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
3. 下载数据集:
   TinyStory: huggingface.co/datasets/roneneldan/TinyStories/resolve/main/
   OpenWebText: skylion007.github.io/OpenWebTextCorpus/
4. 学习了解 uv 的基础使用方法

薇言大义 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 adpater.py 是测试文件的

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

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) ↔ 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:

$$4 \rightarrow 0100$$
$$E \rightarrow 1110$$
$$2 \rightarrow 0010$$
$$D \rightarrow 1101$$

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\xa1\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*

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，词汇表高达到几十万甚至上百万。
  - **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
词级 (word)	词	语义清晰	词表极大,OOV	“I love NLP” → [”I”, ”love”, ”NLP”]
字符级 (character)	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”]
字节级 (byte)	字节	能统一多语言、符号	可读性差，序列长	“Hi” → [72, 105](ASCII 码)
子词级 (subword)	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. 初始化词汇表

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"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\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”, ”abababab”。
	|	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 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–>{low: 5, lower: 2, widest: 3, newest: 6}
4. **BPE 合并**: 首先把预分词的结果再去划分一下–>{(l,o,w): 5, (l,o,w,e,r): 2, (w,i,d,e,s,t): 3, (n,e,w,e,s,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:{lo: 7, ow: 7, we: 8, er: 2, wi: 3, id: 3, de: 3, es: 9, st: 9, ne: 6, 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)，合适的办法是**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**可以来帮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' -> [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') -> 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 -> [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 -> [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' -> 10, b'e' -> 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] -> b'the' + b' c' + b'a' + b't' + b' at' + b'e' -> 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

1. **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: bytes/token = 总字节数/总 token 数

2. **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。

3. **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。

4. **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：$0\ 65,5352^{16}-1$，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

1. 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。

3

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 $P(w_n \mid w_1, w_2, ..., w_{n-1})$，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

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 σ 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 初始思路一致。

  - **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 σ 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:$\mathcal{N}(\mu = 0, \sigma^2 = \frac{din+dout}{2})$, 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 ± 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=self.device, dtype=self.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 + 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：LayerNorm(x + 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 + Sublayer(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：**

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

  - 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_model。
  - μ 是**平均值（mean）**，计算的是 x 在其最后一个维度上的均值。
  - σ 是**标准差（standard deviation）**，计算的是 x 在其最后一个维度上的标准差。
  - γ 和 β 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：**

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

  - x 是输入特征向量。

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

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

- 这里的 D 是 x 的特征维度大小（即 hidden_size）。
- γ 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 β。

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 β 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**。

```python
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. $[1.227, -0.162] \odot [-1.6, 1.5] = [1.227 \text{»} (-1.6), -0.162 \text{»} 1.5] = [-1.963, -0.243]$
- 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：

$$SwiGLU(x, W_1, W_2, W_3) = W_2 \left(SiLU\left(W_1 x\right) \odot W_3 x\right) \tag{3.4}$$

- $x \in \mathbb{R}^{d_{model}}$ 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），
- $W_1, W_3 \in \mathbb{R}^{d_{f1} \text{»} d_{model}}$ 是线性变换的权重矩阵，
- $W_2 \in \mathbb{R}^{d_{model} \text{»} d_{ff}}$ 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/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 $Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$，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 $QK^T = W_qXX^TW_k$ 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。

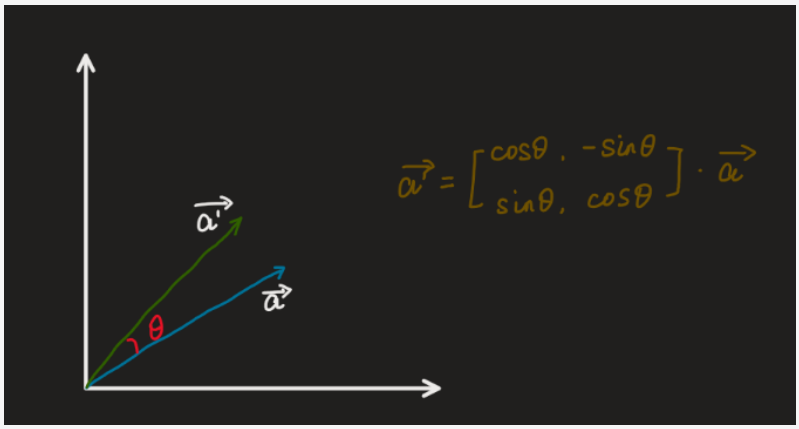
手性指左手右手坐标系，类似物理上的左手定则，右手定则；

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 θ，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 ± 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 α, 再乘 $M(\beta)$ 代表再选择 β, 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 θ 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 θ 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： $e^{i\theta} = cos\theta + isin\theta$

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：$< f_q(x_m, m), f_k(x_n, n) >= g(x_m, x_n, mn)$

二维场景

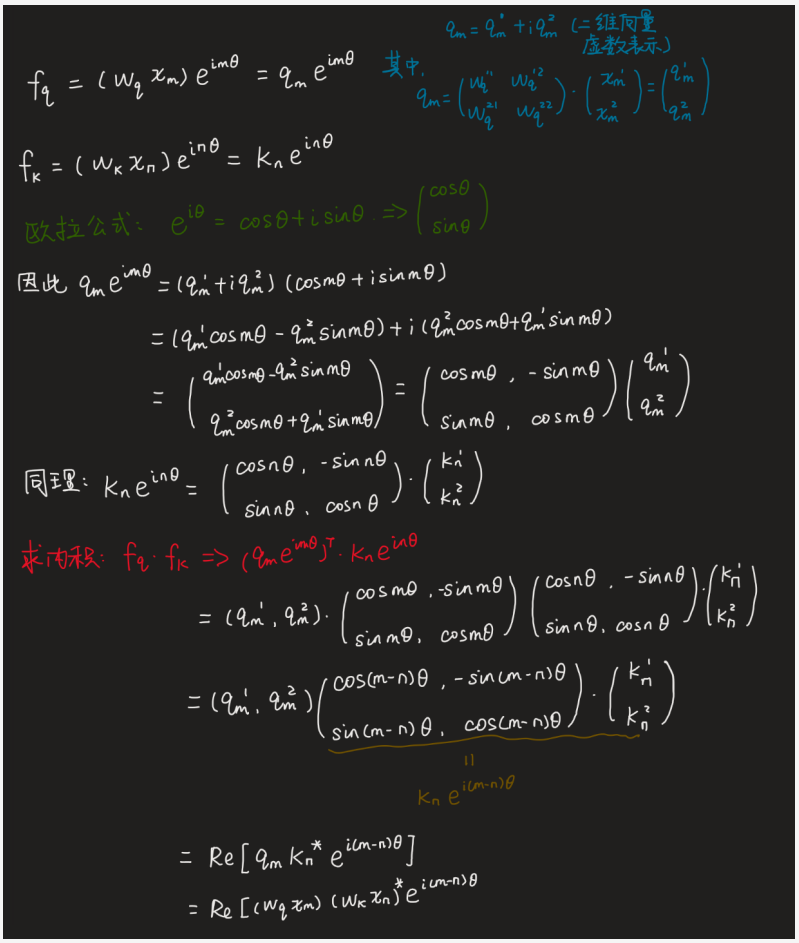
在二维场景下，定义为

$$f_q(x_m, m) = (W_q x_m) e^{im\theta} = q_m e^{im\theta}$$
$$f_k(x_n, n) = (W_k x_n) e^{in\theta} = k_n e^{in\theta}$$
$$g(x_m, x_n, m-n) = \mathrm{Re}\left[(W_q x_m)(W_k x_n)^* e^{i(m-n)\theta}\right]$$

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**



- **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 $< f_q, f_k >=< M(m\theta)q_m, M(n\theta)k_n >= q_m^T M(m\theta)^T M(n\theta)k_n = q_m^T M(m\theta)M(n\theta)k_n = q_m^T M((n-m)\theta)k_n$ 之后仿照证明方法 1 也可得到 $g =< f_q, f_k >= q_m^T M((n-m)\theta)k_n$

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。

$$\begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_0 & \cos m\theta_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_1 & -\sin m\theta_1 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_1 & \cos m\theta_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2-1} & -\sin m\theta_{d/2-1} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2-1} & \cos m\theta_{d/2-1} \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{d-2} \\ q_{d-1} \end{pmatrix}$$

对于此，因为有很多零参与计算会很浪费，优化的方法是：

$$\begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{d-2} \\ q_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_0 \\ \cos m\theta_0 \\ \cos m\theta_1 \\ \cos m\theta_1 \\ \vdots \\ \cos m\theta_{d/2-1} \\ \cos m\theta_{d/2-1} \end{pmatrix} + \begin{pmatrix} -q_1 \\ q_0 \\ -q_3 \\ q_2 \\ \vdots \\ -q_{d-1} \\ q_{d-2} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_0 \\ \sin m\theta_0 \\ \sin m\theta_1 \\ \sin m\theta_1 \\ \vdots \\ \sin m\theta_{d/2-1} \\ \sin m\theta_{d/2-1} \end{pmatrix}$$

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 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**经验数值**，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”

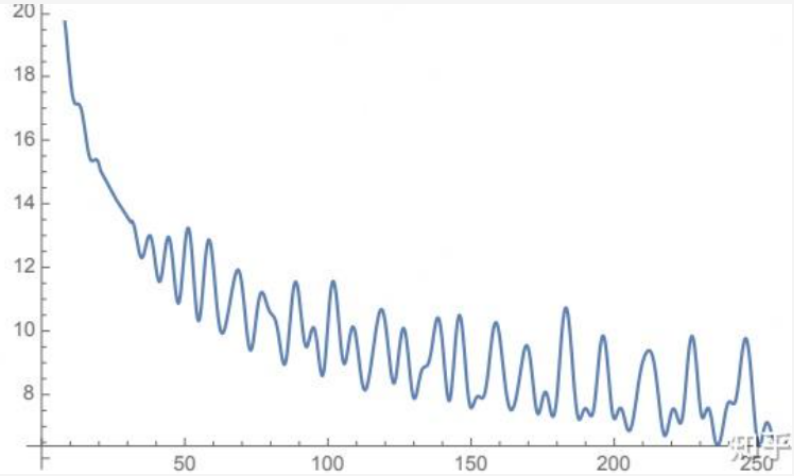
逻辑性可能没那么紧密，token 关联就小一点，自注意力机制结果也应该小一些。经过位置编码的结果也类似，极端情况下,n=m 即 n-m=0 两者同一个位置的时候，$cos(n-m)\theta$ 就是 1，$-sin(n-m)\theta = 0$, 注意力最强。

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

