



# cs336 动手学大模型

*Stanford CS336 | Language Modeling from Scratch*

小红书同名作者：喂喂薇

# 2

## Assignment 1 基础篇:Building a Transformer LM

### 2.1. Assignment 1 Overview

#### 参考资料 2.1

- uv 官方文档: <https://docs.astral.sh/uv/>
- uv 中文基础使用教程: <https://www.runoob.com/python3/uv-tutorial.html>
- 服务器租借平台: <https://www.autodl.com/>(平台不止这一个, 我感觉这个比较方便)

#### 2.1.1. 每一部分的实验任务

1. BPE 分词器 (Byte-pair encoding (BPE) tokenizer)
2. Transformer 语言模型 (Transformer language model (LM))
3. 交叉熵损失函数与 AdamW 优化器
4. 完整实现模型训练

#### 2.1.2. 准备工作

1. 准备一台配备 GPU 和 Linux 系统的电脑;
2. 下载代码仓库;

Assignment 1 仓库地址: [github.com/stanford-cs336/assignment1-basics](https://github.com/stanford-cs336/assignment1-basics)

3. 下载数据集:

TinyStory: [huggingface.co/datasets/roneneldan/TinyStories/resolve/main/](https://huggingface.co/datasets/roneneldan/TinyStories/resolve/main/)

OpenWebText: [skylion007.github.io/OpenWebTextCorpus/](https://skylion007.github.io/OpenWebTextCorpus/)

4. 学习了解 uv 的基础使用方法

#### 微言大义 2.1

cs336 中的实验代码是 uv 管理的, 因此需要学习了解 uv 的基础使用方法 (uv 是比较流行的现代 python 项目管理工具, 学了不亏 ✕)。

另外 cs336 的 5 个 Assignment 的目录格式基本都是一个 basics/, 一个 tests/, 其中 basics/ 目录下可以用来写自己的实验代码 (当然也可以写作别的地方), tests/ 目录下是测试代码。此外 tests/ 目录下的 adapter.py 是测试文件的

#### 2.1 Assignment 1

Overview . . . . .	11
2.1.1 每一部分的实验任务 . . . . .	11
2.1.2 准备工作 . . . . .	11

#### 2.2 Byte-pair encoding

(BPE) tokenizer . . . . .	12
2.2.1 Unicode 字符集 . . . . .	12
2.2.2 subword tokenizer(子词分词器) . . . . .	17

#### 2.3 BPE Tokenizer

Training . . . . .	19
2.3.1 初始化词汇表 . . . . .	20
2.3.2 预分词 . . . . .	20
2.3.3 BPE 合并 . . . . .	21
2.3.4 特殊标记 . . . . .	22
2.3.5 BPE 分词器训练示例 . . . . .	22
2.4 BPE 分词器训练实操	22
2.5 利用 BPE 分词器进行编码和解码 . . . . .	24
2.5.1 编码 . . . . .	24
2.5.2 解码 . . . . .	26
2.6 BPE 讲义相关 Problems . . . . .	27
2.7 BPE 章节实验 . . . . .	27

适配器, 类似一个接口, 作业文件需要实现这个接口才能被测试文件正确调用。

## 2.2. Byte-pair encoding (BPE) tokenizer

### 参考资料 2.2

UTF-8 编码详解-CSDN 博客: <https://blog.csdn.net/whahu1989/article/details/118314154>

UTF-8 编码原理及与 ASCII 的兼容性-CSDN 博客: [https://blog.csdn.net/baidu\\_25299117/article/details/139633315](https://blog.csdn.net/baidu_25299117/article/details/139633315)

2.2 实验目标: 实现一个 BPE 分词器。对应测试文件: tests/test\_tokenizer.py

### 2.2.1. Unicode 字符集

在计算机发展的早期, 不同国家和地区为了表示本国语言的字符, 分别制定了自己的编码系统, 例如: 美国的**ASCII**, 中国的**GB2312**等。但这些编码系统互不兼容, 就很不方便。

而 Unicode(统一码) 是一种统一的字符集标准, 为世界上所有文字和符号分配了一个唯一的编号 (称为“码点” code point)。比如汉字“牛”的码点是“U+29275”, 其中“U+”是一个无意义前缀, 表示这是一个 Unicode 码点, “29275”表示这个码点的十进制值。

### Extension 2.1 Unicode 字符集代码实操

在 python 里, 我们可以使用**ord()**函数获取一个字符的码点, 使用**chr()**函数获取一个码点对应的字符。比如:

```
ord('牛')
>>> 29275
chr(29275)
>>> '牛'
```

Unicode 定义了“**字符↔码点**”的映射, 而 UTF(Unicode Transformation Format) 则进一步定义如何把**码点**编码为**字节**。常见的 UTF 编码有 UTF-8, UTF-16, UTF-32 等。

### UTF-8 编码

UTF-8 是一种**变长编码**, 使用 **1~4 个字节**表示。unicode 字符, 是互联网的主导编码格式 (占所有网页的 98% 以上)。

变长编码也就是说不同的字符编码的结果长度不一定相同, 有的是 1 个字节, 有的是 2 个字节或 3、4 个字节。

UTF-8 编码和 Unicode 码点范围的对应关系:

表 2.1: UTF-8 编码规则对照表

字节数	UTF-8 字节序列 (二进制)	Unicode 码点范围 (十六进制)	Unicode 码点范围 (十进制)
1	0xxxxxxx	U+0000 至 U+007F	0~127 (7 bits)
2	110xxxxx 10xxxxxx	U+0080 至 U+07FF	128~2047 (11 bits)
3	1110xxxx 10xxxxxx 10xxxxxx	U+0800 至 U+FFFF	2048~65535 (16 bits)
4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	U+10000 至 U+10FFFF	65536~1114111 (21 bits)

所有存储的字节被分为两类:**领头字节**(Leading Byte) 和**后续字节**(Continuation Byte)。

### 1. 单字节字符 (ASCII 范围)

- **规则:** 如果一个字节的最高位(第 1 位)是‘0’,那么它就是一个单字节字符。
- **格式:**‘0xxxxxxx’
- **解读:**1 字节时,UTF-8 完全等价于 ASCII:ASCII 0x41(A)  $\leftrightarrow$  UTF-8 0x41,UTF-8 编码格式是 0xxxxxxx,正好能容纳 ASCII 0~127。

### 2. 多字节字符

- **规则:** 如果一个字节的开头是‘1’,那它就是一个多字节字符的其中一部分(**开头有几个连续的 1 就代表几个字节**)。
- **领头字节:**
  - ‘110xxxxx’: 表示这是一个**双字节**字符的第一个字节。
  - ‘1110xxxx’: 表示这是一个**三字节**字符的第一个字节。
  - ‘11110xxx’: 表示这是一个**四字节**字符的第一个字节。
- **后续字节:**
  - ‘10xxxxxx’: 所有非领头的后续字节,都必须以‘10’开头。

#### Example 2.1 (UTF-8 编码示例)

汉字“中”的 Unicode 码点是 U+4E2D。U+4E2D 在 U+0800 到 U+FFFF 之间,根据规则,它需要用 3 个字节来编码。

首先将 U+4E2D 转换为二进制:

```

4 → 0100
E → 1110
2 → 0010
D → 1101

```

所以,U+4E2D 的二进制是 0100 1110 0010 1101。(总共 16 位)

3 字节的 UTF-8 模板是:1110xxxx 10xxxxxx 10xxxxxx,将二进制填入模板中,得到:11100100 10111000 10001011。

所以, 汉字“中”的UTF-8编码是:11100100 10111000 10001011。

### Extension 2.2 Unicode 编码实操

在 python 里, 对于 string 类型数据, 我们可以使用 encode() 方法将字符串编码为 UTF-8 编码, 使用 decode() 方法将 UTF-8 编码解码为字符串。比如:

```
test_string = "hello! こんにちは!"
utf8_encoded = test_string.encode("utf-8")
>>> b'hello! \xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\x81\xe3\x81\xaf!'
list(utf8_encoded)
>>> [104, 101, 108, 108, 111, 33, 32, 227, 129, 147, 227, 130, 147, 227, 129, 171, 227,
    ← 129, 161, 227, 129, 175, 33]
utf8_encoded.decode("utf-8")
>>> "hello! こんにちは!"
```

通过 UTF-8 编码将 Unicode 码点转换为字节序列, 我们本质上是在将码点序列(0 到 154997 范围内的整数)转换为字节值序列(0 到 255 范围内的整数)。256 长度的字节词汇表处理起来要容易得多。使用字节级分词时, 我们无需担心词汇表外的标记, 因为我们知道**任何输入文本都可以表示为 0 到 255 的整数序列**。

### overlong 编码

UTF-8 要求用**最短的字节数**编码每个字符。

比如 U+002F(十进制下为 47, 十六进制下为 0x2F, 表示符号为 “/”)按照规则只能用 1 个字节编码, 但如果我们非要把它编码为 2 个字节(0xC0 0xAF), 它解码出来:

- 0xC0 0xAF → 11000000 10101111
- 11000000 10101111 → 47 → U+002F

同样也是 “/”, 这就是**overlong 编码**。

overlong 编码不守 UTF8 的最小字节编码的规矩, 因此是被明确禁止的, 如果采用 overlong 编码有可能绕过防火墙啥的, 有很大**危险性**。

## Unicode 相关 Problem

### Problem 2.1

1. **chr(0) 返回什么 Unicode 字符？**

答: 返回空字符

2. **这个字符的字符串表示（\_\_repr\_\_()）与其打印表示有何不同？**

答: 它的打印表示通常是不可见的（没有视觉输出），而其字符串表示  
\_\_repr\_\_() 是一个明确的转义序列'  
x00'。

3. **当这个字符出现在文本中会发生什么？**

答: 会出现一个截断，效果类似空格。

## Problem 2.2

**1. 选择在 UTF-8 编码字节而非 UTF-16 或 UTF-32 上训练分词器的原因有哪些?**

答:1. UTF-8 是一种变长编码, 对英文、数字和常用符号等只用 1 个字节表示, 而像汉字等字符通常用 3 个字节。相比之下,UTF-16 至少需要 2 个字节,UTF-32 固定为 4 个字节。对于以英文为主的语料库,UTF-8 的存储和处理效率远高于后两者

2. 一个分词器的词汇表 (Vocabulary) 大小是有限的。如果直接在 Unicode 字符 (UTF-32) 上操作, 遇到词汇表中没有的字符 (例如一个新的 Emoji 或一个罕见的汉字), 就只能将其标记为<UNK>。而基于 UTF-8 字节的分词器, 其基础词汇表是固定的 256 个字节 (从 0x00 到 0xFF)。任何未知的、罕见的字符, 甚至是乱码, 都可以被分解成一串已知的字节序列来表示。这样就从根本上消除了<UNK> 符号, 使得模型能够处理任何形式的文本输入, 而不会丢失信息。

