TRANSDUCING LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern language models define distributions over strings, but their outputs are not always suited to downstream task. For instance, a model generating byte-pair strings may not be suitable when word-level predictions are needed, and a DNA model may not fit applications requiring amino acids. In such cases, a deterministic string-to-string transformation can convert the model's output to the desired form. This is a familiar pattern in probability theory: applying a function f to a random variable $X \sim p$ yields a transformed random variable $f(X)$ with an induced distribution. While such transformations are occasionally used in language modeling, they are not treated as yielding new, fully functional language models. We formalize this perspective and introduce a general framework for language models derived from deterministic string-to-string transformations. Focusing on transformations representable as finite-state transducers—a commonly used state-machine abstraction for efficient string-to-string mappings—we develop algorithms that compose a language model with an FST to *marginalize* over source strings mapping to a given target. This allows us to propagate probabilities through the transducer without altering model parameters and to *condition* on transformed outputs. We present an exact algorithm, an efficient approximation, and a theoretical analysis. We conduct experiments in three domains: converting token-level language models to character-level language models, token-level language models to word-level models, and deriving amino-acid models from DNA models. This demonstrates inference-time adaptation of pretrained language models to match application-specific output requirements.

1 INTRODUCTION

Language models (LMs) define distributions over strings. Yet, the strings they produce often fail to align with the requirements of downstream applications, leading practitioners to apply ad hoc post-processing. We call this the **string mismatch problem**. For example, in natural language processing, modern language models typically generate byte-pair encoded strings (Sennrich et al., 2016), while downstream tasks may require words or characters instead (see Ex. (2), below). Similarly, DNA language models generate nucleobase sequences, but many biological applications require strings of the corresponding amino acids (see Ex. (5), below).

Adding a string-to-string transformation to a generation pipeline is a practical and common engineering solution. Such as normalizing output, or mapping bytes to UTF-8. Formally, this defines a new language model over *transformed* strings. However, while sampling remains straightforward—simple operations like computing the probability of a transformed variable are no longer available, and conditioning on transformed strings is off the table. Consider, for instance, the mapping from a string in any casing to its lowercase version, as in the use-case depicted in Fig. 1. While lowercasing a given input is trivial, converting the original distribution to a distribution over lowercased words, is not.

In this work, we promote string-to-string transformations to first-class citizens in the language modeling pipeline. We enable direct reasoning about the transformed distributions of string-valued random variables and introduce practical algorithms that operate on the transformed models at inference time. This approach to the string mismatch problem is principled, modular, and often far more achievable than retraining a language model to generate transformed strings directly. Moreover, the transformations often guarantee adherence to the requirements of the downstream applications.

We now provide a motivating example in the context of a language model over English utterances. Consider the following sentence:

(1) *Dr. Lemaître was flabbergasted.* 🤯

The byte-pair encoding used by GPT4o (OpenAI, 2024) encodes Ex. (1) as the following string of tokens:

(2)

Dr	.	␣L	ema	\C3\AEtre	␣was	␣fl	ab
5822	13	451	4603	29135	673	1548	378

berg	asted	.	␣\F0\9F\A4	\AF
9667	23030	13	93643	107

The segmentation of Ex. (1) into tokens induced by the byte-pair encoding is based on character substring frequency. However, many applications would benefit from a different choice of units. For instance, in computational psycholinguistics (e.g., (Giulianelli et al., 2024)) and controlled generation (e.g., (Lew et al., 2023; Xefteri et al., 2025)). This also holds if one wishes to derive distributions over grammatical words, such as those defined by the Penn Treebank (PTB) annotation guidelines (Marcus et al., 1993), a variation of which is shown below:

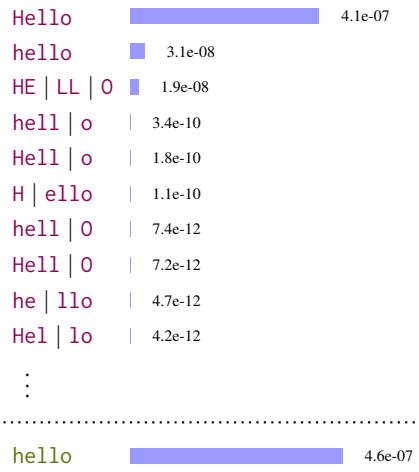


Figure 1: GPT-2 probabilities for the BPE `token strings` that, when lowercased, match `hello`. The total probability of all such sequences is at the bottom.

(3) DR. LEMAÎTRE WAS FLABBERGASTED . 🤯

For other applications, e.g., spelling correction, we might also wish to represent Ex. (2) using a string of characters or UTF-8 byte representation as follows: [1]

(4)

D	r	.	␣	L	e	m	a	\C3\AE	t	r	e	␣	w	a	s	␣	f	l	a	b	b	e	r	g	a	s	t	e	d	.	␣	\F0\9F\A4\AF
68	114	46	32	76	101	109	97	195	174	116	114	101	32	119	97	115	32	102	108	97	98	98	101	114	103	97	115	116	101	100	46	32	240	159	164	175

In genetics, we have another example of varying representations. Consider the DNA sequence given in Ex. (5). The sequence is one of many that translate into the hormone *oxytocin*, typically represented by the amino acid sequence in Ex. (6), as represented below, along with their integer encodings:

(5)

T	G	T	T	A	C	A	T	A	C	A	A	A	A	T	T	G	T	C	C	T	C	T	A	G	G	T
3	1	3	3	0	2	0	3	0	2	0	0	0	0	3	3	1	3	2	2	3	2	3	0	1	1	3

(6)

C	Y	I	Q	N	C	P	L	G
1	19	7	13	11	1	12	9	5

Transforming a language model is, in general, non-trivial, and success depends heavily on the complexity of the mapping. Recent papers explore methods for obtaining probabilities over bytes using subword models (Phan et al., 2024; Hayase et al., 2025, *inter alia*). In particular, Vieira et al. (2025a) addresses this problem in the case of *strict-prefix monotone transformations*, such as converting token-based models into character-based ones. This paper generalizes that approach to handle more complex conversions—where target units need not be direct constituents of the source units as in ex. (6).

We introduce a foundational framework for such conversions, involving the composition of pretrained language models and string-to-string functions, encoded by transducers, referred to as *transduced language models*. We develop exact and approximate algorithms for efficient sampling, scoring, and conditioning on transformed strings, all without modifying the underlying language model.

To validate our approach, we construct FSTs for the three use cases above: (i) by converting tokens to characters, (ii) inserting orthographic boundaries following the Penn Treebank tokenizer, and (iii) converting DNA sequences to sequences over amino acids. We then employ commonly used pretrained language models over the input units of the FSTs, and compose them with the FSTs to obtain language models over the output tokens. Finally, we use these settings to benchmark the theoretical and algorithmic contributions. In particular, we find that using an approximation of the exact algorithm is sufficient to obtain a good approximation at a fraction of the computational cost.

[1] Note that UTF-8 allows multiple encodings of some strings. For example, the character î can be encoded *composed* (`\C3\AE`) or *decomposed*(`i\CC\82`). See https://unicode.org/reports/tr15/.

2 BACKGROUND

We now introduce core background material. A glossary of our notation is given in App. A.

STRINGS. Let $\mathcal{X}$ be an **alphabet** (i.e., a finite, non-empty set). Let $\mathcal{X}^*$ denote the set of all finite strings over $\mathcal{X}$, including the empty string $\varepsilon_{\mathcal{X}}$. When there is no risk of ambiguity with other alphabets, we simply write ε. We use $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}^*$ to denote strings and $Z, Z' \subseteq \mathcal{X}^*$ to denote sets of strings. Let $\boldsymbol{x}\boldsymbol{x}'$ denote **concatenation**, similarly, we define the concatenation of sets of strings as $ZZ' \stackrel{\text{def}}{=} \{\boldsymbol{x}\boldsymbol{x}' \mid \boldsymbol{x} \in Z, \boldsymbol{x}' \in Z'\}$, and in the singleton case as $\boldsymbol{x}Z' \stackrel{\text{def}}{=} \{\boldsymbol{x}\}Z'$, and $Z\boldsymbol{x}' \stackrel{\text{def}}{=} Z\{\boldsymbol{x}'\}$. We write $\boldsymbol{x} \preceq \boldsymbol{x}'$ when $\boldsymbol{x}$ is a **prefix** of $\boldsymbol{x}'$, and $\boldsymbol{x} \prec \boldsymbol{x}'$ when it is a **strict prefix**.

LANGUAGE MODELS. A **language model** $p_{\mathcal{X}}$ is a probability distribution over a set of strings $\mathcal{X}^*$. Let $\text{EOS} \notin \mathcal{X}$ be a special **end-of-string symbol**. We define the **prefix probability** of $p_{\mathcal{X}}$ as the probability that a string $X \sim p_{\mathcal{X}}$ starts with a given prefix $\boldsymbol{x}$[2], and the **conditional prefix probability** as fractions:

$$\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{x}' \in \mathcal{X}^*} p_{\mathcal{X}}(\boldsymbol{x}\boldsymbol{x}') \qquad \overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}' \mid \boldsymbol{x}) \stackrel{\text{def}}{=} \frac{\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}\boldsymbol{x}')}{\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x})} \qquad \overrightarrow{p_{\mathcal{X}}}(\text{EOS} \mid \boldsymbol{x}) \stackrel{\text{def}}{=} \frac{p_{\mathcal{X}}(\boldsymbol{x})}{\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x})} \quad (1)$$

when $\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}) > 0$; otherwise we set $\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}' \mid \boldsymbol{x}) \stackrel{\text{def}}{=} 0$ and $\overrightarrow{p_{\mathcal{X}}}(\text{EOS} \mid \boldsymbol{x}) \stackrel{\text{def}}{=} 1$. Therefore, $\overrightarrow{p_{\mathcal{X}}}(\cdot \mid \boldsymbol{x})$ is a probability distribution over $\mathcal{X} \sqcup \{\text{EOS}\}$ for all $\boldsymbol{x} \in \mathcal{X}^*$, [3] where $\sqcup$ is used to indicate that $\mathcal{X}$ and $\{\text{EOS}\}$ are disjoint. Using this structure, any language model $p_{\mathcal{X}}$ may be factorized as $p_{\mathcal{X}}(\boldsymbol{x}) = \overrightarrow{p_{\mathcal{X}}}(\text{EOS} \mid \boldsymbol{x}) \prod_{t=1}^{|\boldsymbol{x}|} \overrightarrow{p_{\mathcal{X}}}(x_t \mid \boldsymbol{x}_{<t})$ where each $\overrightarrow{p_{\mathcal{X}}}(x_t \mid \boldsymbol{x}_{<t})$ corresponds to the probability of $x_t \in \mathcal{X} \sqcup \{\text{EOS}\}$, given the prefix $\boldsymbol{x}_{<t} \in \mathcal{X}^*$.[4] This factorization defines a left-to-right generative process: starting from $\boldsymbol{x}$ equals ε, we repeatedly sample $x' \sim \overrightarrow{p_{\mathcal{X}}}(\cdot \mid \boldsymbol{x})$; if $x' = \text{EOS}$, we stop, otherwise we update $\boldsymbol{x}$ to $\boldsymbol{x}x'$. Conditional generation simply starts from the conditioning prefix instead of the empty string. We refer to these operations as the **interface** to the language model $p_{\mathcal{X}}$.

CYLINDRICAL SETS. Cylindrical sets define strings with a given prefix and are inherent to the definition of prefix probabilities. Let $Z, Z' \subseteq \mathcal{X}^*$, we define the **cylinder** over Z as $\langle Z \rangle \stackrel{\text{def}}{=} Z\mathcal{X}^*$. We say that Z is **cylindrical** if $Z = \langle Z \rangle$. Note that the union of cylinder sets is a cylinder set. We define the **basic cylinder** for $\boldsymbol{x}$ as $\langle \boldsymbol{x} \rangle \stackrel{\text{def}}{=} \langle \{\boldsymbol{x}\} \rangle$. The **prefix-base** operation $\text{pf}(Z)$ is defined by $\text{pf}(Z) \stackrel{\text{def}}{=} \{\boldsymbol{x} \in Z \colon \nexists \boldsymbol{x}' \in Z \text{ such that } \boldsymbol{x}' \prec \boldsymbol{x}\}$. The prefix-base operation uniquely partitions $\langle Z \rangle$ into basic cylinders over $\text{pf}(Z)$. We say that Z is **prefix-free** if $\text{pf}(Z) = Z$. Crucially, for all $\boldsymbol{x}, \boldsymbol{x}' \in \text{pf}(Z)$ where $\boldsymbol{x} \neq \boldsymbol{x}'$, the basis sets are disjoint $\langle \boldsymbol{x} \rangle \cap \langle \boldsymbol{x}' \rangle = \emptyset$ and exhaustive, $\bigcup_{\boldsymbol{x} \in \text{pf}(Z)} \langle \boldsymbol{x} \rangle = \langle Z \rangle$. Thus, $\text{pf}(Z)$ is a set of representative elements such that $\langle Z \rangle = \bigsqcup_{\boldsymbol{x} \in \text{pf}(Z)} \langle \boldsymbol{x} \rangle$.

TRANSDUCERS. A **transducer** is a state-machine that encodes string-to-string relations $f \subseteq \mathcal{X}^* \times \mathcal{Y}^*$. Transducers are a key component in our work, when we express a relationship defined by f as a transducer, we expose the computational structure required for developing efficienct algorithms. Formally, a **finite-state transducer**[5] **(FST)** f is a tuple $(\mathsf{S}, \mathcal{X}, \mathcal{Y}, \mathsf{T}, \mathsf{I}, \mathsf{F})$ where S is a finite set of **states** and $\mathcal{X}, \mathcal{Y}$ are alphabets of **input** and **output** symbols. The sets $\mathsf{I}, \mathsf{F} \subseteq \mathsf{S}$ are the **initial** and **final** states. $\mathsf{T} \subseteq \mathsf{S} \times (\mathcal{X} \cup \{\varepsilon\}) \times (\mathcal{Y} \cup \{\varepsilon\}) \times \mathsf{S}$ is a set of **transitions**. We render transitions $(s, x, y, s') \in \mathsf{T}$ as $s \xrightarrow{x:y} s'$, we say the transition **scans** x and **emits** y. We denote the set of outgoing transitions from state s with $\mathsf{T}(s)$, and $\mathsf{T}(s, x)$ to denote the set of outgoing transitions from s that scan for x.[6]

The transducer f defines a set of **paths** Π. Each **path** $\boldsymbol{\pi} \in \Pi$ is a sequence of transitions of the form $s_0 \xrightarrow{x_1:y_1} s_1 \xrightarrow{x_2:y_2} s_2 \cdots s_{N-1} \xrightarrow{x_N:y_N} s_N$. We sometimes describe $\boldsymbol{\pi}$ as a generalized transition $s_0 \overset{\boldsymbol{x}:\boldsymbol{y}}{\rightsquigarrow} s_N$ that scans $\boldsymbol{x} = x_1 x_2 \cdots x_N$ and emits $\boldsymbol{y} = y_1 y_2 \cdots y_N$, suppressing the intermediate transitions. We call $\boldsymbol{\pi}$ an **accepting path** if $s_0 \in \mathsf{I}$ and $s_N \in \mathsf{F}$. The **relation defined** by f is given by $[\![\mathsf{f}]\!] \stackrel{\text{def}}{=} \{(\boldsymbol{x}, \boldsymbol{y}) \mid (s \overset{\boldsymbol{x}:\boldsymbol{y}}{\rightsquigarrow} s') \in \Pi \colon s \in \mathsf{I}, s' \in \mathsf{F}\}$, i.e., each accepting path contributes (not necessarily uniquely) a pair of scanned and emitted strings. We give more details about transducers in App. C

[2] Note that the prefix probability is not a probability distribution over $\mathcal{X}^*$, but the probability of the event $\boldsymbol{x} \preceq X$, i.e., $\overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}) = \Pr_{X \sim p_{\mathcal{X}}}[\boldsymbol{x} \preceq X]$.

[3] However, note that $\overrightarrow{p_{\mathcal{X}}}(\cdot \mid \cdots \text{EOS} \cdots)$ is undefined.

[4] See Cotterell et al. (2024, Theorem 2.4.2) for a proof.

[5] We refer to Pin (2021, Ch. 2 & 3) for a detailed treatment of transducers.

[6] I.e., $\mathsf{T}(s'') \stackrel{\text{def}}{=} \{(s \xrightarrow{x:y} s') \in \mathsf{T} \colon s = s''\}$, and $\mathsf{T}(s'', x'') \stackrel{\text{def}}{=} \{(s \xrightarrow{x:y} s') \in \mathsf{T} \colon s = s'', x = x''\}$.

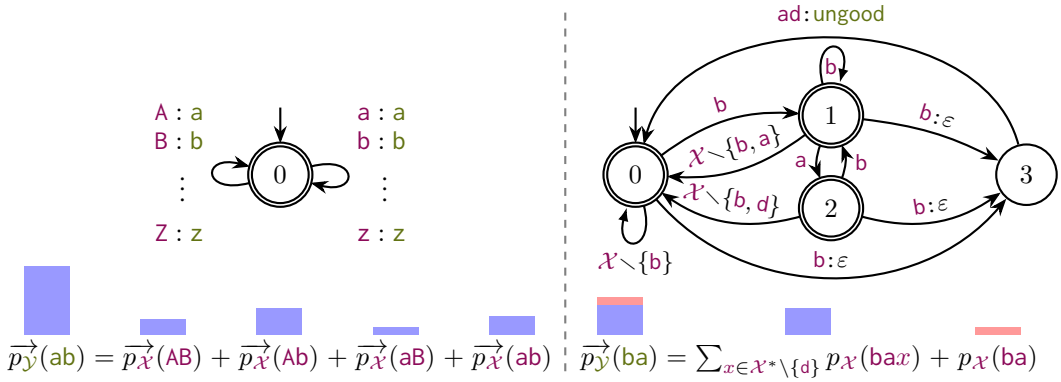


Figure 2: Two examples of transformations: (Left) A transducer that maps alphabetic strings to their lowercase form. Each lowercase output corresponds to a set of input strings that differ only in casing. (Right) A *newspeak* (Orwell, 1949) transducer, where the word 'bad' is not permitted. So $\langle \mathsf{bad} \rangle$ does not contribute to $\overrightarrow{p_\mathcal{Y}}(\mathsf{ba})$. We use a single symbol for copy transitions, e.g., b to denote $\mathsf{b} : \mathsf{b}$.

