–18375, 2024.

[53] I. Provilkov, D. Emelianenko, and E. Voita. Bpe-dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[54] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100,000+ questions for machine comprehension of text. In J. Su, K. Duh, and X. Carreras, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, Nov. 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL `https://aclanthology.org/D16-1264`.

[55] M. T. Ribeiro. A guidance language for controlling large language models, 2023. URL `https://github.com/guidance-ai/guidance?tab=readme-ov-file#text-not-tokens`.

[56] R. S. 4d masks support in transformers, 2024. URL `https://huggingface.co/blog/poedator/4d-masks`.

[57] M. Schuster and K. Nakajima. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[58] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In K. Erk and N. A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, Aug. 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL `https://aclanthology.org/P16-1162`.

[59] R. Shi, Y. Chen, Y. Hu, A. Liu, H. Hajishirzi, N. A. Smith, and S. S. Du. Decoding-time language model alignment with multiple objectives. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[60] W. Shi, X. Han, M. Lewis, Y. Tsvetkov, L. Zettlemoyer, and W.-t. Yih. Trusting your evidence: Hallucinate less with context-aware decoding. In K. Duh, H. Gomez, and S. Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 783–791, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-short.69. URL `https://aclanthology.org/2024.naacl-short.69`.

[61] Y. Tay, V. Q. Tran, S. Ruder, J. Gupta, H. W. Chung, D. Bahri, Z. Qin, S. Baumgartner, C. Yu, and D. Metzler. Charformer: Fast character transformers via gradient-based subword tokenization. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=JtBRnrlOEFN`.

[62] L. Team. The llama 3 herd of models, 2024. URL `https://arxiv.org/abs/2407.21783`.

[63] L. Team. Introducing llama 3.1: Our most capable models to date, 2024. URL https://ai.meta.com/blog/meta-llama-3-1/.

[64] L. Team. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models, 2024. URL https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/. Accessed: 2025/05/10.

[65] O. Team. Olmo release notes, 2025. URL https://allenai.org/olmo/release-notes#olmo-2-1b. Accessed: 2025/05/10.

[66] Q. Team. Qwen3: Think deeper, act faster, 2025. URL https://qwenlm.github.io/blog/qwen3/. Accessed: 2025/05/10.

[67] K. Tran. From english to foreign languages: Transferring pre-trained language models. *arXiv preprint arXiv:2002.07306*, 2020.

[68] B. Tunguz. 200,000+ jeopardy! questions, 1019. URL https://www.kaggle.com/datasets/tunguz/200000-jeopardy-questions.

[69] B. Tunguz. 200,000+ jeopardy! questions, 2019. URL https://www.kaggle.com/datasets/tunguz/200000-jeopardy-questions.

[70] A. Turaga. Character prefix conditioning with back tokenization, 2025. URL https://anilturaga.github.io/cpc.

[71] H. van Antwerpen and A. Neubeck. So many tokens, so little time: Introducing a faster, more flexible byte-pair tokenizer, 2025. URL https://github.blog/ai-and-ml/llms/so-many-tokens-so-little-time-introducing-a-faster-more-flexible-byte-pair-tokenizer/. Accessed: 2025/05/10.

[72] T. Vieira, B. LeBrun, M. Giulianelli, J. L. Gastaldi, B. DuSell, J. Terilla, T. J. O'Donnell, and R. Cotterell. From language models over tokens to language models over characters. *arXiv preprint arXiv:2412.03719*, 2024.

[73] T. Vieira, T. Liu, C. Pasti, Y. Emara, B. DuSell, B. LeBrun, M. Giulianelli, J. L. Gastaldi, T. J. O'Donnell, and R. Cotterell. Language models over canonical byte-pair encodings. *arXiv preprint arXiv:2506.07956*, 2025.

[74] J. Wang, T. Gangavarapu, J. N. Yan, and A. M. Rush. Mambabyte: Token-free selective state space model. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=X1xNsuKssb.

[75] Y. Xu, J. Chen, J. Wu, and J. Zhang. Hit the sweet spot! span-level ensemble for large language models. *arXiv preprint arXiv:2409.18583*, 2024.

[76] Y. Xu, J. Lu, and J. Zhang. Bridging the gap between different vocabularies for llm ensemble. 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 7133–7145, 2024.

[77] Y. Xu, J. Chen, J. Wu, and J. Zhang. Hit the sweet spot! span-level ensemble for large language models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 8314–8325, 2025.

