Deep Learning

Tokenização em Modelos de Linguagem

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

Visão Geral

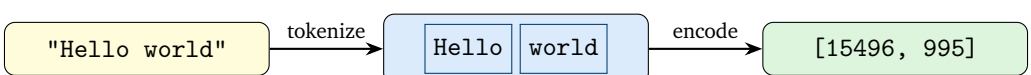
1. **Por que a Tokenização Importa**
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

O que é Tokenização?

Tokenização é o processo de converter texto bruto em uma sequência de unidades discretas (**tokens**) que um modelo pode processar.

O tokenizer define:

1. O **vocabulário**: quais tokens existem
2. A **segmentação**: como o texto é dividido em tokens
3. O **mapeamento**: tokens ↔ IDs inteiros



Por que isso importa?

A tokenização **molda silenciosamente** tudo que vem a seguir:

Problemas causados pela tokenização

- O GPT não consegue contar letras de forma confiável em `"strawberry"`
- Desigualdade multilíngue: o inglês usa menos tokens que outros idiomas
- `"123 + 456"` pode ser dividido de forma inconsistente entre os dígitos

Trade-offs de design

- Vocabulário pequeno → sequências longas
- Vocabulário grande → mais parâmetros e tokens raros subtreinados
- Dificilmente encontraremos uma "granularidade correta"

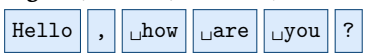
O tokenizer não é um detalhe de pré-processamento; é uma decisão arquitetural.

O Problema da Fertility

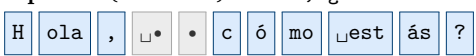
Fertility = número de tokens necessários para representar uma palavra ou frase.

Segmentações reais do GPT-2 para a mesma saudação em três idiomas:

Inglês (6 tokens): `"Hello, how are you?"`



Espanhol (11 tokens): `"Hola, ¿cómo estás?"`



Japonês (19 tokens): こんにちは、お元気ですか？

␣ indica espaço embutido no token; as caixas cinza (•) são fragmentos de bytes de caracteres sem token próprio (¿ e parte dos caracteres japoneses); veremos o porquê no byte-level BPE.

Consequência

Maior fertility = mais tokens = mais custo computacional, janela de contexto efetiva menor, e frequentemente pior desempenho para esse idioma.

Visão Geral

1. Por que a Tokenização Importa
2. **Abordagens Ingênuas**
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

Nosso Exemplo Recorrente

Nas seções de BPE e WordPiece usamos o mesmo **corpus de treinamento**:

Palavra	Frequência no corpus
`low`	5
`lower`	2
`newest`	6
`widest`	3

Por que as frequências importam

Um tokenizer é **aprendido** a partir de um corpus. A frequência de cada palavra determina quais tokens são adicionados ao vocabulário.

Tokenização em Nível de Caractere

Ideia: Cada caractere é um token.

Exemplo Recorrente

`"low lower newest widest"`

→ | l | o | w | _ | l | o | w | e | r | _ | n | e | w | e | s | t | _ | w | i | d | e | s | t |

Vocabulário: { `l, o, w, _, e, r, n, s, t, i, d` } $|\mathcal{V}| = 11$

✓ Vantagens

- Vocabulário reduzido (~256 para ASCII)
- Sem palavras fora do vocabulário (OOV)
- Agnóstico ao idioma

× Desvantagens

- Sequências muito longas
- Perde a semântica em nível de palavra
- O modelo precisa “aprender” a ortografia

Tokenização em Nível de Palavra

Ideia: Dividir por espaço em branco (e pontuação). Cada palavra é um token.

Exemplo Recorrente

`"low lower newest widest"`

→ `low` | `lower` | `newest` | `widest`

Vocabulário: { `low, lower, newest, widest` } $|\mathcal{V}| = 4$

✓ **Vantagens**

- Sequências curtas
- Tokens carregam significado semântico
- Simples de implementar

× **Desvantagens**

- Vocabulário gigantesco (100K+)
- Não lida com palavras OOV
- Sem compartilhamento morfológico: `low` e `lowest` são tokens sem relação

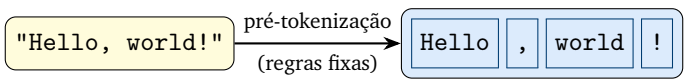
O mais utilizado é: **Tokenização por subwords**

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. **Pré-tokenização**
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

Pré-tokenização: O Passo Zero

Antes de qualquer algoritmo de subword, o texto é dividido em **pré-tokens** por regras fixas (espaço, pontuação ou uma regex). A tokenização opera **dentro** dessas fronteiras.



Consequências

- **Merges nunca cruzam fronteiras**: nenhum token pode unir duas palavras
- **Treinamento tratável**: o corpus vira uma tabela de palavras com frequências, exatamente o formato do nosso exemplo recorrente

O SentencePiece dispensa esta etapa tratando o espaço como caractere comum; veremos na seção de frameworks.

De Quem É o Espaço?

