Tokenization 在 NLP 中的工作原理

杨子凡

Dec 12, 2025

想象一下，你输入一句简单的英文「Hello, world!」到 NLP 模型中，它如何理解这个句子？首先，这段文本会被拆分成一个个小单元，比如「Hello」、「,」、「world」、「!」这样的 token。这些 token 就像语言的积木块，是人类自然语言转化为机器可处理的数字序列的桥梁。Tokenization 作为 NLP 管道的第一步，直接决定了后续 embedding 和模型推理的质量。如果 tokenization 出问题，整个模型的性能都会受影响，比如罕见词汇无法正确拆分，导致准确率下降。

在 NLP 的核心流程中，文本首先经过 tokenization 转换为 token 序列，然后映射到词汇表 ID，再通过 embedding 层变成向量，最后输入 Transformer 等模型进行处理。这个过程看似简单，却至关重要：它影响计算效率，因为模型有最大序列长度限制；也影响准确率，因为好的 tokenization 能更好地捕捉语义，比如处理「人工智能」这样的中文词时，需要考虑无空格特性。本文将深入解释 tokenization 的原理、类型、算法、挑战及实际应用，面向初学者到中级开发者，提供 Python 代码示例，帮助你从零掌握这项基础技术。

通过阅读，你将理解为什么 BERT 和 GPT 使用不同的 tokenizer，以及如何在自己的项目中训练自定义 tokenizer。让我们从基础开始，一步步揭开这个 NLP 基石的神秘面纱。你准备好探索了吗？

1　什么是 Tokenization?

Tokenization 的本质是将原始文本拆分成更小的单元，这些单元称为 token，可以是完整的单词、子词片段，甚至单个字符。这个过程解决了 NLP 模型的一个根本问题：Transformer 等神经网络只能处理数字序列，而非人类语言的连续字符串。通过 tokenization，文本被转化为固定词汇表中的 ID 序列，便于后续向量化。

为什么需要 tokenization？因为直接用字符序列会让序列过长，计算开销巨大；用完整单词又会遇到 OOV（Out-of-Vocabulary）问题，即训练时未见过的词无法处理。Tokenization 巧妙平衡了这两者，提供了一个高效的桥梁。例如，在句子「Don't stop!」中，单词级 tokenization 可能输出「Don't」、「stop」、「!」，而子词级则进一步拆成「Don」、「'」、「t」、「stop」、「!」。

Token 的类型多样化，以适应不同场景。单词级按空格和标点拆分，简单直观，但对新词不友好；子词级如 BPE 将词拆成常见子单元，处理 OOV 更好；字符级则最细粒度，每个字符一个 token，灵活但序列长。在实际模型中，还有特殊 token 增强功能，比如 BERT 中的 [CLS] 用于分类任务的聚合表示，[SEP] 分隔句子，[PAD] 填充序列到固定长度，[UNK] 代表未知 token。这些特殊标记确保输入标准化，提高模型鲁棒性。

文本经过 tokenization 后，会映射到唯一 ID，比如词汇表大小为 30k 的模型中，「hello」可能对应 ID 101，然后生成 attention mask 区分真实 token 和填充部分。这个流程可视化为：原始文本 → token 列表 → ID 序列 → 模型输入。你知道自己的 NLP 项目中，tokenization 如何影响结果吗？

2 Tokenization 的工作原理

Tokenization 的核心流程分为几个步骤，首先是预处理，包括小写转换、标点规范化以及 Unicode 标准化，以减少噪声。然后是拆分，使用规则或统计模型将文本切分成 token。接着，词汇表映射将每个 token 转为唯一 ID，通常词汇表大小在 30k 到 100k 之间。最后是编码，生成 ID 序列和 attention mask；解码则是逆过程，从 ID 还原文本。