**2. 为什么如下函数是错误的 ?**

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    return "".join([bytes([b]).decode("utf-8") for b in
                   bytestring])
>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8"))
'hello'
```

答:UTF-8 是一种变长编码, 一个 Unicode 字符可能由 1 到 4 个字节组成。

对于标准的 ASCII 字符 (如'h', 'e', 'l', 'l', 'o'), 它们在 UTF-8 中确实只由单个字节表示, 所以这个函数对纯 ASCII 字符串"hello".encode("utf-8") 能够侥幸成功。

但是, 对于任何非 ASCII 字符, 需要用多个字节表示, 比如汉字 “中”, 它的 UTF-8 编码是 b'\xe4\xb8\xad', 由三个字节组成。当 decode\_utf8\_bytes\_to\_str\_wrong 函数处理 b'\xe4\xb8\xad' 时, 它会: 取出第一个字节 b'\xe4', 并尝试执行 bytes([b'\xe4']).decode("utf-8")。0xe4 (二进制 11100100) 是一个多字节字符的“起始字节”, 它告诉解码器“后面还跟着 2 个字节”。单独解码它必然会失败, 因为它的序列不完整。

此时,Python 会抛出 UnicodeDecodeError 异常, 因为遇到了一个不完整或无效的 UTF-8 序列。**正确的解码方式必须在完整的字节序列上进行, 而不是逐字节地割裂进行。**

**3. 提供一个双字节序列, 该序列无法解码为任何 Unicode 字符**

答: 一个双字节字符的起始字节的二进制格式必须是 110xxxxx 10xxxxxx, 因此只要不满足这个格式, 就不能解码。

## 薇言大义 2.2

Unicode 尤其 UTF8 编码是后面训练 BPE 分词器的基础概念, 内容实际上就是信息论的延伸, 理解了会对很多方面都有帮助。

### 2.2.2. subword tokenizer(子词分词器)

#### 参考资料 2.3

*jieba 分词:* [https://blog.csdn.net/qq\\_33957603/article/details/124640588](https://blog.csdn.net/qq_33957603/article/details/124640588)

**核心知识:**词级 (word-level) 分词器、字符级 (character-level) 分词器、字节级 (byte-level) 分词器、子词级 (subword-level) 分词器之间的区别和联系

#### 词级 (word-level) 分词器

- 核心思想: 最符合人类直觉的方式, 直接将句子按照空格或标点符号切分成一个个独立的单词。对于中文等没有天然分隔符的语言, 则需要依赖特定的分词算法 (如 **jieba 分词**)。
- 工作流程:
  1. **预处理 (可选)**: 可能包括小写化、去除标点、处理特殊字符等。
  2. **切分**: 主要根据空格和标点符号将句子切分成单词。例如, "Hello, world!" 可能会被切分为 ["Hello", "", "world", "!"] 或 ["Hello", "world"] (如果标点被移除或单独处理)。
  3. **词汇表构建**:
    - 在训练数据上统计所有出现的词语及其**频率**。
    - 选择频率最高的 N 个词语构成词汇表 (vocabulary)。
    - 词汇表之外的词语 (未登录词, **Out-Of-Vocabulary, OOV**) 通常会被映射到一个特殊的 <UNK>(unknown) 标记。
  4. **Token ID 映射**: 将每个词语映射到其在词汇表中的唯一整数 ID。
- 优点: 语义完整, 每个 token 都是一个有完整意义的词, 非常直观。
- 缺点:
  - **词汇表巨大**: 需要为语言中几乎所有的词都创建一个条目, 词汇表高达几十万甚至上百万。
  - **OOV (Out-of-Vocabulary) 问题严重**: 当遇到一个词汇表中没有的词 (如新词、拼写错误、专业术语), 分词器就无法处理, 通常会将其替换为一个特殊的 <UNK>(unknown) 符号, 导致信息丢失。例如, 模型没见过 "chatbot", 就会将其视为 <UNK>。
  - **无法处理词形变化**: run、running、ran 会被视为三个完全不同的词, 模型无法直接看出它们之间的关联, 增加了学习负担。

### 字符级 (character-level) 分词器

- 核心思想: 将文本拆分成一个个独立的字符。
- 示例:
  1. 英文: "I am a pig" → ["I", "a", "m", "a", "p", "i", "g"]
  2. 中文: "我是一头猪" → ["我", "是", "一", "头", "猪"]
- 优点:
  - **词汇表小**: 词汇表只包含所有基本字符 (如 a-z, A-Z, 0-9, 标点, 中文字符等), 大小非常可控。
  - **无 OOV 问题**: 任何单词都可以由字符组成, 因此不存在未知词的问题。
- 缺点:
  - **序列过长**: 一个单词会被切成多个字符, 过于琐碎导致输入序列的长度急剧增加, 对模型的计算和内存都是巨大挑战。
  - **语义丢失**: 单个字符通常不具备独立的语义, 模型需要从头学习如何将字符组合成有意义的词, 学习效率非常低。

### 字节级 (byte-level) 分词器

- 核心思想: 比字符级更底层的切分方式, 它直接操作文本的原始字节 (Bytes)。所有文本最终都以字节形式存储 (如 UTF-8 编码), 一个英文字母通常占 1 个字节, 一个汉字可能占 3 个字节。
- 示例 (UTF-8 编码):
  1. 英文: "cat" → ["c", "a", "t"] → [99, 97, 116]
  2. 中文: "猫" → [227, 149, 131] (三个字节共同表示一个“猫”字)
- 优点:
  - **词汇表小且固定**: 字节的取值范围永远是 0-255, 所以词汇表大小固定为 256。
  - **无 OOV 问题**: 任何文本都可以由字节组成, 因此不存在未知词的问题。
- 缺点:
  - **序列更长**: 比字符级分词器更长, 因为一个字符可能由多个字节组成。
  - **语义几乎破碎**: 模型需要学习从毫无关联的字节序列中重构语义, 学习难度极大。

### 子词级 (subword-level) 分词器

- 核心思想: 目前 LLM(如 GPT、BERT 系列) 的标配, 介于字符级和词级之间, 它将单词拆分成更小的子词 (Subword)。核心思想是: 高频词汇作为一个整

体保留, 低频词汇或未见过的词则拆分为更小的、有意义的子词单元。我们要实现的 BPE 分词器就是一种子词级分词器。

- **示例:**

1. "the" → ["the"] 由于"the" 高频出现, 所以作为一个整体保留。
2. "wonderful" → ["wonder", "ful"] 由于"wonderful" 低频出现, 所以拆分为"wonder" 和"ful" 两个子词。

- **优点:**

- **平衡了词汇量和序列长度:** 词汇表大小适中(通常3万-10万), 序列长度也比字符/字节级短得多。
- **有效处理 OOV 问题:** 任何新词都可以由已知的子词组合而成, 例如模型不认识"webinar", 但可能认识"web" 和"inar", 可以将其切分为["web", "inar"], 从而理解其含义。
- **更灵活的词形变化处理:** 例如"laughing" 和"laughed" 都可以被拆分为["laugh", "ing"] 和["laugh", "ed"], 模型能轻易捕捉到"laugh" 这个共同的词根, 理解不同词形间的关系。

- **缺点:**

- **词汇表大小不易控制:** 需要手动调整合并频率来平衡词汇量和模型效果, 这在实际应用中可能比较麻烦。
- **训练成本较高。**

类型	单位	优点	缺点	举例
<b>词级 (word)</b>	词	语义清晰	词表极大, OOV	"I love NLP" → ["I", "love", "NLP"]
<b>字符级 (character)</b>	字符	词表极小, 无未知词	语义太碎, 序列太长	"I love" → ["I", " ", "l", "o", "v", "e"]
<b>字节级 (byte)</b>	字节	能统一多语言、符号	可读性差, 序列长	"Hi" → [72, 105](ASCII 码)
<b>子词级 (subword)</b>	词根词缀	权衡两者, 处理新词	稍复杂, 需要训练	"unbelievable" → ["un", "believ", "able"]

### 薇言大义 2.3

非常重要的基础知识, 是后面开始训练 BPE 分词器的基础。

## 2.3. BPE Tokenizer Training

### 参考资料 2.4

正则表达式相关知识: <https://www.runoob.com/regexp/regexp-intro.html>

整个 BPE 分词器训练过程可以分为以下三个步骤:

1. **初始化词汇表**
2. **预分词**

### 3. 迭代合并

#### 2.3.1. 初始词汇表

之前说到,BPE 分词器是子词级分词器,它的词汇表是由子词组成的。对应初始的词汇表等价于字节级的,也就是**固定为 256 个字节**。

BPE 算法后续的每一步都是“**找到频率最高的相邻符号对并合并**”,这会逐步改变我们初始的词汇表。也就是说原始词汇表可能是:(a:0x01,b:0x02,c:0x03,...), 经过一轮合并之后可能就变成 (a:0x01,b:0x02,c:0x03,...,ab:0x257)。即基础 256 个字节加上后面合并的子词。

最终这个词汇表里的每一个条目,也就是我们常说的**token**,都会对应一个唯一的整数 ID。

#### 2.3.2. 预分词

##### ● 为什么需要预处理分词?

按理说,有了初始词汇表,就可以遍历语料库,把最频繁出现的字节对开始合并,形成更大的 token 了。但这样做有两个缺点:

1. 语料库遍历很费计算。
2. 如果没有预分词,BPE 算法会直接处理一整串字符,比如”goes.”。它可能会发现’s’ 和’.’ 在语料中经常一起出现(例如在很多句末),于是把它俩合并成一个新的 token ’s.’。这显然是不合理的,因为它混淆了单词本身(go 的第三人称单数形式)和句法结构(句号)。

预分词的作用就是先进行一次清晰的切分,告诉 BPE: “goes 是一个独立的单元,. 是另一个独立的单元,你可以在 goes 内部进行合并,但绝对不能把 goes 的尾巴和. 合并在一起。”

##### ● 预分词的做法:

预分词像是一种**对词汇表粗颗粒度的分词**,常用的预分词方法是使用**正则表达式 (Regular Expression)**来切分文本。例如,GPT2 论文中使用的是:

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

#### Extension 2.3 正则表达式

正则表达式 (Regular Expression, 常缩写为 regex 或 regexp) 是一个强大的文本模式匹配工具。它本质上是一种用特殊符号编写的“规则字符串”,可以用来**查找、替换、分割或验证任何符合该规则的文本**。可以理解为 ctrl+f 的超级加强版。

表 2.2: 正则表达式符号表

类型	符号/语法	解释说明	示例
普通字符	a, b, 1, 2	匹配它们自身。	cat 会精确匹配字符串”cat”。
元字符（任意字符）	.	匹配除了换行符以外的任意单个字符。	c.t 会匹配”cat”, ”cot”, ”c_t” 等。
元字符（重复次数）	*	匹配前面的元素 0 次或多次。	ca*t 会匹配”ct”, ”cat”, ”caaat”。
	+	匹配前面的元素 1 次或多次。	ca+t 会匹配”cat”, ”caaat”, 但不匹配”ct”。
	?	匹配前面的元素 0 次或 1 次。	colou?r 会匹配”color” 和”colour”。
字符集	[...]	匹配方括号内的任意一个字符。	c[ao]t 只会匹配”cat” 和”cot”。
	[^...]	匹配不在方括号内的任意一个字符。	[^0-9] 会匹配任何非数字字符。
分组与或	(...)	将括号内的内容视为一个整体, 可以对整体做重复。	(ab)+ 会匹配”ab”, ”abab”, ”ababab”。
		表示”或” (OR) 逻辑。	cat dog  会匹配”cat” 或者”dog”。
预定义字符类	\d	匹配任意一个数字(Digit), 等同于 [0-9]。	\d\d\d 会匹配”123”, ”987”。
	\w	匹配任意一个单词字符, 包括字母、数字、下划线。	\w+ 会匹配一个完整的单词或数字。
	\s	匹配任意一个空白字符, 包括空格、制表符、换行符。	

**Example 2.2** (在 python 代码中使用正则表达式)

re.findall 和 re.finditer 的区别:

re.findall 返回所有匹配的子字符串, 返回一个列表。

re.finditer 则是**惰性的**, 返回一个迭代器, 每次只返回一个匹配的子字符串, 需要手动调用 next() 方法来获取下一个匹配的子字符串。正因如此, 无论文本有多大, 匹配项有多少, **内存占用都极低**, 因为它一次只处理一个匹配项。这是处理大文件的唯一可行方法。讲义中也推荐使用 re.finditer。