```python
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)
```

```python
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
```

薇言大义 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

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

- **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^TK 相乘后的形状为 $R^{n\times m}$，那么 *mask* 矩阵的形状也应该是 $R^{n\times m}$

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

scaled dot-product attention hw

```python
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)
```

```python
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)

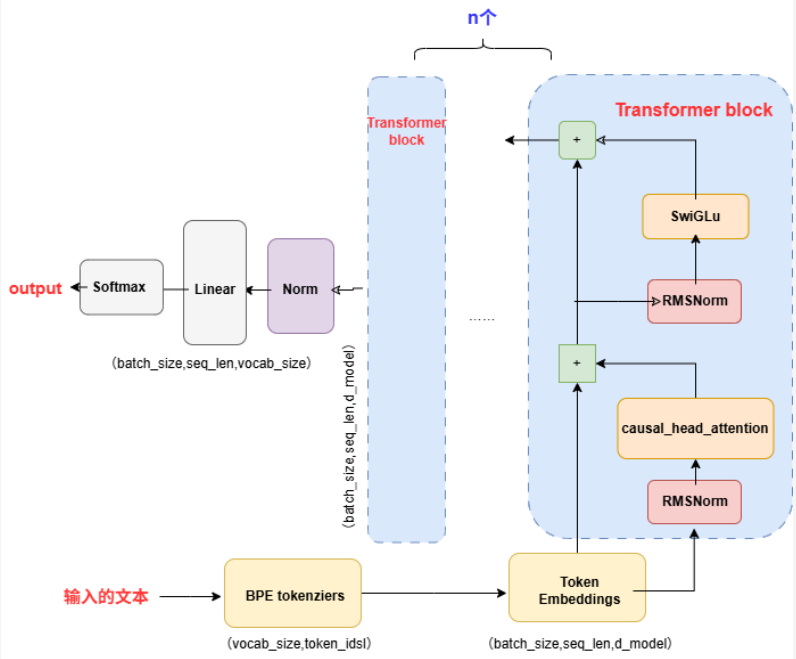
1. 对于输入，要先用 W_q, W_k, W_v 线性变换得到 q,k,v
2. 如果有 n_heads 个头，就把最后一个维度切分成 n_heads 份，q,k,v 每一部分都切分成 d_model//n_heads 的维度，
3. 对于每个头，对于 q,k,v 都去做 attention 操作。
4. 最后把所有的头按照最后一个维度 concat 起来，然后做一次线性变换。

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

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

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**: 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)。

2. **SiLU (Sigmoid Linear Unit) / Swish 激活函数**:
   - **定义**: $SiLU(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$ 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**: $GLU(x, W1, W2) = (W1x)W2x$。
   - σ 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 (batch_size, sequence_length, d_model)，此后继续归一化，Linear，Softmax。

归一化自然就是稳定梯度

Linear 是将 (batch_size, sequence_length, d_model)—>(batch_size, sequence_length, 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。

4

Transformer 模型的训练

前面的章节讨论的是 Transformer 模型的架构和理论知识, 这一章节侧重讲解 Transformer 模型的训练方法 (事实上对于绝大多数深度学习的训练都适用)。主要涉及:

- **损失函数**: 需要定义交叉熵损失函数
- **优化器**: 需要定义 AdamW 优化器来最小化损失
- **训练循环**: 需要构建完整的基础设施来加载数据、保存检查点和管理训练过程

4.1. 信息论前置知识 (简易版)

4.1.1. 信息 (Information)

它用来**度量事件发生所消除的不确定性 (Uncertainty) 的大小**。

1. 直观理解
   - “明天太阳会照常升起”: 这句话包含的信息量很小。因为它几乎是必然发生的, 没有消除我们什么不确定性。
   - “明天北京会下雪 (假设在夏天)”: 这句话包含的信息量极大。因为它是一个极小概率事件, 一旦发生, 就极大地消除了我们的不确定性。
2. 量化定义: 自信息 (Self-Information) 信息论使用 “自信息” 来量化单个事件发生时所提供的信息量。一个事件 x 的自信息量 I(x) 定义为:
   $I(x) = -\log_b(p(x))$
   - p(x): 事件 x 发生的概率。
   - $\log_b$: 对数函数。底数 b 的选择决定了信息量的单位。
     - b=2: 单位是 **比特 (bit)**, 这是最常用的单位, 对应于二进制世界。
     - b=e: 单位是 **奈特 (nat)**。
     - b=10: 单位是 **哈特利 (hartley)**。
   - 负号: 因为概率 p(x) 在 [0, 1] 之间, 它的对数是小于等于 0 的。加上负号可以确保信息量是一个非负数, 这符合我们的直觉。
3. 示例
   - 假设我们抛一枚均匀的硬币:

- “正面朝上”的概率 p(正面) = 0.5。
  它提供的信息量 $I() = -\log_2(0.5) = -\log_2(2^{-1}) = -(-1) = 1bit$。
- 同样，“反面朝上”提供的信息量也是 1 bit。
  这告诉我们，要确定一个硬币的正反面，我们需要 1 bit 的信息。

4.1.2. 熵 (Entropy)

如果我们想衡量的是一个**系统 (随机变量) 的整体不确定性**，而不是单个事件，那就要用到“熵”的概念。

1. 直观理解
   - 熵是**信息量的期望值 (数学期望)**。它衡量了一个随机变量所有可能结果的平均不确定性。
   - **一个系统越混乱、越不可预测，它的熵就越高**。
   - **一个系统越稳定、越可预测，它的熵就越低**。
2. 量化定义
   - 对于一个离散随机变量 X，它有多种可能的取值 $\{x_1, x_2, ..., x_n\}$，对应的概率为 $\{p(x_1), p(x_2), ..., p(x_n)\}$。那么，这个随机变量 X 的熵 H(X) 定义为:
   - $H(X) = E[I(X)] = \sum_{i=1}^{n} p(x_i)I(x_i) = -\sum_{i=1}^{n} p(x_i)\log_b(p(x_i))$
3. 示例
   - 比较两枚硬币的熵:
     - 均匀硬币 (Fair Coin):p(正面)=0.5, p(反面)=0.5
     - 不均匀硬币 (Biased Coin):p(正面)=0.9, p(反面)=0.1
     - 两面都是正面的硬币 (Two-headed Coin):p(正面)=1, p(反面)=0
   - 均匀硬币 (Fair Coin):p(正面)=0.5, p(反面)=0.5
     $H(X) = -[0.5 \times \log_2(0.5) + 0.5 \times \log_2(0.5)] = 1bit$
   - 不均匀硬币 (Biased Coin):p(正面)=0.9, p(反面)=0.1
     $H(X) = -[0.9 \times \log_2(0.9) + 0.1 \times \log_2(0.1)] \approx 0.469bit$
   - 两面都是正面的硬币 (Two-headed Coin):p(正面)=1, p(反面)=0
     $H(X) = -[1 \times \log_2(1) + 0 \times \log_2(0)] = 0 (0 \log 0 = 0)$

4.1.3. 熵的相关扩展概念

理解了熵之后，其他几个重要概念就很容易理解了，它们描述了多个随机变量之间的关系。

1. **联合熵 (Joint Entropy)**
   - 衡量**两个或多个随机变量共同**的不确定性。对于两个变量 X 和 Y，其联合熵 H(X, Y) 为:
   - $H(X,Y) = -\sum_{x \in X}\sum_{y \in Y} p(x,y)\log_2(p(x,y))$

