从零实现大语言模型

Lec1：BPE 分词系统的算法设计与实现

Penghao Kuang

SIST, ShanghaiTech
ACM, DataTech, Geekpie

2025.10.19

Overview

PART 1
设计动机

大语言模型能否直接识别自然语言文本？

```
Hello helo, I'm
```

- 计算机只能识别处理数字编码，无法直接处理自然语言文本!
- 如何设计一种处理方法，使得自然语言文本能无损地转化成数字编码？

核心思路：按词表映射

假设拥有一个完整的词表：

- 词表中每一条目包含：词汇、词汇对应的数码编号
- 对照词表，将文本逐词映射为数码序列

例如词表按整词分割规则制定：

```
"Hello":0
" ":1
"helo":2
",":3
"I'm":4
```

则原文本 `Hello helo, I'm` 的映射结果为：`[0,1,2,3,1,4]`

核心思路：按词表映射

问题：

- 自然语言非常复杂且语种众多。如何制作完整的词表？
- 词表总大小是极为恐怖的
- 如何正确分词编码？
  - "United Kingdoms" 作为整词还是两个独立词？
  - 中文等无明显分词界限的语言如何分词？（“我一把把把把住了”）

整词分割规则显然有明显的局限性

尝试思路 2：utf-8 编码分割

我们存在一种现成的文本编码方式：

```
>>> print("h".encode("utf-8"))
b'h'
>>> print("你".encode("utf-8"))
b'\xe4\xbd\xa0'
>>> print("ô".encode("utf-8"))
b'\xc3\xb4'

>>> test_string="hello,你好"
>>> utf_encoded=test_string.encode("utf-8")
>>> print(utf_encoded)
b'hello,\xe4\xbd\xa0\xe5\xa5\xbd'
>>> list(utf_encoded)
[104, 101, 108, 108, 111, 44, 228, 189, 160, 229, 165, 189]
```

尝试思路 2：utf-8 编码分割

我们只需 256 种数字编码，就可表示所有类型的字符！

```
>>> test_string="hello,你好"
>>> utf_encoded=test_string.encode("utf-8")
>>> print(utf_encoded)
b'hello,\xe4\xbd\xa0\xe5\xa5\xbd'
>>> list(utf_encoded)
[104, 101, 108, 108, 111, 44, 228, 189, 160, 229, 165, 189]
```

我们可以直接按照现有的 utf-8 编码规范，将文本转化成 utf-8 编码序列！

尝试思路 2：utf-8 编码分割

是否存在问题？

- 相比整词分割，文本序列长度将显著膨胀
- 字符的信息密度显著下降，不利于模型捕捉文本信息

能否找到介于“整词分割”与“utf-8 编码”之间的折中方案？
⇒BPE 分词算法将有效解决词表制作与分词规则的上述问题！
⇒BPE 算法，可以理解成**基于统计规律，将分散的 utf-8 逐步合成一个个“整词”**

PART 2
核心算法

一、词表制作的流程

1. 初始条件与 utf-8 编码完全一致：拥有一个大小为 256 的映射词表
2. 每次选取语料库中相邻频率最高的字符对进行合并，形成一个全新词条

一、词表制作的流程

对于原文本 `Hello helo, I'm`：

1. 建立大小为 256 的原词表（编号 0-255）
2. 映射为序列 [72, 101, 108, 108, 111, 32, 104, 101, 108, 111, 44, 32, 73, 39, 109] (即:[H,e,l,l,o,' ',h,e,l,o,',' ⋯⋯])
3. 计算字符对相邻频率：(e,l):2 (l,o):2 (H,e):1 (o,' '):1 (o,',')⋯⋯
4. 取最大频率中字典序最大的词条 (l,o)：所有相邻的 (l,o) 合并成新字符"lo"
5. 词表增加词条：`lo:256`
6. 语料库变为:`H,e,l,lo, ,h,e,lo, ,I,',m`
7. 下一轮合并的相邻频率变为：(e,l):1 (e,lo):1 (lo,):2⋯⋯

一、词表制作的流程

但在实际工程中，我们还需要考虑更多的细节

- 语料库可能由大量无关的文本拼接而成，由”<|endoftext|>” 符号分开
- 我们倾向于将整个文本粗糙地分成一系列 “整词”，再运行 BPE 算法
  - 例如原文本期望分割成：`"Hello" " helo" "," " I" "'m"`
  - 这样可以有效降低空格、标点、特殊符号对词结构本身的影响
  - 并且避免无意义的跨词合并: 若出现了一个合成结果`"Hello helo,"`——这是否有实际意义？

```
u don't have to be scared of the loud dog, I'll protect you
kept him safe. The mole had found his best friend.
<|endoftext|>
Once upon a time, in a warm and sunny place, there was a bi
Tom asked his friend, Sam, to help him search for the ball.
Sam and Tom went close to the pit. They were scared, but th
They went into the pit to search. It was dark and scary. Th
but no one could hear them. They were sad and scared, and t
<|endoftext|>
```