[78] L. Xue, A. Barua, N. Constant, R. Al-Rfou, S. Narang, M. Kale, A. Roberts, and C. Raffel. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022. doi: 10.1162/tacl_a_00461. URL https://aclanthology.org/2022.tacl-1.17.

[79] L. Yu, D. Simig, C. Flaherty, A. Aghajanyan, L. Zettlemoyer, and M. Lewis. MEGABYTE: Predicting million-byte sequences with multiscale transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=JTmO2V9Xpz.

[80] Y.-C. Yu, C. C. Kuo, Y. Ziqi, C. Yucheng, and Y.-S. Li. Breaking the ceiling of the llm community by treating token generation as a classification for ensembling. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1826–1839, 2024.

[81] W. Zheng, Y. Chen, W. Zhang, S. Kundu, Y. Li, Z. Liu, E. P. Xing, H. Wang, and H. Yao. Citer: Collaborative inference for efficient large language model decoding with token-level routing. *arXiv preprint arXiv:2502.01976*, 2025.

A Related Work

Byte-level language models Although our method is able to convert a model using a traditional BPE tokenizer into a byte-level model, allowing it to be used in situations where byte-level models are required, it may not enjoy the benefits of being trained natively at the byte level. Training byte-level models are an active area of research [12, 78, 74]. However, byte-level language models may still implicitly aggregate multiple bytes into a single "patch" to help reduce the required sequence length. These patches can be segmented either statically [61, 79] or dynamically [45, 49, 1] which *may* lead to issues analogous the Prompt Boundary Problem at the patch level, depending on the architecture.

Tokenizer transfer Methods to adapt a model to use tokenizers other than the one they are trained with have been proposed. These methods may rely on interventions during training [7], continued training for a subset of the model with the new tokenizer [39], using self-distillation [44], careful initialization of a new embedding matrix, followed by fine-tuning [42, 17, 67, 36, 14], or zero shot transfer using a hypernetwork [43]. While these methods can, in principle, be used to convert any model into a byte-level model, they will inevitably introduce some distortion into the model's sampling distribution.

Ensembles of language models Many methods to address the mismatching vocabularies one counters when ensembling models have been proposed. These include bridging the vocabularies using a mapping based on model features [22] or edit distance [40] as well as sampling from the union [80] or intersection [76] of multiple vocabularies. There are also several methods that sample multiple tokens of continuation from each model and then select the best one using a scoring metric [35, 77, 38]. For a survey of such methods, including ones that require training or additional data, see Chen et al. [8]. However, unlike our exact method, all of these methods may introduce distortion into the model's outputs.

Word level probabilities The popular decision to include whitespace with the following word in most modern tokenizers presents a challenge when computing the next word probability [46, 52], which is closely related to the Prompt Boundary Problem.

Nondeterministic tokenizers Our analysis crucially relies on the determinism of BPE, however nondeterministic tokenizers such as Unigram [26] and BPE dropout [53] are of interest to the community. Lundberg [37] remarks that nondeterministic tokenizers may reduce the severity of the prompt boundary problem, but it cannot do so perfectly. It is possible that more advanced techniques may be able to fully correct the PBP for these tokenizers as well.

B Experimental Details

In this appendix, we report additional experimental details.

B.1 Calculation of the Naive Method

The naive method is simple to state. We merely report the average probability that the next character sampled after the prompt will be the correct one. However, some complexity arises when considering multibyte characters, which occur occasionally in English text and essentially constantly in Chinese. A multibyte character may correspond to multiple tokens under a byte-level BPE tokenizer, which means that multiple sampling steps may be necessary to form the next character. To handle this properly, we compute the tree of all token sequences which start with the desired character (depicted in Fig. 2b) and score the log-probability of all of its leaves to determine the exact probability that the desired next character will be generated. Note that we do not perform the pairwise pruning in this step, as we describe in Fig. 2c and Section 3.1. It is not strictly necessary, since a single character can be at most four bytes under UTF-8, so the size of the tree will always be small, and omitting the pruning step presents the baseline in the best light.

B.2 Details for Ensemble Evaluations

For the ensemble evaluations we use few-shot prompting with five in-context examples for each query. We choose the few-shot examples randomly to avoid any bias and ensure that the question being tested is not among the examples. We sample the continuation greedily and test whether the resulting text contains the correct answer.