3 Transduced Language Models

A **transduced language model** $p_\mathcal{Y}$ arises from applying a string-to-string **transformation** $f : \mathcal{X}^* \to \mathcal{Y}^*$, encoded by a transducer f, to a string drawn from a **source language model** $p_\mathcal{X}$. Formally, if $X \sim p_\mathcal{X}$, then $f(X)$ has the following probability mass function:

$$p_\mathcal{Y}(\boldsymbol{y}) \stackrel{\text{def}}{=} \Pr_{X \sim p_\mathcal{X}}[\boldsymbol{y} = f(X)] = \sum_{\boldsymbol{x} \in f^{-1}(\boldsymbol{y})} p_\mathcal{X}(\boldsymbol{x}) \tag{2}$$

where $f^{-1}(\boldsymbol{y})$ is the **preimage** of $\boldsymbol{y}$, $f^{-1}(\boldsymbol{y}) \stackrel{\text{def}}{=} \{\boldsymbol{x} \in \mathcal{X}^* : \boldsymbol{y} = f(\boldsymbol{x})\}$. Put differently, in Eq. (2), we sum over the strings $\boldsymbol{x}$ such that $f(\boldsymbol{x}) = \boldsymbol{y}$. Unfortunately, evaluating $p_\mathcal{Y}(\boldsymbol{y})$ exactly using Eq. (2) is generally intractable, even though exact sampling from $p_\mathcal{Y}$ is efficient.

Like all language models, a transduced language model $p_\mathcal{Y}$ has prefix and conditional prefix probability functions. The prefix probability $\overrightarrow{p_\mathcal{Y}}$ is given by

$$\overrightarrow{p_\mathcal{Y}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \Pr_{X \sim p_\mathcal{X}}[\boldsymbol{y} \preceq f(X)] = \sum_{\boldsymbol{x} \in \mathcal{P}(\boldsymbol{y})} p_\mathcal{X}(\boldsymbol{x}) \tag{3}$$

where $\mathcal{P}(\boldsymbol{y})$ is the **precover** of $\boldsymbol{y}$, w.r.t. f, defined as $\mathcal{P}(\boldsymbol{y}) \stackrel{\text{def}}{=} \{\boldsymbol{x} \in \mathcal{X}^* : \boldsymbol{y} \preceq f(\boldsymbol{x})\}$.[7] Here, we sum over the strings $\boldsymbol{x}$ that transform to strings $f(\boldsymbol{x})$ that have $\boldsymbol{y}$ as a prefix, $\boldsymbol{y} \preceq f(\boldsymbol{x})$.

Prefix probabilities allow us to define conditional probabilities over string prefixes, giving a factorization of a string's probability (see §2). This enables efficient conditional generation from a language model via a simple left-to-right autoregressive sampling procedure. We develop a method in §4 that allows us to compute the sum in Eq. (3) in *finite* time for a general class of mappings, such as those mentioned in the introduction (i.e., normalizing text, inserting orthographic word boundaries, or converting DNA to amino-acid sequences). In §5, we present algorithms to compute these quantities.

4 Decomposing the Precover

In §3, we saw that if we can sum over the precover of $\boldsymbol{y}$, we can calculate $\overrightarrow{p_\mathcal{Y}}(\boldsymbol{y}) = \sum_{\boldsymbol{x} \in \mathcal{P}(\boldsymbol{y})} p_\mathcal{X}(\boldsymbol{x})$ (Eq. (3)), unlocking an interface to the transduced language model. This sum may have an infinite number of terms, preventing us from applying the full equation in practice. Consider the transducer in Fig. 2 (left) that lowercases a string. For the string ab, the precover is given by the infinite set $\mathcal{P}(\mathsf{ab}) = \langle\{\mathsf{AB}, \mathsf{Ab}, \mathsf{aB}, \mathsf{ab}\}\rangle$. Recall that $\overrightarrow{p_\mathcal{X}}(\boldsymbol{x}) = \sum_{\boldsymbol{x}' \in \langle \boldsymbol{x} \rangle} p_\mathcal{X}(\boldsymbol{x}')$ (§2). Following Eq. (3), we get the derivation (4a-4d). Eq. (4d) has a remarkable property: if we can decompose the precover into a union of basis sets, we can express the target prefix probability as a sum of prefix probabilities on the source side.

[7] Note that the precover depends on f, we suppress this dependency when it is clear from context.

$$\overrightarrow{p_{\mathcal{Y}}}(\mathsf{ab}) = \sum_{\boldsymbol{x} \in \mathcal{P}(\mathsf{ab})} p_{\mathcal{X}}(\boldsymbol{x}) \tag{4a}$$

$$= \sum_{\boldsymbol{x}' \in \{\mathsf{AB},\mathsf{Ab},\mathsf{aB},\mathsf{ab}\}} \sum_{\boldsymbol{x} \in \langle \boldsymbol{x}' \rangle} p_{\mathcal{X}}(\boldsymbol{x}) \tag{4b}$$

$$= \overrightarrow{p_{\mathcal{X}}}(\mathsf{AB}) + \overrightarrow{p_{\mathcal{X}}}(\mathsf{Ab}) \tag{4c}$$

$$+ \overrightarrow{p_{\mathcal{X}}}(\mathsf{aB}) + \overrightarrow{p_{\mathcal{X}}}(\mathsf{ab}) \tag{4d}$$

$$\overrightarrow{p_{\mathcal{Y}}}(\mathsf{ba}) = \sum_{\boldsymbol{x} \in \langle \mathsf{ba} \rangle \setminus \langle \mathsf{bad} \rangle} p_{\mathcal{X}}(\boldsymbol{x}) \tag{5a}$$

$$= \sum_{\boldsymbol{x} \in \bigcup_{x \in \mathcal{X} \setminus \{\mathsf{d}\}} \langle \mathsf{ba}x \rangle \cup \{\mathsf{ba}\}} p_{\mathcal{X}}(\boldsymbol{x}) \tag{5b}$$

$$= p_{\mathcal{X}}(\mathsf{ba}) + \sum_{x \in \mathcal{X} \setminus \{\mathsf{d}\}} \overrightarrow{p_{\mathcal{X}}}(\mathsf{ba}x) \tag{5c}$$

The transducer on the right in Fig. 2 converts any mention of the word bad into the word ungood while all other substrings remain the same, meaning that $\langle \mathsf{ba} \rangle \not\subseteq \mathcal{P}(\mathsf{ba})$ since $\mathsf{bad} \in \langle \mathsf{ba} \rangle$. This is an example of a transducer for which a decomposition into prefix probabilities is not always possible. Instead, we can at most decompose the precover of ba into a union of disjoint sets, one of which is the union of basis sets, and the other its complement in the precover. We refer to the former set as the *quotient*, and the latter as the *remainder*. The prefix probability of ba is derived in equations (5a-5c). The final step illustrates a **computational shortcut**—for any $\boldsymbol{y}$ we can decompose $\overrightarrow{p_{\mathcal{Y}}}(\boldsymbol{y})$:

$$\overrightarrow{p_{\mathcal{Y}}}(\boldsymbol{y}) = \underbrace{\sum_{\boldsymbol{x} \in \mathcal{R}(\boldsymbol{y})} p_{\mathcal{X}}(\boldsymbol{x})}_{\text{Remainder}} + \underbrace{\sum_{\boldsymbol{x} \in \mathcal{Q}(\boldsymbol{y})} \overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x})}_{\text{Quotient}}. \tag{6}$$

The reader who is convinced by these examples and is eager to see the algorithm for calculating Eq. (6) can jump to §5. We give a formal definition of the remainder and quotient in the next section, derive the general decomposition, and consider when the computational shortcut can be calculated in finite time.

4.1 The Prefix Decomposition of the Precover

Let $f \colon \mathcal{X}^* \to \mathcal{Y}^*$ be a map. For each $\boldsymbol{y} \in \mathcal{Y}^*$, define $\mathcal{C}(\boldsymbol{y}) \stackrel{\text{def}}{=} \{\boldsymbol{x} \in \mathcal{X}^* \mid \langle \boldsymbol{x} \rangle \subseteq \mathcal{P}(\boldsymbol{y})\}$, as the largest cylinder contained in $\mathcal{P}(\boldsymbol{y})$. By construction, this is the union of all basic cylinders fully included in $\mathcal{P}(\boldsymbol{y})$. We then define the **quotient** and **remainder** of $\boldsymbol{y}$ with respect to f as

$$\mathcal{Q}(\boldsymbol{y}) \stackrel{\text{def}}{=} \text{pf}(\mathcal{C}(\boldsymbol{y})) \qquad \text{and} \qquad \mathcal{R}(\boldsymbol{y}) \stackrel{\text{def}}{=} \mathcal{P}(\boldsymbol{y}) \setminus \mathcal{C}(\boldsymbol{y}). \tag{7}$$

The pair $(\mathcal{R}(\boldsymbol{y}),\ \mathcal{Q}(\boldsymbol{y}))$ is called the **prefix decomposition** of $\mathcal{P}(\boldsymbol{y})$. We then get

$$\overrightarrow{p_{\mathcal{Y}}}(\boldsymbol{y}) = \sum_{\boldsymbol{x} \in \mathcal{P}(\boldsymbol{y})} p_{\mathcal{X}}(\boldsymbol{x}) = \sum_{\boldsymbol{x} \in \mathcal{R}(\boldsymbol{y}) \sqcup \mathcal{C}(\boldsymbol{y})} p_{\mathcal{X}}(\boldsymbol{x}) \tag{8a, by definition}$$

$$= \sum_{\boldsymbol{x} \in \mathcal{R}(\boldsymbol{y})} p_{\mathcal{X}}(\boldsymbol{x}) + \sum_{\boldsymbol{x} \in \mathcal{Q}(\boldsymbol{y})} \sum_{\boldsymbol{x}' \in \mathcal{X}^*} p_{\mathcal{X}}(\boldsymbol{x}\boldsymbol{x}') \tag{8b}$$

$$= \sum_{\boldsymbol{x} \in \mathcal{R}(\boldsymbol{y})} p_{\mathcal{X}}(\boldsymbol{x}) + \sum_{\boldsymbol{x} \in \mathcal{Q}(\boldsymbol{y})} \overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}). \tag{8c}$$

This generalizes the example given in Eq. (5a-5c, 6). We next consider when the terms are finite.

4.2 Sufficient Conditions for Finite Quotients and Remainders

We say that a map $f \colon \mathcal{X}^* \to \mathcal{Y}^*$ is **strict-prefix monotone** if and only if $\forall \boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}^* \colon \boldsymbol{x} \prec \boldsymbol{x}' \implies f(\boldsymbol{x}) \prec f(\boldsymbol{x}')$. Similarly f is **prefix monotone** if and only if $\boldsymbol{x} \preceq \boldsymbol{x}' \implies f(\boldsymbol{x}) \preceq f(\boldsymbol{x}')$. We say that a map f is **prefix-continuous** if, for every $\boldsymbol{y} \in \mathcal{Y}^*$, the set $\mathcal{P}(\boldsymbol{y})$ is cylindrical. Equivalently

$$\mathcal{P}(\boldsymbol{y}) = \langle \mathcal{Q}(\boldsymbol{y}) \rangle \quad \iff \quad \mathcal{R}(\boldsymbol{y}) = \emptyset. \tag{9}$$

This assumption allows Vieira et al. (2025a) to effectively omit the remainder elements. We make this clear in the following proposition:

Proposition 4.1. *Let $f \colon \mathcal{X}^* \to \mathcal{Y}^*$ be any map. The following statements are equivalent: (i) f is prefix monotone (ii) $f(\langle \boldsymbol{x} \rangle) \subseteq \langle f(\boldsymbol{x}) \rangle$ for all $\boldsymbol{x} \in \mathcal{X}^*$ (iii) $\mathcal{P}(f(\boldsymbol{x})) = \mathcal{C}(f(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \mathcal{X}^*$ (iv) f is prefix-continuous. The proof is given in App. F.*

Proposition 4.1 shows how our work generalizes and extends Vieira et al. (2025a); our framework encompasses the strict-prefix monotone case, as well as enabling more expressive transformations. These are practical results. An empty (or finite) remainder and a finite quotient set mean that we get a finite-time computable interface to the transduced language model using the decomposition in Eq. (8c) and the interface described in §2. What remains is to consider when the remainder is finite.

We saw above that prefix monotonicity implies an empty remainder and, in App. E, that it implies a bounded quotient. This ensures that $\overrightarrow{p_{\mathcal{Y}}}(\boldsymbol{y})$ (Eq. (8c)) can be computed in finite time for $\boldsymbol{y} \in \mathcal{Y}^*$. Lemma 4.1 considers functions that need not be prefix monotone, yet guarantee a finite remainder and quotient. It relies on the notion of a *finite closure*. We say that a transducer $\mathtt{f}$ has **finite closure** at s if the relation defined by force-starting at the state is finite, i.e., if $|[\![\mathtt{f}_{[s]}]\!]| < \infty$.

Lemma 4.1. *Let $\mathtt{f}$ be a finite-state transducer encoding $f : \mathcal{X}^* \to \mathcal{Y}^*$, and $\boldsymbol{y} \in \mathcal{Y}^*$. If (i) there is a finite number of paths in $\mathtt{f}$ that emit $\boldsymbol{y}$ and (ii) every state is either universal or has finite closure, then $\mathcal{Q}(\boldsymbol{y})$ and $\mathcal{R}(\boldsymbol{y})$ are of finite size for all $\boldsymbol{y} \in \mathcal{Y}^*$. The proof is given in App. F.*

Next, we consider efficient algorithms for prefix decompositions and probability approximations.

5 Algorithms

We now present algorithms for calculating precover decompositions by taking advantage of the explicit structure of a transducer that encodes the function. Combined with Eq. (8c), these enable an interface to transduced language models. We then propose a pruning-based beam-search approximation.

To illustrate the general approach, we first give an abstract algorithm (on the left in Fig. 3) for decomposing the precover. The algorithm explicitly enumerates $\boldsymbol{x} \in \mathcal{X}^*$, starting from the empty string and proceeding from shortest to longest. Only when a quotient element is reached, or a conflict with the target sequence occurs, does the algorithm stop enumerating extensions of the element. Each prefix $\boldsymbol{x}$ undergoes three sequential checks following the definition of the precover:

Continuity If $\langle \boldsymbol{x} \rangle \subseteq \mathcal{P}(\boldsymbol{y})$, then $\boldsymbol{x}$ is attributed to the quotient $\mathcal{Q}(\boldsymbol{y})$ and we do not extend it.

Discontinuity If $\boldsymbol{x} \notin \mathcal{Q}(\boldsymbol{y})$ but $\boldsymbol{x} \in \mathcal{P}(\boldsymbol{y})$, we place it in the remainder $\mathcal{R}(\boldsymbol{y})$ and continue to extend.

Candidacy If $\boldsymbol{x} \notin \mathcal{Q}(\boldsymbol{y})$, we consider which single-symbol extensions are on track to reach the precover. Only extensions that might cover $\boldsymbol{y}$ are retained.

Since strings are processed from shortest to longest, each element has no prefix already in the quotient—since all prefixes of the current element were already considered, and if found to be in the quotient, not extended. Once the queue is exhausted, the algorithm returns the prefix decomposition.

5.1 Transducer-Based Algorithm with Determinism and Projection

The high-level algorithm in §5 relies on the three checks to compute the optimal decomposition. In practice, we represent the transformation f with a transducer $\mathtt{f}$ (see §2 and App. C for details on the notation and definitions used in this section). We now describe a transducer-based implementation.

Continuity. We wish to assess whether $\boldsymbol{x} \in \mathcal{Q}(\boldsymbol{y})$ holds when $\boldsymbol{y} \preceq \mathtt{f}(\boldsymbol{x})$. To do so, we test whether any state reached after scanning $\boldsymbol{x}$ is **universal**—that is, whether the transducer, when force started at that state, accepts all strings in $\mathcal{X}^*$. More details on the universality check are given in App. G.1.

Discontinuity. In case **Continuity** does not pass, $\boldsymbol{x} \notin \mathcal{Q}(\boldsymbol{y})$, we consider whether $\boldsymbol{x} \in \mathcal{R}(\boldsymbol{y})$. It suffices to check whether the state reached after scanning $\boldsymbol{x}$ is final, i.e., if $\mathtt{f}$ accepts $\boldsymbol{x}$ and $\boldsymbol{y} \preceq \mathtt{f}(\boldsymbol{x})$.

Candidacy. We compose $\mathtt{f}$ with a $\boldsymbol{y}\mathcal{Y}$ copy-transducer and project the composition onto the input side, obtaining $\mathrm{proj}_{\mathcal{X}}(\mathtt{f} \circ \boldsymbol{y}\mathcal{Y}^*)$.[8] This machine, denoted by $\mathtt{P}$, accepts exactly the precover of $\boldsymbol{y}$ w.r.t. f and discards all strings that cannot cover the target $\boldsymbol{y}$, allowing us to omit the explicit check whether $\boldsymbol{x}$ is a candidate for covering $\boldsymbol{y}$, i.e., whether $\boldsymbol{x}$ is a prefix of some element in $\mathcal{P}(\boldsymbol{y})$.[9]

[8] A copy-transducer is one where every transition emits the same symbol as it scans, see §2 and App. C for details on the input projection and the construction of copy transducers.

[9] This can also be expressed as $\mathcal{P}(\boldsymbol{y}) = f^{-1}(\boldsymbol{y}\mathcal{Y}^*) = [\![\mathtt{f} \circ \boldsymbol{y}\mathcal{Y}^*]\!]$.

```
1  def abstract_decomposition(y):
2    Q ← QUEUE()
3    Q.push(ε)
4    (R, Q) ← (∅, ∅)
5    while |Q| > 0:
6      x ← Q.pop()
7      if ⟨x⟩ ⊆ P(y):
8        Q.add(x)
9        continue
10     if x ∈ P(y):
11       R.add(x)
12     for x′ ∈ X:
13       if ∃x″ ∈ X*: xx′x″ ∈ P(y):
14         Q.push(xx′)
15   return (R, Q)
16 def is_universal(f, s):
17   return (⟦f_[s]⟧ = X*)
```

```
18 def determinized_decomposition(y):
19   # Determ. FST encoding ⟦P⟧ = P(y)
20   P ← trim(determinize(proj_X(f ∘ yY*)))
21   (_, _, _, T, I, F) ← P
22   Q ← QUEUE()
23   for s ∈ I: Q.push((s, ε))
24   (R, Q) ← (∅, ∅)
25   while |Q| > 0:
26     (s, x) ← Q.pop()
27     if is_universal(P, s):
28       Q.add(x)
29       continue
30     if s ∈ F:
31       R.add(x)
32     for (_ --x′--> s′) ∈ T(s):
33       Q.push((s′, x x′))
34   return (R, Q)
```

Figure 3: (Left) An abstract algorithm for deriving the decomposition of the precover. (Right) A transducer-based implementation.