```
import regex as re
PAT = r"""(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^s]\p{L}\p{N}]+|\s+(?!S)|\s+"""
re.findall(PAT, "some text that i'll pre-tokenize")
>>> ['some', 'text', 'that', 'i', "'ll", 'pre', '-', 'tokenize']
```

**2.3.3. BPE 合并**

等到预分词进行粗颗粒度划分之后, 每一个划分后的部分再转换成 UTF-8 序列, 就可以开始 BPE 合并(即训练 BPE 分词器)了。比如原始输入文本是”I am a pig”, 预分词后得到 [”I”, ”am”, ”a”, ”pig”], 其中每一部分再转换成 UTF-8 序列得到 ”I </w>”: 1, ”a m </w>”: 1, ”a </w>”: 1, ”p i g </w>”: 1,(注意这里的”</w>”是特殊符号, 表示一个词的结束)然后开始 BPE 合并。

从高层次来看,BPE 算法会迭代统计每个字节对, 并识别出现**频率最高的字节对**(”A”, ”B”)。然后将这个最频繁出现的字节对(”A”, ”B”)的所有实例进行合并, 即替换为一个新标记”AB”。这个新合并的标记会被添加到我们的词汇表中; 因此,BPE 训练后的最终词汇表大小等于初始词汇表(在我们的案例中是 256 个), 加上训练过程中执行的 BPE 合并操作次数。

BPE 在合并的时候不考虑跨边界合并。例: 若预处理将”dog!”和”dog.”切分为两个独立标记, 则”dog!”中的 b’g’ 与 b’!’ 可合并, 但”dog!”的 b’!’ 与”dog.”的 b’d’

不会被统计（因属于不同标记，会有`</w>`符号将其分开）。

BPE 在频率相同时，采用选择**字典序更大的对优先的原则**。例如，若字节对（“A”，“B”）、（“A”，“C”）、（“B”，“ZZ”）和（“BA”，“A”）的频率均为最高，则我们会选择合并（“BA”，“A”）。

字典序一般就是位数多的字符排在位数低的后面，位数相同就按照英文单词顺序排，比如“A”在“B”之前，“B”在“C”之前。

#### 2.3.4. 特殊标记

在文本编码过程中，经常会使用特定字符串（如`<|endoftext|>`）来存储元数据（例如文档间的分界标记）。进行编码时，通常需要将某些字符串视为“特殊标记”即表示这些标记**永远不应被拆分为多个子标记（即始终作为独立标记保留）**。比如说，序列终止符`<|endoftext|>`必须始终作为独立标记（对应单一整数 ID）存在，以便语言模型知晓何时停止生成内容。这类特殊标记必须一开始就被加入词汇表，从而获得对应的固定标记 ID。

#### 2.3.5. BPE 分词器训练示例

1. 原始文本(末尾包含`<|endoftext|>`):