- 其中 p(x, y) 是 X=x 和 Y=y 同时发生的联合概率。

2. **条件熵 (Conditional Entropy)**
   - 在**已知一个随机变量 X 的情况下, 另一个随机变量 Y 剩下的不确定性**。记为 H(Y|X)。
   - $H(Y|X) = \sum_{x \in X} p(x) H(Y|X = x)$
   - 它表示, 知道了 X 之后, 对 Y 的不确定性还剩下多少。
3. **互信息 (Mutual Information)**
   - **一个随机变量 X 的信息中, 有多少是与另一个随机变量 Y 共享的**。它衡量了两个变量之间的相关性。记为 I(X; Y)。
   - **直观理解**: 知道了 X 之后,Y 的不确定性减少了多少。
   - **计算公式**:
     - $I(X;Y) = H(Y) - H(Y|X)$(**Y 的总不确定性 - 知道 X 后 Y 剩下的不确定性**)
     - $I(X;Y) = H(X) - H(X|Y)$(**对称的**)
     - $I(X;Y) = H(X) + H(Y) - H(X,Y)$

Venn 图关系你可以把熵想象成一个集合, 那么这些概念的关系就非常清晰了:

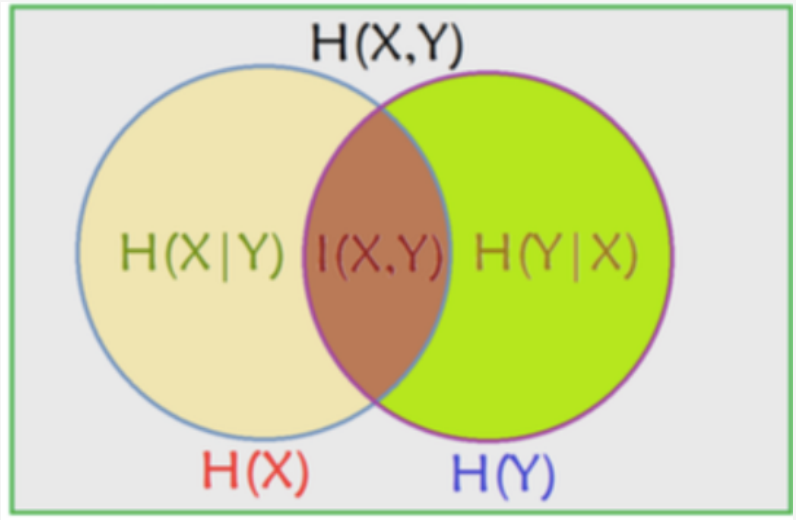


图 4.1: 信息论概念的 Venn 图关系

- 左边的圆圈是 H(X)
- 右边的圆圈是 H(Y)
- 两个圆圈的重叠部分是**互信息 I(X; Y)**
- H(X) 中不重叠的部分是**条件熵 H(X|Y)**
- H(Y) 中不重叠的部分是**条件熵 H(Y|X)**
- 整个两个圆圈覆盖的区域是**联合熵 H(X, Y)**

4.1.4. 一个生动的比喻: 猜数字游戏

这个比喻可以把所有概念串起来。

- **游戏**: 我从 1 到 8 之间想一个数字, 你来猜。
- **熵 H(X)**: 在游戏开始前, 这个数字是什么？你完全不知道, 有 8 种可能性, 每种可能性概率为 1/8。这个系统的总不确定性是 $H(X) = -\sum_{i=1}^{8} \frac{1}{8} \log_2(\frac{1}{8}) = \log_2(8) = 3bits$ 。这恰好是你用“二分法”猜中这个数字所需的最少问题数 (例如:“比 4 大吗?”“比 6 大吗?”“是 7 吗?”)。
- **信息 I(x)**: 我告诉你答案是“5”。这个具体事件提供的信息量是 $I(\text{"5"}) = -\log_2(1/8) = 3bits$ 。一旦你知道了这个信息, 所有不确定性都消除了。
- **互信息 I(X;Y)**: 现在, 我不直接告诉你答案, 而是给你一个提示 **Y**:“这个数字是奇数”。
  - 这个提示 **Y** 本身也有不确定性 (可能是奇数或偶数), 它的熵是 **H(Y)=1 bit**。
  - 这个提示 **Y** 给你提供了多少关于 **X** 的信息？这就是互信息。知道了它是奇数, 可能性从 1,2,3,4,5,6,7,8 缩小到了 1,3,5,7。不确定性大大降低。
- **条件熵 H(X|Y)**: 你知道了“数字是奇数” **(Y)** 之后, 对数字 **(X)** 还剩下多少不确定性?
  - 现在只剩下 4 种可能性 1,3,5,7, 每种概率为 1/4。
  - 剩下的不确定性 (条件熵) 是 $H(X|Y = \text{"奇数"}) = -\sum_{i\in\{1,3,5,7\}} \frac{1}{4} \log_2(\frac{1}{4}) = \log_2(4) = 2bits$。
  - 这意味你还需要问 2 个问题才能猜到。
- **关系验证**:
  - $I(X;Y) = H(X) - H(X|Y) = 3 - 2 = 1bit$
  - 这说明,“数字是奇数”这个提示, 为你提供了 1 bit 的信息。

中文名称	描述	关键点
自信息	度量**单个具体事件**发生所消除的不确定性。	概率越小, 信息量越大。
熵	度量**整个系统 (随机变量)** 的平均不确定性。	信息量的数学期望。系统越混乱, 熵越高。
联合熵	度量**两个或多个系统**共同的不确定性。	把多个变量看成一个整体。
条件熵	在**已知一个变量**后, 另一个变量**剩下**的不确定性。	$H(Y\|X)$, 知道 X 后 Y 还有多乱。
互信息	两个变量**共享**的信息量, 即相关性程度。	$I(X;Y)$, 知道 X 能帮我们消除 Y 多少不确定性。

4.2. 交叉熵 (Cross-Entropy) 损失与 KL 散度

参考资料 4.1

交叉熵损失函数:*https://blog.csdn.net/SongGu1996/article/details/99056721*

建议先熟悉信息论中的基本概念，如熵、相对熵、交叉熵等。

4.2.1. KL 散度 (相对熵)

假设随机变量 X 的真实概率分布为 P(X)，而我们在处理实际问题时使用了一个近似的分布 Q(X) 来进行建模。由于我们使用的 Q(X) 是而不是真实的，所以我们在具体化的取值时需要一些**附加的信息**来抵消分布不同造成的影响。我们需要的平均附加信息量可以使用**相对熵**，或者叫**KL 散度 (Kullback-Leibler Divergence)**来计算,KL 散度可以用来**衡量两个分布的差异**:

$$D_{KL}(P||Q) = -\sum_{i=1}^{n} P(x_i)logQ(x_i) - (-\sum_{i=1}^{n} P(x_i)logP(x_i)) = \sum_{i=1}^{n} P(x_i)log\frac{P(x_i)}{Q(x_i)}$$

KL 散度的性质:

1. KL 散度不是一个对称量 $D_{KL}(P||Q) \neq D_{KL}(Q||P)$
2. KL 散度永远大于等于 0(因为最好也就是 $Q(X) = P(X)$，此时额外信息量为 0)

4.2.2. 交叉熵

实际上就是 KL 散度的第一项值

$$D_{KL}(P||Q) = -\sum_{i=1}^{n} P(x_i)logQ(x_i) - (-\sum_{i=1}^{n} P(x_i)logP(x_i)) = -H(P) + H(P,Q)$$

$$H(P,Q) = H(P) + D_{KL}(P||Q) = -\sum_{i=1}^{n} P(x_i)\log Q(x_i)$$

对于真实分布 $P(X)$ 来讲，它的熵 $H(P)$ 是一个固定值,**真正决定 KL 散度值的还是交叉熵**。

我们训练神经网络的目标就是希望近似分布 Q 逼近真实分布 P。

4.2.3. 交叉熵损失函数

二分类

只有两类，设:

- 真实标签为 $y \in \{0, 1\}$
- 模型输出的预测概率为 $\hat{y} \in [0, 1]$

则交叉熵损失函数为:

$$L = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

```
loss_fn = nn.BCELoss()
    loss = loss_fn(y_pred, y_true)
```

多分类

设:

- 真实标签 $y \in \{0, 1, ..., K-1\}$ 是类别索引
- 模型预测的是一个概率分布向量 $\hat{y} = [p_1, ..., p_K]$

一般来讲,p_i 是要经过 softmax 处理的, 即 $p_i = \frac{e^{x_i}}{\sum_{k=0}^{C-1} e^{x_k}}$

多分类交叉熵公式为:

CrossEntropy Loss $= -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$

```
loss_fn = nn.CrossEntropyLoss()
    loss = loss_fn(logits, targets)# 注意 torch 里面会自动计算 softmax, 所以不需要人为再去
    ↪ 计算一遍了
```