一、词表制作的流程

- 原始词表大小调整为 257:256 个基本映射词条 + ("<|endoftext|>": 256）词条
- 利用正则表达式进行“整词”粗略分割，以此限定合并的极限

这就是我们的"Pre-tokenization"（预词元分割）步骤。

词表制作：Pre-tokenization

如何对一个语料库进行预词元分割？

```
intj intj intj intj intj tech tech<|endoftext|>

Héllò hôw <|endoftext|><|endoftext|> are ü?
```

1. 先以 <|endoftext|> 为界，拆成若干个文本单位
2. 在每个文本单位中，用正则表达式进行“整词”拆分

词表制作：Pre-tokenization

```
intj intj intj intj intj tech tech<|endoftext|>

Héllò hôw <|endoftext|><|endoftext|> are ü?
```

分割结果：

```
chunk a text:b'intj intj intj intj intj tech tech'
chunk a text:b'<|endoftext|>'
chunk a text:b'\n\nH\xc3\xa9ll\xc3\xb2 h\xc3\xb4w '
chunk a text:b'<|endoftext|>'
chunk a text:b'<|endoftext|>'
chunk a text:b' are \xc3\xbc? '
```

词表制作：Pre-tokenization

采用正则表达式分割。GPT-2 的分词标准为：

```
r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"""
```

部分文本的切割效果：

```
chunk a text:b'intj intj intj intj intj tech tech'
>>>>>split a token:b'intj'
>>>>>split a token:b' intj'
>>>>>split a token:b' intj'
>>>>>split a token:b' intj'
>>>>>split a token:b' intj'
>>>>>split a token:b' tech'
>>>>>split a token:b' tech'
chunk a text:b'<|endoftext|>'
>>>>>split a token:b'<|endoftext|>'
```

代码讲解：Pre-tokenization

词表制作：合并

修改后的合并方法是：

- Pre-tokenization 生成一系列粗略的整词
- 将每个“整词”全部打散为单个 utf-8 编码组成的序列
- 跑遍整个语料库的所有打散序列，统计相邻词对频率
- 按照 BPE 算法，选择最高频词对合并
- 再次跑遍整个语料库的所有打散序列，更改需要合并的打散字符
- 一次合并完成，循环进行后续合并

词表制作：合并

对于原文本 `Hello helo, I'm`：

1. 建立大小为 256+1 的原词表（编号 0-255 与 256 号的 <|endoftext|>）
2. Pre-tokenization:["Hello"; " helo" ; "," ; " I" ; "'m"]
3. 计算字符对相邻频率：(e,l):2 (l,o):2 (H,e):1 · · · · · · (o,' ') 与 (o,',') 不复存在!
4. 取最大频率中字典序最大的词条 (l,o)：所有相邻的 (l,o) 合并成新字符"lo"
5. 词表增加词条：`lo:257`
6. 语料库修改，进行下一轮循环

词表制作：合并

最后得出的词表示例：

418: b' by', 419: b'os', 420: b'ant', 421: b' G', 422: b' from',
423: b'ive', 424: b' al', 425: b"'s", 426: b' su', 427: b'el',
……
5569: b' turning', 5570: b' Please', 5571: b' Chief', 5572: b' quant'
……
9801: b' Elizabeth', 9802: b'igate', 9803: b'TR', 9804: b' cognitive'

简单词根与小词 → 高频整词 → 专有名词、次高频词根、次高频词等
往往只需要万级的词表大小，就可以高效表示出大多数的整词及几乎所有的词根
⇒ 高频词可直接整体编码、低频词可按词根编码

二、文本编码

合并语料：`in the`→[i,n][,t,h,e]

- 为防止跨词合并，同样先进行 Pre-tokenization
- 在形成词表的过程中同步记录合并的词对。如 (b' ', b't'), (b' ', b'a'), (b'h', b'e'), (b'i', b'n'), (b'r', b'e'), (b' t', b'he'), (b'o', b'n'), (b'e', b'r')
- 对于待编码的文本：按照合并列表的顺序进行合并：
- 将语料库中所有相邻的 (b' ', b't') 合并为 b' t'→[i,n][t,h,e]
- 将语料库中所有相邻的 (b' ', b'a') 合并为 b' a'→[i,n][t,h,e]
- 将语料库中所有相邻的 (b'h', b'e') 合并为 b'he'→[i,n][t,he]
- 将语料库中所有相邻的 (b'i', b'n') 合并为 b'in'→[in][t,he]