Dividir por espaço **descarta** o espaço. Para a tokenização ser **reversível**, cada convenção guarda a fronteira de um jeito:

Convenção	`"the cat"`	**Usada por**
Espaço embutido no token seguinte	`the` `␣cat`	GPT-2, tiktoken
Espaço vira o caractere _	`_the` `_cat`	SentencePiece
Marcador de fim de palavra	`the_` `cat_`	BPE original (nosso exemplo)
Espaço descartado (não reversível)	`the` `cat`	BERT (WordPiece)

Consequência: `"world"` ≠ `"␣world"`

São tokens distintos, com IDs e embeddings distintos. No primeiro slide da aula, `"Hello world"` virou `[15496, 995]`: o ID 995 já embute o espaço. A mesma palavra no início e no meio da frase recebe representações diferentes.

Internamente o GPT-2 grafa o espaço embutido como Ġ (ex.: `Ġworld`); veremos o porquê no byte-level BPE.

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. **Byte Pair Encoding (BPE)**
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

BPE

Byte Pair Encoding (Sennrich et al., 2016) mescla iterativamente o par de símbolos adjacentes mais frequente.

Algoritmo

1. Começa com um vocabulário de todos os caracteres individuais
2. Conta todos os pares de símbolos adjacentes no corpus
3. Mescla o par mais frequente em um novo símbolo
4. Repete os passos 2–3 por k merges (hiperparâmetro)

Propriedades principais:

- Bottom-up, guloso, baseado em frequência
- A tabela de merges é aprendida a partir de um corpus de treinamento
- Codificação determinística na inferência (aplica os merges na ordem aprendida)

Utilizado em: GPT, GPT-2, GPT-3, RoBERTa, CodeGen, …

BPE: Configuração do Exemplo Simplificado

Corpus já pré-tokenizado (frequências de palavras e marcador de fim de palavra _):

Palavra	**Freq**	**Divisão inicial**
`low_`	5	`l` `o` `w` `_`
`lower_`	2	`l` `o` `w` `e` `r` `_`
`newest_`	6	`n` `e` `w` `e` `s` `t` `_`
`widest_`	3	`w` `i` `d` `e` `s` `t` `_`

Vocabulário inicial: $\mathcal{V} = \{$ `l, o, w, _, e, r, n, s, t, i, d` $\}$ $\quad |\mathcal{V}| = 11$

Vamos executar os $k = 7$ merges passo a passo…

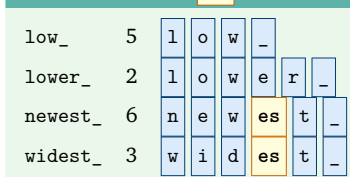
BPE: Merge Passo 1

Contagem de todos os pares adjacentes:

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(e, s)	**9**	(l, o)	7	(o, w)	7	(s, t)	9
(e, w)	6	(w, _)	5	(t, _)	9	(e, r)	2
(w, e)	8	(n, e)	6	(w, i)	3	(d, e)	3
(i, d)	3	(r, _)	2				

Mais frequentes: **(e, s)**, **(s, t)** e **(t, _)** têm contagem 9.
(Desempate: escolher (e, s) primeiro.)

Merge 1: (e, s) → **es** (contagem: 9)



$\mathcal{V} = \{$ `l, o, w, _, e, r, n, s, t, i, d, es` $\}$ $\quad |\mathcal{V}| = 12$

BPE: Merge Passo 2

Contagem dos pares adjacentes (após o merge 1):

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(es, t)	**9**	(t, _)	9	(l, o)	7	(o, w)	7
(n, e)	6	(e, w)	6	(w, es)	6	(w, _)	5
(w, i)	3	(i, d)	3	(d, es)	3	(w, e)	2
(e, r)	2	(r, _)	2				

Mais frequentes: **(es, t)** e **(t, _)** têm contagem 9.
(Desempate: escolher (es, t).)

Merge 2: (es, t) → `est` (contagem: 9)

`low_`	5	`l` `o` `w` `_`
`lower_`	2	`l` `o` `w` `e` `r` `_`
`newest_`	6	`n` `e` `w` `est` `_`
`widest_`	3	`w` `i` `d` `est` `_`

$\mathcal{V} = \{$ `l`, `o`, `w`, `_`, `e`, `r`, `n`, `s`, `t`, `i`, `d`, `es`, `est` $\}$ $\quad |\mathcal{V}| = 13$

BPE: Merge Passo 3

Contagem dos pares adjacentes (após o merge 2):

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(est, _)	**9**	(l, o)	7	(o, w)	7	(n, e)	6
(e, w)	6	(w, est)	6	(w, _)	5	(w, i)	3
(i, d)	3	(d, est)	3	(w, e)	2	(e, r)	2
(r, _)	2						

Mais frequente: **(est, _)**, único par com contagem 9.

Merge 3: (est, _) → **est_** (contagem: 9)

low_	5	l	o	w	_		
lower_	2	l	o	w	e	r	_
newest_	6	n	e	w	est_		
widest_	3	w	i	d	est_		

$\mathcal{V} = \{$ l, o, w, _, e, r, n, s, t, i, d, es, est, **est_** $\}$ $|\mathcal{V}| = 14$

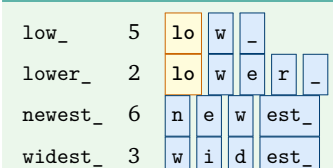
BPE: Merge Passo 4

Contagem dos pares adjacentes (após o merge 3):

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(l, o)	**7**	(o, w)	7	(n, e)	6	(e, w)	6
(w, est_)	6	(w, _)	5	(w, i)	3	(i, d)	3
(d, est_)	3	(w, e)	2	(e, r)	2	(r, _)	2

Mais frequentes: **(l, o)** e **(o, w)** têm contagem 7.
(Desempate: escolher (l, o).)

