

Tokens, tokenização e embeddings

Escola de Verão 2026

Alexandre Teles <alexandre.teles@inctdd.org>

Índice

1. Introdução
2. Tokens
3. Tokenização
4. Tokenizadores
5. Treinando um tokenizador
6. Embeddings
7. Análise de similaridade e relações vetoriais

Introdução

Modelos de linguagem trabalham com texto, mas não entendem o texto da mesma forma que humanos. Para que modelos de linguagem possam processar o texto, ele precisa ser convertido em uma forma que o modelo possa entender. Uma vez que modelos de linguagem não "leem" texto como humanos, eles precisam de uma representação numérica do texto.

Para transformar linguagem em números, usamos três ideias-chave:

1. Tokens: as unidades básicas do texto.
2. Tokenização: o processo de dividir o texto em tokens.
3. Embeddings: representações numéricas com significado.

Entender esses conceitos é essencial para compreender como LLMs funcionam por dentro.

Tokens

■ O que são tokens?

Tokens são as unidades básicas processadas por modelos de linguagem. Podem ser palavras, partes de palavras, símbolos ou até mesmo caracteres individuais, dependendo do tokenizador utilizado.

É importante notar que tokens não são necessariamente iguais a palavras. Por exemplo, a palavra "incrível" pode ser dividida em dois tokens: `in` e `crível`, dependendo do tokenizador. Modelos **não veem texto**, apenas **IDs de tokens**.

Tokens nos permitem:

1. Lidar com **vocabulários extensos**
2. Tratar **palavras raras ou novas**
3. Manter **consistência** na representação do texto
4. Padronizar o tamanho das entradas

Tokens

■ Como escolher tokens?

- **Tamanho do vocabulário**
 - pequeno → sequências longas
 - grande → mais memória e parâmetros
- **Nível de granularidade**
 - caracteres ↔ palavras ↔ subpalavras
- **Eficiência computacional**
 - menos tokens = processamento mais rápido
- **Generalização**
 - subpalavras lidam melhor com palavras novas
- **Custo**
 - treinamento, memória, inferência

(Tokenização é sempre um compromisso, não uma escolha perfeita.)

Tokenização

Tokenização é o processo de **converter texto bruto em uma sequência de tokens** que um modelo de linguagem consegue processar. Como modelos operam apenas sobre números, cada token é posteriormente **mapeado para um identificador inteiro**.

Esse processo define **como o modelo “enxerga” a linguagem**. A mesma frase pode ser tokenizada de formas diferentes, dependendo das regras adotadas (palavras, subpalavras, caracteres). Por isso, a tokenização influencia diretamente **eficiência, generalização e custo computacional**.

Na prática, tokenizar envolve etapas como **normalização do texto**, divisão em unidades menores e mapeamento para IDs em um vocabulário.

Tokenização

■ Exemplo conceitual

```
from typing import List, Dict

def tokenize(text: str, vocab: Dict[str, int]) -> List[int]:
    """
    Converte texto em uma sequência de IDs de tokens.
    """
    # Normalização simples
    text = text.lower()

    # Divisão em tokens (exemplo didático)
    tokens: List[str] = text.split()

    # Mapeamento de tokens para IDs
    token_ids: List[int] = [vocab[token] for token in tokens]

    return token_ids

vocab = {"olá": 1, "mundo": 2}

tokenize("Olá mundo", vocab) # → [1, 2]
```

(Tokenizadores reais usam subpalavras e métodos estatísticos, mas a ideia central é a mesma.)

Tokenização

Quase todos os tokenizadores modernos (BPE, Unigram, WordPiece, etc.) partem da mesma ideia estatística:

Aprender, a partir das frequências no corpus, uma forma de segmentar o texto que seja estatisticamente eficiente e reutilizável.

ou, em outras palavras:

Dividir o texto em partes que aparecem com frequência suficiente para valer a pena tratá-las como unidades.