Putting It All Together. Fig. 3 (right) presents the algorithm. Starting at the initial state, the paths in P are enumerated. Every accepted path P contributes to the precover. At a universal state, the scanned input $\boldsymbol{x}$ is added to the quotient $\mathcal{Q}(\boldsymbol{y})$ and not expanded further. If it enters a nonuniversal accepting state, we add $\boldsymbol{x}$ to the remainder $\mathcal{R}(\boldsymbol{y})$ and continue to extend. The quotient is prefix-free since we detect if the state is universal as early as possible and **continue** on lines 9 and 29. The checks on lines 10 and 31 ensure precover membership, and that $\boldsymbol{x}$ is added to the remainder if not in the quotient. Since the remainder is defined by set difference (Eq. (7)), the algorithm returns a valid remainder.

5.2 Optimizations for Scalability

Lazy determinization. In practice, explicit determinization is often computationally expensive for large transducers. In App. G.2 (Fig. 9), we present an alternative algorithm that lazily determinizes on the fly by maintaining *power states*—the set of all possible states reachable after scanning a prefix. A string that covers the target sequence is added to the quotient exactly when the power state is universal. When it is not, it is added to the remainder if any single states in the power state are final.

Memoization and precomputation. In App. G.3, we propose two additional optimizations: a memoized recursion that derives the precover of a single symbol extension, $\boldsymbol{yy}$, starting from a precomputed decomposition for the prefix $\boldsymbol{y}$. We also employ a direct enumeration of the quotient and remainder using the transducer f, bypassing the often costly composition with the copy transducer. This allows for efficient precomputation of universal states and caching of transducer properties.

Approximation via probability mass pruning. When the precover decomposition grows large, it becomes time-consuming to enumerate and expensive to score. In these cases, we rely on a probability mass pruning strategy that sorts candidates based on prefix probability and removes those whose cumulative probability mass falls below a specified threshold τ. The strategy is described in App. G.4.

6 Experiments

We now consider three examples of transduced language models. For each use case, we follow the approach in Vieira et al. (2025a) and measure the Jensen–Shannon divergence (JSD) between the distributions obtained using the approximation via probability mass pruning mentioned in §5.2 and a reference distribution we get by choosing a low pruning threshold τ. We also report cross-entropy loss in App. L.3, indicative of the cost of getting probabilities of specific sequences as opposed to getting the full distribution. We conduct experiments using GPT-2 Large (p_{gpt2}) (Radford et al., 2019), LlaMA 3.2-1B (p_{llama1B}) and LlaMA 3.1-8B (p_{llama8B}) (Llama Team, 2024),

and a DNA-model that we train over the human genome (p_{dna}).[10] For the experiments in §6, we use GenLM.bytes[11] to convert token-level models into byte-level models. See App. K for details on the training and evaluation setup, and App. J for details on the transducers we consider.

Recall that a transduced language model $p_{\mathcal{Y}}$ is characterized by a source model $p_{\mathcal{X}}$ and a transducer $\mathtt{f}$ encoding some function f (§3). To make this dependency explicit, we also write $p_{\mathcal{X}} \circ \mathtt{f} \stackrel{\text{def}}{=} p_{\mathcal{Y}}$ and refer to the act as composing. All experiments are done using the optimizations described in §5.2 (outlined in Fig. 10), and the algorithms computing the language model interface given in Fig. 13.

From Tokens to Characters. We now revisit the algorithm for converting models over tokens to models over characters, as presented by Vieira et al. (2025a). This algorithm can be realized by a simple transducer $\mathtt{f}_\alpha$ which consists of $\mathcal{O}(|\mathcal{X}|)$ states. Specifically, the transducer contains a chain for each token present in the input vocabulary $\mathcal{X}$. An example of such a machine is given in Fig. 4.

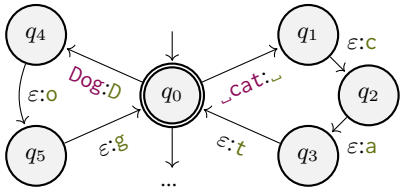


Figure 4: An FST for converting a token model into a character model. Paths for ␣cat, and Dog.

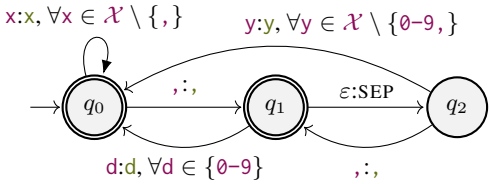


Figure 5: An FST that inserts a separator (SEP) after commas followed by a non-digit character.

We benchmark the algorithms in §5 using the transducers $p_{\text{gpt2}} \circ \mathtt{f}_\alpha$, $p_{\text{llama1B}} \circ \mathtt{f}_\alpha$, and $p_{\text{llama8B}} \circ \mathtt{f}_\alpha$ on the first ten paragraphs of the `wikitext-103-v1` dataset (Merity et al., 2017) (corresponding to the first 7684 bytes). As shown in Fig. 6 (left) and Tab. 7 in App. L, lower pruning thresholds (τ) give lower JSD values at the cost of throughput, measured in bytes per second[12]. We confirm that the JSD values are similar to those achieved by Vieira et al. (2025a) in App. L, Tab. 6.

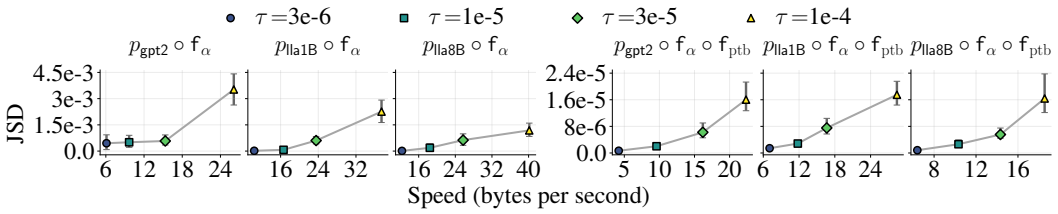


Figure 6: Average Jensen–Shannon distance (JSD) and throughput (bytes/s) across thresholds τ (relative to $\tau = 10^{-6}$). Left: $p_{\mathcal{X}} \circ \mathtt{f}_\alpha$. Right: $p_{\mathcal{X}} \circ \mathtt{f}_\alpha \circ \mathtt{f}_{\text{ptb}}$. Bars show 95% bootstrapped CIs.

From Tokens to Orthographic Word Boundaries. The next use case we consider is the conversion of language models over subwords to language models over *orthographic* words. The precise definition of what constitutes such a word varies depending on the application. In some settings, contractions such as "`wouldn't`" should be treated as a single unit. While in other settings, a more natural segmentation would split it in two: "`would`" and "`n't`". The proposed approach can accommodate any definition that can be described with a transformation in the form of a transducer.

Linguistic tokenizers, such as the PTB tokenizer (Marcus et al., 1993), use contextual information to segment raw English text into linguistically meaningful units, suitable for downstream NLP tasks.We construct an FST encoding the PTB tokenizer, $\mathtt{f}_{\text{ptb}}$, details are in App. I. An example of a rule in the transducer is given in Fig. 5, which inserts SEP after a comma if a digit does not follow.

[10] Links to models: `https://huggingface.co/openai-community/gpt2-large`, `https://huggingface.co/meta-llama/Llama-3.2-1B`, `https://huggingface.co/meta-llama/Meta-Llama-3-8B`.

[11] `https://github.com/genlm/genlm-bytes`

[12] For higher thresholds $\tau >$1e-4 (Tab. 7), throughput is less consistent.When this occurs, we relax the threshold and recompute the decomposition until a valid path is found (see App. G.4).

Composing $p_{\mathcal{X}} \circ \mathtt{f}_{\alpha} \circ \mathtt{f}_{\mathrm{ptb}}$ yields a transduced language model, which maps subword tokens to PTB tokens. To reduce the state count of the composed FST and improve efficiency, we represent $p_{\mathcal{X}} \circ \mathtt{f}_{\alpha}$ using the byte-transformed models from GenLM.bytes[13]. In Fig. 6 (right), we plot the average JSD against the throughput for different thresholds τ, using the dataset from §6. As with the experiments in §6, we observe lower JSD at lower thresholds, albeit at the cost of throughput. For experiments using higher thresholds, see App. L, Tab. 8.

Converting DNA Models to Models Over Amino Acids. To model the translation from DNA to proteins, we use a transducer that converts sequences of the four DNA nucleotides to sequences over the twenty-two amino acids. That is we have $\mathcal{X} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and $\mathcal{Y} = \{\mathtt{A}, \mathtt{R}, \mathtt{N}, \mathtt{D}, \mathtt{C}, \mathtt{Q}, \mathtt{E}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{L}, \mathtt{K}, \mathtt{M}, \mathtt{F}, \mathtt{P}, \mathtt{S}, \mathtt{T}, \mathtt{W}, \mathtt{Y}, \mathtt{B}, \mathtt{Z}, \star\}$. The transducer $\mathtt{f}_{\mathrm{dna2aa}}$ (partially shown in App. B) maps each *codon*-sequence corresponding to three nucleotides to a corresponding amino acid, with multiple codons often encoding the same amino acid. Let $p_{\mathsf{dna}} \circ \mathtt{f}_{\mathrm{dna2aa}}$ denote the transduced model that converts DNA nucleotides into amino acids. To evaluate our approach, we sample 65 human proteins[14].In Tab. 9, we show the average JSD and throughput for different thresholds τ. Note that this transducer is particularly challenging since the set of candidates in the decomposed precover grows exponentially with the sequence length. To mitigate the combinatorial blow-up, we cap the candidate-set size and report throughput and JSD while varying the cap.

Benchmarking the Quotient. We benchmark the value of the computational shortcut inherent in the prefix decomposition described in §4.1, which allows us to decompose the precover into remainder and quotient representatives. We generate suboptimal decompositions by randomly selecting a proportion of the universal states in the transducer (see §4) and treat them as non-universal states in our algorithms (see §5). This gradually decreases the size of the quotient and increases the remainder correspondingly. Tab. 1 shows the average JSD between the original and modified distributions over the first 256 bytes of the `wikitext-103-v1` dataset (Merity et al., 2017). For each value of n we repeat the sampling of new non-universal states five times, and report the mean JSD. The distributions diverge quickly, and after converting approximately 20-25% of universal states, the algorithm keeps running into dead ends and can no longer recover any valid sequence.

Table 1: Average Jensen–Shannon distance (JSD) for τ =1e-4 after converting n% of the universal states to non-universal. See Tab. 10 for results with p_{gpt2} and Tab. 2 for the number of states.

%	$p_{\mathsf{llama1B}} \circ \mathtt{f}_{\alpha}$		$p_{\mathsf{llama1B}} \circ \mathtt{f}_{\alpha} \circ \mathtt{f}_{\mathrm{ptb}}$	
Conv.	States	avg. JSD / byte	States	avg. JSD / byte
0	0	(not applicable)	0	(not applicable)
5	8849	1.8e-2 ± 4.0e-3	1	9.9e-7 ± 3.2e-7
10	17699	1.4e-2 ± 3.0e-3	3	4.0e-5 ± 2.7e-5
15	26548	1.6e-2 ± 3.0e-3	4	1.1e-5 ± 9.1e-6
20	35398	5.1e-2 ± 6.0e-3	6	1.2e-3 ± 6.8e-4
25	44247	6.4e-2 ± 7.0e-3	7	1.1e-2 ± 1.5e-3

Training on The Target Domain. There are many reasons why one might prefer transducing a language model instead of training a new one. For instance, the inductive bias of shared semantic subwords may result in a better model, as the popularity of BPE demonstrates. We confirm this by training byte-level models and comparing them to a transduced model in App. L.4. Training a model may also be unfeasible or impractical for, e.g., prototyping. Finally, we may simply want to know how a specific model behaves under a transformation or different units, e.g., for comparison between models.

7 Conclusion

We have introduced a foundational framework for transforming language models into new language models using transducers. In particular, this allows one to convert language models defined over one set of units into models over another using transducers. Empirically, we have shown that our beam-search approximation efficiently transduces token-based LLMs into models over characters, words, and even amino acids, without requiring retraining. Our theoretical analysis characterizes the conditions under which such mappings can be performed exactly. The proposed approach is an effective way to repurpose existing language models and accurately compute probabilities for any kind of unit and transformation defined via a transducer. We discuss the limitations of our approach in App. D.

[13] `https://github.com/genlm/genlm-bytes` implementing (Vieira et al., 2025a).

[14] Sampled from `https://www.uniprot.org/uniprotkb?query=Human`, see Tab. 3 for the accession numbers.

REFERENCES

Cyril Allauzen, Bill Byrne, Adrià de Gispert, Gonzalo Iglesias, and Michael Riley. Pushdown automata in statistical machine translation. *Computational Linguistics*, 40(3), September 2014. doi: 10.1162/COLI_a_00197. URL https://aclanthology.org/J14-3008/.

Lisa Beinborn and Yuval Pinter. Analyzing Cognitive Plausibility of Subword Tokenization. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://aclanthology.org/2023.emnlp-main.272/.

Martin Berglund and Brink van der Merwe. Formalizing BPE Tokenization. In *Proceedings of the International Workshop on Non-Classical Models of Automata and Applications*, volume 388, 2023. URL https://cgi.cse.unsw.edu.au/~eptcs/paper.cgi?NCMA2023.4.pdf.

Martin Berglund, Willeke Martens, and Brink van der Merwe. Constructing a BPE Tokenization DFA. In *Implementation and Application of Automata*, 2024. ISBN 978-3-031-71112-1. URL https://link.springer.com/chapter/10.1007/978-3-031-71112-1_5.

Jean Berstel. *Transductions and Context-Free Languages*, volume 38. Teubner, Stuttgart, Germany, 1979. URL https://www.worldcat.org/oclc/06364613.

Harry E. Blanchard, Alexander Pollatsek, and Keith Rayner. The acquisition of parafoveal word information in reading. *Perception & Psychophysics*, 46(1), 1989. ISSN 0031-5117. URL https://link.springer.com/content/pdf/10.3758/BF03208078.pdf.

Filippo Bonchi and Damien Pous. Hacking nondeterminism with induction and coinduction. *Communications of the ACM*, 2015. URL https://doi.org/10.1145/2713167.

Christian Choffrut. Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3), 1977. ISSN 0304-3975. URL https://doi.org/10.1016/0304-3975(77)90049-4.

Marco Cognetta, David Pohl, Junyoung Lee, and Naoaki Okazaki. Pitfalls, Subtleties, and Techniques in Automata-Based Subword-Level Constrained Generation. In *Tokenization Workshop*, Vancouver, Canada, 2025. URL https://openreview.net/forum?id=DFybOGeGDS. To appear.

Ryan Cotterell, Anej Svete, Clara Meister, Tianyu Liu, and Li Du. Formal aspects of language modeling, 2024. URL https://arxiv.org/abs/2311.04329.

M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: a New Algorithm for Checking Universality of Finite Automata. In *Proceedings of the International Conference on Computer Aided Verification*, Berlin, Heidelberg, 2006. URL https://doi.org/10.1007/11817963_5.

Edo Dotan, Gal Jaschek, Tal Pupko, and Yonatan Belinkov. Effect of tokenization on transformers for biological sequences. *Bioinformatics*, 40(4), 2024. ISSN 1367-4811. URL https://doi.org/10.1093/bioinformatics/btae196.

Philip Gage. A new algorithm for data compression. *The C Users Journal archive*, 12, 1994. URL https://api.semanticscholar.org/CorpusID:59804030.

Renato Geh, Honghua Zhang, Kareem Ahmed, Benjie Wang, and Guy Van Den Broeck. Where is the signal in tokenization space? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Miami, Florida, USA, November 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.emnlp-main.230/.

Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. Generating topical poetry. In Jian Su, Kevin Duh, and Xavier Carreras (eds.), *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1126. URL https://aclanthology.org/D16-1126/.

Mario Giulianelli, Luca Malagutti, Juan Luis Gastaldi, Brian DuSell, Tim Vieira, and Ryan Cotterell. On the proper treatment of tokenization in psycholinguistics. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://aclanthology.org/2024.emnlp-main.1032.

Kyle Gorman. Pynini: A Python library for weighted finite-state grammar compilation. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, 2016. URL https://aclanthology.org/W16-2409/.

Jonathan Hayase, Alisa Liu, Noah A. Smith, and Sewoong Oh. Sampling from your language model one byte at a time. In *Tokenization Workshop*, 2025. URL https://openreview.net/forum?id=DM8D9Nq9uO.

Yanrong Ji, Zhihan Zhou, Han Liu, and Ramana V Davuluri. DNABERT: Pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome. *Bioinformatics*, 37(15), 2021. ISSN 1367-4803. URL https://doi.org/10.1093/bioinformatics/btab083.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BDBdblmyzY.

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018. URL https://aclanthology.org/P18-1007/.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023. URL https://doi.org/10.1145/3600006.3613165.

Alexander K. Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash Mansinghka. Sequential monte carlo steering of large language models using probabilistic programs. In *ICML 2023 Workshop: Sampling and Optimization in Discrete Space*, 2023. URL https://openreview.net/forum?id=Ul2K0qXxXy.

(Meta) Llama Team. The Llama 3 Herd of Models, 2024. URL https://arxiv.org/abs/2407.21783.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 1993. URL https://aclanthology.org/J93-2004/.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the Annual Symposium on Switching and Automata Theory*, 1972. URL https://doi.org/10.1109/SWAT.1972.29.

Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2), 1997. URL https://aclanthology.org/J97-2003/.

Sathvik Nair and Philip Resnik. Words, Subwords, and Morphemes: What Really Matters in the Surprisal-Reading Time Relationship? In *Findings of the Association for Computational Linguistics: EMNLP*, 2023. URL https://aclanthology.org/2023.findings-emnlp.752/.