4.2.4. 实际代码的应用

Exp-normalize 技巧 + LogSoftmax + NLLloss

Exp-normalize 技巧

Exp-normalize 技巧的目标是为了实现后续 softmax 计算数值的稳定性, 避免出现上溢或者下溢的情况。具体来说,

先将输入减去向量最大值:$z = x - max(x)$

这样做, 所有的 $x_i \subseteq (-\infty, 0]$, 也就是 e^x 的值域只会在 y 轴左半部分, 这样做的好处是:

- 最大的值变为 0, 避免其可能原始数值很大, 导致上溢指数爆炸;
- 至少有一个 exp(0)=1, 防止 softmax 里的分母为 0 下溢。
- 这种变换不改变最终 softmax 的输出, 因为指数任意平移不改变比值。

$\text{softmax}(x)i = \frac{e^{xi}}{\sum_j e^{x_j}} = \frac{e^{x_i - m}}{\sum_j e^{x_j - m}} = \text{softmax}(z)_i$

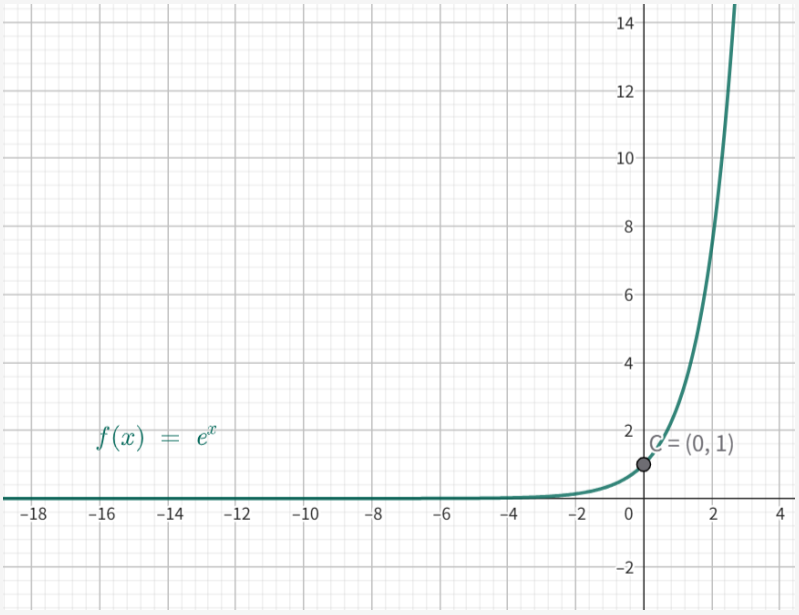


图 4.2: Exp-normalize 技巧

LogSoftmax

实际上就是把 log 和 softmax 两种运算给组合起来了。

softmax 运算:

$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

LogSoftmax 运算:

$\log \frac{e^{x_i}}{\sum_j e^{x_j}} = x_i - \log \sum_j e^{x_j}$

直接使用 Logsoftmax 而不是分步使用 log 和 softmax 的好处在于:

- 减少计算开销, 直接使用了 x_i 等价先 $softmax(x_i)$ 再 log
- 直接使用 Logsoftmax 计算梯度会更加稳定

NLLloss 负对数似然损失函数

公式:nllloss $= -\frac{1}{N}\sum_{k=1}^{N} P(y_k) \cdot (\log\_softmax)$

其实就是 Logsoftmax 后续的补充操作, 使得结果与 Crossentropy 结果一致。

```
logp = F.log_softmax(logits, dim=1)    # 计算 log 概率:shape = [batch, K],
loss = F.nll_loss(logp, targets)       # targets 是 [batch] 维度的真实类索引
```

这两行代码等价于直接用 crossentropy

```
loss = F.cross_entropy(logits, targets)
```

4.2.5. 具体在 Transformer 中的应用

Transformer 作为一个**自回归的语言模型**, 其核心任务是**预测下一个词**。给定一个词序列的前 i 个词 $x_{1:i}$, 它需要预测出第 i+1 个词 x_{i+1} 的概率分布 $p_\theta(x_{i+1}|x_{1:i})$。这里的 θ 代表模型的所有参数 (权重)。

$\ell(\theta; D) = \frac{1}{|D|m}\sum_{x\in D}\sum_{i=1}^{m} -\log p_\theta\left(x_{i+1} \mid x_{1:i}\right)$

$\ell(\theta; D)$ 就是这个损失函数。我们来把它拆解一下:

- $|D|$ 是数据集中序列的数量。
- m 是我们考虑的序列长度。
- $\sum_{x\in D}\sum_{i=1}^{m}$ 这个双重求和符号代表我们要计算数据集中每一条序列里, 从第 1 个位置到第 m 个位置的每一次预测的损失, 然后把它们全部加起来。
- $-\log p_\theta(x_{i+1}|x_{1:i})$ 这是整个损失函数的核心。
- $p_\theta(...)$ 是模型预测的、下一个词恰好是正确答案 x_{i+1} 的概率。
- **我们的目标是让这个概率尽可能高, 理想情况下趋近于 1**。
- log 函数在 (0, 1] 区间内是单调递增的, 所以让 p 最大化等同于让 $\log(p)$ 最大化。
- 加上一个负号 -, 最大化 $\log(p)$ 就变成了最小化 $-\log(p)$。这正是机器学习中梯度下降的目标: 最小化损失函数。

- $\frac{1}{|D|m}$: 最后除以总的预测次数 (序列数 × 序列长), 这是为了计算平均损失, 避免数据集大小对损失值产生影响。

模型在内部并不会直接输出概率, 而是先输出一个叫做**logits**的原始数值向量 o_i。这个向量的维度等于你的词汇表大小 (vocab_size), 其中每个元素可以看作是对应单词的 **“得分”或“置信度”**, 这个得分可以是任意实数 (可正可负)。

为了把这些原始得分 o_i 转换成一个合法的概率分布 (即所有值都在 0 到 1 之间, 且加起来等于 1), 我们使用**Softmax**函数。

$p(x_{i+1}|x_{1:i}) = \text{softmax}(o_i)[x_{i+1}] = \frac{\exp(o_i[x_{i+1}])}{\sum_{a=1}^{\text{vocabsize}} \exp(o_i[a])}$

这个公式展示了如何计算正确词 x_{i+1} 的概率。

- 分子 $\exp(o_i[x_{i+1}])$: 对正确词的 logit 分数取指数, 确保其为正数。
- 分母 $\sum_{a=1}^{\text{vocabsize}} \exp(o_i[a])$: 对词汇表中所有词的 logit 分数取指数然后求和, 这是一个归一化项。
- 两者相除, 就得到了正确词 x_{i+1} 的最终概率。

总结一下整个流程:

对于训练数据中的一个序列, 在每一个位置 i:

- Transformer 模型接收前面的序列 $x_{1:i}$ 作为输入。
- 模型输出一个大小为 vocab_size 的 logits 向量 o_i。
- 通过 Softmax 函数将 o_i 转换为概率分布。
- 取出真实下一个词 x_{i+1} 对应的概率, 取负对数得到该位置的损失 $-\log(p)$。
- 将所有位置、所有序列的损失加起来求平均, 得到最终的总损失, 然后通过反向传播来更新模型参数 θ 以最小化这个损失。

4.2.6. Perplexity(困惑度)

Perplexity(困惑度) 是**衡量语言模型预测能力的指标**, 它反映了模型对测试数据的不确定性。**Perplexity 越低, 说明模型对数据的预测越准确; Perplexity 越高, 说明模型对数据的预测越不确定**。

Perplexity 的数学定义

Perplexity(PP) 与交叉熵 (Cross-Entropy) 密切相关, 数学定义如下:

$PP(W) = 2^{H(W)}$

其中,$H(W)$ **是语言模型的交叉熵**, 计算公式如下:

$H(W) = -\frac{1}{N}\sum_{i=1}^{N} \log_2 P(w_i|w_1, w_2, ..., w_{i-1})$

- **W**: 一整段文本
- **N**: 文本中的单词总数
- $P(w_i|w_1, ..., w_{i-1})$: 模型对单词 w_i 在给定前面单词的条件下的预测概率

直观理解

1. **如果模型完美预测每个单词的概率都是 1**, 那么 **PP = 1**, 即模型完全不困惑。
2. **如果模型随机猜测 (每个单词概率相等)**, 那么 **PP 会很高**, 表示模型的预测不确定。

Perplexity 的意义

衡量语言模型的好坏:

1. **低 Perplexity(PP 低)** → 说明模型更擅长预测下一个单词, 效果更好。
2. **高 Perplexity(PP 高)** → 说明模型对语言的理解较差, 需要优化。

用于对比不同模型的性能:

- 例如,GPT-3 的 Perplexity 低于 GPT-2, 说明 GPT-3 在相同数据集上的预测能力更强。

优化语言模型的目标:

- 训练过程中, 优化目标通常是**最小化交叉熵 (降低 Perplexity)**, 使模型更擅长语言理解和生成。

4.3. Optimizer: 优化器

参考资料 4.2

通俗易懂理解 (梯度下降) 优化算法:*https://blog.csdn.net/Invokar/article/details/86768571*
优化算法《动手学深度学习》:*http://zh.gluon.ai/chapter_optimization/index.html*
神经网络中 warmup 策略为什么有效:*https://www.zhihu.com/question/338066667*

Optimizer(优化器) 是深度学习中的一种核心算法, 其主要作用是根据模型的损失函数 (Loss Function) 计算出的梯度 (Gradient), 来指导并更新网络中的参数 (如权重和偏置), 目标是让损失函数的值越来越小, 从而使模型的预测结果越来越准确。

常见的 Optimizer 包括:**SGD**、**Momentum**、**Adagrad**、**RMSprop**、**Adam 等**。

4.3.1. SGD (随机梯度下降)

SGD 是一般的梯度下降法的近似版本, 传统的梯度下降法是针对**整个训练集**进行求梯度, 而 SGD(mini SGD) 是对训练集的**一小批量**进行求梯度并更新。

公式 (和梯度下降法一样):

$\theta_{t+1} = \theta_t - \alpha \nabla_\theta J(\theta_t)$

其中:

- θ_t: 第 t 步的参数 (或者权重);

- α: 学习率;
- $\nabla_\theta J(\theta_t)$: 损失函数对参数的梯度。$\nabla_\theta J(\theta_t) = \frac{1}{|\mathcal{B}_t|}\sum_{i\in\mathcal{B}_t} f(x_i, \theta)$ 其中 $\mathcal{B}_t$ 代表这个批次

相较于 GD ,SGD **计算损耗更少**, 但梯度的估计存在噪声, 导致收敛过程会发生震荡, 并且**不保证能收敛到全局最优点**。

4.3.2. 动量法 (Momentum)

直接利用 SGD 的话, 可能出现相邻两个时刻梯度差距过大, 来回**震荡**, 动量法利用了上一时刻的梯度值, 在 SGD 的基础上引入“动量”, 即**历史梯度的线性组合 (加权平均)**, 以缓解震荡、加速收敛。

Keynote 4.1

为什么动量法要叫动量法？

这是从山坡上最快降到谷底这一物理过程上类比得出来的, 实际上叫”惯性法“更加合适。对于梯度法, 它没有考虑历史时刻的速度也就是惯性, 而动量法通过将当前梯度与上一时刻的速度进行加权平均的方式利用了惯性。打个比方, 一个小球从山坡上降到谷底, 梯度法不考虑它的速度, 没到一个地方就计算这个地方的梯度, 那么在接近谷底的陡峭点 (梯度高), 那么小球就直接冲到山谷另一侧了, 而如果我们保留了历史的速度, 在这个位置冲到另一侧时就会由于动量的存在冲的没那么厉害, 减少震荡。

公式:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha v_t$$

其中:

- v_t: 动量变量, 表示历史梯度的指数加权平均;
- β: 动量衰减系数 (惯性系数), 常取 0.9。

动量法的两大好处

1. **在梯度方向一致时加速**:
   在山谷的一侧坡上, 梯度方向基本不变 (一直指向谷底)。每一时刻的梯度 (加速度) $J(\theta_t)$ 都会和上一时刻的速度 v_{t-1} 方向**相同**, 速度 v_t 就会不断累积, 变得越来越大, 从而让参数 的更新步伐越来越大, 实现**加速**效果。
2. **在梯度方向震荡时抑制**:
   在山谷的陡峭两侧, 梯度方向会来回变化 ($J(\theta_t)$ 和 $J(\theta_{t-1})$ 方向相反)。这时,$J(\theta_t)$ 会和 v_{t-1} 中的震荡方向分量相互**抵消**, 使得 v_t 在这个方向上的值变小。这就起到了**抑制震荡**、平滑更新路径的作用。

但相较于 GD、SGD 来说, 动量法需要**额外维护一个与模型参数量等大的动量变量** v_{t-1}, 其空间复杂度为 O(N),N 为模型参数数量。

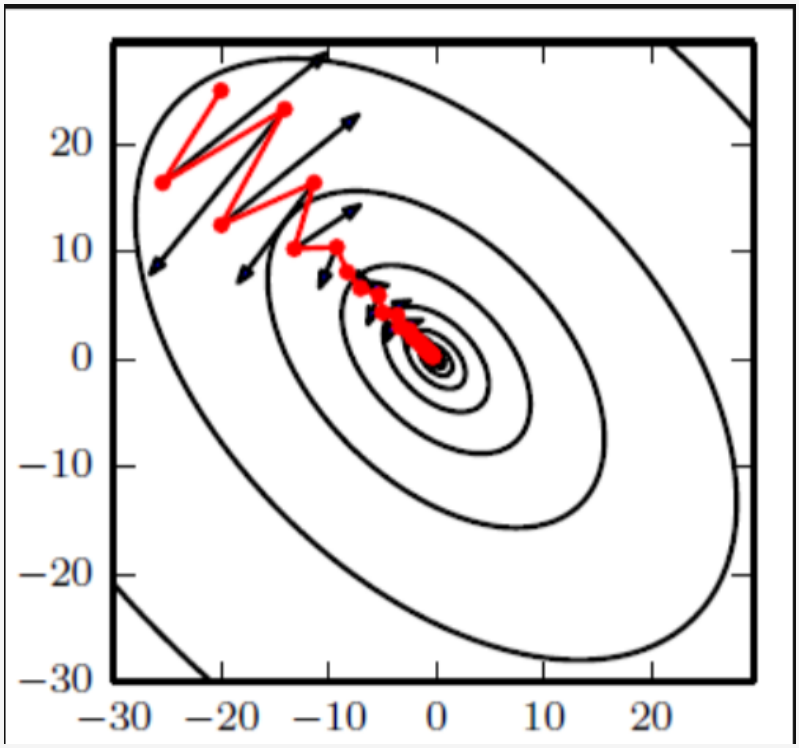


图 4.3: 动量法示意图

4.3.3. AdaGrad

核心思想: 为每一个参数分配**不同**的**自适应学习率**

像 GD、SGD 这些算法的学习率 α 一般都是固定的,AdaGrad 认为有些参数更新频繁, 有些很少更新, 不同的参数应根据其**历史梯度**有不同学习率

公式:

$G_t = G_{t-1} + g_t g_t$

$x_t = x_{t1} \frac{\alpha}{+G_t} g_t$

其中:

- g_t: 第 t 步的梯度;
- G_t: 从第 1 步到第 t 步, 梯度平方的累加;
- 表示**逐元素相乘**

e.g.:

假设在二维平面上的梯度 g=[4,9], 初始时 G=0, 则 G_1 =[16,81], 我们对第一维度进行更新时, 采用 $x_t^{(1)} = x_{t-1}^{(1)} - \frac{\eta}{\epsilon+\sqrt{16}} \times 4$

对第二维度进行更新时, 采用 $x_t^{(2)} = x_{t-1}^{(2)} - \frac{\eta}{\epsilon+\sqrt{81}} \times 9$

从上面我们就可以看出, 当梯度的某一个方向上比较大时, 分母会约束其变小, 而当梯度的某一个方向上比较小时, 分母会令其相对变大。从中其实可以体现出归一化的效果, 有点类似于下图的转换结果:

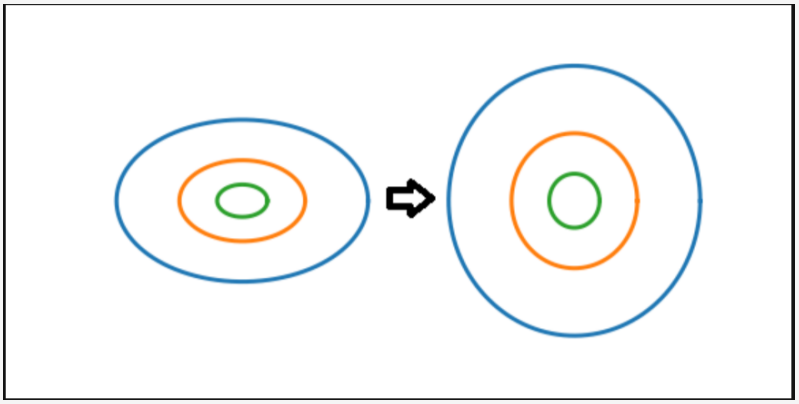


图 4.4: AdaGrad 示意图

AdaGrad 的一个重大缺点是 G_t 是**无上限累计的**, 这样到后期学习率基本上单调递减到 0 了, 就不会训练了。

4.3.4. RMSProp

为了克服 AdaGrad 的缺点,RMSProp 不是直接累加所有历史梯度, 而是使用**滑动窗口式的加权平均**, 只“记住**最近**的梯度行为”。

设梯度为 g_t, 均方梯度为 $E[g^2]_t$:

1. 估计梯度平方的指数平均:
   $E[g^2]_t = \beta \cdot E[g^2]_{t-1} + (1-\beta) \cdot g_t^2$
2. 更新参数:
   $\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$

其中:

- g_t^2: 表示梯度按元素平方的值

$$
\begin{aligned}
E\left[g^2\right]_1 &= (1-\gamma)g_1^2 \\
E\left[g^2\right]_2 &= \gamma E\left[g^2\right]_1 + (1-\gamma)g_2^2 = \gamma(1-\gamma)g_1^2 + (1-\gamma)g_2^2 \\
E\left[g^2\right]_3 &= \gamma E\left[g^2\right]_2 + (1-\gamma)g_3^2 \\
&= \gamma^2(1-\gamma)g_1^2 + \gamma(1-\gamma)g_2^2 + (1-\gamma)g_3^2 \\
&\cdots \\
E\left[g^2\right]_t &= (1-\gamma)\sum_{i=1}^{t}\gamma^{t-i}g_i^2
\end{aligned}
$$

最近的梯度平方权重大,**越早之前的梯度影响越小**(指数级衰减).