```
low low low low low
lower lower widest widest widest
newest newest newest newest newest
```

2. **初始化词汇表**: 256 个固定字节以及特殊标记`<|endoftext|>`
3. **预分词**: 按照空格划分结果 $\rightarrow \{\text{low: } 5, \text{lower: } 2, \text{widest: } 3, \text{newest: } 6\}$
4. **BPE 合并**: 首先把预分词的结果再去划分一下 $\rightarrow \{(\text{l}, \text{o}, \text{w}): 5, (\text{l}, \text{o}, \text{w}, \text{e}, \text{r}): 2, (\text{w}, \text{i}, \text{d}, \text{e}, \text{s}, \text{t}): 3, (\text{n}, \text{e}, \text{w}, \text{e}, \text{s}, \text{t}): 6\}$ ; 然后不断迭代合并统计频率最高的字节对，得到新的词汇表。第一轮频率统计: $\{\text{lo: } 7, \text{ow: } 7, \text{we: } 8, \text{er: } 2, \text{wi: } 3, \text{id: } 3, \text{de: } 3, \text{es: } 9, \text{st: } 9, \text{ne: } 6, \text{ew: } 6\}$ , (`'es'`) 和 (`'st'`) 并列频率最高，按照字典序最大原则选择 (`'st'`) 作为这一轮合并的 token 添加到字典里，然后继续下一轮，以此类推。
5. 一般来讲，在大型的 BPE 合并时，训练终止的条件是字词大小到达某个预设值，比如预设 5000，我们初始词典大小时 256(假设无特殊标记)，那么合并 4744 次之后，词汇表大小达到 5000，训练终止。

#### 2.4. BPE 分词器训练实操

##### 参考资料 2.5

*cProfile* 相关知识: <https://docs.python.org/3/library/profile.html>  
*Scalene* 相关知识: <https://github.com/plasma-umass/scalene>

接下来要实现在 TinyStory 数据集上训练 BPE 分词器。(TinyStory 数据集可从 github 上下载)

我们之前说了像 <|endoftext|> 这样的特殊标记必须一开始就被加入词汇表，从而获得对应的固定标记 ID。在预分词的时候也要特殊对待一下特殊符号，一般就会把特殊符号也作为一个分隔符。

我们之前所说的 BPE 训练的办法是基本、朴素的原理，但在实际操作中这种方法效率会比较低，速度较慢（我一开始用的就是这种朴素的算法，可以说非常慢了），合适的方法是直接记录下来所有词的频率到计数器中去，每次 merge 只需要把对应的计数器内容给改了就好。虽然都是两重循环，但第二种办法只用该计数器，它避免了对重复内容的重复处理，因此会快不少。举个例子：

#### ● 第一种方法（低效）：

1. 拿到班级花名册，上面写着每个学生的姓名
2. 花名册上有：“张三、李四、张三、王五、张三、李四、张三...”（总共 1000 个名字）
3. 你一个一个地数：张三、李四、张三、王五、张三、李四...
4. 每个名字都要处理一遍，即使是重复的

#### ● 第二种方法（高效）：

1. 你先把花名册整理成：张三（500 次）、李四（300 次）、王五（200 次）
2. 然后直接统计：张三出现 500 次，李四出现 300 次，王五出现 200 次
3. 只需要处理 3 个唯一的名字

另外一些实践技巧：利用 **cProfile 或 scalene 等工具** 来帮助我们分析代码中的瓶颈；在直接测试 TinyStory 数据集时，可以先测试小部分数据快速检验效果，避免因为数据量太大而浪费时间。

#### Extension 2.4 cProfile 和 scalene

cProfile 是 Python 内置的**性能分析（Profiler）模块**，用于测量程序运行过程中**各个函数的执行时间、调用次数等性能数据**，帮助开发者定位程序中的性能瓶颈。cProfile 会在程序运行时记录每个函数被调用的次数、每次调用消耗的时间以及总耗时，最终生成一份详细的统计报告。

基本使用方法：python -m cProfile test.py

cProfile 会输出文本报告统计，保存到 result.prof，也可以结合 SnakeViz 来查看可视化报告，即 snakeviz result.prof

Scalene 是一个高精度的 Python 性能分析器（profiler），比 cProfile 更先进。它不仅能分析 CPU 时间，还能同时分析：**CPU 使用（Python 与本地代码分离）、内存使用（包括分配与释放）、GPU 使用（可选）、行级别的性能分析结果**

可以在命令行中直接使用查看文本报告：scalene test.py

也可以生成网页可视化报告，即 scalene --html example.py

## 2.5. 利用 BPE 分词器进行编码和解码

### 参考资料 2.6

BPE 分词器编码和解码相关知识: <https://github.com/huggingface/tokenizers>

我们训练 BPE tokenizers 的最终目的还是希望它对新的文本进行编解码, 以便后续的操作。

### 2.5.1. 编码

目标是利用我们已经训练好的 BPE 分词器将一个原始的文本字符串转换成一个整数 Token ID 列表。具体实现步骤和训练 BPE 分词器的过程类似, 但是不需要进行 BPE 合并。

#### 1. 处理特殊符号:

(a). **最优先处理**: 在进行任何其他操作之前, 你需要先将文本中的特殊符号替换为它们对应的占位符或直接分割出来。

(b). 策略:

- 如果特殊符号在 vocab 中已经有预定义的 ID (通常是在训练 BPE 之前就加入的), 可以用一个独特的、不会在普通文本中出现的字节序列来临时替换它们。或者先用特殊符号将整个文本分割成多个部分。例如, 如果文本是 "Hello <|endoftext|> World", 你可以先将其分割成 ["Hello", "<|endoftext|>", "World"]。

- 对于每个非特殊符号的文本块, 进行下面的预分词和合并。

- 对于特殊符号块, 直接查找其在 vocab 中的 ID。

#### 2. Pre-tokenize (预分词):

(a). 目的: 将文本分割成一些“词块”(word chunks)。BPE 合并只在这些词块内部进行, 不会跨越词块边界。这通常是为了防止合并无意义的字符组合(比如一个词的末尾和下一个词的开头)。

(b). 方法: 使用与 BPE 训练时相同的正则表达式。这个正则表达式通常会根据空格、标点符号等来切分文本。

(c). 输出: 一个字符串列表, 每个字符串是一个预分词块。例如, "Hello world!" 可能被预分词为 ["Hello", "world", "!"] (注意空格可能被归属到某个块)。

#### 3. Apply the merges (应用合并规则):

(a). 对每一个预分词块单独执行以下操作:

- 转换为字节序列列表: 将预分词块(字符串)编码为 UTF-8 字节序列, 然后将这个字节序列拆分成单个字节的列表。例如, "the" -> b'the' -> [b't', b'h', b'e']。

- 迭代应用合并规则:

- 遍历 merges 列表中的每一条合并规则 (pair\_A, pair\_B)。
- 在当前的字节序列列表中, 查找所有连续出现的 (pair\_A, pair\_B)。■
- 将找到的第一个 (或所有, 取决于实现策略, 但通常是迭代地、贪婪地合并最先出现的) (pair\_A, pair\_B) 替换为它们合并后的新字节序列 pair\_A + pair\_B。(**重要:** 每应用一次合并, 字节序列列表的结构就可能发生变化。你需要重新从 merges 列表的开头开始检查, 或者更高效地只检查与新合并的 token 相关的可能合并。)
- 例如, 当前序列是 [b't', b'h', b'e'], merges 中有 (b't', b'h')。应用后变成 [b'th', b'e']。然后假设 merges 中还有 (b'th', b'e'), 应用后变成 [b'the']。
- 查找 Token ID: 当一个预分词块不能再进行任何合并时, 它内部的每个 (可能是合并后的) 字节序列都应该对应 vocab 中的一个 Token ID。将这些字节序列转换为它们的 ID。

(b). 拼接结果: 将所有预分词块 (以及特殊符号) 得到的 Token ID 列表按顺序拼接起来, 得到最终的编码结果。

举例: 输入→ 'the cat ate'

- vocab: 0: b'', 1: b'a', 2:b'c', 3: b'e', 4: b'h', 5: b't', 6: b'th', 7: b' c', 8: b' a', 9:b'the', 10: b' at'
- merges: [(b't', b'h'), (b' ', b'c'), (b' ', b'a'), (b'th', b'e'), (b' a', b't')]
- special\_tokens: (假设没有)

处理过程:

1. **Pre-tokenize:** ['the', ' cat', ' ate'] (注意空格的归属)

2. **处理 'the':**

- 初始字节: [b't', b'h', b'e']
- 遍历 merges: (b't', b'h') 可应用 -> [b'th', b'e']
- 从头遍历 merges (对于 [b'th', b'e']): (b'th', b'e') 可应用 -> [b'the']
- 从头遍历 merges (对于 [b'the']): 没有可应用的。
- 查找 ID: b'the' -> 9. 结果: [9]

3. **处理 'cat':** (假设预分词包含前导空格)

- 初始字节: [b' ', b'c', b'a', b't'] (UTF-8 编码的空格字节, 这里用 b' ' 示意)
- 遍历 merges: (b' ', b'c') 可应用 -> [b' c', b'a', b't']
- 从头遍历 merges (对于 [b' c', b'a', b't']): 根据讲义结果 [7, 1, 5], 它实际上是: b' c' -> 7 b'a' -> 1 b't' -> 5 这意味着在 [b' c', b'a', b't'] 状态下,

没有进一步的合并可以应用了，或者  $(b'a', b't')$  这个合并规则不存在或顺序靠后。或者，更可能的是，'cat' 预分词结果是  $[', 'cat']$ ，然后  $'cat' \rightarrow [b'c', b'a', b't']$ ，没有合并，直接查 ID 得  $[2,1,5]$ 。但讲义结果是  $[7,1,5]$ ，对应  $b' c', b'a', b't'$ 。这暗示了'cat' 的预分词结果就是'cat'，然后它被字节化为  $[b', b'c', b'a', b't']$ 。第一个合并是  $(b', b'c') \rightarrow b'c'$  (ID 7)。剩下  $[b'a', b't']$ 。这两个不能再合并，所以分别是  $b'a'$  (ID 1) 和  $b't'$  (ID 5)。

#### 4. 处理 'ate': (假设预分词包含前导空格)

- 初始字节:  $[b', b'a', b't', b'e']$
- 遍历 merges:  $(b', b'a')$  可应用  $\rightarrow [b'a', b't', b'e']$
- 从头遍历 merges (对于  $[b'a', b't', b'e']$ ):  $(b'a', b't')$  (这里指  $b'a'$  和  $b't'$  合并) 可应用  $\rightarrow [b'at', b'e']$  (注意  $b'at'$  是 ID 10)
- 从头遍历 merges (对于  $[b'at', b'e']$ ): 没有可应用的。
- 查找 ID:  $b'at' \rightarrow 10, b'e' \rightarrow 3$ . 结果:  $[10, 3]$

#### 5. 最终结果: $[9] + [7, 1, 5] + [10, 3] = [9, 7, 1, 5, 10, 3]$ 。

### 2.5.2. 解码

目标是将一个整数 Token ID 列表转换回原始的文本字符串。

详细步骤:

#### 1. ID to Bytes (ID 转字节序列):

- 遍历输入的 Token ID 列表。
- 对于每个 ID，使用 vocab 查找其对应的字节序列。
- 将所有查找到的字节序列拼接起来，形成一个单一的字节串。
- 例如， $[9, 7, 1, 5, 10, 3] \rightarrow b'the' + b'c' + b'a' + b't' + b'at' + b'e' \rightarrow b'the'cat ate'$ 。

#### 2. Bytes to String (字节串转字符串):

- 使用 UTF-8 解码器将拼接后的字节串转换回 Unicode 字符串。