让我们通过 Python 示例直观理解，使用 Hugging Face 的 BertTokenizer：

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
tokens = tokenizer.tokenize("Hello, world!")
ids = tokenizer.convert_tokens_to_ids(tokens)
print(tokens) # 输出 : ['hello', ',', 'world', '!']
print(ids) # 输出 : [10176, 1010, 2088, 10008]（实际 ID 依模型而定）
```

这段代码首先加载预训练的 BERT 分词器，`from_pretrained` 从 Hugging Face Hub 下载 bert-base-uncased 模型的 tokenizer 配置，包括词汇表文件。`tokenizer.tokenize(Hello, world!)` 执行核心拆分：它先规范化文本（小写、去除多余空格），然后用 WordPiece 算法拆分成子词 token，输出 ['hello', ',', 'world', '!']。注意逗号独立成 token，这是为了捕捉标点语义。接着 `convert_tokens_to_ids` 查询词汇表，将每个 token 映射到整数 ID，比如 'hello' 可能为 10176。这些 ID 是模型实际输入，加上特殊 token 如 [CLS] 和 [SEP] 后，形成完整序列。这个示例展示了从文本到数字的端到端转换，实际使用时还需调用 `tokenizer.encode_plus` 添加 mask 和截断。

词汇表构建是从大规模语料库中训练而来：统计高频 token，设置频率阈值或采样低频部分，避免词汇表爆炸。训练过程迭代优化，确保常见词完整表示，罕见词拆分成子词。这个原理确保了 tokenization 的高效性和泛化能力。你试过调试自己的 tokenizer 输出吗？

3 常见 Tokenization 方法与算法

规则-based 方法是最基础的，比如单词级 tokenization 依赖空格和正则表达式拆分，NLTK 的 word_tokenize 就是典型实现。它简单快速，适合英文，但 OOV 问题突出：新词如「COVID-19」只能用 [UNK] 表示。

统计-based 方法更强大，其中 Byte-Pair Encoding（BPE）被 GPT 系列广泛采用。BPE 算法从字符级开始，迭代合并语料中最频字符对构建子词词汇表。具体步骤：初始化每个字符为 token，统计相邻对频率，重复合并最高频对，直至达到词汇表大小。例如，从「low lower lowest」开始，第一轮可能合并「l o」成「lo」，逐步形成「low」、「lowest」等子词。这种贪婪合并高效处理 OOV，「un」+「known」可拆成已知子词。

WordPiece 是 BERT 的选择，类似 BPE 但在合并时选择最大 likelihood 提升的子词，提高 perplexity。Unigram Language Model 则相反，从大词汇表开始，概率采样删除低频 token 直到目标大小，SentencePiece 常用此法支持无空格语言如中文。

这些方法的对比鲜明：BPE 高效处理 OOV，需训练语料；WordPiece 优化语言建模但计算密集；Sentence-

Piece 多语言友好但词汇表较大。下面是 BPE 简化实现演示：

```
def simple_bpe(texts, num_merges=10):
    words = [list(w) for w in texts.split()] # 初始化字符列表
    merges = {}
    for i in range(num_merges):
        pairs = {}
        for word in words:
            for j in range(len(word)-1):
                pair = (word[j], word[j+1])
                pairs[pair] = pairs.get(pair, 0) + 1
        if not pairs:
            break
        best_pair = max(pairs, key=pairs.get)
        merges[best_pair] = i
        new_words = []
        for word in words:
            new_word = word
            while ''.join(new_word[:-1]) in merges: # 贪婪合并
                new_word = new_word[:-2] + [''.join(new_word[-2:])]
            new_words.append(new_word)
        words = new_words
    return merges, words