Tokenizadores aprendem tokens a partir da distribuição estatística do corpus, portanto:

- Sequências **frequentes** viram tokens únicos
- Sequências **raras** são decompostas em partes menores
- Isso está intimamente ligado a ****compressão de dados**

Tokenizadores

■ BPE (Byte Pair Encoding)

1. Começa com caracteres
2. Junta pares mais frequentes repetidamente

Aqui a intuição é a seguinte: **se algo aparece muito, vale a pena virar um símbolo próprio.**

De forma geral, dado um corpus **D** representado como uma sequência de **caracteres**:

$$D = (s_1, s_2, \dots, s_N)$$

(**D** é uma sequência de símbolos **s 1, s 2, ..., s N.**)

Em cada iteração, escolhe-se o par adjacente mais frequente:

$$(a^*, b^*) = \arg \max_{a,b} \text{freq}(a, b)$$

(**(a*, b*)** é o par **(a, b)** que maximiza a frequência **freq(a, b)** no corpus.)

Cria-se um novo token **c** a partir desse par:

(**c** é o novo token formado pela concatenação do par mais frequente **(a*, b*)**.)

Tokenizadores

■ Unigram (LM)

Ideia: ter um vocabulário V de tokens candidatos (subpalavras) e um modelo que atribui uma probabilidade a cada token:

$$p(t) \text{ for } t \in V$$

(para todo token t em V , define-se uma probabilidade $p(t)$.)

Inferência (tokenizar um texto x): escolhe a segmentação mais provável.

$$s^* = \arg \max_s p(s \mid x)$$

(s^* é a segmentação s que maximiza $p(s \mid x)$.)

Uma forma de pensar: a probabilidade do texto soma sobre tokenizações possíveis.

$$p(x) = \sum_s p(s \mid x)$$

($p(x)$ é a soma, para toda segmentação s possível de x , de $p(s \mid x)$.)

Treinando um tokenizador

Um dos principais hiperparâmetros ao treinar um tokenizador é o tamanho do vocabulário. Um vocabulário maior resulta em menos tokens por texto, mas aumenta o custo computacional e a complexidade do modelo. Já um vocabulário menor reduz o custo, mas pode levar a sequências de tokens mais longas e menos eficientes.

Artigos recentes sugerem utilizar a proximidade da frequência dos tokens à lei de Zipf como critério de pontuação automática uma vez que a linguagem natural segue distribuição Zipfiana: palavras frequentes são muito comuns, palavras raras são extremamente raras $f(r) \propto 1/r^\alpha$.

Vocabulários que produzem tokens com distribuição Zipfiana vão apresentar melhor desempenho downstream porque:

- **Alinhamento estrutural:** tokenização reflete a estrutura estatística natural do corpus
- **Vocabulário muito pequeno:** força sobre-segmentação, achata a curva, modelo precisa aprender padrões longos para reconstruir palavras básicas
- **Vocabulário muito grande:** muitos tokens raros sub-treinados, cauda pesada, parâmetros desperdiçados

Zipf ótimo = granularidade ideal: unidades frequentes mapeiam para poucos tokens frequentes; unidades raras são composições de tokens comuns (onde possível).

Treinando um tokenizador

Clique aqui para abrir o notebook de treinamento de tokenizador (
<https://colab.research.google.com/drive/1qplja9Bm8Vc9JoNmivKGK8hz1rfmguBy>)

Embeddings

■ Tokens vs embeddings: qual a diferença?

Tokens são unidades **discretas** (palavras/subpalavras/símbolos) escolhidas pelo tokenizador. O modelo não processa texto diretamente, e sim uma sequência de **IDs inteiros** (um ID por token).

Embeddings são vetores **contínuos** (números reais) que o modelo **aprende** para esses IDs. Eles colocam tokens em um espaço geométrico onde relações como *semelhança* e *contexto* aparecem como **distâncias/direções**.

Pipeline típico:

```
texto → tokens → IDs → lookup na tabela de embeddings → vetores → rede neural
```

Embeddings

■ Exemplo conceitual

O token vira ID, e o ID seleciona uma linha de uma matriz E (tabela de embeddings):

```
import numpy as np

# IDs de tokens (saída do tokenizador)
token_ids = [1, 2, 2, 1]

# Tabela de embeddings do modelo: (tamanho_do_vocab) × (dimensão)
E = np.random.randn(10, 4)

# Embedding de cada token da sequência (um vetor por ID)
X = E[token_ids] # shape: (len(token_ids), 4)
```

Em resumo:

- token/ID = **rótulo** (discreto, definido pela tokenização)
- embedding = **representação** (contínua, aprendida no treinamento)

(Embeddings são parâmetros do modelo: não existem “antes” do treinamento. E não se limitam a tokens – modelos também usam embeddings de posição, por exemplo.)

