

Tokenization-文本的数字化

Token

Token释义:

在自然语言处理（NLP）中，“token”是指文本中的一个基本单元或组成部分。Token化（Tokenization）是将文本分割成这些单元的过程，这些单元可以是单词、数字、符号或者它们的组合。Token化的目的是为了更容易地处理文本数据，因为大多数NLP任务都需要以某种方式分析文本中的词汇和结构。

例如，考虑句子：“我今天去了图书馆。”在这个例子中，通过token化，这句话可以被分解为以下tokens:

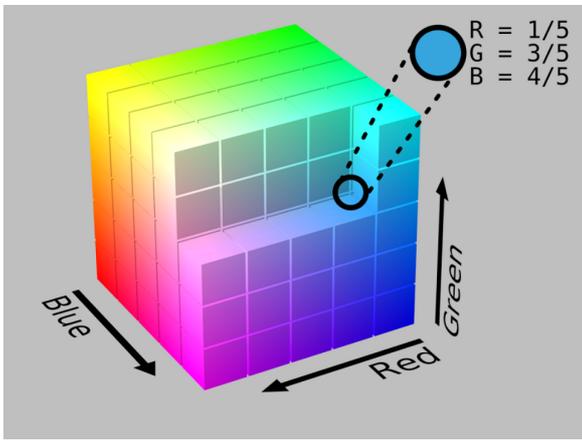
- 我
- 今天
- 去了
- 图书馆
- 。（标点符号也可以作为一个单独的token）

不同的NLP任务可能需要不同粒度的token化。例如，在一些情况下，你可能希望保留标点符号作为单独的tokens，而在其他情况下，则可能希望将其与相邻的单词合并。此外，对于像中文这样的语言，由于单词之间没有明显的空格分隔，token化可能会更加复杂，需要依赖于专门的算法来正确识别词语边界。

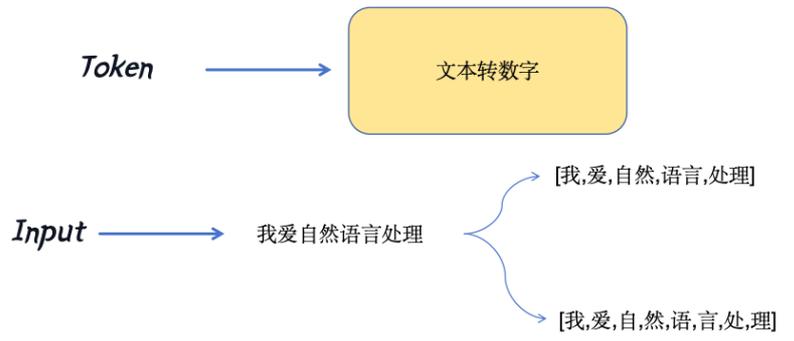
Token化是许多NLP流程的第一步。

通俗来讲，自然语言处理中文本由token组合而成，而每个token就是文本中的最小单元，文本本质上就是一连串token的序列。

为什么需要token?