由此可将最初的一长串 utf-8 编号不断合并，合并为一系列可表征整词、有意义词根的词条编号！

耗时问题

- 对于极小规模的语料库，不需要考虑耗时问题
- 正规的词表制作中，语料库往往是几百 MB 规模
- 每一次合并都需跑遍整个语料库进行统计和更改……
- 对于文本编码的合并操作同样需要大量遍历整个语料库
- 上万大小的词表将对应上万次的语料库遍历

以 130KB 大小的语料库为例，上述算法的实际运行时长约为 7 秒。对于几百 MB 级别的语料库，这种算法的时间开销是不可承受的！

PART 3

算法优化 1：词表训练中的合并算法

原算法的耗时问题在于：

- 每次合并，都对整个语料库中（包括需要合并和无需合并）的内容进行遍历。
- 每次合并都需要生成全新的词对计数表（许多词对相比合并前并没有变化）

所以我们期望设计出一种算法：

- 能够维护语料库中的词对信息，使得每次合并都能精准更新**产生变化的词对**
- 能够维护各词对的计数，使得每次合并只需精准更新计数表的少量条目，**而非暴力更新语料库**

token_idx 的预处理：将 tokens 定位编号

```
intj intj intj intj intj tech tech<|endoftext|>
```

```
Héllò hôw <|endoftext|><|endoftext|> are ü?
```

```
Token idx: 0, Token: 105, Char: b'i'
Token idx: 1, Token: 110, Char: b'n'
Token idx: 2, Token: 116, Char: b't'
Token idx: 3, Token: 106, Char: b'j'
Token idx: 4, Token: 32, Char: b' '
Token idx: 5, Token: 105, Char: b'i'
Token idx: 6, Token: 110, Char: b'n'
Token idx: 7, Token: 116, Char: b't'
Token idx: 8, Token: 106, Char: b'j'
Token idx: 9, Token: 32, Char: b' '
Token idx: 10, Token: 105, Char: b'i'
Token idx: 11, Token: 110, Char: b'n'
Token idx: 12, Token: 116, Char: b't'
Token idx: 13, Token: 106, Char: b'j'
Token idx: 14, Token: 32, Char: b' '
Token idx: 15, Token: 105, Char: b'i'
Token idx: 16, Token: 110, Char: b'n'
Token idx: 17, Token: 116, Char: b't'
Token idx: 18, Token: 106, Char: b'j'
Token idx: 19, Token: 32, Char: b' '
Token idx: 20, Token: 105, Char: b'i'
Token idx: 21, Token: 110, Char: b'n'
Token idx: 22, Token: 116, Char: b't'
Token idx: 23, Token: 106, Char: b'j'
Token idx: 24, Token: 32, Char: b' '
Token idx: 25, Token: 116, Char: b't'
Token idx: 26, Token: 101, Char: b'e'
Token idx: 27, Token: 99, Char: b'c'
Token idx: 28, Token: 104, Char: b'h'
Token idx: 29, Token: 32, Char: b' '
```

```
Token idx: 30, Token: 116, Char: b't'
Token idx: 31, Token: 101, Char: b'e'
Token idx: 32, Token: 99, Char: b'c'
Token idx: 33, Token: 104, Char: b'h'
Token idx: 47, Token: 10, Char: b'\n'
Token idx: 48, Token: 10, Char: b'\n'
Token idx: 49, Token: 72, Char: b'H'
Token idx: 50, Token: 195, Char: b'\xc3'
Token idx: 51, Token: 169, Char: b'\xa9'
Token idx: 52, Token: 108, Char: b'l'
Token idx: 53, Token: 108, Char: b'l'
Token idx: 54, Token: 195, Char: b'\xc3'
Token idx: 55, Token: 178, Char: b'\xb2'
Token idx: 56, Token: 32, Char: b' '
Token idx: 57, Token: 104, Char: b'h'
Token idx: 58, Token: 195, Char: b'\xc3'
Token idx: 59, Token: 180, Char: b'\xb4'
Token idx: 60, Token: 119, Char: b'w'
Token idx: 61, Token: 32, Char: b' '
Token idx: 88, Token: 32, Char: b' '
Token idx: 89, Token: 97, Char: b'a'
Token idx: 90, Token: 114, Char: b'r'
Token idx: 91, Token: 101, Char: b'e'
Token idx: 92, Token: 32, Char: b' '
Token idx: 93, Token: 195, Char: b'\xc3'
Token idx: 94, Token: 188, Char: b'\xbc'
Token idx: 95, Token: 63, Char: b'?'
Token idx: 96, Token: 32, Char: b' '
```