Eric Nguyen, Michael Poli, Marjan Faizi, Armin W Thomas, Michael Wornow, Callum Birch-Sykes, Stefano Massaroli, Aman Patel, Clayton M. Rabideau, Yoshua Bengio, Stefano Ermon, Christopher Re, and Stephen Baccus. HyenaDNA: Long-Range Genomic Sequence Modeling at Single Nucleotide Resolution. In *Proceedings of the Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=ubzNoJjOKj.

Byung-Doh Oh and William Schuler. Leading Whitespaces of Language Models' Subword Vocabulary Pose a Confound for Calculating Word Probabilities. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://aclanthology.org/2024.emnlp-main.202/.

OpenAI. Tiktoken: A fast BPE tokeniser for OpenAI's models, 2024. URL https://github.com/openai/tiktoken.

George Orwell. *Nineteen Eighty-Four*. Secker & Warburg, 1949. URL https://en.wikipedia.org/wiki/Nineteen_Eighty-Four.

Buu Phan, Marton Havasi, Matthew Muckley, and Karen Ullrich. Understanding and mitigating tokenization bias in language models, 2024. URL https://arxiv.org/abs/2406.16829.

Tiago Pimentel and Clara Meister. How to compute the probability of a word. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://aclanthology.org/2024.emnlp-main.1020/.

Jean-Éric Pin. *Mathematical Foundations of Automata Theory*. Pin, Jean-Éric, 2025. URL https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf.

Jean-Éric Pin. *Handbook of Automata Theory*. European Mathematical Society Publishing House, Zürich, Switzerland, 2021. ISBN 978-3-98547-006-8. URL https://doi.org/10.4171/Automata.

Lifeng Qiao, Peng Ye, Yuchen Ren, Weiqiang Bai, chaoqi liang, Xinzhu Ma, Nanqing Dong, and Wanli Ouyang. Model Decides How to Tokenize: Adaptive DNA Sequence Tokenization with MxDNA. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=AQ1umQL7dZ.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8), 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Keith Rayner, Arnold D. Well, Alexander Pollatsek, and James H. Bertera. The availability of useful information to the right of fixation in reading. *Perception & Psychophysics*, 31(6), 1982. URL https://doi.org/10.3758/BF03204186.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2016. URL https://aclanthology.org/P16-1162/.

Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast WordPiece Tokenization. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://aclanthology.org/2021.emnlp-main.160/.

Felix Stahlberg, Christopher Bryant, and Bill Byrne. Neural grammatical error correction with finite state transducers. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *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)*, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1406. URL https://aclanthology.org/N19-1406/.

Tim Vieira, Ben LeBrun, Mario Giulianelli, Juan Luis Gastaldi, Brian DuSell, John Terilla, Timothy J. O'Donnell, and Ryan Cotterell. From language models over tokens to language models over characters. In *Proceedings of the International Conference on Machine Learning*, 2025a. URL https://arxiv.org/abs/2412.03719.

Tim Vieira, Tianyu Liu, Clemente Pasti, Yahya Emara, Brian DuSell, Benjamin LeBrun, Mario Giulianelli, Juan Luis Gastaldi, John Terilla, Timothy J. O'Donnell, and Ryan Cotterell. Language models over canonical byte-pair encodings. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=eCVrfVDNSY.

Ning Wang, Jiang Bian, Yuchen Li, Xuhong Li, Shahid Mumtaz, Linghe Kong, and Haoyi Xiong. Multi-purpose RNA language modelling with motif-aware pretraining and type-guided fine-tuning. *Nat. Mac. Intell.*, 6(5), 2024. URL https://doi.org/10.1038/s42256-024-00836-4.

Brandon T. Willard and Rémi Louf. Efficient Guided Generation for Large Language Models, 2023. URL https://arxiv.org/abs/2307.09702.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020. URL https://aclanthology.org/2020.emnlp-demos.6.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016. URL https://arxiv.org/abs/1609.08144.

Vicky Xefteri, Tim Vieira, Ryan Cotterell, and Afra Amini. Syntactic control of language models by posterior inference. In *Findings of the Association for Computational Linguistics: ACL*, 2025. URL https://arxiv.org/abs/2506.07154.

A Notation Glossary

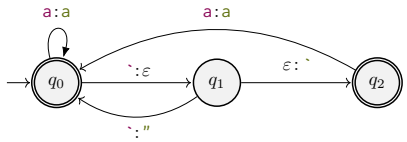
Notation	Gloss
ε	Empty string
EOS	End-of-string symbol
$x, x' \in \mathcal{X}$	Symbols in the source alphabet $\mathcal{X}$
$\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}^*$	Source strings
$\mathcal{X}^*$	Set of all source strings
$Z, Z' \subseteq \mathcal{X}^*$	Sets of source strings
$\boldsymbol{x}\boldsymbol{x}'$	Concatenation (strings)
ZZ'	Concatenation (sets of strings)
$\boldsymbol{y}, \boldsymbol{y}' \in \mathcal{Y}^*$	Target strings
$y, y' \in \mathcal{Y}$	Symbols in the target alphabet
$\mathcal{Y}^*$	Set of all target strings
$f\colon \mathcal{X}^* \to \mathcal{Y}^*$	Transformation from source strings $\mathcal{X}^*$ to target strings $\mathcal{Y}^*$
$\boldsymbol{x} \preceq \boldsymbol{x}'$	$\boldsymbol{x}$ is a prefix of $\boldsymbol{x}'$
$\boldsymbol{x} \prec \boldsymbol{x}'$	$\boldsymbol{x}$ is a strict prefix of $\boldsymbol{x}'$
$\langle Z \rangle$	Cylinder set spanned by Z
$\mathrm{pf}(Z)$	Prefix-base of Z
$p_{\mathcal{X}}$	Language model over source strings $\mathcal{X}^*$
X	$\mathcal{X}^*$-valued random variable $X \sim p_{\mathcal{X}}$
$\overrightarrow{p_{\mathcal{X}}}$	Prefix probability of $p_{\mathcal{X}}$
$p_{\mathcal{Y}}$	Language model over target strings $\mathcal{Y}^*$
$f(X)$	$\mathcal{Y}^*$-valued random variable $f(X) \sim p_{\mathcal{Y}}$
$\overrightarrow{p_{\mathcal{Y}}}$	Prefix probability of $p_{\mathcal{Y}}$
f	Transducer implementation of f
$\hookrightarrow \mathsf{S}$	Set of states
$\hookrightarrow \mathcal{X}$	Input alphabet
$\hookrightarrow \mathcal{Y}$	Output alphabet
$\hookrightarrow \mathsf{T}$	Set of transitions
$\hookrightarrow \mathsf{I} \subseteq \mathsf{S}$	Set of initial states
$\hookrightarrow \mathsf{F} \subseteq \mathsf{S}$	Set of final states
$\hookrightarrow \mathsf{U} \subseteq \mathsf{S}$	Set of universal states
$s, s' \in \mathsf{S}$	States
$(s \xrightarrow{x:y} s') \in \mathsf{T}$	Transition from s to s' that scans x and emits y
$\mathsf{T}(s) \subseteq \mathsf{T}$	Outgoing transitions from state s
Π	Set of all paths
$\boldsymbol{\pi} \in \Pi$	Path
$(s \overset{\boldsymbol{x}:\boldsymbol{y}}{\rightsquigarrow} s') \in \Pi$	Path from s to s' that scans $\boldsymbol{x}$ and emits $\boldsymbol{y}$
$\mathsf{f} \circ \mathsf{f}'$	Transducer composition
$f \circ f'$	Relation composition
$\mathsf{proj}_{\mathcal{X}}(\mathsf{f})$	Input projection
$\mathsf{f}_{[s]}$	Force-start f in state s
$\boldsymbol{y}\mathcal{Y}^*$	Either a copy-transducer that accepts $\boldsymbol{y}\mathcal{Y}^*$ or the corresponding relation.
$\mathsf{f} \circ \boldsymbol{y}\mathcal{Y}^*$	A transducer whose paths are restricted to those that accept $\boldsymbol{y}\mathcal{Y}^*$.
$f^{-1}(\boldsymbol{y})$	Preimage of the target string $\boldsymbol{y}$ (§3)
$f^{-1}(\mathcal{Y})$	The preimage of a set $\mathcal{Y}$, $\{f^{-1}(\boldsymbol{y}) \mid \boldsymbol{y} \in \mathcal{Y}\}$ (§3)
$\mathcal{P}(\boldsymbol{y})$	Precover of the target string $\boldsymbol{y}$ (§3)
$\mathcal{C}(\boldsymbol{y})$	The largest cylinder set contained in $\mathcal{P}(\boldsymbol{y})$ (§4)
$\mathcal{Q}(\boldsymbol{y})$	Quotient (§4)
$\mathcal{R}(\boldsymbol{y})$	Remainder (§4)

A.1 Example Decompositions

We give two concrete examples of decompositions and prefix probabilities.

Example 1. *The transducer below merges consecutive backticks (`` ` ` ``) into a quotation mark (`"`) but otherwise keeps the symbols* a *and* `` ` `` *fixed. It can be seen as a minimal example of text normalization.*

$$\mathsf{f} = \left[\quad \right] \tag{10a}$$



When computing the prefix probability $\overrightarrow{p_{\mathcal{Y}}}(\texttt{`})$*, sequences starting with two backticks get mapped to a quote and thus do not appear in the precover. Instead, a single backtick remains in the remainder, while a backtick followed by* a *goes into the quotient, as any further extensions preserve the initial backtick. We thus have (by visually canceling out continuations that get mapped to quotation marks),*

$$\mathcal{P}(\texttt{`}) = \{\texttt{`}, \cancel{\texttt{``}}, \cancel{\texttt{```}}, \cancel{\texttt{``a}}, \ldots\} \cup \{\texttt{`a}, \texttt{`a`}, \texttt{`aa}, \ldots\} \tag{10b}$$

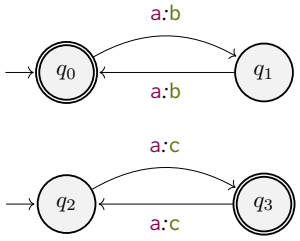
$$= \{\texttt{`}\} \sqcup \{\texttt{`a}\}\,\mathcal{X}^* \tag{10c}$$

Thus, $\overrightarrow{p_{\mathcal{Y}}}(\texttt{`}) = p_{\mathcal{X}}(\texttt{`}) + \overrightarrow{p_{\mathcal{X}}}(\texttt{`a})$*. Similarly, we have* $\mathcal{P}(\texttt{"}) = \{\texttt{`}\}\mathcal{X}^*$ *and thus* $\overrightarrow{p_{\mathcal{Y}}}(\texttt{"}) = \overrightarrow{p_{\mathcal{Y}}}(\texttt{` `})$*.*

Example 1 demonstrates the efficiency of using the remainder and the quotient; both sets only have a single element, yet they fully specify what probabilities contribute to $\mathcal{P}(\texttt{`})$. There are, however, edge cases that do not lend themselves to such efficient cover functions.

Example 2. *Consider the following mapping* f *and its representation as a transducer* f*:*

$$f(\mathsf{a}^n) = \mathsf{b}^n \textbf{ if } n \textit{ is even } \textbf{else } \mathsf{c}^n \qquad \mathsf{f} = \left[\quad \right] \tag{11a}$$



with $\mathcal{X} = \{\mathsf{a}\}$ *and* $\mathcal{Y} = \{\mathsf{b}, \mathsf{c}\}$*. The precover for* b *is given by*

$$\mathcal{P}(\mathsf{b}) = \{\cancel{\varepsilon}, \cancel{\mathsf{a}}, \mathsf{aa}, \cancel{\mathsf{aaa}}, \mathsf{aaaa}, \cancel{\mathsf{aaaaa}}, \mathsf{aaaaaa}, \ldots\} \tag{11b}$$

This example is, unfortunately, challenging for our approach as the remainder set is not finite:

$$\mathcal{R}(\mathsf{b}) = \left\{\mathsf{a}^{2^n} \mid n > 0\right\}, \mathcal{Q}(\mathsf{b}) = \emptyset \tag{11c}$$

In other words, the mapping never releases the evenness constraint, and we sum forever.

Finally, we give an example of a simple transducer with a single remainder element in Example 3

Example 3. *Consider the following mapping* f*, visualized in Fig. 7.*

$$f(\mathsf{a}^n) = \mathsf{b}^n \textbf{ if } n \neq 2 \textbf{ else } \mathsf{c} \tag{12a}$$

$\mathcal{X} = \{\mathsf{a}\}$ *and* $\mathcal{Y} = \{\mathsf{b}, \mathsf{c}\}$*. Suppose we want to compute the prefix probability* $\overrightarrow{p_{\mathcal{Y}}}(\mathsf{b})$*.*

$$\mathcal{P}(\mathsf{b}) = \{\mathsf{a}, \cancel{\mathsf{aa}}, \mathsf{aaa}, \mathsf{aaaa}, \mathsf{aaaaa}, \ldots\} \tag{12b}$$

$$\mathcal{R}(\mathsf{b}) = \{\mathsf{a}\}, \mathcal{Q}(\mathsf{b}) = \{\mathsf{aaa}\} \tag{12c}$$

Thus,

$$\overrightarrow{p_{\mathcal{Y}}}(\mathsf{b}) = \sum_{\boldsymbol{x} \in \mathcal{R}(\mathsf{b})} p_{\mathcal{X}}(\boldsymbol{x}) + \sum_{\boldsymbol{x} \in \mathcal{Q}(\mathsf{b})} \overrightarrow{p_{\mathcal{X}}}(\boldsymbol{x}) \tag{12d}$$

$$= p_{\mathcal{X}}(\mathsf{a}) + \overrightarrow{p_{\mathcal{X}}}(\mathsf{aaa}) \tag{12e}$$

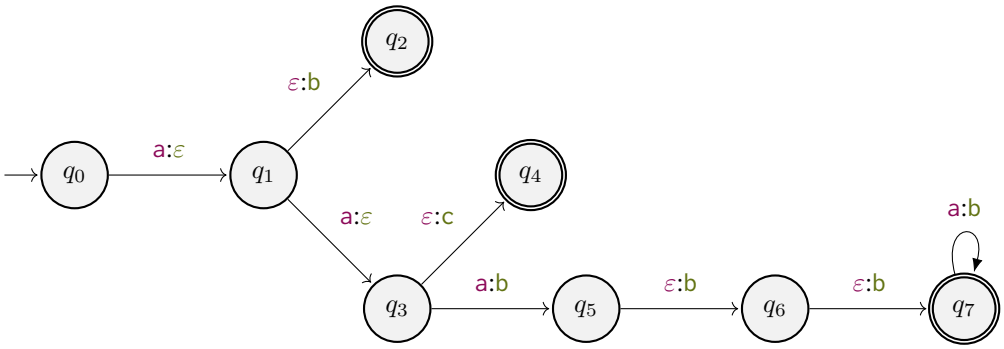


Figure 7: An FST that maps n occurrences of 'a' to the same number of 'b's, except when the input is exactly two 'a's, which are mapped to 'c'.

B Transducer Diagrams

DNA to amino acid transducer. The DNA to amino-acid transducer, described in §6 is partially shown in App. B.

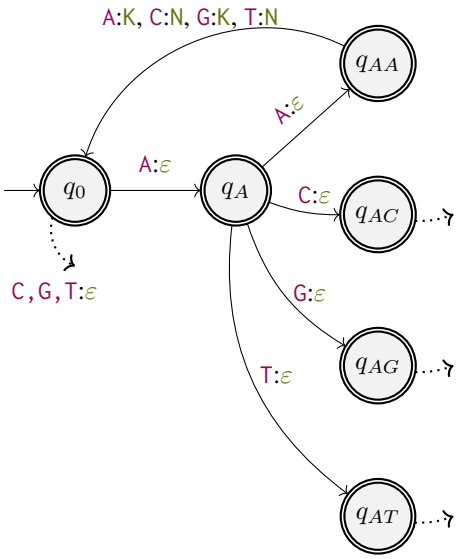


Figure 8: An FST for converting DNA sequences to amino acids. Each triplet of nucleobases maps to one of 20 different amino acids. We only show a proportion of the machine.

C Background on Transducers

Transducer variants. We say a transducer is **functional** if it defines a function, and **partially functional** if it defines a partial function. A transducer is **input-deterministic** if for every state $s \in \mathsf{S}$, $|\mathsf{T}(s, \varepsilon)| = 0$ and $|\mathsf{T}(s, x)| \leq 1$ for all $x \in \mathcal{X}$. For input-deterministic transducers, each source string $\boldsymbol{x} \in \mathcal{X}^*$ has at most one accepting path that scans $\boldsymbol{x}$ and, therefore, can emit at most one target string $\boldsymbol{y} \in \mathcal{Y}^*$. Thus, every input-deterministic transducer defines a (partial) function. A **copy-transducer** is one where every transition emits the same symbol as it scans, i.e., every transition is of the form $s \xrightarrow{x:x} s'$. We abbreviate copy-transitions as $s \xrightarrow{x} s'$.[15] Copy-transducers define partial identity functions: they map each string in a designated subset to itself, and drop all others. In the same way that we abbreviate copy-transitions, we may implicitly map them to a set of strings via $(\boldsymbol{x}, \boldsymbol{x}) \mapsto \boldsymbol{x}$.

[15]For readers familiar with finite-state automata, every copy-transducer is isomorphic to an *acceptor*.

Operations. Transducers support **composition**: given transducers $\mathtt{f}$ and $\mathtt{f}'$, their composition $\mathtt{f} \circ \mathtt{f}'$ is a transducer denoting $[\![\mathtt{f} \circ \mathtt{f}']\!] \stackrel{\text{def}}{=} [\![\mathtt{f}]\!] \circ [\![\mathtt{f}']\!]$.[16][17] We denote a transducer that encodes the relationship $[\![\mathtt{f}]\!] \circ \{(\boldsymbol{y}', \boldsymbol{y}') \mid \boldsymbol{y}' \in \langle \boldsymbol{y} \rangle\}$ with $\mathtt{f} \circ \boldsymbol{y}\mathcal{Y}^*$. In general, we omit defining these type coercions explicitly when it should be clear from context what is meant. We define the **input projection** operation encoding the relationship $[\![\mathsf{proj}_{\mathcal{X}}(\mathtt{f})]\!] = \{(\boldsymbol{x}, \boldsymbol{x}) \mid (\boldsymbol{x}, \cdot) \subseteq [\![\mathtt{f}]\!]\}$ as $\mathsf{proj}_{\mathcal{X}}(\mathtt{f}) \stackrel{\text{def}}{=} (\mathsf{S}, \mathcal{X}, \mathcal{X}, \{(s \xrightarrow{x:x} s') \mid s \xrightarrow{x:y} s' \in \mathsf{T}\}, \mathsf{I}, \mathsf{F})$, which is a copy-transducer. Let $\mathtt{f}_{[s]}$ denote the operation of **force-starting** $\mathtt{f}$ in state s, $\mathtt{f}_{[s]} \stackrel{\text{def}}{=} (\mathsf{S}, \mathcal{X}, \mathcal{Y}, \mathsf{T}, \{s\}, \mathsf{F})$; this operation yields a machine defining the set of source–target suffix pairs that are generated by paths starting at a given s and ending in a final state. We say that a state s is **universal** if $[\![\mathsf{proj}_{\mathcal{X}}(\mathtt{f}_{[s]})]\!] = \mathcal{X}^*$. Let $\mathsf{U} \subseteq \mathsf{S}$ denote the set of universal states.