Merge 4: (l, o) → `lo` (contagem: 7)



$\mathcal{V} = \{$ `l, o, w, _, e, r, n, s, t, i, d, es, est, est_, lo` $\}$ $\quad |\mathcal{V}| = 15$

BPE: Merge Passo 5

Contagem dos pares adjacentes (após o merge 4):

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(lo, w)	**7**	(n, e)	6	(e, w)	6	(w, est_)	6
(w, _)	5	(w, i)	3	(i, d)	3	(d, est_)	3
(w, e)	2	(e, r)	2	(r, _)	2		

Mais frequente: **(lo, w)**, único par com contagem 7.

Merge 5: (lo, w) → `low` (contagem: 7)

`low_`	5	`low` `_`
`lower_`	2	`low` `e` `r` `_`
`newest_`	6	`n` `e` `w` `est_`
`widest_`	3	`w` `i` `d` `est_`

$\mathcal{V} = \{$ `l`, `o`, `w`, `_`, `e`, `r`, `n`, `s`, `t`, `i`, `d`, `es`, `est`, `est_`, `lo`, `low` $\}$ $\quad |\mathcal{V}| = 16$

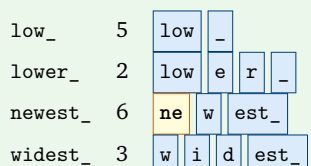
BPE: Merge Passo 6

Contagem dos pares adjacentes (após o merge 5):

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(n, e)	**6**	(e, w)	6	(w, est_)	6	(low, _)	5
(w, i)	3	(i, d)	3	(d, est_)	3	(low, e)	2
(e, r)	2	(r, _)	2				

Mais frequentes: **(n, e)**, **(e, w)** e **(w, est_)** têm contagem 6.
(Desempate: escolher (n, e).)

Merge 6: (n, e) → **ne** (contagem: 6)



$\mathcal{V} = \{$ `l, o, w, _, e, r, n, s, t, i, d, es, est, est_, lo, low, ne` $\}$ $\quad |\mathcal{V}| = 17$

BPE: Merge Passo 7

Contagem dos pares adjacentes (após o merge 6):

Par	Contagem	Par	Contagem	Par	Contagem	Par	Contagem
(ne, w)	**6**	(w, est_)	6	(low, _)	5	(w, i)	3
(i, d)	3	(d, est_)	3	(low, e)	2	(e, r)	2
(r, _)	2						

Mais frequentes: **(ne, w)** e **(w, est_)** têm contagem 6.
(Desempate: escolher (ne, w).)

Merge 7: (ne, w) → `new` (contagem: 6)

low_	5	low _
lower_	2	low e r _
newest_	6	new est_
widest_	3	w i d est_

$\mathcal{V}$ = { `l, o, w, _, e, r, n, s, t, i, d, es, est, est_, lo, low, ne, new` } $|\mathcal{V}| = 18$

Após os $k = 7$ merges, observe como `est_` é compartilhado entre “newest” e “widest”!

BPE: Codificando Novo Texto

Na inferência, aplica-se os merges aprendidos **em ordem**:

Codificando a palavra "lowest"

1. Início: `l` `o` `w` `e` `s` `t` `_`
2. Aplicar merge 1 (e,s)→es: `l` `o` `w` `es` `t` `_`
3. Aplicar merge 2 (es,t)→est: `l` `o` `w` `est` `_`
4. Aplicar merge 3 (est,_)→est_: `l` `o` `w` `est_`
5. Aplicar merge 4 (l,o)→lo: `lo` `w` `est_`
6. Aplicar merge 5 (lo,w)→low: `low` `est_`

Resultado: `low` `est_` compartilha subwords com "low", "newest", "widest"!

Mesmo que "lowest" **nunca tenha aparecido no corpus de treinamento**, o modelo consegue compô-la a partir de subwords aprendidos.

Byte-Level BPE: Motivação

Problema: O BPE padrão opera sobre caracteres Unicode. Mas o Unicode define 150K+ caracteres, cada um com um ID numérico (ponto de código); o vocabulário base é enorme, e scripts raros geram muitos tokens `<unk>`.

Solução: Byte-Level BPE (Radford et al., 2019, GPT-2)

- Opera sobre **bytes brutos** (sempre 256 tokens base)
- Todo texto pode ser codificado: **nunca há tokens desconhecidos**
- Em seguida, aplica-se os merges padrão do BPE sobre os bytes