pair_positions 的预处理：将词对定位编号

```
intj intj intj intj intj tech tech<|endoftext|>

Héllò hôw <|endoftext|><|endoftext|> are ü?
```

```
Pair: (105, 110), Char_Pair:((b'i', b'n')),  Positions: [0, 5, 10, 15, 20]
Pair: (110, 116), Char_Pair:((b'n', b't')),  Positions: [1, 6, 11, 16, 21]
Pair: (116, 106), Char_Pair:((b't', b'j')),  Positions: [2, 7, 12, 17, 22]
Pair: (32, 105), Char_Pair:((b' ', b'i')),  Positions: [4, 9, 14, 19]
Pair: (32, 116), Char_Pair:((b' ', b't')),  Positions: [24, 29]
Pair: (116, 101), Char_Pair:((b't', b'e')),  Positions: [25, 30]
Pair: (101, 99), Char_Pair:((b'e', b'c')),  Positions: [26, 31]
Pair: (99, 104), Char_Pair:((b'c', b'h')),  Positions: [27, 32]
Pair: (72, 195), Char_Pair:((b'H', b'\xc3')),  Positions: [49]
Pair: (195, 169), Char_Pair:((b'\xc3', b'\xa9')),  Positions: [50]
Pair: (169, 108), Char_Pair:((b'\xa9', b'l')),  Positions: [51]
Pair: (108, 108), Char_Pair:((b'l', b'l')),  Positions: [52]
Pair: (108, 195), Char_Pair:((b'l', b'\xc3')),  Positions: [53]
Pair: (195, 178), Char_Pair:((b'\xc3', b'\xb2')),  Positions: [54]
Pair: (32, 104), Char_Pair:((b' ', b'h')),  Positions: [56]
Pair: (104, 195), Char_Pair:((b'h', b'\xc3')),  Positions: [57]
Pair: (195, 180), Char_Pair:((b'\xc3', b'\xb4')),  Positions: [58]
Pair: (180, 119), Char_Pair:((b'\xb4', b'w')),  Positions: [59]
Pair: (32, 97), Char_Pair:((b' ', b'a')),  Positions: [88]
Pair: (97, 114), Char_Pair:((b'a', b'r')),  Positions: [89]
Pair: (114, 101), Char_Pair:((b'r', b'e')),  Positions: [90]
Pair: (32, 195), Char_Pair:((b' ', b'\xc3')),  Positions: [92]
Pair: (195, 188), Char_Pair:((b'\xc3', b'\xbc')),  Positions: [93]
```

pair_counter 的预处理：只需维护词对计数

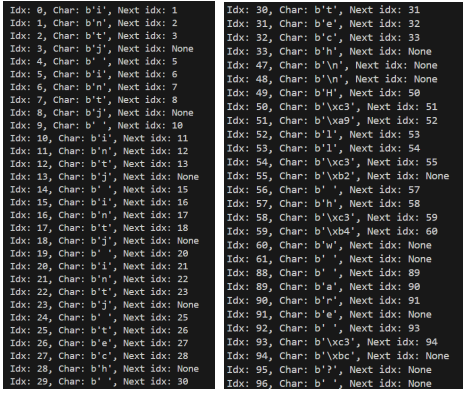
```
intj intj intj intj intj tech tech<|endoftext|>

Héllò hôw <|endoftext|><|endoftext|> are ü?
```

```
Pair: (105, 110), Char_Pair:((b'i', b'n')),  Counted: 5
Pair: (110, 116), Char_Pair:((b'n', b't')),  Counted: 5
Pair: (116, 106), Char_Pair:((b't', b'j')),  Counted: 5
Pair: (32, 105), Char_Pair:((b' ', b'i')),  Counted: 4
Pair: (32, 116), Char_Pair:((b' ', b't')),  Counted: 2
Pair: (116, 101), Char_Pair:((b't', b'e')),  Counted: 2
Pair: (101, 99), Char_Pair:((b'e', b'c')),  Counted: 2
Pair: (99, 104), Char_Pair:((b'c', b'h')),  Counted: 2
Pair: (72, 195), Char_Pair:((b'H', b'\xc3')),  Counted: 1
Pair: (195, 169), Char_Pair:((b'\xc3', b'\xa9')),  Counted: 1
Pair: (169, 108), Char_Pair:((b'\xa9', b'l')),  Counted: 1
Pair: (108, 108), Char_Pair:((b'l', b'l')),  Counted: 1
Pair: (108, 195), Char_Pair:((b'l', b'\xc3')),  Counted: 1
Pair: (195, 178), Char_Pair:((b'\xc3', b'\xb2')),  Counted: 1
Pair: (32, 104), Char_Pair:((b' ', b'h')),  Counted: 1
Pair: (104, 195), Char_Pair:((b'h', b'\xc3')),  Counted: 1
Pair: (195, 180), Char_Pair:((b'\xc3', b'\xb4')),  Counted: 1
Pair: (180, 119), Char_Pair:((b'\xb4', b'w')),  Counted: 1
Pair: (32, 97), Char_Pair:((b' ', b'a')),  Counted: 1
Pair: (97, 114), Char_Pair:((b'a', b'r')),  Counted: 1
Pair: (114, 101), Char_Pair:((b'r', b'e')),  Counted: 1
Pair: (32, 195), Char_Pair:((b' ', b'\xc3')),  Counted: 1
Pair: (195, 188), Char_Pair:((b'\xc3', b'\xbc')),  Counted: 1
```