CV中的像素值都是连续的，可以直接拿去训模型

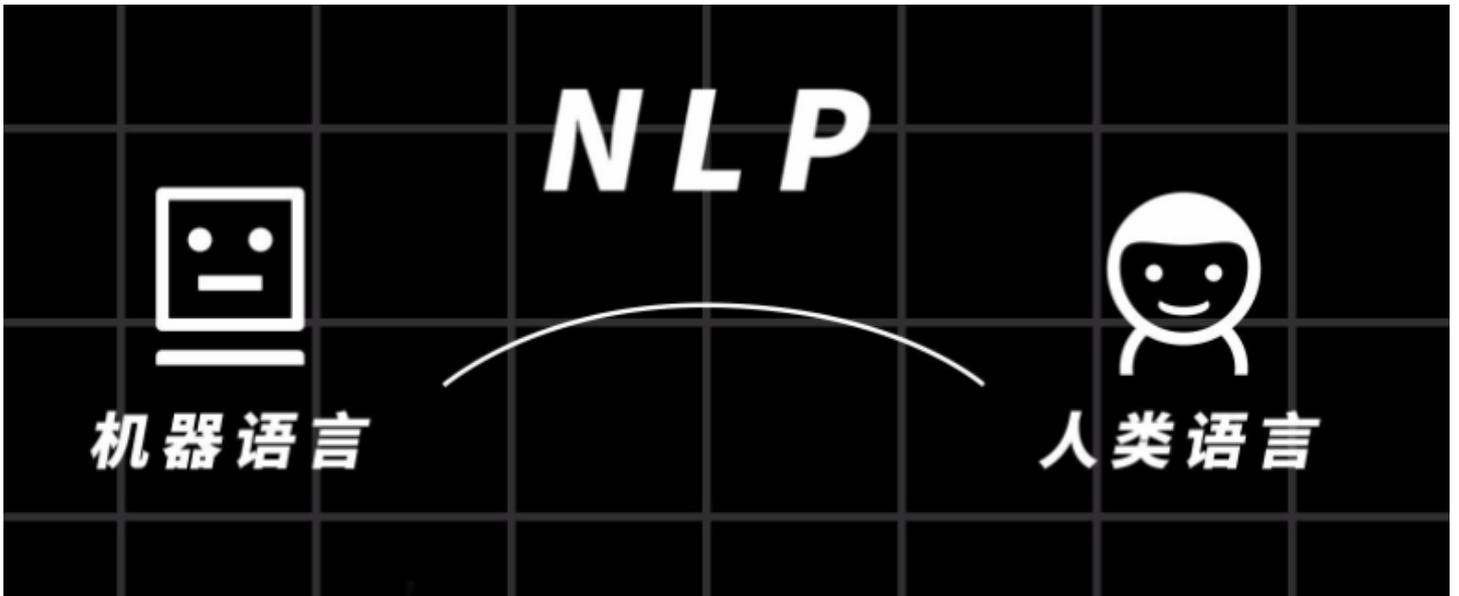


NLP中的文本是离散的，如何转换成计算机能够拿去运算的连续值，这是一个问题

Tokenization

Tokenization

Tokenization就是将一段文本切分为若干个token的过程。这也是我们将文本这种人类语言转化为机器能够看懂的机器语言的第一步，先切分。



NLP的前处理就是在完成这样的事情

Tokenization不同的切分策略

Tokenization

Token →

文字转数字

Input →

小沈阳江西演唱会邀请

小沈阳江西演唱会邀请了沈春阳，
演唱会一共开了3个多小时，
.....
中央调拨10亿发展东北新城村
建设。2024年国民GDP超过...

方法一

小 → 1,
沈 → 2,
阳 → 3,
江 → 4,
西 → 5,
演 → 6,
唱 → 7,
会 → 8,
邀 → 9,
请 → 10

简单匹配

小 → 1
沈 → 2
阳 → 3
... → ...
... → ...
遵 → 10000

Vocab_size = 10000

方法二

tiktoken(GPT):

小 → 31809,
沈 → 31106, 230,
阳 → 83175,
江 → 70277,
西 → 61786,
演 → 78256, 242,
唱 → 84150, 109,
会 → 38093,
邀 → 45932, 222,
请 → 15225

tiktoken("cl100k_base")

! → 0
" → 1
请 → 15225
... → ...
... → ...
Conveyor → 100255

Vocab_size = 100256

context_length = 16

tokenization的过程

朴素方法

word-based tokenization

 这是最常见的方式，即把文本按照空格和标点符号分割成单词序列。例如，句子 "Hello, world!" 可以被 tokenization 成 ["Hello", ",", "world", "!"]。

缺点

- 词汇表庞大：对于大规模语料库，词汇表可能非常大，导致内存消耗增加。
- OOV (Out-of-Vocabulary) 问题：未见过的单词无法被正确处理，特别是在处理专有名词、新词或拼写错误时。
- 语言依赖：对于没有明显单词边界的语言（如中文），这种方法效果不佳。