直观理解

1. RMSProp 旨在估计一个“近似**局部**梯度方差”;
2. 如果梯度波动很大 (即平方值变化大), 那 $E[g^2]_t$ 就会大;

3. RMSProp 用这个值来**对当前梯度进行缩放**, 让不同参数维度的学习率自适应:

$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$ 这意味着:

- 如果某个参数维度的梯度在近期波动很大 (不稳定), 我们就**减少它的学习率**;
- 如果某个维度的梯度变化小 (较稳定), 就**保持或稍大更新步幅**。

超参数的取值:

- β 取值:
  - 接近 1(如 0.99): 长记忆 (依赖更长的历史)
  - 接近 0(如 0.5): 短记忆 (几步就忘)
  - 通常设置为 0.9: 平衡短期与长期趋势

4.3.5. Adam

Adam 是把之前的优化器的特点都融合起来了, 不仅**看当前位置的梯度**(就像普通 SGD 那样), 还**考虑过去梯度的平均值**(动量) 和**梯度平方的平均值**(像 RMSProp 那样), 并进行**自适应的学习率调整**, 从而在不同参数维度上使用不同的更新步长。

假设我们在训练中第 t 步, 梯度为 g_t。Adam 的更新公式如下:

1. **一阶矩估计 (动量)**:
   $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
2. **二阶矩估计 (RMS)**:
   $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
3. **偏差校正**:
   $\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$
4. **参数更新**:
   $\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$

其中:

- θ 为当前模型参数 - g_t 为当前梯度 (损失对参数的导数) - m_t 为梯度的指数加权平均 (动量) - v_t 为梯度平方的指数加权平均 - β_1 控制动量的衰减率, 常设为 0.9 - β_2 控制梯度平方平均的衰减率, 常设为 0.999 - α 代表学习率, 默认 0.001 - ϵ 是防止除以 0 的微小常数, 通常设为 10^{-8}

Keynote 4.2

为什么要进行偏差矫正？ 由于 v_t 是从 $v_0 = 0$ 开始的，所以在前几步迭代时，v_t 的值明显偏小 *(因为它平均值主要还在积累阶段)*。例如第 *1* 步：$v_1 = (1-\beta)g_1$ 这个值太小了，因为历史还没积累。于是，我们用一个“修正系数”将它还原为更合理的期望值：$\hat{v}_t = \frac{v_t}{1-\beta^t}$
其中：

- v_t 是带偏的动量估计；
- $\hat{v}_t$ 是**无偏估计***(bias-corrected)*；
- $1-\beta^t$ 是第 t 步的修正系数。

随着 t 增大，β^t 趋近于 0，修正项变为 1，所以只在初期有显著影响。

Keynote 4.3

因为 $v_0 = 0$ 不是一个真实的估计，它肯定比现实值要小，所以 v_t 是被低估了。而由上面的推导我们知道 $E\left[g^2\right]_t = (1-\gamma)\sum_{i=1}^{t}\gamma^{t-i}E[g_i^2]$，假设每一步期望采样独立同分布，令 $E[g_i^2] = \mu$ 因此有 $E[v_t] = (1)\sum_{i=1}^{t}{}^{ti}E[g_i^2] = (1)\sum_{i=1}^{t}{}^{ti}$
$= (1)\sum_{i=1}^{t}{}^{ti} = (1)\sum_{j=0}^{t-1}{}^{j}(\check{a}j = ti) = (1)\frac{1^t}{1}$
所以有 $E[v_t] = E[g^2](1^t)$ 因此误差估计的修正项就是 $\frac{1}{1-\beta^t}$ 最终的 $\hat{v}_t$ 是无偏估计

4.3.6. AdamW

在原始的 Adam 中，我们常常使用 **L2 正则项** 来防止过拟合，形式是：$L(\theta) + \frac{\lambda}{2}\|\theta\|^2$ 这就会带来一个额外的梯度项 $\lambda \cdot \theta$，被加入到梯度更新中。

所以，很多框架 (如 PyTorch) 默认会把它**加到梯度上**：

$g_t^{\text{new}} = g_t + \lambda \cdot \theta$

然后再执行 Adam 的更新逻辑，这样附加项会对原有正常结果产生影响。

AdamW 就是希望**将正则项“从梯度更新中剥离”，变成一个独立的权重衰减项。**

即：

$\theta_{t+1} = \theta_t - \alpha\left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} + \lambda \cdot \theta_t\right)$

Keynote 4.4

注意：*weight decay(L2 衰减)* 是直接对参数 θ_t 进行缩小，而不是对梯度进行修改！

4.4. 自适应学习率调整 (学习率调度器)

学习率是调整梯度下降步长的超参数，学习率过大过小都会导致模型无法收敛。在实际训练中，我们常常会根据训练的轮数来调整学习率，这就是学习率调整。

Keynote 4.5

为什么需要学习率调度器？

以一个“一个盲人下山”为例子。

- **山:** 代表整个复杂的损失函数 *(Loss Landscape)*。山的海拔高度就是损失函数的值 *(Loss)*。
- **盲人:** 代表我们的模型。
- **盲人的位置:** 代表模型当前的一组参数 *(权重和偏置)*。
- **目标:** 走到山谷的最低点，也就是让损失函数的值最小。
- **拐杖:** 用来感知脚下地面的坡度，这个“坡度”就是梯度 *(Gradient)*。梯度会指向当前位置最陡峭的上升方向，那么它的反方向就是下降最快的方向。
- **如何走路 *(策略)*:** 这就是 *Optimizer(优化器)*。它决定了盲人听从拐杖的建议后，下一步该往哪里走，走多大一步。
- **步子的大小:** 这就是学习率 *(Learning Rate)*。步子太大可能会直接跨过最低点 *(“扯着蛋”)*，步子太小则下山速度太慢。

如果整个下山过程都用固定的步长 *(固定的学习率)*，会遇到两个问题：

- **步子太大:** 在山坡上 *(训练初期)* 走得很快，但快到山谷 *(最优点)* 附近时，因为步子太大，很容易一步跨过最低点，然后在谷底来回 **“震荡”**，始终无法精确到达谷底中心。
- **步子太小:** 从一开始就走得很慢，虽然最终能精确到达谷底，但整个下山过程会非常非常耗时 *(训练时间过长)*。

一个聪明的盲人会怎么做？他会先迈大步，快速接近谷底的大致区域；感觉脚下的路越来越平坦时，就放慢脚步，小心翼翼地小步探索，找到最低的那个点。这就是学习率调度器的核心思想。
学习率调度器就是用来调整学习率的，它通常是一个函数，输入是当前的迭代次数，输出是当前的学习率。

常见的学习率调度器包括:**StepLR、CosineAnnealingLR、ReduceLROnPlateau 等**。

4.4.1. StepLR (阶梯式下降)

StepLR 是一种简单的学习率调度器，它根据训练的轮数来调整学习率，通常是**阶梯式下降**。

- **公式:**
  $\alpha_t = \alpha_0 \cdot \gamma^t$ 其中 α_0 是初始学习率，γ 是衰减因子，t 是当前的迭代次数。
- **特点:** 学习率的变化是跳跃式的，像下楼梯一样，每隔一段时间就下降一个台阶。**简单有效，但不够平滑**。

4.4.2. CosineAnnealingLR (余弦退火)

余弦退火的方式就是预先**设定一个初始学习率**, 然后逐步衰减。然后余弦的**周期性**让学习率在降到最低后再到最高值。

公式:

(Warm-up) If $t < T_w$, then $\alpha_t = \frac{t}{T_w}\alpha_{max}$.

(Cosine annealing) If $T_w \leq t \leq T_c$, then $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t-T_w}{T_c-T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$.

(Post-annealing) If $t > T_c$, then $\alpha_t = \alpha_{min}$.