Embeddings

■ Vector Space (espaço vetorial)

Um **espaço vetorial** é um “mundo” matemático onde cada item é representado por um **vetor de números reais**. Em NLP, normalmente pensamos em:

$$e(t) \in \mathbb{R}^d$$

(**$e(t)$** é o *embedding* do token **t** , um vetor em **d** dimensões.)

A ideia central é: **significado vira geometria**.

- Tokens com usos parecidos no corpus tendem a ter embeddings **próximos**.
- Relações semânticas aparecem como **direções** e **regiões** do espaço (não como regras explícitas).
- “Semelhança” é medida por operações simples, como **produto interno** ou **cosseno**:


Embeddings

Como modelos usam isso na prática:

1. **Lookup:** cada ID de token aponta para um vetor (uma linha da matriz de embeddings).
2. **Composição contextual:** a rede (especialmente atenção) combina vetores para produzir representações que dependem do **contexto**.
3. **Predição:** as camadas finais escolhem o próximo token porque certos vetores ficam mais alinhados (maior similaridade) com a saída do modelo.

Intuição importante:

- Um embedding “isolado” captura **tendências gerais** do token.
- A representação **contextual** (saída de camadas do Transformer) captura o significado *na frase*.

(Mesmo sendo difícil visualizar em  grande (ex.: 768, 1024, ...), o modelo explora exatamente essas operações geométricas para organizar e recuperar significado.)

Embeddings

■ Positional embeddings (por que posição importa?)

Se eu te der apenas os tokens `{homem, mordeu, cachorro}`, eu perdi algo essencial: **a ordem**.

- “o **cachorro** mordeu o **homem**” \neq “o **homem** mordeu o **cachorro**”
- Sem posição, o modelo tende a enxergar um “**saco de palavras**” (bag of words).

Positional embeddings resolvem isso dando a cada posição **i** na sequência um vetor **p_i** (na mesma dimensão do embedding do token). Em Transformers, a entrada em cada posição costuma ser a soma:

$$x_i = e(t_i) + p_i$$

(**e(t_i)** é o embedding do token na posição **i**, e **p_i** codifica “onde” ele está na sequência.)

O que isso “representa” na prática?

- **p_i** não é “significado de palavra”; é **informação de posição**.
- Mas ao ser combinado com **e(t_i)**, ele permite que a atenção aprenda padrões como:
 - dependências locais (“adjetivo costuma vir perto do substantivo”)
 - estrutura (início/fim, separação de frases, listas)
 - relações de longo alcance (sujeito \leftrightarrow verbo, pares de parênteses, etc.)

Observação: há várias formas de codificar posição (aprendida, senoidal, relativa, RoPE). A ideia central é sempre a mesma: **transformar ordem em vetores** para que o modelo possa usar geometria e atenção para interpretar sequência.

Embeddings

■ Relações vetoriais: analogias e “direções de significado”

No espaço vetorial, não olhamos apenas para “proximidade”. Muitas vezes, **diferenças entre vetores** capturam um atributo (uma “direção”):

- `rainha ≈ rei - homem + mulher`
- `Roma ≈ Paris - França + Itália`

Em termos geométricos, você cria um vetor-consulta `q` e procura o token cujo embedding é mais similar a `q`:

$$q = e(\text{rei}) - e(\text{homem}) + e(\text{mulher})$$

e então:

(ou seja: o token `t` é o mais próximo de `q` por cosseno/produto interno.)

Dois detalhes importantes:

- Essas analogias tendem a funcionar melhor com **embeddings estáticos** (ex.: word2vec).
- Em LLMs modernos, o “significado” mais fiel costuma estar em **representações contextuais** (o vetor do token *dentro* da frase). Ainda assim, a intuição de “direções semânticas” continua sendo uma ferramenta útil para entender **por que vizinhos no espaço vetorial parecem relacionados**.

Análise de similaridade e relações vetoriais

Clique aqui para abrir o notebook de embeddings (

https://colab.research.google.com/drive/1fM0J3BRzMXJRKl8brcqPv_mxH-YX0I3o)