1. **Arithmetic** contains simple arithmetic problems [5].[10] We use the `2da`, `2dm`, and `2ds` splits for addition, multiplication, and division of (up to) 2-digit numbers.
2. **DROP** contains questions about passages, potentially requiring reasoning over multiple pieces of information in the passage [15].
3. **Jeopardy** contains open-ended questions from the "Jeopardy!" quiz show [69].
4. **LAMBADA** contains narratives without the last word, which is inferrable given the context [50]. This task requires models to attend to the full narrative instead of only the local context.
5. **SQuAD** contains passages paired with questions about the passage [54]. The answer is always a span from the passage.
6. **TriviaQA** contains open-ended questions about world knowledge [24].
7. **BIG-bench WikidataQA** require models to complete factual statements with the correct continuation [4].

To save compute, we randomly subsample large datasets down to 5,000 examples.

B.3 Details for Proxy-Tuning Evaluations

Following Liu et al. [33], we use the proper instruct template for OLMO2-INSTRUCT and use a basic Question/Answer format for the base models. Unlike in the previous section, we use a more varied evaluation setup.

1. For **AlpacaEval 2**, we prompt using the instruction as the question and take the response as the answer. This is done with no chain of thought prompting or in-context examples. We use the default AlpacaEval 2 judge and report the length-controlled win-rate in our results.
2. For **GSM8k**, we use five in-context examples, which naturally cause the model to produce chains of thought. We extract the final number produced by the model and test if it exactly matches the answer (removing any commas).
3. For **MMLU**, we use no in-context examples and use the chain-of-thought prompt from Lambert et al. [30] to elicit chains of thought resulting in a multiple-choice answer. Unlike with the other datasets, we do not truncate MMLU to 5,000 examples since its examples are distributed across various domains. We report the multiple-choice accuracy in our results.

These evaluations were intended to benefit from instruction-following capabilities and general knowledge model performance.

B.4 Compute Resources

Our experiments were conducted with a variety of computing resources, including Nvidia A40, L40S, and A100 GPUs. Our method only requires one GPU at a time and features minimal memory overhead compared to regular sampling. We estimate that the total compute required to reproduce all of our results is less than 200 L40S hours.

C Implementation Details and Optimizations

In this appendix, we report implementation details that improve performance and ensure correctness.

C.1 Inference Optimizations

To ensure that our method is practical we employ a number of optimizations. In order to quickly compute the Valid Cover Tree, we maintain a cache of token masks which are valid following a given token and a separate cache for masks specifying tokens that begin with certain common byte prefixes. Then given a node of the tree, we can quickly expand it, as described in Algorithm 1 by fetching the relevant masks from both caches and intersecting them on the GPU to find the valid children to add.

[10] `https://huggingface.co/datasets/EleutherAI/arithmetic`

When evaluating the probabilities of the leaves of the Valid Cover Tree, we use 4D attention masks [56] to perform inference for the entire tree in a single query. Additionally, while sampling we use KV-caching to avoid redundant computation. Combining these two techniques can lead to excessive memory usage because tokens corresponding to branches that are ultimately not selected by sampling take up space in the KV cache. To address this, we implement a copying garbage collector for the KV cache which discards such tokens from the cache. Since the GC can be run one layer at a time, its total memory overhead is negligible. When using the GC, the KV cache will store exactly one set of keys and values for each token in the *current* Valid Cover Tree, reducing the memory overhead compared to naive sampling to a constant.

We also implement batching, allowing one to sample multiple sequences of bytes in parallel, which permits better utilization of GPU resources.

C.2 Byte-level vs Character-level BPE

Throughout this work, we assume that BPE is carried out at the byte level. However, the alternative, performing BPE at the character level, is also a popular choice. Our method can be extended to character-level BPE merges in a natural manner. In particular, one can perform our method at the character level instead. All the analysis we provide, including the guarantees for the Valid Cover Tree in Section 3.1 continue to hold regardless of the choice of base vocabulary. The only additional logic that needs to be implemented revolves around the handling of byte fallback, which is a feature that allows the tokenizer to represent characters that were not included in the base vocabulary explicitly using their Unicode encoding. To handle this properly, we will need to "reset" the tree whenever we encounter a character encoded using byte fallback, since BPE merges do not interact with byte fallback (essentially the byte encoded character acts as a pretokenization boundary). In order to condition on an arbitrary *byte* sequence, we must consider the possibility that a partial character will be completed to form one not in the base vocabulary, necessitating the addition of a "byte fallback" branch to the Valid Cover Tree. In all other regards, the approach is the same as the one we outline in Section 3.