merges, tokenized = simple_bpe("low␣lower␣lowest", 5)
print(merges) # 输出类似 {(('l', 'o'), 0), (('lo', 'w'), 1)} 等合并对
print(tokenized) # 输出子词序列
```

这段代码模拟 BPE 训练：`texts.split()` 分词成字符列表，循环统计相邻对频率，选最高频合并并记录在 `merges` 字典。合并后用贪婪方式应用到所有词，确保子词一致。这个简化版忽略了完整词汇表构建，但捕捉了核心迭代逻辑。在 Hugging Face 中，你可比较不同 tokenizer：`tokenizer(Hello)` 输出差异揭示算法特性，如 BPE 更倾向子词拆分。BPE 的合并过程就像逐步构建拼图，高效捕捉语言规律。你最喜欢哪种方法，为什么?

4 Tokenization 在 NLP 模型中的作用

在 Transformer 架构中，tokenization 位于输入 embedding 层之前，直接塑造向量表示的质量。没有它，模型无法处理变长序列。生成 token ID 后，加入位置编码（Positional Encoding），如 sin 和 cos 函数注入顺序信息：$PE(pos, 2i) = \sin(pos/10000^{2i/d})$，确保模型感知位置。

实际影响显著：模型有序列长度限制，如 BERT 的 512 token，超长需截断或填充 [PAD]，attention mask 屏蔽无效部分。多语言场景下，子词 tokenizer 处理中文「人工智能」为「人」、「工」、「智」、「能」，无空格依赖

强。BERT 用 WordPiece 偏好完整英文词，GPT 的 BPE 更碎片化利于生成。
案例中，BERT tokenizer 保留标点独立，适合分类；GPT 优化连续生成。长文本如 Longformer 用滑动窗口 tokenization，动态调整 attention，减少 token 数。Transformer 输入管道从 tokenization 开始，串联 embedding 和模型，任何环节偏差都放大误差。思考你的模型输入如何优化?

5 挑战与解决方案

Tokenization 面临 OOV 问题，未见词用 [UNK] 表示，语义丢失严重，子词方法如 BPE 通过拆分解决，将「neuralink」拆成「neu」、「ralink」。长序列超过 max_length 时，需智能截断保留关键部分，或用层次 tokenization 分块处理。
多语言尤其是低资源语言挑战大，英文依赖空格，中文无此特性，导致过拆；SentencePiece 直接处理原始文本，训练联合模型缓解。噪声文本如表情「🤔」、URL「https://example.com」、拼写错误需自定义预处理规则，先规范化再 tokenization。
性能优化关键，TikToken 为 GPT 设计，用 Rust 实现超快编码，基准测试显示比 Python tokenizer 快 10 倍。这些解决方案让 tokenization 更鲁棒。你遇到过哪些 tokenization 坑?

6 实际应用与工具推荐

在情感分析中，tokenization 确保「I love AI!!」拆成捕捉强调的 token；在机器翻译，子词对齐源语和目标语；在聊天机器人，快速 tokenizer 支撑实时响应。流行库中，Hugging Face Tokenizers 最全面，支持 BPE 等训练自定义模型：

```
from tokenizers import Tokenizer, models, trainers, pre_tokenizers, decoders

tokenizer = Tokenizer(models.BPE())
tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()
trainer = trainers.BpeTrainer(vocab_size=30000, special_tokens=["[UNK]", "[PAD]"])
files = ["corpus.txt"] # 你的语料文件
tokenizer.train(files, trainer)
output = tokenizer.encode("人工智能很强大")
print(output.tokens) # 输出子词如 ['人工', '智能', '很', '强大']
```

这段代码训练自定义 BPE：初始化 BPE 模型，用 Whitespace 预分词，Trainer 设置词汇表大小和特殊 token。`tokenizer.train` 从语料统计合并对，输出 tokenizer 对象。encode 处理新文本，展示子词拆分。这个教程让你 5 分钟上手自定义 tokenizer。NLTK/spaCy 适合规则入门，TikToken 专为 OpenAI 优化速度。基准显示 TikToken 每秒处理 100 万 token。动手试试吧!

7 结论与展望

Tokenization 从规则拆分演进到 BPE、WordPiece 等统计算法，成为 NLP 管道基石，桥接人类语言与数字模型，影响一切从 embedding 到推理的表现。

未来，动态 tokenization 按上下文自适应拆分，稀疏 tokenizer 压缩 token 数提升效率，多模态版本融合文本图像 token。行动起来：实验 Hugging Face 代码，分享你的 tokenizer 项目！

资源推荐：BPE 原论文「Neural Machine Translation of Rare Words with Subword Units」（arXiv:1508.07909），Hugging Face Tokenizers GitHub，tiktoken 在线 demo。你准备好构建下一个 NLP 项目了吗？