Char-based tokenization

 Char-based Tokenization将每个 unicode 和 ascii 字符表示为一个token，可以解决缺失词的问题

缺点

- 计算量大，每个单词需要标识为更长的token序列
- 字符本身缺乏语义，丢失了word本身的语义信息，模型学起来会很困难，缺乏泛化能力。

最佳实践 subword tokenization

最具代表性的算法：BPE算法，字节对编码（BPE, Byte Pair Encoder）。

BPE算法原理

构建词表

- 确定词表大小，即subword的最大个数V；
- 在每个单词最后添加一个</w>，并且统计每个单词出现的频率；
- 将所有单词拆分为单个字符，构建出初始的词表，此时词表的subword其实就是字符；
- 挑出频次最高的**字符对**，比如说 **t** 和 **h** 组成的 **th**，将新字符加入词表，然后将语料中所有该字符对融合（merge），即所有 **t** 和 **h** 都变为 **th**。新字符依然可以参与后续的 merge，有点类似**哈夫曼树**，BPE 实际上就是一种**贪心算法**；

- 1 统计输入中所有出现的单词并在每个单词后加一个单词结束符</w> -> ['hello</w>': 6, 'world</w>': 8, 'peace</w>': 2]
- 2 将所有单词拆成单字 -> {'h': 6, 'e': 10, 'l': 20, 'o': 14, 'w': 8, 'r': 8, 'd': 8, 'p': 2, 'a': 2, 'c': 2, '</w>': 3}

- 合并最频繁出现的单字(l, o) -> {'h': 6, 'e': 10, 'lo': 14, 'l': 6, 'w': 8, 'r': 8, 'd': 8, 'p': 2, 'a': 2, 'c': 2, '</w>': 3}
- 合并最频繁出现的单字(lo, e) -> {'h': 6, 'lo': 4, 'loe': 10, 'l': 6, 'w': 8, 'r': 8, 'd': 8, 'p': 2, 'a': 2, 'c': 2, '</w>': 3}
- 反复迭代直到满足停止条件

编码

词表构建完成后，需要对训练语料进行编码，编码流程如下：

- 将词表中的单词按长度从长到短进行排序；
- 对于语料中的每个单词，遍历排序好的词表，判断词表中的单词/子词（subword）是否是该字符串的子串，如果匹配上了，则输出当前子词，并继续遍历单词剩下的字符串。
- 如果遍历完词表，单词中仍然有子字符串没有被匹配，那我们将其替换为一个特殊的子词，比如 `<unk>`。

举个例子，假设我们现在构建好的词表为：

- "errrr</w>":1
- "tain</w>":2
- "moun":3
- "est</w>":4
- "high":5
- "the</w>":6
- "a</w>":7
- "ukn":8

对于给定的单词 `mountain</w>`，其分词结果为：`[moun, tain</w>]`，最终结果为`[3,2]`

解码

语料解码就是将所有的输出子词拼在一起，直到碰到结尾为 `<\w>`。举个例子，假设模型输出为：

- `[3,2,5,6]`
- 讲词表词汇一一对应可得
- `["moun", "tain</w>", "high", "the</w>"]`

那么其解码的结果为

- `["mountain</w>", "highthe</w>"]`

BPE思想下的改良：

 **BBPE (Byte-Level BPE)**: Byte-level BPE 迈向更通用的Tokenizer

对于英文、拉美体系的语言来说使用BPE分词足以在可接受的词表大小下解决OOV的问题，但面对中文、日文等语言时，其稀有的字符可能会不必要的占用**词汇表**，因此考虑使用字节级别**byte-level**解决不同语言进行分词时OOV的问题。具体的，BBPE考虑将一段文本的UTF-8编码(UTF-8保证任何语言都可以通用)中的一个字节256位不同的编码作为词表的初始化基础Subword。

相比ASCII只能覆盖英文中字符，UTF-8编码创建的本身就是为了通用的将世界上不同的语言字符尽可能全部用一套编码进行编号，同时相比UTF-32对于每个字符都采用4位字节 (byte) 过于冗长。改进的UTF-8编码是一个变长的编码，有1~4个范围的字节(bytes)长度。对于不同语言中字符采用不同长度的字节编码，例如英文字符基本都是1个字节 (byte) ，中文汉字通常需要2~3个字节。