next 的预处理 (prev 同理)

```
intj intj intj intj intj tech tech<|endoftext|>
```

```
Héllò hôw <|endoftext|><|endoftext|> are ü?
```



需要准备的数据结构

词对信息的维护：

- 为所有字符设置“位置编号”（token_idx）
- 建立字典，key 为词对，value 为该词对的所有位置：词对位置设定为第一字符的位置编号（pair_positions）
- 设置 prev、next 字典，key 为位置编号，value 为前、后的位置编号。可能为 None。
- 设置词对计数器字典。key 为词对，value 为该词对的总出现次数。（pair_counter）
- 设置优先队列维护词对计数信息。（heap）

每次合并的更改过程

- 选定需要合并的词对 (pair0,pair1)
- 遍历该词对对应的所有位置
  - 确认字符无误
  - 获取 pair0 的前一字符 prev_token、pair1 的后一字符 next_token
  - prev_token、pair0、pair1、next_token、new 相关词对的计数器更新
  - pair0、pair1 对应位置编号的字符更新
  - 相关词对位置编号列表更新（只增不删）
  - prev、next 更新
- 优先队列更新（采用 lazy update）

实际算法运行

- 选择计数最大的 (b't',b'j'), 合并为 new=b'tj'
- 对于 pair_positions 中 (b't',b'j') 对应的所有位置（所有 pairs）：
  - 该位置 b't' 前一个字符 prev_token、b'j' 后一个字符 next_token
  - (prev_token,b't')(prev_token,new)(b'j',next_token)(new,next_token) 计数器更新
  - b'j' 不复存在，token_idx 该下标改为 None；b't' 处改为 b'tj'
  - (prev_token,new)(new,next_token) 的 pair_positions 列表增加元素（lazy update）
  - b'tj' 下标的 next 改为 next_token 下标；next_token 下标改为 b'tj' 下标
- 优先队列 heap：将所有受影响的 pairs 及计数插入（先前数据不删除）

代码讲解：合并算法优化

PART 4
算法优化 2：文本编码

逆向思维

```
corpus1:tech→["tech"]
corpus2:tech tech→["tech"," tech"]
```

- 预处理中经过了 Pre-tokenization：不同整词之间互不干扰
- 原本编码算法改成：逐个对单一“整词”进行合并编码，正确性不受影响
- **每个整词往往只需要很少几次合并！** ⇒ 能否找出先后有哪几个合并？
- 对于每个“整词”：枚举所有的相邻词对
- 如果该词对可合并（在合并列表中），查询其在合并列表中的位置
- 位置最靠前的一定被最先合并

实际算法运行

```
original_token: tech
pair:(b' ', b't'),number:0
pair:(b't', b'e'),number:465
pair:(b'e', b'c'),number:505
pair:(b'c', b'h'),number:98
finish a merge,token_list:[b' t', b'e', b'c', b'h']
pair:(b' t', b'e'),number:287
pair:(b'e', b'c'),number:505
pair:(b'c', b'h'),number:98
finish a merge,token_list:[b' t', b'e', b'ch']
pair:(b' t', b'e'),number:287
pair:(b'e', b'ch'),number:2614
finish a merge,token_list:[b' te', b'ch']
pair:(b' te', b'ch'),number:5739
finish a merge,token_list:[b' tech']
```

代码讲解：文本编码优化

PART 5
实操演示

我们将实操演示：

- 21MB 语料库上 1000 大小词表的训练
- 32000 大小词表上的文本编码与解码

谢谢！