Exemplo

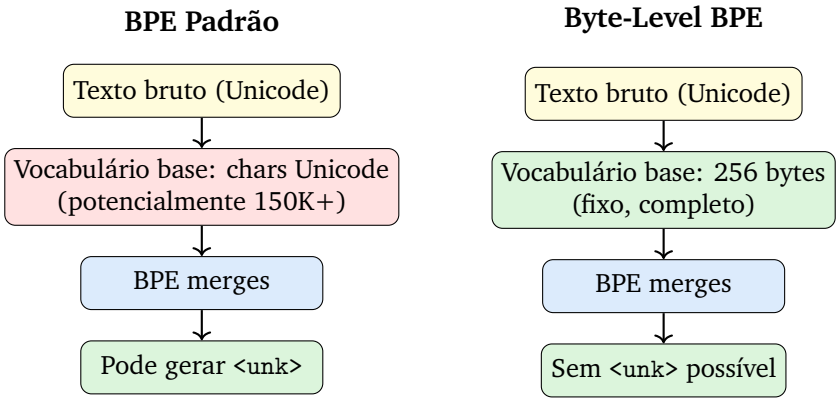
O emoji caveira "💀" (U+1F480) ocupa 4 bytes em UTF-8, `0xF0 0x9F 0x92 0x80`:

→ `0xF0` `0x9F` `0x92` `0x80` (4 tokens de byte)

Mesmo fora do vocabulário, ele **nunca vira `<unk>`**: todo caractere é representável byte a byte, e bytes frequentes ainda podem ser mesclados pelo BPE.

Utilizado em: GPT-2, GPT-3, GPT-4, Codex, LLaMA 3, …

Byte-Level BPE vs. BPE Padrão



O GPT-2 usa um **mapeamento byte-para-unicode**: cada byte é representado como um caractere Unicode imprimível, de modo que o código BPE baseado em strings existente funciona sem modificações.

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. **WordPiece**
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

WordPiece

WordPiece (Schuster & Nakajima, 2012; Wu et al., 2016) é similar ao BPE, mas seleciona merges com base na **likelihood** em vez da frequência.

Diferença principal em relação ao BPE

Em vez de mesclar o par mais *frequente*, o WordPiece mescla o par que **maximiza o ganho de likelihood dos dados de treinamento**. O paper descreve apenas esse critério, sem fórmula fechada nem implementação oficial; a expressão abaixo é a **reconstrução popularizada pelo Hugging Face**:

$$\text{score}(x, y) = \frac{\text{freq}(xy)}{\text{freq}(x) \cdot \text{freq}(y)}$$

Isso favorece mesclar pares cuja combinação é muito mais provável do que o acaso.

Outras diferenças:

- Usa o prefixo `##` para tokens de continuação: "playing" → `play` `##ing`
- Longest-match guloso da esquerda para a direita na inferência

Utilizado em: BERT, DistilBERT, Electra, …

WordPiece: Exemplo Simplificado

Mesmo corpus, scores diferentes:

Par	freq(xy)	freq(x)·freq(y)	Score	
(e, s)	9	$17 \times 9 = 153$	0.0588	
(s, t)	9	$9 \times 9 = 81$	0.111	
(i, d)	3	$3 \times 3 = 9$	**0.333**	← maior
(n, e)	6	$6 \times 17 = 102$	0.0588	
(l, o)	7	$7 \times 7 = 49$	0.143	

O WordPiece mescla (i, d) primeiro, não (e, s)!

Intuição

(i, d) aparecem quase *exclusivamente* juntos → alta informação mútua.
(e, s) aparecem com frequência, mas também de forma independente → score menor.

WordPiece: Limitações do Score

Um par que **sempre** ocorre junto, com contagem n, recebe:

$$\text{score} = \frac{n}{n \cdot n} = \frac{1}{n}$$

O score mistura associação com uma penalidade inversa de frequência: entre dois pares de dependência perfeita, vence o mais **raro**.

Dependência perfeita, evidências diferentes

(e, s) com freq(`e`) = freq(`s`) = freq(`es`) = 1: score = 1.
(a, b) com freq(`a`) = freq(`b`) = freq(`ab`) = 3: score = 1/3.
O WordPiece mescla (e, s), observado uma única vez; o BPE mesclaria (a, b).

- O ganho real de likelihood sob um modelo unigrama é $\approx \text{freq}(xy) \cdot \ln(N \cdot \text{score})$, a associação **pesada pela frequência do par**: esse critério escolheria (a, b)
- O viés pró-raro é, portanto, um artefato da reconstrução, que descarta o peso $\text{freq}(xy)$
- Na prática, um corte de frequência mínima (`min_frequency` no `tokenizers` do Hugging Face) filtra pares raros antes do score

N é o número total de tokens do corpus. Normalizar o score por N não altera o ranking: a constante multiplica todos os pares igualmente.

WordPiece: Inferência com Longest Match

Na inferência, o WordPiece usa **longest-match-first guloso** da esquerda para a direita.

Exemplo: codificando "unhappiness" com um vocabulário estilo BERT

Suponha que o vocabulário contém: `un`, `happy`, `##happi`, `##ness`, `##ly`, ...