Our algorithms use an **input-determinization** transformation $\mathsf{determinize}(\mathtt{f})$ that maps a (partially) functional transducer $\mathtt{f}$ to an equivalent one that is input-deterministic. In general, such a mapping is not always realizable with a finite number of states.[18] However, in the special case of copy-transducers, input-determinization is always possible, but may result in exponential blowup in the worst case.[19] We use $\mathsf{trim}(\mathtt{f})$ to denote a **trimming** operation that removes all states and edges that do not appear on any accepting path. These are standard operations that are implemented in any FST library, more details can be found in (Pin, 2021; 2025).

Visual notation. We use diagrams like those shown in Fig. 2 to represent transducers. Transitions without source states denote initial states; double-lined states indicate final states. Transitions $s \xrightarrow{x:y} s'$ are shown as arrows between states.

Limitations. Finite-state transducers define the class of *rational relations* (Berstel, 1979, Ch. III). Because FSTs only have finitely many states, they are inherently limited in the relations they can represent. For example, FSTs cannot perform transformations that require unbounded matching or counting. In contrast, transducers with unbounded memory extend beyond the rational class, offering greater expressive power, but come with increased complexity and often undecidability of key properties, such as *universality* (see §5.1) and App. G.1.

D Limitations

Although our framework enables transduction of any language model to any unit of interest, given a valid transducer, we test only a limited combination of architectures (GPT-2 and Llama 3) and target specific units (alphabetization, Penn Treebank tokens, and amino acids). Future research could broaden the analysis to a wider range of models, datasets, and units. Our analysis has also focused on functional finite-state transducers and regular languages; future work could consider dynamically built transducers and distributions over more expressive languages. Future work could also consider stochastic maps, encoded by non-functional transducers—here, the notion of universality would need to be adjusted. Finally, the speed of our algorithms and implementations may not suit every use case; while getting the full distribution (for e.g. decoding or model comparisons) speeds of around 10-20 bytes per second are not prohibitive for many tasks, we will consider scaling this up in future work.

E Quotient Bound for Prefix Monotone Maps

The following proposition gives a bound for the size of the quotient when the map is prefix monotone.

[16] Here, **relation composition** is given by $f \circ g \stackrel{\text{def}}{=} \{(x, z) \mid (x, y) \in f, (y, z) \in g\}$, f and g are relations.

[17] Pin (2025, Ch. XIX, sec. 2) gives an efficient method for constructing $\mathtt{f} \circ \mathtt{f}'$.

[18] Choffrut (1977) gives such an algorithm, by using a power set construction on the input side as in the determinization of non-deterministic finite automata. He shows it yields a finite machine if and only if the automaton has *bounded variation*, the constraint that for any two strings whose prefix distance (the combined length of the strings with the longest shared prefix removed) is bounded, then the output prefix distance is also bounded. He also provides a testable condition for determinization, known as the twinning condition: once two runs have read the same input prefix (i.e., we cut the input at the same point), then for every common continuation, they append the same further output.

[19] For readers familiar with finite-state automata, input-determinization of a copy-transducer is isomorphic to the determinization of an equivalent finite-state automaton, see App. C.

Proposition E.1. *Let $f\colon \mathcal{X}^* \to \mathcal{Y}^*$ be a strict-prefix monotone map. Then, for every $\boldsymbol{y} \in \mathcal{Y}^*$:*

1. $f^{-1}(\boldsymbol{y}) = \mathcal{Q}(\boldsymbol{y}) \setminus \bigsqcup_{y\in\mathcal{Y}} \mathcal{Q}(\boldsymbol{y}y)$
2. $\bigsqcup_{y\in\mathcal{Y}} \mathcal{Q}(\boldsymbol{y}y) \subseteq \mathcal{Q}(\boldsymbol{y})\,(\mathcal{X} \sqcup \{\varepsilon\})$
3. $|\mathcal{Q}(\boldsymbol{y})| \leq (|\mathcal{X}|+1)^{|\boldsymbol{y}|}$

Proof. (1) Observe that (using Proposition 4.1) for every $\boldsymbol{y} \in \mathcal{Y}^*$

$$\mathcal{C}(\boldsymbol{y}) = \mathcal{P}(\boldsymbol{y}) = f^{-1}(\boldsymbol{y}) \sqcup \bigsqcup_{y\in\mathcal{Y}} \mathcal{P}(\boldsymbol{y}y) = f^{-1}(\boldsymbol{y}) \sqcup \bigsqcup_{y\in\mathcal{Y}} \mathcal{C}(\boldsymbol{y}y). \tag{13}$$

In particular, the minimality of $\mathcal{Q}(\boldsymbol{y})$ in $\mathcal{P}(\boldsymbol{y})$ implies that

$$\mathcal{Q}(\boldsymbol{y}) \cap \bigsqcup_{y\in\mathcal{Y}} \mathcal{Q}(\boldsymbol{y}y) = \mathcal{Q}(\boldsymbol{y}) \cap \bigsqcup_{y\in\mathcal{Y}} \langle \mathcal{Q}(\boldsymbol{y}y)\rangle.$$

Consequently, we obtain the identity $f^{-1}(\boldsymbol{y}) = \mathcal{C}(\boldsymbol{y}) \setminus \bigsqcup_{y\in\mathcal{Y}} \mathcal{C}(\boldsymbol{y}y) = \langle \mathcal{Q}(\boldsymbol{y})\rangle \setminus \bigsqcup_{y\in\mathcal{Y}} \langle \mathcal{Q}(\boldsymbol{y}y)\rangle$, which in turn yields the inclusion

$$\mathcal{Q}(\boldsymbol{y}) \setminus \bigsqcup_{y\in\mathcal{Y}} \mathcal{Q}(\boldsymbol{y}y) = \mathcal{Q}(\boldsymbol{y}) \setminus \bigsqcup_{y\in\mathcal{Y}} \langle \mathcal{Q}(\boldsymbol{y}y)\rangle \subseteq f^{-1}(\boldsymbol{y}).$$

For the reverse inclusion, let $\boldsymbol{x}' \in f^{-1}(\boldsymbol{y})$. Then there exists a unique $\boldsymbol{x}_q \in \mathcal{Q}(\boldsymbol{y})$ such that $\boldsymbol{x}_q \preceq \boldsymbol{x}'$, by definition of the quotient set. Since f is strict-prefix monotone and $f(\boldsymbol{x}_q) \preceq f(\boldsymbol{x}') = \boldsymbol{y}$, we must have $\boldsymbol{x}_q = \boldsymbol{x}'$. This shows that $f^{-1}(\boldsymbol{y}) \subseteq \mathcal{Q}(\boldsymbol{y})$, and completes the proof of (1).

(2) Fix $y \in \mathcal{Y}$, $\boldsymbol{y} \in \mathcal{Y}^*$ and let $\boldsymbol{x} \in \mathcal{Q}(\boldsymbol{y}y)$. There exists a unique $\boldsymbol{x}_y \in \mathcal{Q}(\boldsymbol{y})$ such that $\boldsymbol{x}_y \preceq \boldsymbol{x}$. By strict monotonicity, it follows that[20]

$$|\boldsymbol{x}_y{}^{-1}\boldsymbol{x}| \leq |f(\boldsymbol{x}_y)^{-1} f(\boldsymbol{x})| = |\boldsymbol{y}^{-1}\boldsymbol{y}y| = |y| = 1$$

which proves the claim.

(3) For any $\boldsymbol{y} \in \mathcal{Y}^*$ and any $y \in \mathcal{Y}$, we have from (2),

$$|\mathcal{Q}(\boldsymbol{y}y)| \leq (|\mathcal{X}|+1)|\mathcal{Q}(\boldsymbol{y})|. \tag{14}$$

Iterating this bound along any string $\boldsymbol{y} \in \mathcal{Y}^*$, we have:

$$|\mathcal{Q}(\boldsymbol{y})| \leq (|\mathcal{X}|+1)^{|\boldsymbol{y}|}|\mathcal{Q}(\varepsilon)| = (|\mathcal{X}|+1)^{|\boldsymbol{y}|}. \qquad \blacksquare$$

F Proofs

Proposition 4.1. *Let $f\colon \mathcal{X}^* \to \mathcal{Y}^*$ be any map. The following statements are equivalent: (i) f is prefix monotone (ii) $f(\langle \boldsymbol{x}\rangle) \subseteq \langle f(\boldsymbol{x})\rangle$ for all $\boldsymbol{x} \in \mathcal{X}^*$ (iii) $\mathcal{P}(f(\boldsymbol{x})) = \mathcal{C}(f(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \mathcal{X}^*$ (iv) f is prefix-continuous. The proof is given in App. F.*

Proof. (1) $\Rightarrow$ (2): Prefix monotonicity means that for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}^*$, if we have that $\boldsymbol{x} \preceq \boldsymbol{x}\boldsymbol{x}'$ then $f(\boldsymbol{x}) \preceq f(\boldsymbol{x}\boldsymbol{x}')$ and thus that there exists a $\boldsymbol{y} \in \mathcal{Y}^*$ such that $f(\boldsymbol{x}\boldsymbol{x}') = f(\boldsymbol{x})\boldsymbol{y}$. And since $\boldsymbol{x}'$ was chosen arbitrarily (2) holds.

(2) $\Rightarrow$ (3): Let $\boldsymbol{x} \in \mathcal{X}^*$. Suppose $f(\langle \boldsymbol{x}\rangle) \subseteq \langle f(\boldsymbol{x})\rangle$. Then, $\langle \boldsymbol{x}\rangle \subseteq f^{-1}(f(\langle \boldsymbol{x}\rangle)) \subseteq \mathcal{P}(f(\boldsymbol{x}))$. Note that, for any $\boldsymbol{x}' \in \mathcal{P}(f(\boldsymbol{x}))$, $\mathcal{P}(f(\boldsymbol{x}')) \subseteq \mathcal{P}(f(\boldsymbol{x}))$. Therefore, $\langle \boldsymbol{x}'\rangle \subseteq \mathcal{P}(f(\boldsymbol{x}')) \subseteq \mathcal{P}(f(\boldsymbol{x}))$ and so $\boldsymbol{x}' \in \mathcal{C}(f(\boldsymbol{x}))$. This shows that $\mathcal{P}(f(\boldsymbol{x})) = \mathcal{C}(f(\boldsymbol{x}))$.

(3) $\Rightarrow$ (4): By assumption, any element in the precover is an extension of a quotient element, and thus a member in the cylinder over quotient elements.

(4) $\Rightarrow$ (1): For $\boldsymbol{x}' \in \mathcal{P}(f(\boldsymbol{x}))$ we have by definition that $f(\boldsymbol{x}) \preceq f(\boldsymbol{x}')$. Assuming f is prefix-continuous, i.e., there are no remainder elements, then $\boldsymbol{x} \preceq \boldsymbol{x}'$ implies $\boldsymbol{x}' \in \mathcal{P}(f(\boldsymbol{x}))$.

$\blacksquare$

[20] For two strings $\boldsymbol{x}, \boldsymbol{x}'$ such that $\boldsymbol{x} \preceq \boldsymbol{x}'$, the string $\boldsymbol{x}^{-1}\boldsymbol{x}'$ denotes the unique string $\boldsymbol{x}''$ such that $\boldsymbol{x}\boldsymbol{x}'' = \boldsymbol{x}'$.

Lemma 4.1. *Let $\mathtt{f}$ be a finite-state transducer encoding $f : \mathcal{X}^* \to \mathcal{Y}^*$, and $\boldsymbol{y} \in \mathcal{Y}^*$. If (i) there is a finite number of paths in $\mathtt{f}$ that emit $\boldsymbol{y}$ and (ii) every state is either universal or has finite closure, then $\mathcal{Q}(\boldsymbol{y})$ and $\mathcal{R}(\boldsymbol{y})$ are of finite size for all $\boldsymbol{y} \in \mathcal{Y}^*$. The proof is given in App. F.*

Proof. Let $\boldsymbol{y} \in \mathcal{Y}^*$ and $\Pi_{\boldsymbol{y}}$ be the set of, not necessarily accepting, paths that emit exactly $\boldsymbol{y}$. For each $\boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}$, let $\boldsymbol{x}_{\boldsymbol{\pi}}$ be the scanned input and $s_{\boldsymbol{\pi}}$ the state reached after emitting $\boldsymbol{y}$. Consider the trimmed automaton $\mathtt{P}_{\boldsymbol{\pi}} \stackrel{\text{def}}{=} \mathsf{trim}(\mathsf{proj}_{\mathcal{X}}(\mathtt{f}_{[s_{\boldsymbol{\pi}}]}))$ consisting of all states and transitions on paths from an initial state to some $s_{\boldsymbol{\pi}}$ and from $s_{\boldsymbol{\pi}}$ to a final state for some $\boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}$. Note that the language accepted by $\mathtt{P}_{\boldsymbol{\pi}}$ may be larger than that of the precover since the automaton may accept prefixes of its members, i.e., $L(\mathtt{P}_{\boldsymbol{\pi}}) \supseteq f^{-1}(\boldsymbol{y}\mathcal{Y}^*)$. By assumption, every state in $\mathtt{P}_{\boldsymbol{\pi}}$ is either universal or has finite closure. We use $\Pi_{\boldsymbol{y}}^u$ to denote the subsets of the paths that end in a universal state and $\Pi_{\boldsymbol{y}}^c$ to denote those that end in a state with finite closure. We can then decompose the precover explicitly as

$$f^{-1}(\boldsymbol{y}\mathcal{Y}^*) = \bigcup_{\boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}} \boldsymbol{x}_{\boldsymbol{\pi}} L(\mathtt{P}_{\boldsymbol{\pi}}) = \bigcup_{\boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}^c} \boldsymbol{x}_{\boldsymbol{\pi}} L(\mathsf{proj}_{\mathcal{X}}(\mathtt{f}_{[s_{\boldsymbol{\pi}}]})) \cup \bigcup_{\boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}^u} \boldsymbol{x}_{\boldsymbol{\pi}} \mathcal{X}^*. \tag{15}$$

Thus $\mathcal{R}(\boldsymbol{y}) \subseteq \bigcup_{\boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}^c} \boldsymbol{x}_{\boldsymbol{\pi}} L(\mathsf{proj}_{\mathcal{X}}(\mathtt{f}_{[s_{\boldsymbol{\pi}}]}))$ and $\mathcal{Q}(\boldsymbol{y}) \subseteq \{\boldsymbol{x}_{\boldsymbol{\pi}} \colon \boldsymbol{\pi} \in \Pi_{\boldsymbol{y}}^u\}$. By assumption $|\Pi_{\boldsymbol{y}}| = |{\Pi_{\boldsymbol{y}}}^u \cup {\Pi_{\boldsymbol{y}}}^c| < \infty$ so $|\mathcal{Q}(\boldsymbol{y})| < \infty$. By the finite-closure assumption $|\boldsymbol{x} L(\mathsf{proj}_{\mathcal{X}}(\mathtt{f}_{[s_{\boldsymbol{\pi}}]}))| < \infty$ for any $\boldsymbol{\pi} \in {\Pi_{\boldsymbol{y}}}^c$, so $|\mathcal{R}(\boldsymbol{y})| < \infty$.

■

In practice $|\Pi_{\boldsymbol{y}}| < \infty$ implies there exists no loop of the form $(s \xrightarrow{x:\varepsilon} s)$. Note also that Eq. (15) does not guarantee the two unions are disjoint; some elements on the left could be included in the right union as made explicit in the proof. An algorithm following the above reading should thus ensure the disjoint property in a post-processing step.

G Algorithms

In §5 (Fig. 3), we introduce an algorithm that uses a transducer to compute the prefix decomposition $(\mathcal{R}(\boldsymbol{y}), \mathcal{Q}(\boldsymbol{y}))$ for a string $\boldsymbol{y}$. The algorithm first builds a transducer that only accepts members of the precover $\mathcal{P}(\boldsymbol{y})$ and then determinizes it. This section introduces more practical variants that skip the expensive determinization step and operate directly on the original transducer, without composing it with the language $\boldsymbol{y}\mathcal{Y}^*$. We also provide supplementary details and algorithms to evaluate probability $p_{\mathcal{Y}}(\cdot)$, prefix probability $\overrightarrow{p_{\mathcal{Y}}}(\cdot)$, and its conditional form $\overrightarrow{p_{\mathcal{Y}}}(\cdot \mid \cdot)$ as described in §2.

G.1 Notes on Universality Checking

In §5.1 we introduce a transducer-based algorithm for deriving the decomposition of the precover. Here we elaborate on the **Continuity** step and, in particular, the *universality check*. We note it is computable, since *universality is decidable* for finite-state automata.[21] For each state s, we can test whether $\mathtt{f}_{[s]}$ accepts $\mathcal{X}^*$ by checking its equivalence against a reference machine for $\mathcal{X}^*$. To ensure we can check for universality at a single state, we *determinize* $\mathtt{P}$ and *trim* away dead-end paths. Without determinization, two paths may yield the same string and end up in different states s_1 and s_2, where neither state is universal on its own, although their union covers the entire language $\mathcal{X}^*$. That is, where $[\![\mathtt{f}_{[s_1]}]\!] \neq \mathcal{X}^*$ and $[\![\mathtt{f}_{[s_2]}]\!] \neq \mathcal{X}^*$, but $[\![\mathtt{f}_{[s_1]}]\!] \cup [\![\mathtt{f}_{[s_2]}]\!] = \mathcal{X}^*$. A non-deterministic transducer could misclassify such strings as remainder elements, leading to a suboptimal decomposition. Determinization merges such states, ensuring that a string is universal if and only if the single state it reaches, for a given input, is universal.