C.3 Handling pretokenization

So far, we have focused on correctly handling byte pair encoding, ignoring the pretokenization conventionally applied beforehand. To illustrate why this is step is important, recall that pretokenization is often used to ensure that tokens cannot span multiple words and that whitespace separating words is merged with the following word and not the preceding one. In order to correctly handle all aspects of modern tokenizers, we must also perform pretokenization in an online fashion, which is challenging in its own right.

Pretokenization is typically implemented using a regular expression: beginning at the start of the text, the longest prefix matching the regular expression is found greedily. This prefix is then extracted into a pretoken and the process is repeated on the suffix. This continues until the entire string has been processed. In order to properly handle pretokenization, we must also perform this splitting online. Due to the expressivity of regular expressions, this requires maintaining a tree of possible splits, which are resolved once enough text is observed, to conclude whether the regex will match or not.

C.3.1 General Solution

In principle, the implementation of this idea is straightforward. We can convert any splitting regular expression into a finite automaton, which allows us to detect matches incrementally. By performing connectivity analysis on the automata's state graph, we can infer *(i)* whether there exists a suffix that could produce a regex match (which would mean that the pretokenization might not end up splitting at this point) and also *(ii)* whether there exists a suffix which would cause the regex to stop matching at this point (which would mean that the pretokenization might end up splitting at this point). This analysis can be precomputed for each state in the automaton, allowing these checks to be performed in constant time for each new byte.

If the verdict is ambiguous (both splitting and not splitting are possible), then we add an additional subtree to the Valid cover Tree which assumes that the split has indeed happened. The portion to the left of the split can only be tokenized one way (since its endpoint is fixed), while the portion to the right of the split will be isomorphic to a new Valid Cover Tree for just the portion of the prefix following the hypothetical split. As we continue to add new bytes, we maintain both branches of the tree, just as we would normally. Once enough bytes are added, we can determine conclusively which option was taken, allowing us to discard the subtree corresponding to the opposite possibility.

Of course, it is possible that a new position may occur where the splitting cannot be determined conclusively before the first one is resolved. This will necessitate further splitting of the tree (potentially in both subtrees). In general, this may lead to trees of exponential size, but for typical pretokenizers in use today, we can still guarantee that the tree will have finite size.

C.3.2 Practical Solution

Unfortunately, the general solution we outlined in the previous section is difficult to implement in practice. First, most regular expression engines in use today support matching features that are not strictly regular, which makes the conversion of its regexes into automata impossible in the general case. While these features are not used by any pretokenizer we are aware of, the possibility thereof has made it difficult to find routines that are able to perform this conversion for existing regex engines.

To provide a correct implementation while avoiding the complexity of writing our own regex engine, we provide bespoke handlers which are able to handle the pretokenization rules in common use. In general most pretokenization regular expressions have the desirable property that any prefix of a match is also a match. We call this property *closed under prefix.* This makes the detection of possible splitting points very easy, since once the regex stops matching new characters, we know there is no suffix that can extend it. There are only a handful of rules which do not have this property:

- Most tokenizers have a lookahead rule which stops matching whitespace one before the last whitespace. Thus given three spaces in a row, followed by a letter, the first two spaces would be one pretoken and the last space and letter would form a second pretoken.
- Many tokenizers have a "contraction merging" rule which forces contraction suffixes such as ⟨'ve⟩ to be individual pretokens. This is tricky because ⟨'ve⟩ is considered a match but ⟨'v⟩ is not.

We provide handlers for expressions that are closed under prefix, as well as the two special cases we listed above. This is enough to correctly support all pretokenizers we are aware of.

C.4 Handling special tokens

Special tokens are tokens that are assigned special meaning and are not used simply to represent text. These tokens can have a variety of uses, including marking the beginning or end of documents or separating turns in a dialog. It is easy to handle special tokens in the prompt: when we see a special token, we terminate the tree at that point (discarding potential continuations) and output it, output the ID of the special token, and then start a new tree.

To handle special tokens in the output, we consider the special token distribution on the branch of the tree that ends exactly at the end of the prompt and add those tokens as generation options with the corresponding probabilities alongside the 256 possible next bytes. When composing multiple models which have different sets of special tokens, we require a mapping to specify which tokens have the same meaning. This mapping is automatically detected for BOS and EOS tokens, but must be manually specified for others.