1. Varredura da esquerda: maior correspondência = `un`
2. Restante: "happiness". Maior correspondência = `##happi`
3. Restante: "ness". Maior correspondência = `##ness`

Resultado: `un` `##happi` `##ness`

- O prefixo `##` indica "este token é uma continuação da palavra anterior" (não é o início de uma palavra)
- Se nenhuma combinação do vocabulário cobre a palavra, ela **inteira** vira o token especial `[UNK]` e seu conteúdo se perde; BPE e Unigram, em contraste, sempre decompõem até caracteres ou bytes

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. **Unigram Language Model**
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

Unigram

Unigram LM (Kudo, 2018) adota a **abordagem oposta** à do BPE:

Algoritmo

1. Começa com um vocabulário inicial **grande** (ex.: todos os substrings do corpus)
2. Define um modelo de linguagem unigrama: $P(x) = \prod_{i=1}^{n} P(x_i)$
3. Estima as probabilidades $P(x_i)$ via **EM**
4. Para cada token, calcula o quanto a likelihood **cai** se ele for removido
5. Remove os tokens com a **menor perda** (menos úteis)
6. Repete os passos 3 a 5 até o vocabulário atingir o tamanho desejado

Diferença principal em relação ao BPE

BPE: começa pequeno, cresce (bottom-up)
Unigram: começa grande, poda (top-down)

Unigram: Exemplo Simplificado (1/6), Vocabulário Inicial

Corpus: “low” (freq 10), “er” (freq 10), “lower” (freq 1)

Usamos um corpus menor que o exemplo recorrente: o vocabulário inicial do Unigram contém **todos os substrings** do corpus, e as tabelas do exemplo anterior ficariam intratáveis num slide.

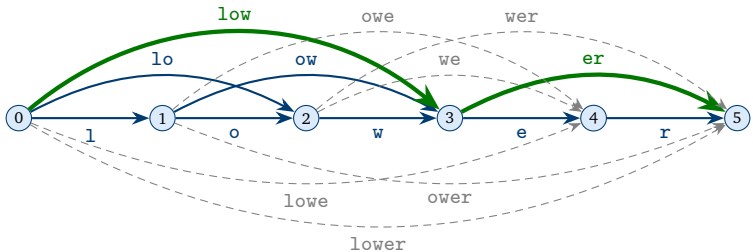
Passo 1: Começa com vocabulário de todos os substrings; $P(t) = \frac{\text{count}(t)}{N}$, onde $N = \sum_{t'} \text{count}(t')$:

Token	$P(\cdot)$	**Token**	$P(\cdot)$	**Token**	$P(\cdot)$
`low`	0.105	`lo`	0.105	`l`	0.105
`er`	0.105	`ow`	0.105	`o`	0.105
`lower`	0.010	`we`	0.010	`w`	0.105
`owe`	0.010	`wer`	0.010	`e`	0.105
`lowe`	0.010	`ower`	0.010	`r`	0.105

Substrings de “low” ou de “er” têm count = 11 (= 10 ocorrências da sua palavra +1 de “lower”); substrings exclusivas de “lower” têm count = 1; $N = 9{\times}11 + 6{\times}1 = 105$.

Unigram: Exemplo Simplificado (2/6), o Reticulado

Segmentar uma palavra é escolher um **caminho** do início ao fim num **reticulado**: cada aresta é um token do vocabulário, com peso $\ln P(\text{token})$.



Arestas cheias: tokens frequentes, $\ln P = -2.26$. Tracejadas: tokens que só ocorrem em "lower", $\ln P = -4.65$. Em **verde**, o melhor caminho: `low` `er`.

A probabilidade de uma segmentação é o **produto** dos P; em log, a **soma** dos pesos do caminho. "lower" admite $2^4 = 16$ caminhos possíveis.

Unigram: Exemplo Simplificado (3/6), Viterbi

A melhor segmentação é encontrada por **programação dinâmica**, o mesmo princípio do algoritmo de **Viterbi** em HMMs: $\delta(j)$ é o log da melhor segmentação do prefixo com j caracteres.

$$\delta(0) = 0, \qquad \delta(j) = \max_{\substack{i<j \\ s_{i:j} \in V}} \left[\delta(i) + \ln P(s_{i:j}) \right]$$

j	**Prefixo**	**Melhor segmentação**	$\delta(j)$	**Alternativa descartada**
1	`l`	`l`	−2.26	
2	`lo`	`lo`	−2.26	`l·o` (−4.51)
3	`low`	`low`	−2.26	`lo·w`, `l·ow` (−4.51)
4	`lowe`	`low·e`	−4.51	`lowe` (−4.65)
5	`lower`	`low·er`	−4.51	`lower` (−4.65)

- Cada $\delta(j)$ reaproveita os $\delta(i)$ anteriores: custo $O(n^2)$ em vez de 2^{n-1} caminhos
- O *backtracking* a partir de $\delta(5)$ recupera a segmentação: `low` `er`

Unigram: Exemplo Simplificado (4/6), Likelihood do Corpus

Corpus: “low” (freq 10), “er” (freq 10), “lower” (freq 1)

Passo 2: Viterbi dá a melhor segmentação de cada palavra do corpus:

Palavra	**Freq**	**Melhor segmentação**	$\ln P$
“low”	10	`low`	$\ln(0.105) \approx -2.26$
“er”	10	`er`	$\ln(0.105) \approx -2.26$
“lower”	1	`low` `er`	$\ln(0.011) \approx -4.51$

A qualidade do vocabulário é medida pela **log-likelihood do corpus**: a soma, ponderada pela frequência, do log da melhor segmentação de cada palavra:

$$\mathcal{L} = \sum_{w \in \text{corpus}} \text{freq}(w)\ \ln P(\text{melhor seg. de } w) = 20 \ln(0.105) + \ln(0.011) \approx -49.6$$