G.2 Lazy Determinization

In Fig. 9 (left), we give an algorithm that enumerates the paths of the transducer $\mathsf{proj}_{\mathcal{X}}(\mathtt{f} \circ \boldsymbol{y}\mathcal{Y}^*)$ without explicit determinization. Instead, the algorithm implicitly tracks *power states*. A power state

[21] Efficient algorithms for this include antichain-based simulation (De Wulf et al., 2006), bisimulation up to congruence (Bonchi & Pous, 2015), and classical equivalence checking (Meyer & Stockmeyer, 1972).

S represents the set of all possible states the transducer may occupy after scanning a string $\boldsymbol{x}$. For the current power state, the algorithm iterates over every input symbol $\boldsymbol{x} \in \mathcal{X}^*$, gathers all outgoing transitions that scan $\boldsymbol{x}$, and takes the ε-closure of their target states to form the next power state S'. The resulting edges $S \xrightarrow{x'} S'$ are analogous to those that explicit determinization with subset construction would create. To determine whether a string belongs to the quotient or the remainder, we check whether the power state is universal (see Fig. 9 on the right). Otherwise, we check whether any individual state of S is final before adding the string to the remainder.

```
def lazy_decomposition(y):
  # Build FST encoding ⟦P⟧ = P(y)
  P ← proj_X(f ∘ yY*)
  (_, _, _, T, I, F) ← P
  Q ← QUEUE()
  Q.push((I, ε))
  (R, Q) ← (∅, ∅)
  while |Q| > 0:
    (S, x) ← Q.pop()
    if powerstate_universal(S):
      Q.add(x)
      continue
    if S ∩ F ≠ ∅: R.add(x)
    for x' ∈ X:
      S' ← next_powerstate(S, x')
      Q.push((S', x x'))
  return (R, Q)

def next_powerstate(S, x'):
  # S --x'--> S' is a determ. transition
  S' ← ∅
  for s in eps_closure(S):
    for (s --x'--> s') ∈ T(s, x'):
      S'.update(eps_closure({s'}))
  return S'
```

```
def powerstate_universal(f, S):
  (S, X, Y, T, I, F) ← f
  s ← State()  # A new state
  S' ← S ∪ {s}
  T' ← T ∪ {(s --ε--> s') | s' ∈ S}
  f' ← (S', X, Y, T', I, F)
  return is_universal(f', s)

def eps_closure(S):
  closure ← ∅
  Q ← QUEUE(S)
  while |Q| > 0:
    s' ← Q.pop()
    if s' in closure:
      continue
    closure.add(s')
    for (s' --ε--> s'') ∈ T(s', ε):
      Q.add(s'')
  return closure
```

Figure 9: An algorithm (left) that computes the optimal decomposition without explicitly determinizing the machine.

G.3 A Recursive Algorithm That Operates Directly on the FST

We now consider a recursive approach for enumerating the prefix decomposition. Let $\boldsymbol{y} \in \mathcal{Y}^*$ and $y \in \mathcal{Y}$. Since $\langle \mathcal{Q}(\boldsymbol{y}y) \rangle \subseteq \langle \mathcal{Q}(\boldsymbol{y}) \rangle$, we have $\mathcal{P}(\boldsymbol{y}y) = (\langle \mathcal{Q}(\boldsymbol{y}) \rangle \cap \langle \mathcal{Q}(\boldsymbol{y}y) \rangle) \sqcup \mathcal{R}(\boldsymbol{y}y)$. This means that once we have computed the prefix decomposition $(\mathcal{R}(\boldsymbol{y}), \mathcal{Q}(\boldsymbol{y}))$ for a string $\boldsymbol{y} \in \mathcal{Y}^*$, the decomposition for any extension $\boldsymbol{y}y$, where $y \in \mathcal{Y}$, can be obtained with incremental work. The algorithm in Fig. 10 (left) performs a recursive enumeration of the quotient and remainder, where each recursive call is realized using a cache lookup. Note that we only store exact matches in the cache to limit memory usage and ensure the cache is prefix-free.

Further, recall that, in §5.1, we composed f with $\boldsymbol{y}\mathcal{Y}^*$ to get a transducer that only accepts strings in the prefix decomposition. For many transducers, this is an expensive operation that must be repeated for every target sequence. Instead, we would like to operate directly on f without the initial composition. Doing so requires careful bookkeeping of partial outputs. Fortunately, it allows us to precompute the set of universal states.

The algorithm proceeds in two phases. The first phase identifies the paths in f whose output $\boldsymbol{y}'$ covers the target, i.e., $\boldsymbol{y}' \succeq \boldsymbol{y}$. Similar to the algorithm in App. G.2, we group partial paths by the scanned input symbol x, so each power state corresponds to a unique input prefix. However, instead of tracking only the power state S, we maintain a frontier set $\mathcal{F}$ of pairs $(s, \boldsymbol{y}')$ that record the current state s and the output sequence $\boldsymbol{y}'$ emitted in that state. We then discard any partial paths whose

emitted output cannot cover the target. In the second phase, once a path has produced a covering output, we stop applying the output-based filter. For each such path, we form the corresponding power state by grouping all transitions that share the same input $S \xrightarrow{x} S'$, as in the algorithm given in Fig. 9 (left).

In contrast to previous algorithms, we process all active candidate paths simultaneously at each iteration, followed by a pruning step to discard low-probability candidates. Although pruning makes the algorithm practical, it results in an approximation of the prefix probability of the target sequence, as we discard elements of the prefix decomposition with low mass. Details of the pruning strategies are provided in App. G.4.

```
def lazy_cached_decomp(y, cache):
  (_, _, _, T, I, F) ← f
  (R, Q) ← (∅, ∅)
  Q ← get_cached(y, cache)
  while |Q| > 0:
    Q_C ← ∅ # New candidates
    for (F, x) in Q:
      match ← {(s, y') ∈ F | y' ⪰ y}
      exact ← {(s, y') ∈ match | y' = y}
      if |exact| > 0: # Update cache
        cache[y].add((exact, x))
      if |match| > 0:
        S ← {s | (s, y') ∈ match}
        if powerstate_universal(S):
          Q.add((match, x))
          continue
        if S ∩ F ≠ ∅:
          R.add((match, x))
      for x' ∈ X:
        F' ← next_frontier_out(F, x')
        F' ← {(s, y'') ∈ F' |
            (y'' ⪯ y) ∨ (y ⪯ y'')}
        if |F'| > 0:
          Q_C.add((F', x x'))
    Q ← prune(Q_C)
  return (R, Q)
```

```
def get_cached(y, cache):
  y_1 ··· y_N ← y
  for i in range(N, 1, −1):
    y_i ← y_1 ··· y_i
    if y_i in cache: return cache[y_i]
  return {({(s, ε) | s ∈ I}, ε)}

def eps_closure_out(F):
  closure ← ∅
  Q ← QUEUE(F)
  while |Q| > 0:
    (s', y') ← Q.pop()
    if s' in closure:
      continue
    closure.add(s')
    for (s' --ε:y--> s'') ∈ T(s', ε):
      Q.add((s'', y'y))
  return closure

def next_frontier_out(F, x):
  F ← eps_closure_out(F)
  F' ← ∅
  for (s, y) in F:
    for (s --x:y--> s') ∈ T(s, x):
      F'.add((s', yy))
  F' ← eps_closure_out(F')
  return F'
```

Figure 10: A lazy, cached, fully FST-based approach that does not use composition.

G.4 Probability Mass Pruning

To make probability calculations relying on the algorithm practical but potentially inexact, we use a threshold-based pruning strategy, which sorts candidates in Q by their prefix probability and then discards those whose cumulative probability mass falls below a specified threshold τ. Smaller values of τ retain more candidates, whereas larger values improve efficiency at the risk of discarding relevant candidates. Furthermore, we use an adaptive threshold that grows with the size of the candidate set. Given a hyperparameter n_{piv} and $n = |\text{Q}|$, if $n \leq n_{\text{piv}}$ we use the base threshold τ, otherwise the threshold increases at a rate controlled by a hyperparameter α and is capped by another hyperparameter τ_{max}. Additionally, we introduce an optional hard size cap n_{max} on the candidate set, that keeps only the top-n_{max} candidates by their probability. The (optional) pruning strategy is given in Fig. 11 on the left with the adaptive threshold given on the right.

Note that our pruning strategy may occasionally reach dead ends at high pruning thresholds, where no valid extension remains. In such cases, we apply a backtracking algorithm that retraces the cache and incrementally relaxes the threshold until a viable extension is found.

```
def prob_mass_prune(p⃗_X, τ, α, τ_max, n_piv, n_max):
  def prune(Q):
    w ← [score(x, p⃗_X) | (_, x) ∈ Q]
    n ← |Q|,  Z ← Σ_{i=1}^{n} w_i,  w ← w/Z
    τ_new ← adapt_thld(n, τ, α, τ_max, n_piv)
    σ ← argsort(w)  # w_σ(1) ≤ ··· ≤ w_σ(n)
    C_k ← Σ_{j=1}^{k} w_σ(j)   (k = 1, ..., n)
    k_τnew ← min{ k ∈ [n] | C_k > τ_new }
    n_cap ← n − min(n_max, n) + 1
    I_keep ← { σ(j) | j ≥ max(k_τnew, n_cap) }
    return [Q[i] for i in I_keep]
  return prune
```

```
def score(x, p⃗_X): # memoized
  x_1 ··· x_M ← x
  if M = 0:
    return 1
  p ← score(x_1 ··· x_{M−1})
  return p · p⃗_X(x_M | x_1 ··· x_{M−1})

def adapt_thld(n, τ, α, τ_max, n_piv):
  if n ≤ n_piv:
    return τ
  τ_new ← τ · (n / n_piv)^α
  return min(τ_new, τ_max)
```

Figure 11: Probability mass pruning algorithm (left); memoized scoring function and adaptive pruning threshold (right).

G.5 Computing Probabilities – Implementing the Language Model Interface

Given a language model $p_{\mathcal{X}}$ and a transducer f, the algorithm given in Fig. 10 recursively enumerates the remainder $\mathcal{R}(\boldsymbol{y})$ and quotient $\mathcal{Q}(\boldsymbol{y})$ for a sequence $\boldsymbol{y}$. We now make it explicit how we employ the interface defined in §2; the algorithms given in Fig. 12 show how to compute $p_{\mathcal{Y}}(\boldsymbol{y})$, $\overrightarrow{p_{\mathcal{Y}}}(\boldsymbol{y})$, and the next character distribution $\overrightarrow{p_{\mathcal{Y}}}(\cdot \mid \boldsymbol{y})$. The direct computation of $\overrightarrow{p_{\mathcal{Y}}}(\cdot \mid \boldsymbol{y})$ loops over every symbol $y \in \mathcal{Y} \cup \{\text{EOS}\}$ and is therefore expensive.

To avoid this, we present a faster version for computing the next character distribution that exploits the recursive structure of the quotient and remainder (see App. G.3). We start by describing an abstract version of the algorithm, which is given in Fig. 13 on the left.

The algorithm computes the prefix decomposition for the given target string $\boldsymbol{y}$ and then iterates over all elements of $\mathcal{Q}(\boldsymbol{y})$ and $\mathcal{R}(\boldsymbol{y})$. For each emitted output $\boldsymbol{y}'$ where $|\boldsymbol{y}'| > |\boldsymbol{y}|$ we accumulate the probability mass for the respective symbols $\boldsymbol{y}'_{|\boldsymbol{y}|+1}$. Any elements of $\mathcal{Q}(\boldsymbol{y})$ not scored in this pass, i.e., where $|\boldsymbol{y}'| = |\boldsymbol{y}|$, are further expanded until a universal power state is reached. We do not expand the remainder elements, since their universal extensions are already contained in $\mathcal{Q}(\boldsymbol{y})$.

The complete transducer-based version of this algorithm is given in Fig. 13 on the right. It uses a similar procedure as the algorithm Fig. 10 for maintaining the frontier $\mathcal{F}$ of state-output pairs.

```
def prefix_prob(y, p⃗_X):
  (R, Q) ← decomposition(y)
  p⃗_y ← 0
  for (_, x) in Q:
    p ← score(x, p⃗_X)
    p⃗_y ← p⃗_y + p
  for (_, x') in R:
    p ← score(x', p⃗_X)
    p⃗_y ← p⃗_y + p · p⃗_X(EOS | x')
  return p⃗_y, R, Q
```

```
def prob(y, p⃗_X):
  (R, Q) ← decomposition(y)
  p_y ← 0
  for (F, x) in Q ∪ R:
    if (∃(_, y') ∈ F : y = y'):
      p_y ← p_y + score(x, p⃗_X)
  return p_y, R, Q

def next_dist(y, p⃗_X):
  p̄ ← {y': 0 for y' ∈ Y ∪ {EOS}}
  Z, * ← prefix_prob(y, p⃗_X)
  for y' ∈ Y ∪ {EOS}:
    p⃗_yy', * ← prefix_prob(yy', p⃗_X)
    p̄[y'] ← p⃗_yy' / Z
  return p̄
```

Figure 12: Algorithms for computing $\overrightarrow{p_{\mathcal{Y}}}(\cdot)$, $p_{\mathcal{Y}}(\cdot)$ and $\overrightarrow{p_{\mathcal{Y}}}(\cdot \mid \boldsymbol{y})$.

```
def abstract_fast_next_dist(y, p⃗_X):
  p̄ ← {y′: 0 for y′ ∈ Y ∪ {EOS}}
  Z, R, Q ← prefix_prob(y, p⃗_X)
  Q ← QUEUE()
  for (s, x, y′) in Q:
    if |y′| > |y|:
      ŷ ← y′_{|y|+1}
      p̄[ŷ] += score(x, p⃗_X)/Z
    else:
      Q.add((s, x, y′))
  for (s, x, y′) in R:
    if |y′| > |y|:
      ŷ ← y′_{|y|+1}
      p̄[ŷ] += score(x EOS, p⃗_X)/Z
  while |Q| > 0:
    (s, x, y′) ← Q.pop():
    for x ∈ X:
      for y ∈ Y:
        ŷ ← y′ y_{|y|+1}
        if ⟨xx⟩ ⊆ P(yŷ):
          p̄[ŷ] += score(x, p⃗_X)/Z
          continue
        elif xx ∈ P(yŷ):
          p̄[ŷ] += score(x EOS, p⃗_X)/Z
        Q_C.push((s, xx, yy))
  return p̄

def merge(p̄, p̄_new):
  for y in p̄_new:
    p̄[y] ← p̄[y] + p̄_new[y]
```

```
def fast_next_dist(y, p⃗_X):
  def score_tgt(F, x):
    p̄_new ← {}
    Y ← {y′_{|y|+1} | (s, y′) ∈ F, y′ ≻ y}
    for y ∈ Y: p̄_new[y] ← score(x, p⃗_X)/Z
    return p̄_new
  p̄ ← {y′: 0 for y′ ∈ Y ∪ {EOS}}
  Z, R, Q ← prefix_prob(y, p⃗_X)
  Q ← ∅
  for (F, x) in Q:
    p̄_new ← score_tgt(F, x)
    if p̄_new ≠ {}:
      merge(p̄, p̄_new)
    else:
      Q.add((F, x))
  for (F, x) in R:
    merge(p̄, score_tgt(F, x EOS))
  while |Q| > 0:
    Q′_C ← ∅
    for (F, x) in Q:
      for x ∈ X:
        F′ ← next_frontier_out(F, x)
        S ← {s | (s, y′) ∈ F′, y′ ≻ y}
        if powerstate_universal(S):
          p̄_new ← score_tgt(F′, xx)
          if p̄_new ≠ {}:
            merge(p̄, p̄_new)
            continue
        elif S ∩ F ≠ ∅:
          merge(p̄, score_tgt(F′, xx EOS))
        Q_C.add((F′, xx))
    Q ← prune(Q_C)
  return p̄
```

Figure 13: Algorithms for computing efficiently computing $\overrightarrow{p_{\mathcal{Y}}}(\cdot \mid \boldsymbol{y})$.

H Related Work

Modern language models define probability distributions over sequences of tokens (see §2). For efficiency and vocabulary (a.k.a. their alphabet) management, they usually rely on sub-word schemes such as BPE (Sennrich et al., 2016; Gage, 1994) or Unigram (Kudo, 2018). Although these approaches have been remarkably successful, their units often don't coincide with linguistic boundaries, and any given string typically admits an exponential number of variations of tokenizations with non-zero probability mass under the language model. Recent work has tackled this issue by enforcing canonical tokenization to remove probability mass from noncanonical encodings (Vieira et al., 2025b), while Geh et al. (2024) have shown that aggregating the probability mass of noncanonical tokenization choices carries a useful signal that can boost downstream accuracy.

Sub-word segmentation also gives rise to the prompt-boundary problem (Vieira et al., 2025a), where imperceptible changes to the final characters of a prompt (e.g., appending a single whitespace) can push the encoded token sequence onto a completely different path in token space, causing the model to abandon otherwise highly probable continuations. To overcome these issues, Vieira et al. (2025a) introduce an algorithm for transforming token-based language models into language models over characters. Although their contribution centers around characters, the underlying idea can be generalized. Various applications could make use of a method for accurately converting the probability mass learned over subword tokens onto other types of units, such as bytes, words, or morphemes in NLP, or amino acids in computational biology.

In psycholinguistics, for instance, researchers often require fine-grained estimates of surprisal, e.g., when predicting a reader's likelihood of skipping a word, based on how predictable its first three characters are (Rayner et al., 1982; Blanchard et al., 1989). To this end, a number of recent studies have tackled the challenges posed by subword tokenization in modern language models (Nair & Resnik, 2023; Beinborn & Pinter, 2023; Pimentel & Meister, 2024; Oh & Schuler, 2024; Giulianelli et al., 2024). For example, recent studies (Oh & Schuler, 2024; Pimentel & Meister, 2024) have argued that leading whitespace tokenization introduces a confound in surprisal estimates and instead, advocate for incorporating the probability of trailing whitespaces into such calculations.