- **errors='replace'**: 这很关键。如果解码过程中遇到无效的 UTF-8 字节序列 (比如用户提供了一个非法的 ID 序列，或者你的 vocab 中存在一些不能组成有效 UTF-8 的字节片段)，`errors='replace'` 会用 Unicode 的替换字符 U+FFFD 来代替这些无效部分，而不是抛出 `UnicodeDecodeError`。

## 2.6. BPE 讲义相关 Problems

### Problem 2.3

- 从 TinyStories 和 OpenWebText 中各抽取 10 份文档样本。使用您先前训练的 TinyStories 和 OpenWebText 分词器（词汇表大小分别为 1 万和 3.2 万），将这些抽样文档编码为整数 ID。这两个分词器的压缩比（字节/标记）分别是多少？

答：对每个文档：计算原始 UTF-8 编码下的字节数。用对应的 tokenizer 编码成 token ID 序列，并统计 token 数。最后可以得到平均压缩率：  
 $\text{bytes/token} = \text{总字节数/总 token 数}$

- 如果用 TinyStories 分词器处理 OpenWebText 样本会发生什么？比较压缩率和/或定性描述产生的结果。

答：如果用 TinyStories 分词器处理 OpenWebText 样本压缩比会降低很多。原因一是因为 TinyStories 分词器的词汇表大小只有 1 万，而 OpenWebText 分词器的词汇表大小有 3.2 万，能表示的 token 会少很多。二是 TinyStories tokenizer 是在简单的儿童故事上训练的，而 OpenWebText 包含各种网络文本（新闻、技术文章、论坛讨论等）。当 tokenizer 遇到训练时未见过的词汇模式时，会将其分解成更多的小片段，降低压缩效率。

- 估算你的分词器吞吐量（例如以字节/秒为单位）。处理 Pile 数据集（825GB 文本）需要多长时间？

答：这个在不同的分词器的实现方式和硬件条件下吞吐量也有很大不同。

- 使用您的 TinyStories 和 OpenWebText 分词器，将相应的训练集和开发集编码为整数标记 ID 序列。我们稍后将用此来训练语言模型。建议将标记 ID 序列化为 uint16 数据类型的 NumPy 数组。为何 uint16 是合适的选择？

答：uint16 范围：0 到  $65,535^{16} - 1$ ，选择 uint16 是因为它可以表示 0 到 65,535 的整数范围，足够覆盖 10K 和 32K 词汇量的所有 token ID，同时相比 uint32 节省了一半的存储空间。

## 2.7. BPE 章节实验

- train\_bpe.py：编写一个函数，给定输入文本文件的路径，训练一个（字节级）BPE 分词器。您的 BPE 训练函数应至少处理以下输入参数：input\_path(输入文本文件路径，这里是 TinyStories 和 OpenWebText 数据集)，vocab\_size(词汇表大小)，special\_tokens(特殊符号列表)，vocab(分词器词汇表，一个从整型（词汇表中的标记 ID）到字节（标记字节）的映射关系。)，merges(训练生成的 BPE 合并操作列表。每个列表项为一个字节元组 (<token1>, <token2>)，表示 <token1> 与 <token2> 进行了合并。这些合并操作应按创建顺序排列。)

uv run pytest tests/test\_train\_bpe.py 来运行测试文件。

2. tokenizer.py : 实现一个分词器类，该分词器在给定词汇表和合并规则列表的情况下，能够将文本编码为整数 ID，并将整数 ID 解码回文本。该分词器还应支持用户提供的特殊标记（若这些标记尚未存在于词汇表中，则将其追加至词汇表）。

uv run pytest tests/test\_tokenizer.py 来运行测试文件。

# Transformer Language Model Architecture (Transformer 架构)

首先要明确，语言模型是一种**自回归**的**预测**模型。

## ● 输入：

- 语言模型不直接处理文字，而是处理数字。每个词、标点符号或更小的语言单位（如“ing”、“un-”）都会被赋予一个唯一的整数 ID（也就是上一章提到的**token**）。
- 利用**并行计算**，模型可以按照 batch 同时处理多段文本。

## ● 输出：

- 对输入序列的每一个词，模型都会预测下一个词是什么，即给出概率分布。
- 所有可能词的概率**必须标准化**，即和为 1。

## ● 训练：

- 现代语言模型（尤其是像 GPT 这样的自回归模型）的预训练过程，本质上是一种基于大规模文本语料库的**自监督学习 (Self-Supervised Learning)**。其核心目标是最大化给定上文序列（Context）条件下，预测下一个词元（Token）的条件概率。
- **目标任务：下一个词元预测**。模型学习的是一个概率分布  $P(w_n | w_1, w_2, \dots, w_{n-1})$ ，即在已知前面所有词的条件下，下一个词  $w_n$  出现的概率。
- **学习范式：自监督学习 (Self-Supervised Learning)**。由于训练的标签是从输入数据本身中**自动获取的**，而非人工标注，因此被称为“**自监督**”。这种范式使得模型可以利用海量的、无标签的原始文本进行学习，是大型语言模型能够成功训练的关键。
- **优化过程：最小化损失函数 (Loss Function Minimization)**。当模型做出预测后，会通过一个名为**交叉熵损失 (Cross-Entropy Loss)**的函数，来量化其预测的概率分布与“真实标签”（即下一个词的**独热编码 one-hot vector**）之间的差距。然后，模型使用**反向传播 (Back-propagation)** 算法来调整内部数以亿计的参数，其目标就是让这个损失值尽可能小。这个不断调整参数以减少误差的过程，就是模型的“**学习**”过程。

3.1	基础模块: 线性层和嵌入层 . . . . .	30
3.1.1	参数初始化 . . . . .	30
3.1.2	实验中参数初始化标准 . . . . .	31
3.2	词嵌入层 (embedding layer) . . . . .	31
3.3	两大基础模块: 线性模块和嵌入模块 . . . . .	31
3.3.1	线性模块 (Linear Module) . . . . .	31
3.3.2	嵌入模块 (Embedding Module) . . . . .	32
3.4	Pre-norm Transformer Block . . . . .	32
3.4.1	RMSNorm 的原理 . . . . .	33
3.4.2	位置级前馈网络 (Position-wise Feed-Forward Network) . . . . .	35
3.4.3	相对位置嵌入 (RoPE 旋转位置编码) . . . . .	37
3.4.4	缩放点积注意力 scaled dot-product attention . . . . .	43
3.4.5	多头因果注意力机制 . . . . .	44
3.5	Transformer 总结篇 . . . . .	45
3.5.1	输入部分 . . . . .	45
3.5.2	BPE tokenzier(预分词) . . . . .	45
3.5.3	Embedding (词嵌入) . . . . .	46
3.5.4	Position encoding (位置编码) . . . . .	46
3.5.5	Transformer block . . . . .	46
3.5.6	RMSnorm . . . . .	46
3.5.7	Causal Multi-head self-attention (多头注意力机制) . . . . .	46
3.5.8	SwiGLU . . . . .	46
3.5.9	最终输出 . . . . .	47

### 3.1. 基础模块: 线性层和嵌入层

#### 参考资料 3.1

He 初始化: <https://zhuanlan.zhihu.com/p/40175178> 《神经网络与深度学习》邱锡鹏相关章节 (讲的最好)

#### 3.1.1. 参数初始化

我们训练神经网络的目的是为了找到一组参数, 使得模型在训练数据上的表现最好。但是, 如果初始参数设置不当, 可能会导致模型无法收敛或者收敛到局部最优解甚至出现**梯度爆炸**、**梯度消失**等问题。因此, 参数初始化也是一门需要研究的学问。

最常见的两种参数初始化方法: **Xavier 初始化**和**He 初始化**。

##### ● Xavier 初始化 (Glorot Initialization)

Xavier 初始化由 Xavier Glorot 和 Yoshua Bengio 在 2010 年提出。它的核心思想是保持每一层激活值的**方差**和反向传播时梯度的方差在前向和反向传播中保持不变。Xavier 初始化适用于**Sigmoid**, **Tanh** 等对称函数。

Xavier 初始化通常有两种分布形式:

- **均匀分布 (Uniform)**: 权重从均匀分布  $U[-r, r]$  中采样, 其中:  $r = \sqrt{\frac{6}{fan_{in} + fan_{out}}}$   
均匀分布方差为  $\frac{(b-a)^2}{12} = \frac{r^2}{3}$ 。这里的  $fan_{in}$  是一层网络输入神经元数量,  $fan_{out}$  是输出的神经元数量。
- **正态分布 (Normal)**: 权重从均值为 0, 标准差为  $\sigma$  的正态分布中采样, 其中:  $\sigma = \sqrt{\frac{2}{fan_{in} + fan_{out}}}$  这是由于每经过一层, 权重的方差就会变成原来的  $\frac{1}{fan}$ , 其中  $fan$  表示这一层的神经元的数量。

工作原理简述: 通过同时考虑输入和输出神经元的数量, Xavier 初始化试图在层与层之间找到一个平衡点, 使得信号的方差既不会在传播中衰减, 也不会无限放大。

##### ● He 初始化 (Kaiming Initialization)

He 初始化由 Kaiming He(何凯明, 残差网络也是他提出的)在 2015 年提出。针对 Xavier 初始化在**ReLU 激活函数**上效果不佳的问题, He 初始化就适配于**ReLU 激活函数**。

ReLU 函数  $f(x) = \max(0, x)$  的特性是, 它会将所有负输入都变为 0。这破坏了 Xavier 初始化所依赖的“激活函数关于原点对称”的假设, 并导致大约一半的神经元输出为 0, 从而改变了输出的方差。

He 初始化考虑到 ReLU 会将一半的输入置为零, 这会使得输出方差减半。为了补偿这一点, He 初始化在计算方差时引入了一个因子 2。其他和 Xavier 初步思路一致。

- **均匀分布 (Uniform)**: 权重从均匀分布  $U[-r, r]$  中采样, 其中:  $r = \sqrt{\frac{6}{fan}}$

- 正态分布 (Normal): 权重从均值为 0, 标准差为  $\sigma$  的正态分布中采样,  
其中:  $\sigma = \sqrt{\frac{2}{fan_{in}}}$

之所以 Kaiming 初始化只考虑  $fan_{in}$ , 是因为这个更注重**前向传播**, 不是很在意反向传播。

### 3.1.2. 实验中参数初始化标准

- 线性层权重:  $\mathcal{N}(\mu = 0, \sigma^2 = \frac{d_{in}+d_{out}}{2})$ , 范围限制在  $\pm 3\sigma$  以内。
- 词嵌入权重:  $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ , 范围限制在  $\pm 3$  以内。
- RMSNorm 权重: 1

## 3.2. 词嵌入层 (embedding layer)

- **输入**: 整数 token ID 序列 (这里已经是由之前的 BPE tokenizer 处理文本之后的结果了)
- **词嵌入 (embedding)** : 目的是为了将 token ID 转换为**稠密向量**; 一个词嵌入层 (embedding layer) 的作用就是把这些离散的、没有语义的整数 ID, 转换为连续的、包含语义信息的**向量**。

比如, “猫” 的向量可能和 “狗” 的向量在某种 “动物” 维度上比较接近, 而和 “汽车” 的向量相距较远。每个嵌入层接收形状为 (`batch_size, sequence_length`) 的整数张量, 并生成形状为 (`batch_size, sequence_length, d_model`) 的向量序列。`d_model` 代表输出维度, 越大代表语义信息越丰富, 模型越强大。

## 3.3. 两大基础模块: 线性模块和嵌入模块

### 3.3.1. 线性模块 (Linear Module)

线性模块是神经网络里面最最基础的模块了, 它就是一个线性变换, 将输入的向量映射到另一个向量。

$$y = Wx + b$$

其中,  $W$  是权重矩阵,  $b$  是偏置向量,  $x$  是输入向量,  $y$  是输出向量。

实现的过程中**别忘了初始化**就好。