Unigram: Exemplo Simplificado (5/6), Poda

Passo 3: Poda do vocabulário. O critério é a **perda** Δ_t: quanto a log-likelihood do corpus cai se o token t for removido e as palavras re-segmentadas:

$$\Delta_t = \mathcal{L} - \mathcal{L}_{\text{sem } t}$$

- Remover `owe`: nenhuma melhor segmentação o utiliza, logo $\Delta_{\texttt{owe}} = 0 \rightarrow$ **podar**
- Remover `lo`: está no reticulado, mas nenhum **melhor caminho** passa por ele ("low" continua sendo `low`), logo $\Delta_{\texttt{lo}} = 0 \rightarrow$ **podar**
- Remover `low`: "low" vira `lo`·`w` (0.105 → 0.011) e "lower" vira `lower` (0.011 → 0.010):

$$\mathcal{L}_{\text{sem low}} = 10\ln(0.011) + 10\ln(0.105) + \ln(0.010) \approx -72.3$$

logo $\Delta_{\texttt{low}} \approx 22.7 \rightarrow$ **manter**

Pelo mesmo critério, $\Delta = 0$ para `lower`, `lowe`, `ower`, `wer`, `we` e `ow`: todos podados. Vocabulário final: `low`, `er` e os caracteres `l`, `o`, `w`, `e`, `r`.

No algoritmo real, quem encerra a poda é um hiperparâmetro de tamanho de vocabulário: cada rodada remove ~20% dos tokens de menor Δ e repete até atingir o alvo (passo 6 do algoritmo). Neste exemplo, podamos todos os tokens com $\Delta = 0$ de uma vez.

Caracteres individuais nunca são podados, mesmo com $\Delta = 0$: eles garantem que qualquer palavra continue segmentável.

Unigram: Exemplo Simplificado (6/6), Iteração Final

Passo 4: Após a poda, as probabilidades são re-estimadas sobre o vocabulário restante (aqui, por recontagem simples; no algoritmo real, via EM):

Token	`low`	`er`	`l`	`o`	`w`	`e`	`r`
count	11	11	11	11	11	11	11
$P(\cdot)$	1/7	1/7	1/7	1/7	1/7	1/7	1/7

Segmentações finais: "low" → `low`, "er" → `er`, "lower" → `low` `er`

$$\mathcal{L}_{\text{final}} = 20\ln(1/7) + \ln(1/7^2) = 22\ln(1/7) \approx -42.8 \quad \text{(antes: } -49.6\text{)}$$

A poda melhora a likelihood

A massa de probabilidade antes espalhada por 15 tokens concentra-se nos 7 que o corpus usa. Num corpus real, o ciclo (EM, cálculo de Δ, poda) repete até atingir o tamanho alvo; aqui, o método redescobriu os morfemas `low` e `er`.

Unigram: Re-estimação via EM (1/2)

No exemplo, inicializamos $P(t)$ com contagens brutas de substrings:

O problema: supercontagem

Cada ocorrência de "low" contou ao mesmo tempo para `l`, `o`, `w`, `lo`, `ow` e `low`. Mas o modelo assume **uma única** segmentação por palavra: um token só deveria contar quando usado.

O algoritmo completo (Kudo, 2018) corrige isso re-estimando $P(t)$ via **EM**:

As duas etapas

E-step: com o P atual, calcula a posterior de cada segmentação e as contagens **esperadas**:

$$P(s \mid w) = \frac{P(s)}{\sum_{s' \in \mathcal{S}(w)} P(s')} \qquad \mathbb{E}[\text{count}(t)] = \sum_{w} \text{freq}(w) \sum_{\substack{s \in \mathcal{S}(w) \\ t \in s}} P(s \mid w)$$

M-step: renormaliza: $P(t) \leftarrow \mathbb{E}[\text{count}(t)] \,/\, \sum_{t'} \mathbb{E}[\text{count}(t')]$

Unigram: Re-estimação via EM (2/2)

E-step: posteriores

"low" (todas as 4 segmentações):

`low`	0.82
`l·ow`, `lo·w`	0.09 cada
`l·o·w`	0.01

"lower" (16 segmentações):

`low` `er`	0.40
`lower`	0.34
`l·ow·er`, `lo·w·er`, `low·e·r`	0.04 cada
demais 11	0.13

Contagem esperada e M-step

O token `low` aparece nas 10 ocorrências de "low" (posterior 0.82) e em duas segmentações de "lower":

$$\mathbb{E}[\text{count}(\texttt{low})] = 10(0.82) + 0.40 + 0.04 \approx 8.6$$

Após renormalizar todos os tokens:

Token	P antiga	P nova
`low`	0.105	0.35
`er`	0.105	0.39
`lo`	0.105	0.04
`lower`	0.010	0.01

A massa migra para os tokens **realmente usados**: `low` e `er` sobem, os demais caem.

Na prática, alternam-se algumas iterações de EM antes de cada rodada de poda, que remove $\approx 20\%$ dos tokens de menor Δ (Kudo, 2018).

Subword Regularization e BPE Dropout

Ideia: A tokenização não precisa ser determinística durante o treinamento!

Subword Regularization (Kudo, 2018)

O Unigram pode amostrar entre múltiplas segmentações válidas:

`"lowest"` pode ser:

- `low` `est` (probabilidade 0.7)
- `lo` `we` `st` (probabilidade 0.2)
- `l` `o` `w` `est` (probabilidade 0.1)

BPE Dropout (Provilkov et al., 2020)

Ignora aleatoriamente alguns merges do BPE durante o treinamento:

Com dropout $p = 0.1$:

- Normal: `low` `est`
- Ignorado: `lo` `w` `est`
- Ignorado: `low` `e` `st`

Ambos atuam como **data augmentation**, tornando os modelos mais robustos a segmentações raras.