C.5 Converting Merge Lists to Normal Form

Throughout this work, we have assumed that the tokenizer is constructed using the BPE training algorithm, which proceeds by iteratively merging the most frequent pair in the partially tokenized training corpus. This assumption leads to merge lists that have three desirable properties: *(i)* every token has a unique merge that forms it, *(ii)* every token can be achieved as the tokenization of some string, and *(iii)* the merges always appear in the order they are merged. We assume that these properties are true in the analysis we present in Section 3.

However in practice, some models use "BPE" tokenizers with merge lists that are not directly produced by the BPE training algorithm. One example of this is the tokenizer of LLAMA 3 [62], which appears to be constructed by extending OpenAI's cl100k tokenizer [48] with additional merges intended to add multilingual capabilities. Because of way this extension is done, the LLAMA 3 tokenizer does not have any of the three properties we outlined above. Despite this, inference with the tokenizer is still possible because some tokenization libraries such as HuggingFace Tokenizers[11] employ a heap-based algorithm which simply applies the earliest merge available until no more merges can be applied, which permits the merges to be applied out of order.

Fortunately, it happens to be the case that every merge list can be converted into a functionally equivalent one in "normal form" which has identical behavior while also satisfying the three properties above. This is done using a two step process: *(1)* for each token, we run the heap-based algorithm on it as a string and track which merges are used during the tokenization process. If the resulting token sequence is *not* just the single corresponding token id, then we mark the token as "unreachable" and drop it (this ensures property *(ii)*). Otherwise, we check which merge was applied last and drop any other merges which form the same token since they are also unreachable (this ensures property *(i)*). Then *(2)*, for every merge we check the position of the merges forming its two inputs and move it to immediately after the later of the two if it appears after the original merge (this ensures property *(iii)*). This procedure allows our method to be used with any tokenizer that can be specified using a merge list, even if it was not trained using BPE.

D Invalid segmentations

D.1 Differences in exact methods

In this work, we consider a method exact if it samples according to the distribution in Eq. (2) modulo the probability mass placed on invalid sequences, which we defined in Section 3.1. Here we describe exactly how these methods differ in their handling of invalid sequences. The method of Turaga [70] and our method condition on a valid covering of the prompt. This corresponds to the distribution

$$\mathsf{P}\left(t_1, \ldots, t_n \,\middle|\, \begin{array}{c} \text{prompt} \sqsubseteq \text{decode}(t_1, \ldots, t_n), [t_1, \ldots, t_k] \text{ is valid} \\ \text{where } k = \min\{i \mid \text{prompt} \sqsubseteq \text{decode}(t_1, \ldots, t_i)\} \end{array}\right). \tag{3}$$

While difficult to notate, this simply means that the portion of the sequence overlapping the prompt is required to be valid. This is roughly similar to common practice described in Eq. (1) of directly tokenizing the prompt and sampling a continuation while avoiding the PBP. Meanwhile, Phan et al. [51] consider a relaxation of the above, which does not require the last pair to be valid. This corresponds to

$$\mathsf{P}\left(t_1, \ldots, t_n \,\middle|\, \begin{array}{c} \text{prompt} \sqsubseteq \text{decode}(t_1, \ldots, t_n), [t_1, \ldots, t_{k-1}] \text{ is valid} \\ \text{where } k = \min\{i \mid \text{prompt} \sqsubseteq \text{decode}(t_1, \ldots, t_i)\} \end{array}\right). \tag{4}$$

The less strict conditioning explains why this method has greater overhead, as seen in Section 4.1.

It may seem desirable to sample from the distribution

$$\mathsf{P}(t_1, \ldots, t_n \,|\, \text{prompt} \sqsubseteq \text{decode}(t_1, \ldots, t_n), [t_1, \ldots, t_k] \text{ is valid}), \tag{5}$$

where the entire sequence is required to be valid. However, it is not clear how to efficiently sample from this distribution. Vieira et al. [73] highlights this difficulty and propose several alternative approaches, including approximations of Eq. (5) and architectural modifications that make it easier to sample from Eq. (5).

When applying ByteSampler iteratively, the validity of the sequence is enforced continuously. Since this is done locally, the resulting distribution corresponds to the "locally canonicalized approximation" of Eq. (5) described in Vieira et al. [73] additionally conditioned on a prompt.