```
class LinearModule(nn.Module):
    def __init__(self, in_features: int, out_features: int, device: torch.device =
        None = None, dtype: torch.dtype | None = None):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.device = device
        self.dtype = dtype
        self.W = nn.Parameter(torch.empty(self.out_features, self.in_features,
                                         device=device, dtype=dtype))
        # self.b = nn.Parameter(torch.empty(out_features, device=device,
                                         dtype=dtype))
        # 对权重进行 Xavier 初始化
        std = 2 / (self.in_features + self.out_features) ** 0.5
        torch.nn.init.trunc_normal_(self.W, std=std, a = -3 * std, b = 3 * std)
```

```
def forward(self, x: torch.Tensor) -> torch.Tensor:  
    return x @ self.W.T
```

### 3.3.2. 嵌入模块 (Embedding Module)

嵌入模块就是我们之前所说的嵌入层所利用的模块，它的功能是将代表文本的整数“词元 ID”(token ID)转换为高维的、模型能够理解的向量表示。

嵌入层用最高效的“**查字典**”方式，实现了在数学上等价于“**一个输入为独热编码的线性层**”的功能。输入为(batch\_size, sequence\_length)代表一个批次每个句子有sequence\_length个token，输出为(batch\_size, sequence\_length, embedding\_dim)代表每个token的embedding向量。

```
class EmbeddingModule(nn.Module):  
    def __init__(self, num_embeddings: int, embedding_dim: int, device: torch.device |  
     → None = None, dtype: torch.dtype | None = None):  
        super().__init__()  
        self.num_embeddings = num_embeddings # 词表 vocab_size 大小  
        self.embedding_dim = embedding_dim # 词向量维度 d_model  
        self.device = device  
        self.dtype = dtype  
  
        self.embedding_matrix = nn.Parameter(torch.empty(self.num_embeddings,  
     → self.embedding_dim, device=self.device, dtype=self.dtype))  
        std = 1  
        torch.nn.init.trunc_normal_(self.embedding_matrix, std=std, a = -3 * std, b = 3  
     → * std)  
  
    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:  
        return self.embedding_matrix[token_ids] # 从词表到词向量的映射的神经网络里读取  
     → token_ids 输出词向量
```

## 3.4. Pre-norm Transformer Block

- **Transformer 块的基本结构：**

- **多头自注意力机制 (Multi-Head Self-Attention mechanism)**：这是Transformer的核心，允许模型在处理序列时对输入的不同部分赋予不同的权重。
- **位置级前馈网络 (Position-wise Feed-Forward Network)**：一个简单的全连接层，独立地应用于序列中每个位置的向量。

- **残差连接 (Residual Connection)**：

- 原始Transformer论文指出，每个子层都使用了残差连接。残差连接的目的是将子层的输入(x)直接加到子层的输出(Sublayer(x))上，即  $x + \text{Sublayer}(x)$ 。
- 这种设计有助于解决深度网络中的**梯度消失问题**，使得信息可以直接通过多层网络传递。

## ● 层归一化 (Layer Normalization) 的不同应用方式:

- **后归一化** (Post-norm) Transformer: 这是原始 Transformer 论文中使用的架构。层归一化是在每个子层的输出之后，残差连接之后应用的。即:  $\text{LayerNorm}(x + \text{Sublayer}(x))$ 。
- **前归一化** (Pre-norm) Transformer: 这是后续研究 (Nguyen and Salazar, 2019; Xiong et al., 2020) 发现的一种改进架构。层归一化是在每个子层的输入之前应用的。即:  $x + \text{Sublayer}(\text{LayerNorm}(x))$ 。此外，在整个 Transformer 堆栈的最后一个 Transformer 块之后，还会有一个额外的**层归一化**。
  - **优点**: 多项工作发现，这种“前归一化”的方式改善了 Transformer 的训练稳定性。
  - **直观解释 (Intuition)**: 前归一化会创建一个“干净的残差流 (residual stream)”，即从输入嵌入到 Transformer 最终输出的路径上，没有任何归一化操作直接作用于残差连接本身。这种设计被认为能改善梯度流动 (gradient flow)，使得梯度更容易有效地反向传播通过多层网络。
  - **当前标准**: 由于其训练稳定性上的优势，“前归一化”已经成为现代大型语言模型（如 GPT-3, LLaMA, PaLM 等）的标准实践。

### 3.4.1. RMSNorm 的原理

RMSnorm 是一种**简化版的层归一化 (Layer Normalization)**。

## ● LN 层归一化的公式是:

$$LN(x) = \gamma \odot \frac{x - \mu}{\sigma} + \beta \quad (3.1)$$

- $x$  是输入特征向量 (对于 NLP 通常是 (batch\_size, sequence\_length, hidden\_size) 中的一个 hidden\_size 维度上的向量)，这里的 hidden\_size 通常就是  $d_{\text{model}}$ 。
- $\mu$  是**平均值 (mean)**，计算的是  $x$  在其最后一个维度上的均值。
- $\sigma$  是**标准差 (standard deviation)**，计算的是  $x$  在其最后一个维度上的标准差。
- $\gamma$  和  $\beta$  是**可学习的缩放 (scale) 和偏移 (shift) 参数**，它们的维度与  $x$  的最后一个维度相同。它们允许归一化后的数据进行仿射变换，从而恢复模型的表达能力。

## ● RMSnorm 的公式是:

$$RMSnorm(x) = \gamma \frac{x}{RMS(x)} \quad (3.2)$$

- $x$  是输入特征向量。

- $RMS(x)$  是  $x$  的 **均方根 (Root Mean Square)**。

$$RMS(x) = \sqrt{\frac{1}{D} \sum_{i=1}^D x_i^2} = \sqrt{mean(x^2)} \quad (3.3)$$

- 这里的  $D$  是  $x$  的特征维度大小 (即 `hidden_size`)。
- $\gamma$  是**可学习的缩放 (scale) 参数**, 其维度与  $x$  的最后一个维度相同。
- **注意:** RMSnorm 没有 (偏移) 参数  $\beta$ 。

移除均值计算可以带来以下好处:

- **计算效率更高:** 计算均值需要对所有元素求和再除以维度, 这本身是一个  $O(D)$  的操作。移除这一步可以减少计算量, 尤其是在硬件层面上可能更高效。
- **内存占用更少:** 不计算和存储均值, 也不需要  $\beta$  参数。
- **简化模型:** 参数更少, 模型更简洁。

让我们再详细解释一下 (`batch_size`, `sequence_length`, `hidden_size`) 这种形状的张量在 Layer Normalization (LN) 或 RMSnorm 中是如何被归一化的。

### 1. (`batch_size`, `sequence_length`, `hidden_size`) 张量的含义

- **`batch_size`:** 批次大小, 表示一次性处理多少个独立的序列 (或句子)。
- **`sequence_length`:** 序列长度, 表示每个序列中有多少个 token (词或子词)。
- **`hidden_size`:** 隐藏层维度, 也常被称为 **d\_model**。这是每个 token 经过词嵌入层或其他层后, 所对应的**稠密向量的维度**。

可以把这个三维张量想象成: `batch_size` 个矩阵, 每个矩阵的形状是 (`sequence_length`, `hidden_size`)。而每个 (`hidden_size`) 向量, 就是对应于序列中某个位置的某个 token 的表示。

### 2. Layer Normalization 和 RMSnorm 的归一化范围

当你看到  $x$  在 LN/RMSnorm 的公式中时, 这个  $x$  指的是:

针对 (`batch_size`, `sequence_length`, `hidden_size`) 张量中的每一个 (`batch_idx`, `sequence_idx`) 对应的 `hidden_size` 维度上的向量。

也就是说: 对于批次中的每个样本 (`batch_idx`), 对于序列中的每个位置 (`sequence_idx`), 都会独立地提取出一个形状为 (`hidden_size`) 的向量。Layer Normalization 或 RMSnorm 的均值/RMS 和标准差, 就是在这个 (`hidden_size`) 维度的向量上计算的。

#### Example 3.1

具体地说: 如果你的输入张量是 `input_tensor`, 其形状为 ( $B$ ,  $S$ ,  $D$ ) ( $B=batch\_size$ ,  $S=sequence\_length$ ,  $D=hidden\_size$ )。当你应用 Layer Normalization 或 RMSnorm 时, 它们会遍历:  $b$  从 0 到  $B-1$ ,  $s$  从 0 到  $S-1$ , 然后, 对于每一个 ( $b$ ,  $s$ ) 对, 它会取 `input_tensor[b, s, :]` 这个子向量 (其

形状为(D,))，并对这个(D,)维度的向量进行归一化。所以，归一化操作是独立地应用于B \* S个这样的D维向量。

3. 词嵌入层之后的稠密向量 batch\_size 中一个，然后这个对应映射后的 hidden\_size 也就是经过词嵌入层之后的稠密向量，对这个向量在进入 Transformer 块之前归一化。

- **batch\_size 中一个**: 对应于 input\_tensor[b, :, :], 即批次中的一个完整序列。
- **这个对应映射后的 hidden\_size**: 对应于 input\_tensor[b, s, :], 即该序列中某个特定 token (在 s 位置) 的 hidden\_size 维度的向量。
- **也就是经过词嵌入层之后的稠密向量**: 正是如此。词嵌入层的输出形状就是 (batch\_size, sequence\_length, embedding\_dim)，其中 embedding\_dim 通常就是 hidden\_size (或 d\_model)。
- **对这个向量在进入 Transformer 块之前归一化**: 这正是“前归一化”(pre-norm) Transformer 的核心思想。在每个子层 (多头自注意力或前馈网络) 的输入端，都会对这个 (hidden\_size,) 维度的向量进行归一化。

**总结一下：**

Layer Normalization 和 RMSnorm 总是沿着**特征维度** (通常是最后一个维度，即 hidden\_size 或 embedding\_dim) 进行归一化，并且这种归一化是**独立地**应用于每个样本的每个序列位置的。它不涉及批次维度或序列长度维度上的统计计算。

这个操作确保了进入 Transformer 子层的每个 token 的向量表示，其数值范围都得到了稳定，从而有助于训练的**稳定性和效率**。