 **补充知识点：**

在计算机中，每个字节(Byte) 有8位的**2进制编码**，在电脑显示时全部用2进制太冗长，因此每个字节(bytes) 通常使用2个**16进制编码** (0~F) 进行表示。例如: 字母 'A'的unicode-8 用十进制表示的值是：65，两位16进制表示就是：41

```
1 ord('A') #如果想知道某个字符utf-8编码，在python中使用内置函数ord()即可# 65
```

Tokenizer

 Tokenizer就是我们用来分词的工具，是一种分词思想的工程化具体实现

常用的Tokenizer：

- SentencePiece: LLama\baichuan
- Tiktoken:GPT

为什么要用？

需要海量文本训练，且重复性劳动大，并且可复用程度很高

注意

每个模型都要用自己对应的tokenizer!

如何使用

 Hugging Face 的 `transformers` 库是一个非常流行和强大的工具，支持多种预训练模型和相应的 tokenizer。以下是使用 `transformers` 库中的 tokenizer 的基本步骤：

使用 GPT 的 tokenizer 可以通过 Hugging Face 的 `transformers` 库来实现。GPT 模型（如 GPT-2 和 GPT-3）的 tokenizer 提供了方便的接口来处理文本数据。以下是详细的步骤：

安装 `transformers` 库

首先，确保你已经安装了 `transformers` 库。如果还没有安装，可以使用以下命令进行安装：

```
1 pip install transformers
```

导入所需的模块

```
1 from transformers import GPT2Tokenizer
```

加载预训练的 tokenizer

```
1 tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
```

对文本进行 tokenization

1. 分词 (Tokenize)

将文本字符串转换为 tokens 列表：

```
1 text = "Hello, how are you?"
2 tokens = tokenizer.tokenize(text)
3 print(tokens) # 输出: ['Hello', ',', ' ', 'How', ' are', ' you', ' ?']
```

注意：GPT-2 的 tokenizer 使用了一个特殊的前缀 `Ġ` 来表示单词的开头。

2. 将 tokens 转换为 IDs

将 tokens 转换为对应的 token IDs：

```
1 token_ids = tokenizer.convert_tokens_to_ids(tokens)
```

```
2 print(token_ids) # 输出: [15496, 11, 318, 326, 5956, 13])
```

3. 直接编码文本为 IDs

可以直接将文本编码为 token IDs，包括添加特殊 token（如 `[CLS]` 和 `[SEP]`）：

```
1 input_ids = tokenizer.encode(text, add_special_tokens=True)
2 print(input_ids) # 输出: [50256, 15496, 11, 318, 326, 5956, 13, 50256]
```

4. 解码 IDs 回文本

将 token IDs 解码回原始文本：

```
1 decoded_text = tokenizer.decode(input_ids)
2 print(decoded_text) # 输出: 'Hello, how are you?'
```

批量处理文本

如果你有一批文本需要处理，可以使用 `batch_encode_plus` 方法：

```
1 texts = ["Hello, how are you?", "I am fine, thank you."]
2 batch_encoding = tokenizer.batch_encode_plus(
3     texts,
4     padding=True,
5     truncation=True,
6     max_length=50,
7     return_tensors='pt' # 返回 PyTorch 张量
8 )
9
10 print(batch_encoding['input_ids'])
11 print(batch_encoding['attention_mask'])
```

处理特殊 token

GPT 模型通常使用 `EOS` 作为特殊 token。你可以通过 `tokenizer.eos_token` 获取这个特殊 token：

```
1 eos_token_id = tokenizer.eos_token_id
2 print(eos_token_id) # 输出: 50256
```



可视化token分享: *

你好, 世界

Clear Show example

Tokens Characters
3 5

你好, 世界

Text Token IDs

GPT-4o & GPT-4o mini

GPT-3.5 & GPT-4

GPT-3 (Legacy)

你好, 世界, 魑魅魍魎

Clear Show example

Tokens Characters
11 10

你好, 世界, 魑魑魑魑

Text Token IDs