D.2 Significance of invalid segmentations

For the most part, we have ignored the contribution of invalid sequences to the language model's distribution. This is done out of necessity, since the number of invalid sequences scales exponentially with the prompt length [72]. However it is worth considering whether these segmentations could contribute meaningfully to the model's capabilities.

This is closely related to the concept of *marginalization* [6]: the idea that calculating the probability of generating a string with a language model requires summing over all segmentations of the string, (including invalid ones). Of note, Chirkova et al. [9] found that $\mathsf{P}([t_1, \ldots, t_n] \text{ is not valid})$ makes up a negligible fraction of the language model's distribution, however later works [19, 73] came to the opposite conclusion.

[11] `https://github.com/huggingface/tokenizers`

E Example Valid Cover Trees

Here we show complete Valid Cover Trees for several example prefixes. Unlike the tree in Fig. 2c, we show the actual tree as calculated by our algorithm. However to allow them to fit on a page, we choose to display *only the internal nodes* of the tree (not the leaves). To denote where the hidden leaves would be, we display nodes that have leaves in **bold font**.

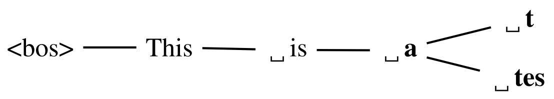


Figure 4: Example Valid Cover Tree for prefix “this is a tes” with the OLMo 2 tokenizer.

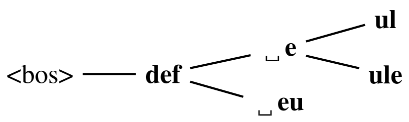


Figure 5: Example Valid Cover Tree for prefix “def eule” with the OLMo 2 tokenizer.

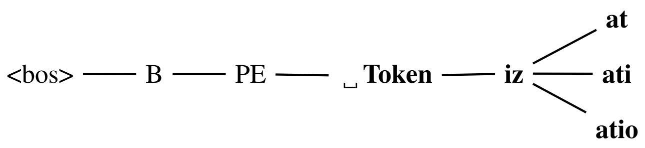


Figure 6: Example Valid Cover Tree for prefix “BPE Tokenizatio” with the OLMo 2 tokenizer.

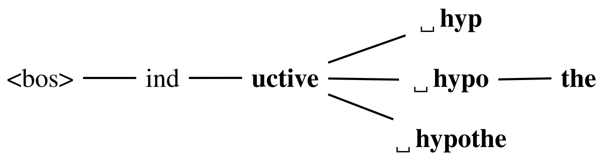


Figure 7: Example Valid Cover Tree for prefix “inductive hypothe” with the OLMo 2 tokenizer.

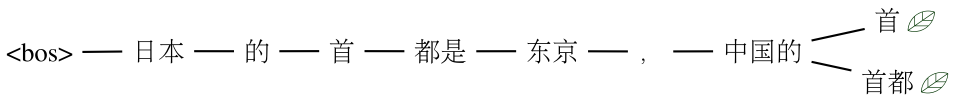


Figure 8: Example Valid Cover Tree for prefix “日本的首都是东京，中国的首都” with the QWEN3 tokenizer. We use 🍃 to denote nodes with leaves omitted.

We hide the leaves because it is typical for nodes that do have leaves to have dozens or even hundreds of them. To see how this can occur, imagine a prompt that ends on a space, and an internal node that ends right before that space. The node’s children will be all valid tokens that begin with a space. Most tokenizers have tens of thousands of tokens which begin with a space and nearly all of them will be valid continuations.

While this may sound problematic, we only need to query the next token distribution for the parent once in order to score all of its children, so this can be done efficiently in combination with the masking cache we describe in Appendix C.1.

F Advanced Decoding Methods

In Section 3, we focused on showing that our method is “exact.” To be precise, this means that sampling bytewise using our method and sampling normally give exactly the same distributions of output text (modulo invalid token sequences, as we discussed in Appendix D). However, this applies only when doing standard sampling from the model, and not when transforming the distribution using popular decoding techniques such as greedy decoding, top-k, top-p [21], or even temperatures other than 1. This is because these transformations have different effects when applied with different granularities (clearly, greedily selecting the most likely next byte is not the same as greedily selecting the most likely next token). It is not immediately clear what advantages or disadvantages are gained by transforming the LM's textual distribution in this way, and we believe this presents an interesting direction for future work.