Furthermore, Pimentel & Meister (2024) gives a bespoke procedure for converting token-based language models to word-based language models. However, their method does not model the contextually sensitive nature of English word segmentation, e.g., it treats both periods in Ex. (1) identically, where English orthography does not. Additionally, the justification of the procedure requires that there exists a set of distinguished end-of-word markers that appear at the end of a token, if at all. We now consider how such a transducer can be constructed. Let $\mathtt{f}_\alpha$ be a transducer that converts a token alphabet to a character alphabet D be the set of delimiters. The transducer $\mathtt{f}_D$ is given in Fig. 14. Given a language model $p_\mathcal{X}$ over $\mathcal{X}$, we can then compose them into a transducer $p_\mathcal{X} \circ \mathtt{f}_\alpha \circ \mathtt{f}_D$ to get a transduced language model over separator-delimited words. However, such an approach would be rather naïve. Unfortunately, delimiter-based separation would not be able to distinguish when the dot should be its own symbols or not as in ex. (3)

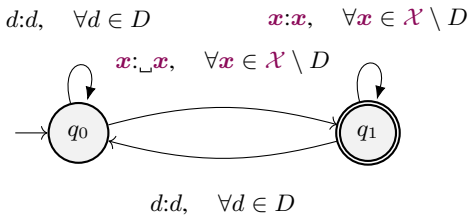


Figure 14: A simple FST that segments character streams into words without contextual information, inserting ␣ at the start of each word.

The delimiter-based approach also breaks for most BPE-based language models due to clustering of delimiter candidates. For instance, GPT-4o's alphabet contains the token !!! 10880, which consists solely of end-of-word symbols. Under PTB guidelines, for example, !!! 10880 should be broken into three consecutive orthographic words ! ! !. In contrast, this paper takes the position that the proper tokenization scheme for psycholinguistic modeling should be specified based on the goals of the study and not based on the properties of any one specific tokenizer.

Tokenization challenges are not unique to modeling natural language. In computational biology, DNA, RNA, and protein sequences are long, unsegmented strings over small alphabets that pose challenges in tokenization. Researchers thus alternate between different tokenization schemas, such as k-mers, learned subwords, and motif-aware segmenters (Ji et al., 2021; Nguyen et al., 2023; Dotan et al., 2024; Wang et al., 2024; Qiao et al., 2024). Because foundational models are often trained under different tokenization schemas, their predictions cannot easily be compared or reused across different tasks.

Transducer-based approaches to tokenization are well-established—WordPiece (Wu et al., 2016) can be implemented as a transducer (Song et al., 2021), and deterministic finite automata have been constructed for BPE (Berglund & van der Merwe, 2023; Berglund et al., 2024). Moreover, transducers also have a long history in language modeling (Mohri, 1997) and have been adopted for constrained decoding, where an FST enforces lexical or structural constraints (Allauzen et al., 2014; Ghazvininejad et al., 2016; Stahlberg et al., 2019; Willard & Louf, 2023; Koo et al., 2024; Cognetta et al., 2025).

In this study, we generalize the character-level conversion and extend Vieira et al. (2025a) to a framework that allows transforming a language model to another language model beyond the limited setting of strict-prefix monotonic mappings. We support conversations between sets of units and unit-preserving transformations, provided that the mapping between them can be described by a finite-state transducer. We give algorithms for doing so accurately and efficiently, and present a formal framework that outlines the conditions under which a mapping between two sets of units can be performed exactly, along with practical algorithms.

I Constructing the PTB Tokenizer

We construct the PTB FST by encoding each tokenizer rule[22] as an FST that segments character sequences by inserting a distinguished separator symbol SEP $\notin \mathcal{Y}$. Note that the resulting transduced language model is thus not a true distribution over PTB tokens, but over characters and separators corresponding to the same boundaries that the PTB tokens would have. This is a pragmatic decision, as the PTB tokenizer can tokenize any sentence into orthographic words. In other words, it would accept an infinite vocabulary. Such a transducer can be built on the fly and would be equivalent to one with infinitely many states. While nothing prevents such a transducer from being constructed and used, we stick to the finite version in the scope of this paper. An example of such a rule is given in Fig. 5, which inserts SEP after a comma if it is not followed by a digit. We then compose these FSTs into a single transducer ($\mathtt{f}_{\mathrm{ptb}}$). Note that context-dependent rules, such as the one given in Fig. 5, introduce non-universal states. For example, state q_2 only accepts $x \in \mathcal{X} \setminus \{\texttt{0-9}, \}$. In fact, out of the 197 states in $\mathtt{f}_{\mathrm{ptb}}$, just 31 are universal.

J Details on Transducers Used in Experiments

Tab. 2 contains the number of states, universal states, and transitions for the transducers described in §6. We construct all finite-state transducers using Pynini (Gorman, 2016). Note that for experiments using the Penn Treebank FST ($\mathtt{f}_{\mathrm{ptb}}$), we realize $p_{\mathcal{X}} \circ \mathtt{f}_{\alpha}$ using GenLM.bytes, thereby keeping the number of states and arcs constant.

Table 2: Number of states, universal states, and transitions.

Model	States	Universal States	Transitions
Tokens to Characters			
$p_{\mathsf{gpt2}} \circ \mathtt{f}_{\alpha}$	75,723	75,723	125,979
$p_{\mathsf{llama1B}} \circ \mathtt{f}_{\alpha}$	176,990	176,990	305,244
$p_{\mathsf{llama8B}} \circ \mathtt{f}_{\alpha}$	176,990	176,990	305,244
Tokens to Words			
$\mathtt{f}_{\mathrm{ptb}}$	197	31	14,584
$p_{\mathsf{gpt2}} \circ \mathtt{f}_{\alpha} \circ \mathtt{f}_{\mathrm{ptb}}$	197	31	14,584
$p_{\mathsf{llama1B}} \circ \mathtt{f}_{\alpha} \circ \mathtt{f}_{\mathrm{ptb}}$	197	31	14,584
$p_{\mathsf{llama8B}} \circ \mathtt{f}_{\alpha} \circ \mathtt{f}_{\mathrm{ptb}}$	197	31	14,584
DNA to amino acids			
$\mathtt{f}_{\mathrm{dna2aa}}$	21	21	84
$p_{\mathsf{dna}} \circ \mathtt{f}_{\mathrm{dna2aa}}$	21	21	84

K Experimental Setup

Here we detail the experimental setup for reproducing the experiments in §6 and §6.

[22] See https://www.nltk.org/_modules/nltk/tokenize/treebank.html#TreebankWordTokenizer for the full specification.

K.1 DATASETS

For the experiments in §6 and §6, we choose the first 10 paragraphs (excluding headers) of the test split in the `wikitext-103-v1` dataset (Merity et al., 2017) (corresponding to the first 7684 bytes) from the 🤗 Hugging Face datasets library. We use the first 256 bytes of the same dataset and split to run the experiments in §6. For the experiments in §6, we sample 65 human proteins[23], each consisting of 4-12 amino acids, with their accession numbers given in Tab. 3.

Table 3: Accession numbers used in this study.

C0HLZ5	P01858	P0DPI4	P0DUS0	P84464
P84465	P0DOY5	P67857	P67858	P67859
P81826	P23210	P85003	P84071	P86168
C0HJF1	C0HJG0	P02729	P81010	P86909
P86922	B3EWE5	P0DMM6	P0DQM6	P0DQM7
P12481	P85002	B3EWR3	P01358	P02728
P0C005	P0DKX2	P0DMM7	P0DQM9	P0DX30
P22103	P84200	P84785	P84868	P86600
A8C8X2	B3A0L6	B3EUR5	C0HJM6	C0HLK7
P0DJC3	P0DJF4	P69208	P85444	P85870
P86942	A0A0A0MT89	B3EWS0	C0HJB6	C0HL84
C0HL88	P0C8I8	P0DQH7	P0DQH8	P0DQX4
P0DQX5	P58805	P69437	P82820	P83127

K.2 MODELS

We conduct experiments using GPT-2 Large (Radford et al., 2019), Llama 3.2-1B, and Llama 3.1-8B[24] from the 🤗 Hugging Face hub (Wolf et al., 2020). We use the GenLM library[25] and the vLLM (Kwon et al., 2023) backend to efficiently evaluate the models.

For the experiments in §6, we use GenLM.bytes[26] to convert token-level models into byte-level models and compose them with $\mathtt{f}_{\mathrm{ptb}}$. We use a beam size of K=5 and a pruning threshold of 0.001. For the experiments in §6, we train a custom GPT-2 Small model on a human DNA dataset[27]. Note that we set the token set to $\mathcal{X} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, eliminating the need for composing the model with a transducer $\mathtt{f}_{\alpha}$ that maps from subword tokens to characters. For training parameters, see Tab. 4, for training and validation metrics, see Tab. 5.

K.3 PARAMETERS

For all experiments, we use the pruning heuristic described in App. G.4 with the parameters $\tau_{\mathtt{max}} = 0.4$, $n_{\mathtt{piv}} = 100$ and $\alpha = 0.7$. For the experiments in §6 we report the results for different values of $n_{\mathtt{max}} \in \{5000, 10000, 15000, 20000\}$. For all other experiments, we set $n_{\mathtt{max}} = \infty$

K.4 GPU USAGE

All experiments in §6 were run on a single NVIDIA GeForce RTX 4090 GPU with 24GB of memory.

[23] https://www.uniprot.org/uniprotkb?query=Human

[24] Available under `openai/gpt2-large`, `meta-llama/Llama-3.2-1B` and `meta-llama/Llama-3.1-8B` at https://huggingface.co.

[25] https://github.com/genlm/genlm-backend

[26] https://github.com/genlm/genlm-bytes

[27] https://huggingface.co/datasets/simecek/Human_DNA_v0.

Table 4: Training Hyperparameters.

Parameter	Value
Learning Rate	0.0003
Optimizer	AdamW
	$\beta_1 = 0.9$, $\beta_2 = 0.999$,
	$\epsilon = 1\text{e}{-8}$
Learning Rate Scheduler	Linear
Warm-up Steps	1000
Train Batches (per device)	64
Eval Batches (per device)	8
Total Train Batches	256 (262,144 tokens)
Total Eval Batches	32
Epochs	10
Seed	42
Distributed Training	Multi-GPU (4 devices)
Mixed Precision	Native AMP

Table 5: Training and Validation Metrics.

Step	Epoch	Train Loss	Val Loss	Acc (%)
5k	0.69	1.1252	1.1206	47.45
10k	1.38	1.0835	1.0814	49.91
15k	2.07	1.0641	1.0639	51.03
20k	2.76	1.0563	1.0547	51.63
25k	3.45	1.0504	1.0486	52.04
30k	4.14	1.0439	1.0439	52.33
35k	4.84	1.0425	1.0407	52.54
40k	5.53	1.0365	1.0380	52.71
45k	6.22	1.0325	1.0361	52.84
50k	6.91	1.0322	1.0341	52.96
55k	7.60	1.0307	1.0328	53.05
60k	8.29	1.0267	1.0316	53.13
65k	8.98	1.0273	1.0306	53.20
70k	9.67	1.0270	1.0299	53.24

L Experimental Results

Here we provide complementary results to the experiments presented in §6.

L.1 Jensen–Shannon distance

We benchmark how well the algorithm given in §5 approximates the baselines when using high pruning thresholds τ in the probability mass pruning described in App. G.4. For all three settings $p_{\mathcal{X}} \circ \mathtt{f}_{\alpha}$ (Tab. 7 and Fig. 6), $p_{\mathcal{X}} \circ \mathtt{f}_{\alpha} \circ \mathtt{f}_{\mathrm{ptb}}$ (Tab. 8 and Fig. 6), and $p_{\mathcal{X}} \circ \mathtt{f}_{\mathrm{dna2aa}}$ (Tab. 9), we observe decreasing Jensen-Shannon distances, at the cost of throughput (bytes per second). Note that runtimes become less consistent for higher thresholds ($\tau >$1e-4) as these settings frequently lead to dead ends and require backtracking (see App. G.3). Importantly, larger models are not slower, as the scoring of the candidates is not the main computational bottleneck.

Table 6: Average Jensen–Shannon distance (JSD) and bytes per second for various thresholds τ using $p_{\mathcal{X}} \circ \mathtt{f}_{\alpha}$ against a reference distribution from Vieira et al. (2025a) with a beam size of K=60. 95% confidence intervals are given in parentheses.

τ	$p_{\mathsf{gpt2}} \circ \mathtt{f}_{\alpha}$ average JSD / byte	$p_{\mathsf{llama1B}} \circ \mathtt{f}_{\alpha}$ average JSD / byte	$p_{\mathsf{llama8B}} \circ \mathtt{f}_{\alpha}$ average JSD / byte
1e-1	7.8e-2 (7.4e-2, 8.1e-2)	6.7e-2 (6.4e-2, 7.0e-2)	5.2e-2 (5.0e-2, 5.5e-2)
3e-2	7.4e-2 (7.1e-2, 7.8e-2)	6.2e-2 (5.9e-2, 6.5e-2)	4.8e-2 (4.6e-2, 5.0e-2)
1e-2	6.1e-2 (5.8e-2, 6.3e-2)	5.4e-2 (5.1e-2, 5.6e-2)	4.3e-2 (4.1e-2, 4.5e-2)
3e-3	3.0e-2 (2.8e-2, 3.2e-2)	3.3e-2 (3.1e-2, 3.5e-2)	3.0e-2 (2.8e-2, 3.2e-2)
1e-3	1.5e-2 (1.4e-2, 1.7e-2)	1.6e-2 (1.4e-2, 1.7e-2)	1.1e-2 (1.0e-2, 1.3e-2)
3e-4	6.5e-3 (5.6e-3, 7.6e-3)	5.7e-3 (4.8e-3, 6.8e-3)	4.3e-3 (3.5e-3, 5.2e-3)
1e-4	3.8e-3 (2.9e-3, 4.7e-3)	2.5e-3 (2.0e-3, 3.2e-3)	1.4e-3 (1.1e-3, 1.8e-3)
3e-5	8.5e-4 (5.6e-4, 1.3e-3)	9.2e-4 (6.9e-4, 1.2e-3)	8.1e-4 (5.1e-4, 1.2e-3)
1e-5	7.4e-4 (4.3e-4, 1.2e-3)	3.8e-4 (2.7e-4, 5.2e-4)	3.8e-4 (2.2e-4, 6.4e-4)
3e-6	6.7e-4 (3.1e-4, 1.1e-3)	3.2e-4 (2.2e-4, 4.5e-4)	2.0e-4 (1.4e-4, 2.7e-4)
1e-6	2.7e-4 (1.4e-4, 4.7e-4)	3.0e-4 (2.0e-4, 4.3e-4)	1.9e-4 (1.2e-4, 2.6e-4)

Table 7: Average Jensen–Shannon distance (JSD) and bytes per second for various thresholds τ and a reference (τ =1e-6) using $p_{\mathcal{X}} \circ \mathtt{f}_\alpha$. 95% confidence intervals are given in parentheses.

τ	$p_{\mathtt{gpt2}} \circ \mathtt{f}_\alpha$ average JSD / byte	$p_{\mathtt{gpt2}} \circ \mathtt{f}_\alpha$ bytes / sec	$p_{\mathtt{llama1B}} \circ \mathtt{f}_\alpha$ average JSD / byte	$p_{\mathtt{llama1B}} \circ \mathtt{f}_\alpha$ bytes / sec
1e-1	7.8e-2 (7.4e-2, 8.1e-2)	22.02 (13.88, 36.15)	6.7e-2 (6.3e-2, 7.0e-2)	38.76 (30.81, 48.63)
3e-2	7.4e-2 (7.0e-2, 7.7e-2)	22.31 (12.72, 41.71)	6.1e-2 (5.8e-2, 6.4e-2)	39.73 (30.65, 51.48)
1e-2	6.0e-2 (5.7e-2, 6.3e-2)	26.08 (13.91, 51.18)	5.4e-2 (5.1e-2, 5.6e-2)	25.61 (15.06, 45.38)
3e-3	3.0e-2 (2.8e-2, 3.2e-2)	27.67 (17.50, 49.67)	3.3e-2 (3.1e-2, 3.5e-2)	42.08 (29.25, 59.36)
1e-3	1.5e-2 (1.3e-2, 1.7e-2)	29.60 (18.88, 51.40)	1.5e-2 (1.4e-2, 1.7e-2)	46.25 (29.10, 75.78)
3e-4	6.2e-3 (5.2e-3, 7.2e-3)	21.91 (13.66, 37.44)	5.4e-3 (4.4e-3, 6.4e-3)	49.40 (40.14, 59.56)
1e-4	3.5e-3 (2.7e-3, 4.4e-3)	26.10 (24.85, 27.39)	2.3e-3 (1.7e-3, 2.9e-3)	37.42 (29.84, 43.84)
3e-5	5.8e-4 (3.4e-4, 9.3e-4)	15.33 (14.69, 16.03)	6.0e-4 (4.1e-4, 8.4e-4)	23.54 (22.37, 24.84)
1e-5	5.0e-4 (2.3e-4, 8.7e-4)	9.66 (9.25, 10.09)	7.1e-5 (4.2e-5, 1.1e-4)	16.62 (15.83, 17.42)
3e-6	4.5e-4 (9.1e-5, 9.4e-4)	6.10 (5.85, 6.39)	1.7e-5 (7.2e-6, 3.2e-5)	10.40 (9.97, 10.86)
1e-6	(not applicable)	2.52 (2.41, 2.64)	(not applicable)	5.41 (5.19, 5.65)

τ	$p_{\mathtt{llama8B}} \circ \mathtt{f}_\alpha$ average JSD / byte	$p_{\mathtt{llama8B}} \circ \mathtt{f}_\alpha$ bytes / sec
1e-1	5.2e-2 (5.0e-2, 5.5e-2)	42.87 (32.92, 57.03)
3e-2	4.8e-2 (4.5e-2, 5.0e-2)	41.40 (30.70, 54.70)
1e-2	4.3e-2 (4.1e-2, 4.5e-2)	34.24 (19.37, 56.79)
3e-3	2.9e-2 (2.8e-2, 3.1e-2)	30.55 (20.06, 48.56)
1e-3	1.1e-2 (1.0e-2, 1.3e-2)	35.00 (20.96, 56.28)
3e-4	4.1e-3 (3.4e-3, 5.0e-3)	25.95 (12.88, 53.91)
1e-4	1.2e-3 (8.8e-4, 1.6e-3)	40.15 (38.14, 42.33)
3e-5	6.3e-4 (3.1e-4, 9.9e-4)	25.70 (22.68, 27.97)
1e-5	1.9e-4 (5.5e-5, 4.3e-4)	18.43 (17.64, 19.30)
3e-6	1.0e-5 (6.8e-6, 1.4e-5)	12.35 (11.83, 12.84)
1e-6	(not applicable)	7.77 (7.45, 8.12)

Table 8: Average Jensen–Shannon distance (JSD) and bytes per second for various thresholds τ and a reference (τ =1e-6) using $p_{\mathcal{X}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$. 95% confidence intervals are given in parentheses.