```
class RMSNorm(nn.Module):
    def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(d_model, device=device, dtype=dtype))
        # weight 对应缩放参数 gamma

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # 题目要求对于不同的精度要先转换为 float32 再进行归一化，最后再转换回原来的精度
        origin_dtype = x.dtype
        x_fp32 = x.to(torch.float32)

        norm_x = x / (x.pow(2).mean(dim=-1, keepdim=True) + self.eps).sqrt()
        x_norm = norm_x.to(origin_dtype)
        return x_norm * self.weight
```

### 3.4.2. 位置级前馈网络 (Position-wise Feed-Forward Network)

**原始 Transformer 的 FFN (基于 ReLU)**

- **结构**: 包含两个线性变换层，中间夹着一个**ReLU 激活函数**。
- **公式**:  $FFN(x) = Linear_2(ReLU(Linear_1(x)))$

- **维度**: 中间隐藏层 (Linear\_1 的输出) 的维度  $d_{ff}$  通常是输入维度  $d_{model}$  的 4 倍 ( $d_{ff} = 4 * d_{model}$ )。

### Extension 3.1 常见激活函数及对应场景

#### Sigmoid 函数

- 公式:  $f(x) = \frac{1}{1+e^{-x}}$
  - 范围:  $(0, 1)$
  - 优点:
    - 输出平滑, 适合于**二分类问题**。
    - 可以将输出映射到  $(0, 1)$  区间。
  - 缺点:
    - 梯度消失: 在输入值很大或很小时, 导数接近 0, 导致更新缓慢。
    - 输出不以 0 为中心, 可能导致优化效率下降。
- Tanh 函数**
- 公式:  $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - 范围:  $(-1, 1)$
  - 优点:
    - 比 Sigmoid 更为平滑, 输出以 0 为中心。
    - 解决了**Sigmoid 的输出不以 0 为中心的问题**。
  - 缺点:
    - 仍然存在**梯度消失**的问题。

#### ReLU 函数

- 公式:  $f(x) = \max(0, x)$
- 范围:  $[0, +\infty)$
- 优点:
  - 计算简单, 收敛速度快。
  - 在正区间内, 梯度恒为 1, 避免了梯度消失问题。
- 缺点:
  - 梯度消失问题: 对于负输入, 梯度为 0, 可能导致 “死亡神经元” 现象。

#### Softmax 函数

- 公式:  $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$  适用于**多分类**任务。

- 范围：输出值在 (0, 1) 之间，且所有输出的和为 1。
- 优点：
  - 将输出转换为概率分布，适合于多分类问题。

### SWiGLU

#### SiLU (Sigmoid Linear Unit) / Swish 激活函数

- 定义： $SiLU(x) = x \cdot \sigma(x) = x/(1 + e^{-x})$ 。
- 特点：类似于 ReLU，但在零点附近是平滑的，这有助于缓解梯度消失问题，并允许负数输入通过。

### 门控线性单元 (Gated Linear Units, GLUs)

- 定义： $GLU(x, W1, W2) = \sigma(W1x) \odot W2x$ 。
- 其中， $\odot$  代表元素级乘法。e.g.  $[1.227, -0.162] \odot [-1.6, 1.5] = [1.227 \gg (-1.6), -0.162 \gg 1.5] = [-1.963, -0.243]$
- 特点：将一个线性变换的结果通过 **sigmoid 函数**（作为门控信号），再与另一个线性变换的结果进行逐元素相乘。
- 优点：建议通过提供一个“线性路径”来“减少深度架构中的梯度消失问题”，同时保留非线性能力。

#### 对于 SwiGLU 的定义：

$$SwiGLU(x, W_1, W_2, W_3) = W_2 (SiLU(W_1x) \odot W_3x) \quad (3.4)$$

- $x \in \mathbb{R}^{d_{model}}$  是输入向量（在实际中，通常是 (batch\_size, sequence\_length, d\_model) 形状张量的最后一个维度），
- $W_1, W_3 \in \mathbb{R}^{d_{f1} \times d_{model}}$  是线性变换的权重矩阵，
- $W_2 \in \mathbb{R}^{d_{model} \times d_{ff}}$  是另一个线性变换的权重矩阵，
- $d_{ff} = \frac{8}{3}d_{model}$ ,  $d_{ff}$  是中间隐藏层的维度，通常是  $d_{model}$  的 8/3 倍（并确保是 64 的倍数）

### 3.4.3. 相对位置嵌入 (RoPE 旋转位置编码)

#### 参考资料 3.2

RoPE 旋转位置编码相关知识：[https://www.bilibili.com/video/BV1CQoaY2EU2?spm\\_id\\_from=333.788.player.player\\_end\\_recommend\\_autoplay&vd\\_source=453c2363ee43bdaa84f759f243a88819](https://www.bilibili.com/video/BV1CQoaY2EU2?spm_id_from=333.788.player.player_end_recommend_autoplay&vd_source=453c2363ee43bdaa84f759f243a88819)  
[https://www.bilibili.com/video/BV1Mj421R7JQ/?spm\\_id\\_from=333.1007.top\\_right\\_bar\\_window\\_history.content.click&vd\\_source=453c2363ee43bdaa84f759f243a88819](https://www.bilibili.com/video/BV1Mj421R7JQ/?spm_id_from=333.1007.top_right_bar_window_history.content.click&vd_source=453c2363ee43bdaa84f759f243a88819)  
<https://zhuanlan.zhihu.com/p/647109286>  
<https://zhuanlan.zhihu.com/p/642884818>

Transformer 中常见的位置编码包括**绝对位置编码**和**相对位置编码**两大类。

Rope 旋转位置编码就是属于相对位置编码，是一种经常被用在大模型（比如 LLaMA、chatGLM）里面的方式。

### 为什么需要位置编码？

当我们想要利用 **Transformer 架构**来训练一个大语言模型时

1. 首先会输入一段文本，或者通俗地讲是一句话，比如“我喜欢你”。
2. 这句话会经过已经训练好的分词器比如 **BPE tokenizer** 进行初步分词，转换为 token 数学向量形式；
3. 之后，token 会经过词嵌入层 (**input embedding**)，其神经网络输入输出形状为 (batch\_size,sequence\_length,d\_model) 转换为**稠密向量 X**；
4. 转换后的稠密向量 X 会输入到**自注意力机制**中，具体的运算就是
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$
，其中 Q(query)、K(key)、V(value) 是 X 分别经过  $W_q, W_k, W_v$  三个神经网络之后的结果。
5. 那么实际上  $QK^T = W_q XX^T W_k$  的运算主要依赖于原始的稠密向量  $XX^T$ 。而对于一句话来说，如果不去关注每一个字的**位置信息**，只是关注字词内容本身，那么每个 token 所转换成的稠密向量 X 肯定是相同的，经过注意力机制后相同文本不同位置的语句结果也不会变化，但这是不符合实际情况的。

#### Example 3.2

比如，“我喜欢你” → [“我”，“喜欢”，“你”] → [[1,2,3],[4,5,6],[7,8,9]]  
这句话如果改为“你喜欢我” → [“你”，“喜欢”，“我”] → [[7,8,9],[4,5,6],[1,2,3]]  
可以看到，虽然“我喜欢你”和“你喜欢我”这两句话语义有很大的不同，但最后转换成的稠密向量 X 在具体的值上没有变化，都是 [1,2,3,4,5,6,7,8,9]。  
我们引入位置编码的目标就是希望额外保留原有文本的位置（索引）信息，使得“你喜欢我”变成类似  
[[74,86,96],[433,53,62],[12,42,63]] 这种和“我喜欢你”生成的 X 有明显区别。

这个章节主要讲解相对位置编码下的 **Rope (旋转位置编码)**

### 什么样的位置编码是好的位置编码

1. **相对位置**
2. **外推性好**

外推性好代表着如果你训练时最长文本是 1000，但真实推理时是 5000，也能很好适应。

## 什么是旋转矩阵？

旋转矩阵是在乘以一个向量的时候改变了向量的方向但不改变大小的效果并保持了手性的矩阵。

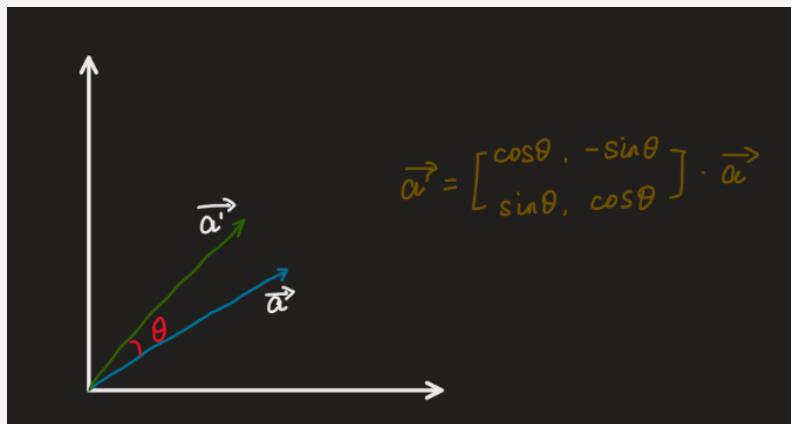
手性指左手右手坐标系，类似物理上的左手定则，右手定则；

## 性质

设  $M$  是任何维的一般旋转矩阵:  $M \in \mathbb{R}^{n \times n}$

- 两个向量的内积在它们都被一个旋转矩阵操作之后保持不变:  $a^T \cdot b = (Ma)^T \cdot Mb$
- 旋转矩阵的逆矩阵是它的转置矩阵:  $MM^{-1} = MM^T = I$  这里的  $I$  是单位矩阵。
- 若用  $M(\theta)$  表示逆时针旋转的角度为  $\theta$ , 那么有  $M(\alpha + \beta) = M(\alpha)M(\beta)$
- 旋转矩阵的转置等于原矩阵角度取负, 即  $M(\theta)^T = M(-\theta)$
- 一个矩阵是旋转矩阵, 当且仅当它是正交矩阵并且它的行列式是 1。正交矩阵的行列式是  $\pm 1$ ; 如果行列式是  $-1$ , 则它包含了一个反射而不是真旋转矩阵。

直观上理解就是乘  $M(\alpha)$  代表先旋转  $\alpha$ , 再乘  $M(\beta)$  代表再选择  $\beta$ , 合计旋转了  $\alpha + \beta$ , 等效于乘  $M(\alpha + \beta)$ 。



## 二维空间下的旋转矩阵

在二维空间中, 旋转可以用一个单一的角  $\theta$  定义。作为约定, 正角表示逆时针旋转。把笛卡尔坐标的列向量关于原点逆时针旋转  $\theta$  的矩阵是:

$$M(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} = \cos \theta \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \sin \theta \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \exp \left( \theta \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \right)$$

欧拉公式:  $e^{i\theta} = \cos \theta + i \sin \theta$

## 二维向量和复数域

一般的复数表示法为  $z = a + ib$ , 其中  $i$  为复数单位, 我们就可以根据实轴和虚轴对一个负数向量化表示, 比如  $z_1 = 3 + 4i \rightarrow (3, 4)$

### 将位置信息注入到稠密向量中

前面我们讲到，原始不加入位置信息的经过词嵌入层之后的稠密向量  $X$  存在着较大缺陷，因此，我们要在原始向量的基础上添加位置信息，即  $f(x_i, i)$ ，其中  $i$  代表该元素的位置信息（索引）。

相对位置编码希望能利用上 token 之间的相对位置信息，假定 query 向量  $q_m$  和 key 向量  $k_n$  之间的内积操作可以被一个函数  $g$  表示，该函数  $g$  的输入是词嵌入向量  $x_m$ ,  $x_n$  和它们之间的相对位置  $m-n$ :  $\langle f_q(x_m, m), f_k(x_n, n) \rangle = g(x_m, x_n, mn)$

### 二维场景

在二维场景下，定义为

$$f_q(x_m, m) = (W_q x_m) e^{im\theta} = q_m e^{im\theta}$$

$$f_k(x_n, n) = (W_k x_n) e^{in\theta} = k_n e^{in\theta}$$

$$g(x_m, x_n, m-n) = \operatorname{Re} [(W_q x_m) (W_k x_n)^* e^{i(m-n)\theta}]$$

其实和无位置编码的 Transformer 架构比起来的区别只是增加了  $e^{im\theta}$  项

然后对  $QK^T$  (即求内积) =  $g$  的推导如下所示：

#### ● 方法一：暴力推导

$$\begin{aligned}
 & f_q = (W_q x_m) e^{im\theta} = q_m e^{im\theta} \quad \text{其中 } q_m = q_m^1 + i q_m^2 \quad (\text{二维向量虚数表示}) \\
 & q_m = \begin{pmatrix} w_q^{11} & w_q^{12} \\ w_q^{21} & w_q^{22} \end{pmatrix} \cdot \begin{pmatrix} x_m^1 \\ x_m^2 \end{pmatrix} = \begin{pmatrix} q_m^1 \\ q_m^2 \end{pmatrix} \\
 & f_k = (W_k x_n) e^{in\theta} = k_n e^{in\theta} \\
 & \text{欧拉公式: } e^{i\theta} = \cos\theta + i \sin\theta \Rightarrow \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \\
 & \text{因此 } q_m e^{im\theta} = (q_m^1 + i q_m^2) (\cos m\theta + i \sin m\theta) \\
 & = (q_m^1 \cos m\theta - q_m^2 \sin m\theta) + i (q_m^2 \cos m\theta + q_m^1 \sin m\theta) \\
 & = \begin{pmatrix} q_m^1 \cos m\theta - q_m^2 \sin m\theta \\ q_m^2 \cos m\theta + q_m^1 \sin m\theta \end{pmatrix} = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_m^1 \\ q_m^2 \end{pmatrix} \\
 & \text{同理: } k_n e^{in\theta} = \begin{pmatrix} \cos n\theta & -\sin n\theta \\ \sin n\theta & \cos n\theta \end{pmatrix} \cdot \begin{pmatrix} k_n^1 \\ k_n^2 \end{pmatrix} \\
 & \text{求内积: } f_q \cdot f_k \Rightarrow (q_m e^{im\theta})^T \cdot k_n e^{in\theta} \\
 & = (q_m^1, q_m^2) \cdot \begin{pmatrix} \cos n\theta & -\sin n\theta \\ \sin n\theta & \cos n\theta \end{pmatrix} \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \cdot \begin{pmatrix} k_n^1 \\ k_n^2 \end{pmatrix} \\
 & = (q_m^1, q_m^2) \underbrace{\begin{pmatrix} \cos(m-n)\theta & -\sin(m-n)\theta \\ \sin(m-n)\theta & \cos(m-n)\theta \end{pmatrix}}_{\text{II}} \cdot \begin{pmatrix} k_n^1 \\ k_n^2 \end{pmatrix} \\
 & = \operatorname{Re} [q_m K_n^* e^{i(m-n)\theta}] \\
 & = \operatorname{Re} [(W_q x_m) (W_k x_n)^* e^{i(m-n)\theta}]
 \end{aligned}$$

## ● 方法二：利用旋转矩阵的性质

对于  $f_q = q_m e^{im\theta}$  可以理解为  $q_m$  逆时针旋转的角度为  $m\theta$ , 即  $M(m\theta)$  而  $f_k = k_n e^{in\theta}$  可以理解为  $k_n$  逆时针旋转的角度为  $n\theta$ , 即  $M(n\theta)$

那么两者内积  $\langle f_q, f_k \rangle = \langle M(m\theta)q_m, M(n\theta)k_n \rangle = q_m^T M(m\theta)^T M(n\theta)k_n = q_m^T M(m\theta)M(n\theta)k_n = q_m^T M((n-m)\theta)k_n$  之后仿照证明方法 1 也可得到  $g = \langle f_q, f_k \rangle = q_m^T M((n-m)\theta)k_n$

## 扩展至多维场景

实际的词嵌入维度 ( $d_{model}$ ) 一般都不是二维的, 不过处理的方式也不复杂, 是**两两一分组**, 每一组还是上面二维场景下的操作。

$$\left( \begin{array}{ccccccc} \cos m\theta_0 & -\sin m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_1 & -\sin m\theta_1 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_1 & \cos m\theta_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2-1} & -\sin m\theta_{d/2-1} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2-1} & \cos m\theta_{d/2-1} \end{array} \right) \left( \begin{array}{c} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{d-2} \\ q_{d-1} \end{array} \right)$$

对于此, 因为有很多零参与计算会很浪费, 优化的方法是:

$$\left( \begin{array}{c} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{d-2} \\ q_{d-1} \end{array} \right) \otimes \left( \begin{array}{c} \cos m\theta_0 \\ \cos m\theta_0 \\ \cos m\theta_1 \\ \cos m\theta_1 \\ \vdots \\ \cos m\theta_{d/2-1} \\ \cos m\theta_{d/2-1} \end{array} \right) + \left( \begin{array}{c} -q_1 \\ q_0 \\ -q_3 \\ q_2 \\ \vdots \\ -q_{d-1} \\ q_{d-2} \end{array} \right) \otimes \left( \begin{array}{c} \sin m\theta_0 \\ \sin m\theta_0 \\ \sin m\theta_1 \\ \sin m\theta_1 \\ \vdots \\ \sin m\theta_{d/2-1} \\ \sin m\theta_{d/2-1} \end{array} \right)$$

即对对应的元素相乘相加, 这样做在代码里面可以用 `arrange` 函数处理了。

此外, 对 RoPE 不是对所有维度都用同一个  $\theta$  进行旋转, 而是给**不同维度分配不同的旋转速度 (频率)**。将  $d_{model}$  维的向量两两分组, 共有  $\frac{d_{model}}{2}$  个组。第  $i$  组 ( $i$  from 0 to  $\frac{d_{model}}{2} - 1$ ) 的旋转角度  $i$  定义为:  $i = 10000^{(-\frac{2i}{d_{model}})}$

这里底数设置成 10000 是**经验值**, 针对 4096 这类维度绰绰有余, 实际上如果把底数设计的更大一些, 比如 50000, 会有更好的**外推性**, 因为表示的频率范围会扩大。

## 直观解释

由旋转矩阵的性质可知, 原始 Q K 经过内积之后并不会改变原始绝对值大小, Rope 编码最后结果  $q_m^T M((n-m)\theta)k_n$  里有一项是  $(n-m)\theta$ , 这就说明了每一项注意力计算结果会和两个 token 的**相对位置 n-m 有关系**。

直观上, 位置离的越近的两个 token 按理说关联应该越强, 比如 “锦瑟无端五十弦” 和后面一句 “一弦一柱思华年” 关系比较紧密, 但和 “望帝春心托杜鹃”

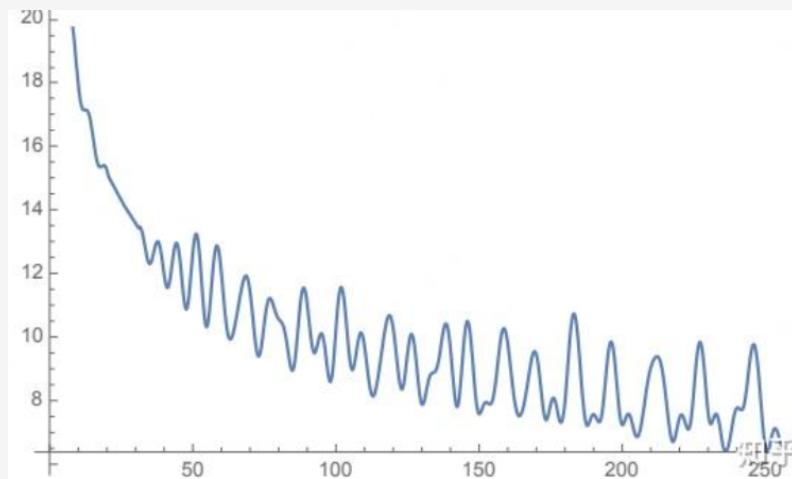
逻辑性可能没那么紧密，token 关联就小一点，自注意力机制结果也应该小一些。经过位置编码的结果也类似，极端情况下,  $n=m$  即  $n-m=0$  两者同一个位置的时候， $\cos(n-m)\theta$  就是 1,  $-\sin(n-m)\theta = 0$ , 注意力最强。

### 旋转位置编码的远程衰减性

直观上想，距离越远的 token 之间越不相干，也就是**注意力分数比较低**，即  $QK^T$  值比较低

我们知道由于  $QK^T$  内积运算的结果包括周期函数（即  $\cos\theta$  和  $\sin\theta$ ），其最终求和的结果也一定是周期函数，而我们这里所讨论的远程衰减性实际上指一个周期就已经很长了，可能达到上万，而在这段区域内，内积的值是**振荡衰减**的，所以我们称为远程衰减性。

证明有些复杂，可见：<https://zhuanlan.zhihu.com/p/647109286>



### Rope homework