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. **Frameworks de Tokenização**
8. Tokenização na Prática
9. De Tokens a Embeddings

SentencePiece: O Framework

SentencePiece (Kudo & Richardson, 2018) é um **framework**, não um algoritmo.

O que torna o SentencePiece especial?

1. **Agnóstico ao idioma:** trata a entrada como um fluxo bruto de bytes/unicode
2. **Sem pré-tokenização:** não depende de divisão por espaço em branco
3. O espaço em branco é tratado como um caractere regular (substituído por _)
4. Suporta **tanto** BPE quanto Unigram como algoritmo subjacente

Exemplo

`"New York"` → `_New` `_York`

O _ codifica o espaço, tornando a tokenização **totalmente reversível**.

Utilizado em: T5, LLaMA, ALBERT, XLNet, Gemma, …

SentencePiece vs. tiktoken

Duas bibliotecas dominam o ecossistema; a diferença fundamental é o **papel** de cada uma:

	SentencePiece (Google, 2018)	**tiktoken** (OpenAI, 2022)
Papel	treino **e** inferência	inferência apenas
Algoritmos	BPE e Unigram	byte-level BPE
Vocabulário base	chars Unicode	256 bytes
Normalização	NFKC configurável	nenhuma (preserva bytes)
Pré-tokenização	opcional; espaço vira _	obrigatória, via regex
OOV	`<unk>` ou `byte_fallback`	impossível por construção
Vocabulários	treinados pelo usuário	prontos (`cl100k`, `o200k`, …)
Modelos	T5, LLaMA 1/2, Mistral, Gemma	GPT-3.5/4/4o, LLaMA 3

- O `byte_fallback` (LLaMA 1/2) decompõe chars fora do vocabulário em bytes: evita o `<unk>` sem operar nativamente em byte-level
- A regex de pré-tokenização do tiktoken codifica decisões de design (no `cl100k`, dígitos são agrupados em blocos de até 3; voltaremos a isso)
- No tiktoken, tokens especiais no texto do usuário geram **erro** por padrão (`disallowed_special`): o escape é responsabilidade explícita da aplicação

O HuggingFace `tokenizers` reimplementa ambos os formatos: é o que o `AutoTokenizer` carrega por baixo dos panos.

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. **Tokenização na Prática**
9. De Tokens a Embeddings

Tokens Especiais

É comum reservar **tokens especiais** no vocabulário:

Token	Modelo	Função
`[CLS]`	BERT	representação agregada para classificação
`[SEP]`	BERT	separa pares de sentenças
`[MASK]`	BERT	posição a prever no pré-treino (MLM)
`[PAD]`	vários	preenche o lote; ignorado via *attention mask*
`<|endoftext|>`	GPT-2	fim de documento (ID 50256)
`<s>`, `</s>`	LLaMA	início (BOS) e fim (EOS) de sequência

Propriedades

- São **atômicos**: nunca são divididos pela pré-tokenização nem surgem de merges
- Texto do usuário que imita um token especial precisa ser **escapado** pelo tokenizer, senão documentos podem injetar marcadores estruturais (*prompt injection*)

Tokens Especiais em Modelos de Chat

Modelos de chat estruturam a conversa com tokens especiais de **papel** (*role*), definidos por um **chat template**:

Formato ChatML (usado por GPT, Qwen e outros)

```
<|im_start|>user
Olá!<|im_end|>
<|im_start|>assistant
Oi! Como posso ajudar?<|im_end|>
```

- O template varia por modelo: o LLaMA 3, por exemplo, usa `<|start_header_id|>` e `<|eot_id|>`
- No HuggingFace, `tokenizer.apply_chat_template()` aplica o formato correto
- O modelo aprende no fine-tuning (aula de treinamento de LLMs) a encerrar a resposta gerando o token de fim de turno; é assim que a geração "sabe" parar

Tokenização de Números e Código

Lembra do “123 + 456” da abertura? Segmentações reais:

Texto	GPT-2	GPT-4 (cl100k)
1999	1999	199 | 9
2023	20 | 23	202 | 3
123 + 456	123 | ␣+ | ␣4 | 56	123 | ␣+ | ␣ | 456

- **GPT-2**: quebras irregulares, ditadas pela frequência no corpus, que desalinham os dígitos entre operandos e dificultam a aritmética
- **GPT-4**: dígitos agrupados em blocos de até 3; **LLaMA 1/2**: um token por dígito

Código: indentação

Linha indentada com 8 espaços: o GPT-2 gasta **um token por espaço** (13 tokens no exemplo); o cl100k mescla a indentação em um único token (7 tokens).

Glitch Tokens: Subtreinamento ao Extremo

O vocabulário do GPT-2/3 foi aprendido num corpus **diferente** do usado para treinar o modelo.