τ	$p_{\mathsf{gpt2}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$ average JSD / byte	$p_{\mathsf{gpt2}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$ bytes / sec	$p_{\mathsf{llama1B}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$ average JSD / byte	$p_{\mathsf{llama1B}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$ bytes / sec
1e-1	5.6e-3 (5.1e-3, 6.2e-3)	32.91 (31.78, 33.98)	5.1e-3 (4.5e-3, 5.7e-3)	33.08 (21.15, 58.03)
3e-2	1.9e-3 (1.7e-3, 2.2e-3)	30.74 (29.98, 31.50)	1.9e-3 (1.5e-3, 2.3e-3)	53.87 (52.51, 55.42)
1e-2	8.8e-4 (7.0e-4, 1.2e-3)	30.29 (29.51, 31.12)	8.0e-4 (6.1e-4, 1.1e-3)	49.09 (47.80, 50.42)
3e-3	3.4e-4 (2.7e-4, 4.7e-4)	27.67 (26.95, 28.37)	3.1e-4 (2.3e-4, 4.7e-4)	45.28 (44.07, 46.50)
1e-3	1.7e-4 (1.0e-4, 3.0e-4)	28.01 (27.20, 28.91)	1.4e-4 (9.4e-5, 2.2e-4)	37.22 (35.44, 38.84)
3e-4	7.4e-5 (3.5e-5, 1.5e-4)	26.29 (25.50, 27.00)	5.5e-5 (3.3e-5, 9.3e-5)	36.42 (35.57, 37.32)
1e-4	1.6e-5 (1.3e-5, 2.1e-5)	22.34 (21.69, 22.95)	1.7e-5 (1.4e-5, 2.2e-5)	28.42 (27.58, 29.20)
3e-5	6.2e-6 (4.6e-6, 9.0e-6)	16.19 (15.50, 16.83)	7.5e-6 (5.7e-6, 1.0e-5)	16.65 (16.00, 17.40)
1e-5	2.0e-6 (1.7e-6, 2.5e-6)	9.59 (9.02, 10.21)	2.8e-6 (2.5e-6, 3.3e-6)	11.79 (11.12, 12.45)
3e-6	6.7e-7 (6.1e-7, 7.7e-7)	4.20 (3.78, 4.63)	1.4e-6 (1.2e-6, 1.8e-6)	7.06 (6.63, 7.53)
1e-6	(not applicable)	2.52 (2.32, 2.77)	(not applicable)	4.38 (4.07, 4.72)

τ	$p_{\mathsf{llama8B}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$ average JSD / byte	$p_{\mathsf{llama8B}} \circ \mathsf{f}_\alpha \circ \mathsf{f}_{\mathrm{ptb}}$ bytes / sec
1e-1	4.3e-3 (3.8e-3, 5.0e-3)	30.37 (29.48, 31.19)
3e-2	1.6e-3 (1.3e-3, 2.0e-3)	27.92 (27.25, 28.64)
1e-2	5.9e-4 (5.1e-4, 7.4e-4)	26.14 (25.50, 26.81)
3e-3	2.8e-4 (2.0e-4, 4.1e-4)	24.23 (23.64, 24.88)
1e-3	1.5e-4 (8.4e-5, 2.8e-4)	21.21 (20.68, 21.79)
3e-4	5.3e-5 (3.1e-5, 9.4e-5)	20.19 (19.71, 20.69)
1e-4	1.6e-5 (1.2e-5, 2.4e-5)	18.54 (18.12, 18.96)
3e-5	5.6e-6 (4.4e-6, 7.5e-6)	14.33 (13.93, 14.74)
1e-5	2.6e-6 (1.9e-6, 3.9e-6)	10.33 (9.92, 10.74)
3e-6	8.3e-7 (7.4e-7, 9.6e-7)	6.40 (6.02, 6.83)
1e-6	(not applicable)	4.41 (4.10, 4.72)

Table 9: Average Jensen–Shannon distance (JSD) and bytes per second for various thresholds τ and a reference (τ =1e-6) using $p_{\mathcal{X}} \circ \mathsf{f}_{\mathrm{dna2aa}}$. 95% confidence intervals are given in parentheses. We limit the size of the candidate set to mitigate the combinatorial blow-up with increasing sequence length.

	$p_{\mathsf{dna}} \circ \mathsf{f}_{\mathrm{dna2aa}}$ ($n_{\max}$ = 5000)		$p_{\mathsf{dna}} \circ \mathsf{f}_{\mathrm{dna2aa}}$ ($n_{\max}$ = 10000)	
τ	average JSD / byte	byte / sec	average JSD / byte	byte / sec
1e-1	4.9e-3 (3.9e-3, 5.9e-3)	24.31 (21.16, 28.25)	5.1e-3 (4.0e-3, 6.3e-3)	44.33 (38.29, 52.75)
3e-2	2.0e-3 (1.4e-3, 2.6e-3)	24.04 (18.51, 32.81)	2.1e-3 (1.5e-3, 2.8e-3)	23.45 (17.86, 31.51)
1e-2	8.4e-5 (5.9e-5, 1.1e-4)	12.55 (8.83, 19.52)	1.1e-4 (8.2e-5, 1.5e-4)	12.07 (8.64, 18.96)
3e-3	1.4e-5 (9.1e-6, 1.9e-5)	10.98 (7.68, 17.42)	1.4e-5 (9.2e-6, 2.0e-5)	8.51 (5.63, 15.35)
1e-3	4.0e-6 (2.3e-6, 5.9e-6)	10.57 (7.46, 15.76)	3.9e-6 (2.3e-6, 5.9e-6)	8.11 (5.39, 14.43)
3e-4	2.8e-7 (1.7e-7, 4.2e-7)	10.34 (7.29, 15.85)	2.3e-7 (1.3e-7, 3.6e-7)	7.84 (5.22, 14.16)
1e-4	7.6e-8 (3.0e-8, 1.3e-7)	10.39 (7.52, 15.90)	3.2e-8 (1.7e-8, 4.9e-8)	8.24 (5.48, 14.22)
3e-5	2.1e-8 (5.0e-9, 4.5e-8)	10.29 (7.47, 16.18)	1.3e-8 (5.1e-9, 2.1e-8)	8.44 (5.62, 14.41)
1e-5	4.2e-9 (9.2e-10, 8.3e-9)	10.30 (7.40, 15.89)	4.2e-9 (9.2e-10, 8.5e-9)	8.48 (5.82, 14.70)
3e-6	3.5e-10 (0.0e+00, 9.0e-10)	10.36 (7.34, 16.22)	3.5e-10 (0.0e+00, 9.0e-10)	8.50 (5.70, 14.98)
1e-6	(not applicable)	1.17 (1.06, 1.31)	(not applicable)	0.70 (0.62, 0.78)

	$p_{\mathsf{dna}} \circ \mathsf{f}_{\mathrm{dna2aa}}$ ($n_{\max}$ = 15000)		$p_{\mathsf{dna}} \circ \mathsf{f}_{\mathrm{dna2aa}}$ ($n_{\max}$ = 20000)	
τ	average JSD / byte	byte / sec	average JSD / byte	byte / sec
1e-1	5.2e-3 (4.2e-3, 6.2e-3)	47.07 (40.13, 54.66)	5.2e-3 (4.1e-3, 6.3e-3)	47.99 (41.76, 56.14)
3e-2	2.2e-3 (1.6e-3, 2.8e-3)	23.67 (18.30, 32.60)	2.2e-3 (1.6e-3, 2.9e-3)	25.89 (20.38, 35.54)
1e-2	1.3e-4 (8.9e-5, 1.8e-4)	12.44 (8.65, 19.50)	1.4e-4 (8.9e-5, 1.9e-4)	12.82 (9.09, 19.69)
3e-3	1.6e-5 (1.1e-5, 2.2e-5)	8.66 (5.60, 15.33)	1.8e-5 (1.2e-5, 2.3e-5)	8.94 (5.77, 15.76)
1e-3	4.2e-6 (2.6e-6, 5.8e-6)	5.90 (3.51, 12.80)	5.1e-6 (3.3e-6, 7.1e-6)	5.88 (3.59, 13.27)
3e-4	2.3e-7 (1.3e-7, 3.4e-7)	5.07 (3.05, 12.07)	3.2e-7 (1.8e-7, 4.7e-7)	4.91 (2.96, 12.98)
1e-4	2.5e-8 (1.3e-8, 4.1e-8)	4.77 (2.84, 11.74)	3.1e-8 (1.6e-8, 4.8e-8)	4.57 (2.65, 10.69)
3e-5	1.1e-8 (3.7e-9, 1.8e-8)	4.73 (2.85, 10.47)	1.5e-8 (7.2e-9, 2.3e-8)	4.42 (2.52, 10.83)
1e-5	4.2e-9 (9.2e-10, 8.3e-9)	4.57 (2.53, 10.82)	6.1e-9 (2.2e-9, 1.1e-8)	4.18 (2.47, 10.06)
3e-6	3.5e-10 (0.0e+00, 9.0e-10)	4.51 (2.59, 11.05)	3.5e-10 (0.0e+00, 9.0e-10)	4.05 (2.27, 9.51)
1e-6	(not applicable)	0.47 (0.42, 0.52)	(not applicable)	0.39 (0.35, 0.44)

L.2 Benchmarking the Quotient

Table 10: Average Jensen–Shannon distance (JSD) for τ =1e-4 after randomly converting n% of the universal states to non-universal.

	$p_{\mathsf{gpt2}} \circ \mathsf{f}_{\alpha}$		$p_{\mathsf{llama1B}} \circ \mathsf{f}_{\alpha}$	
% Converted	Converted States	average JSD / byte	Converted States	average JSD / byte
0	0	(not applicable)	0	(not applicable)
5	3786	3.4e-3 (2.2e-3, 4.9e-3)	8849	1.8e-2 (1.4e-2, 2.2e-2)
10	7572	2.1e-2 (1.7e-2, 2.5e-2)	17699	1.4e-2 (1.1e-2, 1.7e-2)
15	11358	2.9e-2 (2.5e-2, 3.4e-2)	26548	1.6e-2 (1.3e-2, 1.9e-2)
20	15144	5.3e-2 (4.7e-2, 6.0e-2)	35398	5.1e-2 (4.5e-2, 5.7e-2)
25	18930	5.2e-2 (4.6e-2, 5.8e-2)	44247	6.4e-2 (5.7e-2, 7.1e-2)

	$p_{\mathsf{gpt2}} \circ \mathsf{f}_{\alpha} \circ \mathsf{f}_{\mathrm{ptb}}$		$p_{\mathsf{llama1B}} \circ \mathsf{f}_{\alpha} \circ \mathsf{f}_{\mathrm{ptb}}$	
% Converted	Converted States	average JSD / byte	Converted States	average JSD / byte
0	0	(not applicable)	0	(not applicable)
5	1	2.7e-7 (2.2e-7, 3.3e-7)	1	9.0e-7 (6.7e-7, 1.3e-6)
10	3	5.8e-5 (1.4e-5, 1.1e-4)	3	3.6e-5 (1.3e-5, 6.7e-5)
15	4	1.1e-2 (9.9e-3, 1.2e-2)	4	9.3e-6 (1.9e-6, 2.0e-5)
20	6	1.1e-2 (1.0e-2, 1.3e-2)	6	1.1e-3 (5.5e-4, 1.9e-3)
25	7	-	7	1.1e-2 (9.9e-3, 1.3e-2)

L.3 Cross-Entropy

We repeat the experiments from §6 in Tab. 11, this time evaluating the cross-entropy loss using the same dataset. Because this metric does not require computing the full distribution over the next

symbol, we observe a large speedup. Many applications rely on calculating sequence or prefix probabilities; these numbers are indicative of the performance and accuracy trade-offs in such settings. The JSD-numbers, on the other hand, correspond to the time it would take to do full decoding.

Table 11: Average Cross-Entropy for various thresholds τ.

τ	Bytes/s		Bits/byte		Cross-entropy	
	Mean	95% CI	Mean	95% CI	Mean	95% CI
$p_{\text{gpt2}} \circ \mathtt{f}_\alpha$						
1e-1	54.60	(56.23, 108.78)	1.2273	(1.1322, 1.2894)	0.8507	(0.7860, 0.8917)
3e-2	73.70	(74.99, 125.08)	1.2470	(1.1694, 1.2966)	0.8644	(0.8088, 0.8992)
1e-2	83.40	(79.40, 114.86)	1.1701	(1.1085, 1.2092)	0.8110	(0.7652, 0.8383)
3e-3	96.50	(91.86, 128.42)	1.1130	(1.0652, 1.1500)	0.7715	(0.7388, 0.7979)
1e-3	81.90	(77.90, 96.14)	1.0782	(1.0258, 1.1223)	0.7473	(0.7120, 0.7777)
3e-4	49.80	(47.50, 54.26)	1.0456	(0.9963, 1.1039)	0.7248	(0.6924, 0.7611)
1e-4	19.60	(24.42, 34.33)	1.0288	(0.9877, 1.0768)	0.7131	(0.6822, 0.7477)
3e-5	17.30	(16.15, 19.62)	1.0129	(0.9757, 1.0562)	0.7021	(0.6756, 0.7326)
1e-5	11.10	(10.03, 13.28)	1.0127	(0.9771, 1.0503)	0.7020	(0.6760, 0.7280)
3e-6	6.90	(6.24, 9.27)	1.0162	(0.9811, 1.0584)	0.7044	(0.6790, 0.7355)
$p_{\text{llama1B}} \circ \mathtt{f}_\alpha$						
1e-1	116.70	(109.18, 165.89)	1.0595	(0.9453, 1.1047)	0.7344	(0.6614, 0.7690)
3e-2	155.10	(145.12, 203.50)	1.0858	(0.9770, 1.1307)	0.7526	(0.6743, 0.7896)
1e-2	142.30	(134.66, 202.59)	1.0458	(0.9916, 1.0799)	0.7249	(0.6888, 0.7524)
3e-3	165.00	(157.14, 201.86)	0.9672	(0.8978, 1.0067)	0.6704	(0.6270, 0.6982)
1e-3	150.50	(144.11, 184.43)	0.9057	(0.8445, 0.9413)	0.6278	(0.5891, 0.6543)
3e-4	87.60	(83.48, 102.28)	0.8564	(0.8101, 0.8895)	0.5936	(0.5653, 0.6178)
1e-4	48.70	(46.29, 57.93)	0.8415	(0.7961, 0.8678)	0.5833	(0.5526, 0.6025)
3e-5	27.40	(25.57, 33.84)	0.8394	(0.7942, 0.8688)	0.5818	(0.5530, 0.6037)
1e-5	19.60	(18.68, 21.82)	0.8388	(0.7949, 0.8617)	0.5814	(0.5535, 0.6001)
3e-6	12.50	(11.85, 13.98)	0.8353	(0.7901, 0.8633)	0.5790	(0.5488, 0.6004)
$p_{\text{llama8B}} \circ \mathtt{f}_\alpha$						
1e-1	83.90	(79.37, 120.86)	0.8637	(0.6849, 0.9268)	0.5987	(0.4796, 0.6442)
3e-2	100.90	(94.98, 128.25)	0.8644	(0.6925, 0.9279)	0.5991	(0.4736, 0.6414)
1e-2	117.10	(111.90, 135.80)	0.8398	(0.6890, 0.9044)	0.5821	(0.4710, 0.6243)
3e-3	104.80	(99.09, 125.66)	0.8171	(0.6604, 0.8765)	0.5664	(0.4726, 0.6054)
1e-3	101.90	(97.85, 122.57)	0.7325	(0.6059, 0.7772)	0.5077	(0.4242, 0.5372)
3e-4	72.90	(68.89, 90.38)	0.7067	(0.5866, 0.7583)	0.4899	(0.4114, 0.5238)
1e-4	45.10	(41.32, 60.47)	0.7001	(0.5738, 0.7446)	0.4852	(0.4081, 0.5182)
3e-5	30.50	(28.59, 38.91)	0.6944	(0.5821, 0.7404)	0.4813	(0.4024, 0.5118)
1e-5	20.60	(19.15, 26.25)	0.6928	(0.5795, 0.7415)	0.4802	(0.4023, 0.5095)
3e-6	14.20	(13.62, 16.96)	0.6911	(0.5796, 0.7383)	0.4790	(0.4010, 0.5093)

L.4 Fine-Tuning vs. Transducing

To further benchmark our approach, we compare transduction against fine-tuning, a standard method for adapting pretrained language models to new data. We select a simple transformation: converting a pretrained model into one over lowercased strings, using the transducer depicted in Fig. 1. Specifically, we evaluate four variants of p_{gpt2}: (i) the original model, (ii) a fine-tuned model trained on 50M tokens of lowercased data sampled from GPT2-large until validation loss increased, (iii) a byte-adapted model where the embedding matrix is replaced to operate directly on the 256 bytes and special symbols, and (iv) our transduced model applied at the byte level. We use sampled data to prevent overfitting on a given domain during the fine-tuning.

We evaluate these models on 100 paragraphs (24,925 bytes) from a dataset of recent Wikipedia articles (written in the last three years), which are known not to be part of the training data and report the perplexity in bits per byte, with 95% confidence intervals obtained by bootstrapping. Fine-tuning was performed on a single H100 GPU for approximately a day, depending on early stopping. We note that we do not conduct a comprehensive hyper-parameter search but select what we deem to be

reasonable hyperparameters. We use a warm-up of 2000 steps, a learning rate of 5e-5, a batch size of 32, a context window of 1024 tokens, and evaluate every 100 steps with early stopping patience set to 5. The directly fine-tuned model took around 4 epochs, while the byte-adapted model with the new embedding matrix was still improving minimally. After 24 hours of training, it was stopped since the performance was very far from the fine-tuned model and the transduced model. The results in Tab. 12 show that our transducer-based approach yields the highest performance. All pairwise differences are significant at $p < 0.001$. Even if the byte-adaptation might get close to the fine-tuned model over weeks or months of training, this serves as an example of why transducing a model directly is so lucrative. Transduction does not require training, nor hyperparameter tuning and the results may still be better.

Table 12: Mean bits/byte with 95% confidence intervals.

Run	Mean bits/byte $\pm$ CI (95%)
Baseline	1.22 ± 0.03
Finetuned	1.14 ± 0.03
ByteAdapted	1.59 ± 0.05
Transduced (ours)	1.02 ± 0.04