```
class RoPE(nn.Module):
    def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None):
        super().__init__()
        if d_k % 2 != 0:
            raise ValueError("d_k must be even")
        self.theta = theta # 这个是 RoPE 的底数超参数，不是直接的角度
        self.d_k = d_k #d_k 就是 d_model, 即嵌入之后的稠密向量，它必须为偶数
        self.max_seq_len = max_seq_len
        self.device = device
        # 计算频率
        freqs = 1.0 / (self.theta ** (torch.arange(0, self.d_k, 2).float() / self.d_k))
        # 记录每个 token 的位置信息
        positions = torch.arange(self.max_seq_len)
        # 计算正弦和余弦
        sinusoids = torch.outer(positions, freqs) #outer 是外积，即每个位置都与每个频率相乘
        self.register_buffer("cos_cache", sinusoids.cos(), persistent=False) # 利用
        #→ register_buffer 表示这是固定的，不需要学习
        self.register_buffer("sin_cache", sinusoids.sin(), persistent=False)
```

```

def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor:
    # 这里的 x 是输入的稠密向量, token_positions 是 token 的位置信息
    cos = self.cos_cache[token_positions]
    sin = self.sin_cache[token_positions]

    cos = cos.unsqueeze(0)
    sin = sin.unsqueeze(0)

    x1 = x[...,0::2] # 偶数位置
    x2 = x[...,1::2] # 奇数位置

    output1 = x1 * cos - x2 * sin # 偶数位置乘以 cos, 奇数位置乘以 sin
    output2 = x1 * sin + x2 * cos # 偶数位置乘以 sin, 奇数位置乘以 cos
    out = torch.stack([output1, output2], dim=-1) # [batch, seq_len, d_k//2, 2]
    out = out.flatten(-2) # [batch, seq_len, d_k]
    return out

```

### 微言大义 3.1

Rope 旋转位置编码是很重要的思想, 主播在面试中也遇见过相关的问题。

#### 3.4.4. 缩放点积注意力 scaled dot-product attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (3.5)$$

- 定义: 查询向量  $Q$  和所有词的键向量  $K$  进行点积 (Dot-Product) 运算。
- 为什么? 点积是衡量两个向量相似度的一种常用方法。如果  $Q$  和某个  $K$  的方向很接近, 它们的点积就会很大, 代表相关性高。 $K^T$  表示对 Key 矩阵进行转置, 是为了让矩阵乘法能够顺利进行。
- 得到什么? 一个注意力分数 (Attention Score) 矩阵。这个矩阵的每一行表示一个词的 Query, 每一列表示它对句子中其他词的原始注意力分数。

### Keynote 3.1

#### 为什么需要缩放

- 问题: 当  $d_k$  (Key 向量的维度) 比较大时,  $Q \cdot K$  的点积结果的方差也会变大, 这意味着点积的数值可能会非常大或非常小。
- 后果: 如果数值进入 Softmax 函数时过大, Softmax 的输出会趋近于“硬性”的 one-hot 分布 (比如  $[0, 0, 1, 0, 0]$ )。这意味着梯度会变得极小 (梯度消失), 导致模型在训练时很难学习到有效的参数。
- 解决方案: 将点积结果除以  $\sqrt{d_k}$ 。论文作者证明, 这样做可以使得点积结果的方差保持在 1 左右, 从而避免了上述问题, 让训练过程更加稳定。

## Keynote 3.2

### Mask 机制

对于有些我们不希望产生注意力的 *key*, 可以使用 *mask* 矩阵, 在对应位置标上 “*False*” 或者其他的标识用来表示这个位置不要产生注意力, 之后在标记 “*False*” 的位置填充替换为 “ $-\infty$ ”, 这样做的好处就是之后做 softmax 归一化时可以直接  $e^{-\infty} = 0$ , 忽略掉这一部分的注意力。

e.g.  $Q^T K$  相乘后的形状为  $R^{n \times m}$ , 那么 *mask* 矩阵的形状也应该是  $R^{n \times m}$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \otimes \begin{pmatrix} \text{True} & \text{True} & \text{False} \\ \text{True} & \text{False} & \text{True} \\ \text{True} & \text{False} & \text{True} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & -\infty \\ a_{21} & -\infty & a_{23} \\ a_{31} & -\infty & a_{33} \end{pmatrix}$$

### scaled dot-product attention hw

```
def softmax(x: torch.Tensor, dim: int) -> torch.Tensor:
    x_max = x.max(dim=dim, keepdim=True)[0]
    x_exp = torch.exp(x - x_max)
    return x_exp / x_exp.sum(dim=dim, keepdim=True)
```

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, Q: torch.Tensor, K: torch.Tensor, V: torch.Tensor, mask:
        -> torch.Tensor | None = None):
        d_k = Q.shape[-1]
        scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k))
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9) # 如果 mask 为 0, 则将对应位置的
            -> score 设置为-1e9
        attn_weights = torch.softmax(scores, dim=-1) # 对 key 这一维度进行 softmax 归一化
        return torch.matmul(attn_weights, V) # 将 attn_weights 与 value 相乘得到最终的输出
```

### 3.4.5. 多头因果注意力机制

多头注意力的核心思想: 从多个不同的子空间中学习信息, 让模型能够共同关注来自不同位置、不同维度的信息。

主要流程: 原始的输入形状是 (batch\_size, seq\_len, d\_model)

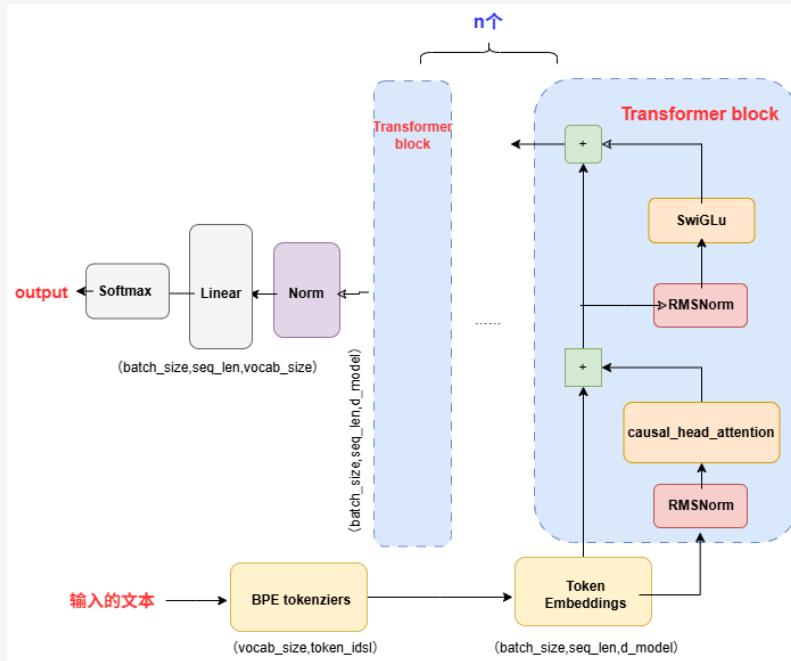
1. 对于输入, 要先用  $W_q, W_k, W_v$  线性变换得到  $q, k, v$
2. 如果有  $n_{\text{heads}}$  个头, 就把最后一个维度切分成  $n_{\text{heads}}$  份,  $q, k, v$  每一部分都切分成  $d_{\text{model}}/n_{\text{heads}}$  的维度,
3. 对于每个头, 对于  $q, k, v$  都去做 attention 操作。
4. 最后把所有的头按照最后一个维度 concat 起来, 然后做一次线性变换。

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \text{ for } \text{head}_i = \text{Attention}(Q_i, K_i, V_i) \quad (3.6)$$

$$\text{MultiHeadSelfAttention}(x) = W_O \text{MultiHead}(W_Q x, W_K x, W_V x) \quad (3.7)$$

### 3.5. Transformer 总结篇

这一小节是第一章最重要的部分，有别于原始论文里提出的模型流程，它从整体上讲解了一些新的主流大模型的底层 Transformer 架构（主要就是 **Llama**，推荐直接阅读 Llama 的源代码，和作业重合度很高）。



#### 3.5.1. 输入部分

模型的输入是**大规模**的文本语料库。其核心训练目标是学习文本的统计规律，从而能够**基于给定的上文 (context)，准确地预测下一个词元 (token)**。通过这种方式，模型可以生成连贯、流畅的文本。

#### 3.5.2. BPE tokenzier(预分词)

原始文本是由字符组成的字符串，无法直接输入神经网络。Tokenizer 的作用是**将原始文本字符串转换为一个整数序列 (Integer IDs)**。我们使用的 BPE (Byte-Pair Encoding) 是一种亚词 (subword) 分词算法，它通过以下步骤工作：

1. **初始化**: 词典由所有单个字节 (0-255) 组成。
2. **迭代合并**: 在训练语料中，不断寻找出现**频率最高的相邻字节对 (或已合并的 token 对)**，将它们合并成一个新的 token，并加入词典。
3. **最终**: 经过指定次数的合并后，形成最终的词典 (vocabulary)。Tokenizer 利用这个词典和合并规则，将任意文本切分成一系列 token，并映射为其在词典中的**唯一整数 ID**。

### 3.5.3. Embedding (词嵌入)

预分词得到的结果维度是 vocab\_size, 即词典的大小, 一般会用**独热码**来表示, 这样做的稀疏向量太多, 太浪费计算资源。为了把稀疏向量映射为**稠密向量**, 维度一般是 d\_model=512。一个词嵌入层 (embedding layer) 的作用就是把这些离散的、没有语义的整数 ID, 转换为连续的、包含语义信息的**向量**。比如, “猫”的向量可能和“狗”的向量在某种“动物”维度上比较接近, 而和“汽车”的向量相距较远。每个嵌入层接收形状为 (batch\_size, sequence\_length) 的整数张量, 并生成形状为 (batch\_size, sequence\_length, d\_model) 的向量序列。d\_model 代表输出维度, 越大代表语义信息越丰富, 模型越强大。

### 3.5.4. Position encoding (位置编码)

在原始论文中, 位置编码被放在 embedding 之后, 但在作业和主流大模型实现里面, 位置编码就直接放在多头注意力那里来只处理 query 和 key 了。

位置编码是由于仅仅只是词嵌入虽然能够保留语义, 但不能很好地表示每个 token 的位置 (索引) 信息。位置编码包括绝对位置编码和相对位置编码两大类, 一般相对位置编码的效果会更好, 作业中使用的方法是 rope 旋转位置编码。

### 3.5.5. Transformer block

作业中的 Transformer block 由 RMSnorm、Causal 多头注意力机制、SwiGlu 这几部分组成。

### 3.5.6. RMSnorm

简化版的层归一化

$$RMSnorm(x) = \frac{x}{RMS(x)} RMS(x) = \sqrt{\frac{1}{D} \sum_{i=1}^D x_i^2} = \sqrt{mean(x^2)}$$

值得注意的是, 对于从 embedding 层输入过来的 (batch\_size, sequence\_length, d\_model(或者 hidden\_dim)), 归一化的目标 x 是最后一维的 d\_model。

### 3.5.7. Causal Multi-head self-attention (多头注意力机制)

1. **多头机制 (Multi-Head)**: 将 d\_model 维的 Q, K, V 向量按照 n\_heads 分成 d\_model//n\_heads 多个头 (head), 让每个头关注输入序列的不同方面, 增强了模型的表达能力, 最后再拼接在一起。
2. **旋转位置编码 (RoPE)**: 为了注入位置信息, RoPE **直接作用于 Q 和 K 向量**, 通过旋转它们来编码其绝对位置, 并使注意力得分自然地依赖于它们的相对位置。V 向量不参与位置编码。
3. **因果 Mask (Causal Masking)**: 在自回归的文本生成任务中, 模型在预测位置 i 的词元时, **只能关注到位置 i 及之前的所有词元**, 不能“偷看”未来的信  
息。这是通过在注意力得分矩阵上应用一个上三角遮罩实现的。”
4. 还有在点积缩放运算里面使用 mask 来给 e 的指数赋值  $-\infty$ , 方便运算。

### 3.5.8. SwiGLU

1. **原始 Transformer 的 FFN (基于 ReLU)**:

- **结构:** 包含两个线性变换层，中间夹着一个 ReLU 激活函数。
- **公式:**  $\text{FFN}(x) = \text{Linear\_2}(\text{ReLU}(\text{Linear\_1}(x)))$
- **维度:** 中间隐藏层( $\text{Linear\_1}$  的输出)的维度  $d_{\text{ff}}$  通常是输入维度  $d_{\text{model}}$  的 4 倍 ( $d_{\text{ff}} = 4 * d_{\text{model}}$ )。

## 2. SiLU (Sigmoid Linear Unit) / Swish 激活函数:

- **定义:**  $\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$  其中， $\sigma(x) = \frac{1}{1+e^{-x}}$  代表 sigmoid 激活函数。
- **特点:** 类似于 ReLU，但在零点附近是平滑的，这有助于缓解梯度消失问题，并允许负数输入通过。

## 3. 门控线性单元 (Gated Linear Units, GLUs):

- **定义:**  $\text{GLU}(x, W1, W2) = (W1x)W2x$ 。
- $\sigma$  代表 Sigmoid 激活函数。
- $\odot$  代表元素级乘法 (element-wise multiplication)。
- **优点:** 建议通过提供一个“线性路径”来“减少深度架构中的梯度消失问题”，同时保留非线性能力。

### 3.5.9. 最终输出

从几个 Transformer block 输出的形状还是  $(\text{batch\_size}, \text{sequence\_length}, d_{\text{model}})$ ，此后继续归一化，Linear，Softmax。

归一化自然就是稳定梯度

Linear 是将  $(\text{batch\_size}, \text{sequence\_length}, d_{\text{model}}) \rightarrow (\text{batch\_size}, \text{sequence\_length}, \text{vocab\_size})$

之后利用 softmax 进行计算 vocab\_size 里面每一个词的概率，选择最大的词作为下一个词。

### 薇言大义 3.2

这一部分内容可以复习或者一开始就看一看，有一个整体的认识。