O caso `SolidGoldMagikarp` (Rumbelow & Watkins, 2023)

- `␣SolidGoldMagikarp` (ID 43453) e `␣petertodd` (ID 37444): usernames do Reddit que entraram no vocabulário BPE, mas foram filtrados do treinamento do modelo
- O embedding desses tokens quase nunca recebeu gradiente e ficou próximo da inicialização aleatória
- Resultado: pedidos simples como “repita SolidGoldMagikarp” faziam o GPT-3 responder com palavras aleatórias, insultos ou evasivas

- É o caso extremo dos “tokens raros subtreinados” do início da aula
- Lição: vocabulário e corpus de treinamento precisam estar **alinhados**
- Há métodos para detectá-los automaticamente (Land & Bartolo, 2024)

Visão Geral

1. Por que a Tokenização Importa
2. Abordagens Ingênuas
3. Pré-tokenização
4. Byte Pair Encoding (BPE)
5. WordPiece
6. Unigram Language Model
7. Frameworks de Tokenização
8. Tokenização na Prática
9. De Tokens a Embeddings

De Token IDs a Embeddings

O tokenizer produz uma sequência de **inteiros** (IDs). O transformer precisa de **vetores densos**.

A tabela de embedding

Uma matriz $E \in \mathbb{R}^{|V| \times d}$, onde cada linha é o vetor de um token:

$$\mathbf{e}_i = E[i] \in \mathbb{R}^d$$

`nn.Embedding` é uma **tabela de lookup**, não uma transformação linear: apenas seleciona uma linha.

ID	$\mathbf{e} \in \mathbb{R}^d$		
0	0.12	−0.43	⋯
⋮	⋮	⋮	⋮
15496	**0.31**	**−1.20**	⋯
⋮	⋮	⋮	⋮
$\lvert V\rvert - 1$	−0.07	0.88	⋯

ID 15496 → linha 15496 → $\mathbf{e} \in \mathbb{R}^{768}$

Pipeline completo

$$\text{texto} \xrightarrow{\text{tokenizer}} [\text{IDs}] \xrightarrow{\texttt{nn.Embedding}} [\text{vetores}] \longrightarrow \text{entrada do transformer}$$

Quantos Parâmetros Custam os Embeddings?

$$\text{params}_{\text{embedding}} = |V| \times d$$

Modelo	$\lvert V\rvert$	d	Params emb.	Total	% do total
GPT-2 Small	50,257	768	38.6M	117M	**33%**
GPT-2 Medium	50,257	1,024	51.5M	345M	**15%**
GPT-2 Large	50,257	1,280	64.3M	762M	**8.4%**
GPT-2 XL	50,257	1,600	80.4M	1.5B	**5.4%**
LLaMA 2 7B	32,000	4,096	131M	7B	**1.9%**

Conclusões

- **Vocabulário maior = mais parâmetros**: escolhas de tokenização têm custo direto no tamanho do modelo
- **Modelos menores pagam mais**: em GPT-2 Small, $\frac{1}{3}$ dos parâmetros estão na tabela de embedding
- **Weight tying**: GPT-2 e muitos outros modelos **reutilizam** a mesma matriz como camada de saída (`lm_head`), sem custo extra de parâmetros para a projeção final

Como Escolher o Tamanho do Vocabulário?

Não existe $|V|$ ótimo universal; a escolha é um trade-off:

A favor de $|V|$ maior

- Fertility menor: mais texto por janela de contexto
- Menos passos de geração por resposta (inferência mais barata)
- Melhor cobertura multilíngue

Contra $|V|$ maior

- Embedding e `lm_head` mais caros ($2 \times |V| \times d$ sem weight tying)
- Tokens de cauda longa aparecem pouco no corpus: subtreinados
- Softmax final sobre mais classes

- Modelos maiores se beneficiam de vocabulários maiores (Tao et al., 2024)
- A prática acompanhou: LLaMA 2 (32K) → LLaMA 3 (128K) → Gemma (256K)
- Na dúvida, avalie a **fertility** do tokenizer candidato nos idiomas e domínios alvo

Leituras Recomendadas

- Radford, A. et al. (2019). Language Models are Unsupervised Multitask Learners. (GPT-2)
- Xue, L. et al. (2022). ByT5: Towards a Token-Free Future. *TACL*.
- Yu, L. et al. (2023). MegaByte: Predicting Million-Byte Sequences. *NeurIPS*.
- Provilkov, I. et al. (2020). BPE-Dropout. *ACL*.
- Rumbelow, J. & Watkins, M. (2023). SolidGoldMagikarp (plus, prompt generation). *LessWrong*.
- Land, S. & Bartolo, M. (2024). Fishing for Magikarp: Automatically Detecting Under-trained Tokens. *EMNLP*.
- Tao, C. et al. (2024). Scaling Laws with Vocabulary. *NeurIPS*.
- OpenAI (2022). tiktoken: a fast BPE tokeniser. *github.com/openai/tiktoken*.
- Hugging Face. The Hugging Face Course, cap. 6: WordPiece tokenization. *huggingface.co/learn*.

Referências

- Sennrich, R., Haddow, B., & Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. *ACL*.
- Schuster, M. & Nakajima, K. (2012). Japanese and Korean Voice Search. *ICASSP*.
- Wu, Y. et al. (2016). Google's Neural Machine Translation System. *arXiv:1609.08144*.
- Kudo, T. (2018). Subword Regularization. *ACL*.
- Kudo, T. & Richardson, J. (2018). SentencePiece. *EMNLP*.