其中 T_w 是预热轮数,T_c 是余弦退火轮数,α_{max} 是初始学习率,α_{min} 是最小学习率。

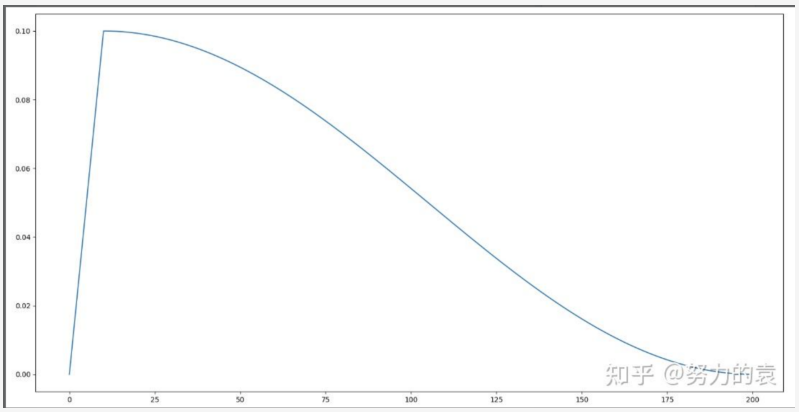


图 4.5: 余弦退火示意图

```python
class CosineSchedule:
    def __init__(self, max_learning_rate, min_learning_rate, warmup_iters,
    ↪  cosine_cycle_iters):
        self.max_learning_rate = max_learning_rate
        self.min_learning_rate = min_learning_rate
        self.warmup_iters = warmup_iters
        self.cosine_cycle_iters = cosine_cycle_iters

    def __call__(self, it):
        if it < self.warmup_iters:
            return self.max_learning_rate * it / self.warmup_iters
        elif it > self.cosine_cycle_iters:
            return self.min_learning_rate
        else:
            return self.min_learning_rate + (self.max_learning_rate -
            ↪  self.min_learning_rate) * (1 + math.cos(math.pi * (it -
            ↪  self.warmup_iters) / (self.cosine_cycle_iters - self.warmup_iters))) / 2
```

4.4.3. ReduceLROnPlateau (遇到平台期就减速)

ReduceLROnPlateau (Reduce Learning Rate On Plateau) 是一种“自适应”学习率调度策略。它通过监控一个指定的性能指标 (通常是**验证集损失 val_loss**), 当这个指标在一定时期内 (称为 **“耐心值” patience**) 不再出现明显改善时, 它就会自动将当前学习率降低一个固定的比例 (factor), 帮助模型“小步慢走”, 从而跳出平台

期, 更精细地寻找最优解。

解释:

我们可以把这个过程想象成“一位有耐心的老师在辅导学生”。

- **学生:** 我们的模型。
- **考试分数:** 被监控的指标 (比如验证集损失 **val_loss**)。
- **学习方法/强度:** 学习率。
- **老师:** ReduceLROnPlateau 调度器。

这位老师的辅导策略是:

- **观察 (Monitor):** 每次考完试 (每个 epoch 结束后), 老师都会记录下学生的分数 (val_loss)。
- **保持耐心 (Patience):** 老师不会因为学生一次没考好就立刻改变策略。他会设定一个**耐心值**, 比如 patience=5, 意味着他会连续观察 5 次考试。
- **判断平台期 (Plateau):** 如果在这连续的 5 次考试中, 学生的最高分都没有被刷新 (val_loss 没有降到更低), 老师就认为学生遇到了“学习瓶颈”或**“平台期”**。
- **调整策略 (Reduce LR):** 一旦确认学生进入平台期, 老师就会调整辅导策略, 说:“看来之前的方法太激进了, 我们放慢点, 把知识点弄得再细一些。”——**这就是将学习率乘以一个衰减因子 (factor), 比如 0.1, 让学习率骤减**。
- **冷静期 (Cooldown):** 调整策略后, 老师会进入一个**“冷静期”**, 比如 cooldown=2。在这两次考试内, 即使学生分数还没起色, 老师也不会再调整策略, 而是给学生时间去适应新的、更慢的学习节奏。

工作流程示例:

假设我们设置如下:

monitor='val_loss', mode='min', patience=3', factor=0.5, lr=0.01

Epoch	val_loss	说明	Patience 计数	学习率 (LR)
1	1.0	初始最佳值	0	0.01
2	0.9	改善, 更新最佳值为 0.9	重置为 0	0.01
3	0.8	改善, 更新最佳值为 0.8	重置为 0	0.01
4	0.82	未改善	1	0.01
5	0.81	未改善	2	0.01
6	0.83	未改善	3	0.01
触发条件:Patience 达到 3,LR 需要降低				
7	0.75	改善, 更新最佳值为 0.75	重置为 0	0.005
8	0.76	未改善	1	0.005

表 4.1: ReduceLROnPlateau 工作流程示例

优缺点总结:

1. **优点:**
   - **自适应和直观:**它的逻辑非常符合直觉, 能够根据模型的真实表现来调整学习率, 而不是依赖一个固定的、需要预先猜测的时间表。
   - **对于精调非常有效:**在训练后期, 当模型性能提升变得困难时, 这种“自动减速”机制对于找到更深、更优的最小值点非常有帮助。
2. **缺点:**
   - **依赖高质量的验证集:**如果验证集的表现有很大的随机性或噪声, 可能会导致调度器过早或错误地降低学习率。
   - **可能反应较慢:**如果 patience 设置得过高, 模型可能会在平台期浪费很多个 epoch, 拖慢整体训练进程。

4.4.4. Warmup (预热)

Warmup (预热) 是一种在深度学习模型训练初期，将学习率从一个**非常小的值 (甚至为 0)**逐步、平滑地增加到预设的初始学习率的策略。其核心目的是为了在模型训练最开始的、最不稳定的阶段，给予模型一个“热身”或“适应”的过程，防止因初始学习率过大而导致的训练不稳定甚至崩溃。它通常作为其他学习率调度器 (如余弦退火) 的前奏部分。

为什么初期需要预热:

在训练刚开始时，由于

1. **参数是完全随机的:**模型对输入数据一无所知，其内部的权重和偏置都是随机初始化的“胡乱猜测”。
2. **初始损失和梯度巨大:**因为参数是随机的，模型第一次进行前向传播时，其预测结果会与真实标签相差十万八千里。这会导致计算出的损失函数的值（Loss）通常非常大。
3. **梯度方向极其不确定:**巨大的损失会反向传播，计算出的梯度（Gradient）也会异常巨大且方向不稳定。这个梯度仅仅反映了从一个完全随机的点该如何“逃离”，这个方向对于全局最优解来说，参考价值很低。

而在真正进入到第一轮训练的时候，每个数据点对模型来说都是新的，模型会很快地进行数据分布修正，如果这时候学习率就很大，极有可能导致开始的时候就对该数据**“过拟合”**，后面要通过多轮训练才能拉回来，浪费时间。当训练了一段时间 (比如两轮、三轮) 后，模型已经对每个数据点看过几遍了，或者说对当前的 batch 而言有了一些正确的先验，较大的学习率就不那么容易会使模型学偏，所以可以适当调大学习率。这个过程就可以看做是 warmup。

那么为什么之后还要 decay 呢？当模型训到一定阶段后 (比如十个 epoch)，模型的分布就已经比较固定了，或者说能学到的新东西就比较少了。如果还沿用较大的学习率，就会破坏这种稳定性，用我们通常的话说，就是已经接近 loss 的 local optimal 了，为了靠近这个 point，我们就要慢慢来。

Warmup 与其他调度器的关系:

非常重要的一点是：Warmup 几乎总是与我们上面提到的其他学习率调度器 (如 CosineAnnealingLR, StepLR) 配合使用。

完整的训练过程的学习率策略被分为两个阶段:

- **第一阶段 (预热期):**学习率从 0 线性增长到 initial_lr。
- **第二阶段 (正常调度期):**学习率从 initial_lr 开始, 按照你选择的主调度器 (如余弦退火) 的规则进行衰减。

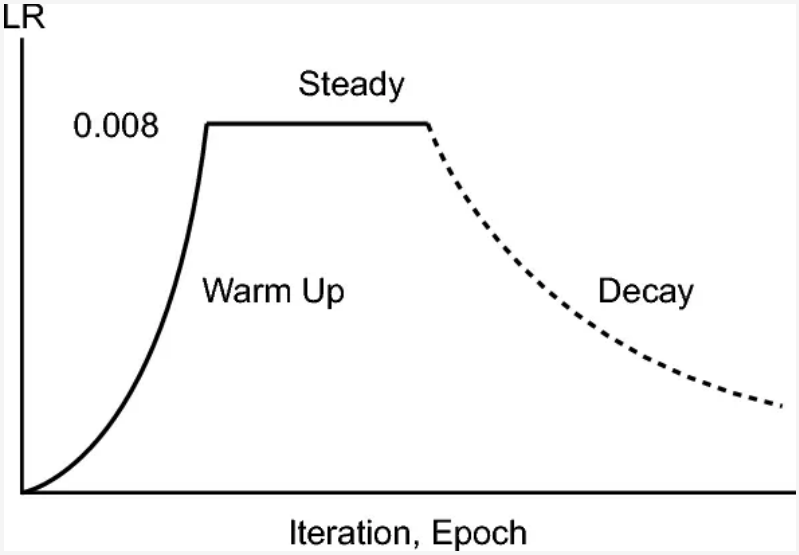


图 4.6: Warmup 与其他调度器配合使用示意图

> **薇言大义 4.1** 超参数的配置不仅在这篇教程中，在所有的 Training 里面都是很重要的，往后大部分科研或者工程上的工作可能实际上就是在调参。

4.5. 梯度裁剪

梯度裁剪是防止梯度爆炸的一种方法, 通过将梯度限制在一个范围内, 从而防止梯度爆炸。

公式:

$g_t = \min(g_t, \text{clip\_value})$

其中 $clip\_value$ 是裁剪阈值。

```
class GradientClip:
    def __init__(self, parameters,max_l2_norm,epslion=1e-6):
        self.parameters = parameters
        self.max_l2_norm = max_l2_norm
        self.epslion = epslion

    def __call__(self):
```

```
        grads = [p.grad for p in self.parameters if p.grad is not None] # 我们求 l2 范数
        ↪  是对所有元素求的，所以要先把所有元素给 flatten
        all_grads = torch.cat([grad.flatten() for grad in grads])
        grad_l2 = torch.norm(all_grads,2)
        if grad_l2 > self.max_l2_norm:
            clip_coeff = self.max_l2_norm / (grad_l2 + self.epslion)
            for grad in grads:
                grad.mul_(clip_coeff)
```

5

泛深度学习训练流程

一般的深度学习训练流程就包括:

1. **数据集加载 (DataLoader)**
   代表**输入环节**，在数据集进入模型之前，需要对数据集进行处理，比如归一化，分批，打乱等。
2. **模型保存 (Checkpoint)**
   代表**输出环节**，在模型训练过程中，需要定期保存模型参数，以便避免在训练过程中因为各种原因导致模型参数丢失。
3. **训练循环 (TrainLoop)**
   这是**最终的脚本**。它负责初始化所有组件，并按照正确的顺序组织整个训练流程，包括数据加载、模型计算、反向传播、参数更新、性能记录和模型保存等。

5.1. 数据集加载 (DataLoader)

参考资料 5.1

https://www.runoob.com/pytorch/pytorch-dataset-dataloader.html?utm_source=chatgpt.com

https://docs.pytorch.org/tutorials/beginner/basics/data_tutorial.html?utm_source=chatgpt.com

PyTorch 提供了两个数据原语：torch.utils.data.DataLoader 和 torch.utils.data.Dataset，它们允许你使用预加载的数据集和你自己的数据。Dataset 存储了样本及其相应的标签，DataLoader 在 Dataset 周围包装了一个可迭代对象，以便于访问样本。AI 为你提供母语级高精翻译

一个完整、符合 PyTorch 习惯的流程图

原始数据--› Dataset (读取样本 + 数据增强 transforms)--› DataLoader (batching, shuffle, num_workers, pin_memory...)--› 训练循环 for batch in DataLoader

5.1.1. 数据增强的两种方式

在线增强（Online Augmentation）

放在 Dataset 里，训练过程中每次取样本时随机增强

特点

- 每次训练加载一个样本时才进行随机增强（如随机裁剪、翻转），不需要提前生成大量数据副本
- 每个 epoch 都可能看新的增强结，提升模型泛化
- 训练期间使用 CPU 对每个样本执行 transform，DataLoader 多进程可**并行加速**

```
class MyDataset(Dataset):
    def __init__(self, data_paths, transform=None):
            self.paths = data_paths
            self.transform = transform  # 数据增强在这里
    def __getitem__(self, idx):
        img = read_image(self.paths[idx])
        if self.transform:
            img = self.transform(img)  # → 在 Dataset 做 augmentation
        return img
```

离线增强 (Offline Augmentation)

提前对原始数据增强好
特点

- Dataset 简单、**读取速度快**
- 没有训练时增强的 CPU 开销 → **DataLoader 更轻松、GPU 利用率更高**
- 非常适合：
  - 大规模数据
  - 训练频繁、增强重复多次
  - 多机训练、分布式训练（offline augmentation 避免重复增强）
  - 图像增强开销特别大的情况（如大图、复杂变换）

实际工程中通常会采用：

混合模式（推荐）

- 离线做一些“耗时重、固定不变”的增强
  - 图像标准化
  - 统一尺寸 resize
  - 去噪、直方图均衡（如果需要）
  - 硬件加速无法轻易做的操作
- 在线做一些“轻量、随机”的增强
  - RandomCrop
  - RandomFlip

- ColorJitter
- RandomRotation

5.1.2. 自定义 Dataset

torch.utils.data.Dataset 是一个抽象类，允许你从自己的数据源中创建数据集。我们需要继承该类并实现以下两个方法：

- ‘__len__(self)‘：返回数据集中的样本数量。
- ‘__getitem__(self, idx)‘：通过索引返回一个样本。(有了这个函数才可以使用索引，比如 dataset[6])

假设我们有一个简单的 CSV 文件或一些列表数据，我们可以通过继承 Dataset 类来创建自己的数据集。

```
import torch
from torch.utils.data import Dataset

# 自定义数据集类
class MyDataset(Dataset):
    def __init__(self, data_paths, transform=None):
        """ 初始化数据集，path 是数据集的路径，transform 代表数据增强的手段"""
        self.paths = data_paths
        self.transform = transform  # 数据增强在这里
        self.X_data = read(path) # 读取数据集
    self.Y_data = ...

    def __len__(self):
        """ 返回数据集的大小"""
        return len(self.X_data)

    def __getitem__(self, idx):
        """ 返回指定索引的数据"""
        x = torch.tensor(self.X_data[idx], dtype=torch.float32)  # 转换为 Tensor
        y = torch.tensor(self.Y_data[idx], dtype=torch.float32)
        return x, y
```

5.1.3. 使用 DataLoader 加载数据

DataLoader 是对 Dataset 的进一步封装，它主要是为了提升训练效率，常用的手段有 **batch 打包（batch_size）**、**打乱顺序（shuffle）**、**多进程加速（num_workers）**、**内存优化（pin_memory）**、**collate 规则（collate_fn）**、textbfdrop_last: 如果数据集中的样本数不能被 ‘batch_size‘ 整除，设置为 ‘True‘ 时，丢弃最后一个不完整的 batch。其中前两个最常用。

```
dataloader = DataLoader(
dataset,
batch_size=32,
shuffle=True,
num_workers=4,
pin_memory=True
```

```
)

# 打印加载的数据
for epoch in range(1):
    for batch_idx, (inputs, labels) in enumerate(dataloader):
        print(f'Batch {batch_idx + 1}:')
        print(f'Inputs: {inputs}')
        print(f'Labels: {labels}')
```

5.1.4. 如果数据集过大无法载入内存怎么办？

我们可以使用名为 mmap 的 Unix 系统调用，它能将磁盘文件映射到虚拟内存，并在访问该内存位置时惰性加载文件内容。这样就能” 假装” 整个数据集已载入内存。

NumPy 通过 np.memmap（或使用 np.load 时设置 mmap_mode='r' 标志，前提是数组最初通过 np.save 保存）实现该功能，它会返回一个类似 numpy 数组的对象，在访问时按需加载数据项。在训练期间从数据集（即 numpy 数组）采样时，务必以内存映射模式加载数据集（通过 np.memmap 或 np.load 的 mmap_mode='r' 标志，具体取决于数组的保存方式）。

5.1.5. 内存映射

数据并不是在磁盘上被直接访问的。CPU 的核心原则是：**它只能直接处理位于物理内存（RAM）中的数据。任何在磁盘上的数据，如果想被 CPU 计算、读取或修改，最终都必须被加载到物理内存中**。

内存映射的快体现在它**极大地优化了数据从磁盘进入物理内存，并最终呈现给用户进程的这个过程**。

传统 I/O (read()) 的数据流：两次 copy

假设程序需要从磁盘读取 1GB 的文件到一块内存中。

1. 第一次拷贝：从 **磁盘 -> 内核缓冲区**
   - 你的程序发起 ‘read()‘系统调用，导致程序从**用户态**切换到**内核态**。
   - 内核向磁盘控制器发出指令。
   - 磁盘控制器通过 **DMA（Direct Memory Access）**，将文件数据从磁盘直接拷贝到内核地址空间的一块缓冲区里。这个缓冲区通常被称为**页缓存（Page Cache）**。
   - **关键**：到目前为止，数据在内核里，你的用户程序还访问不到它。DMA 完成了这次拷贝，CPU 没有参与搬运数据，但它需要等待。
2. 第二次拷贝：从 **内核缓冲区 -> 用户缓冲区**
   - 现在数据已经在内核的页缓存里了，‘read()‘系统调用的下一步，就是把这些数据从**内核空间**拷贝到你程序在**用户空间**指定的缓冲区（就是你传给 ‘read()‘函数的那个 buffer 指针）。
   - **这是性能瓶颈所在**：这次拷贝是由**CPU**亲自执行的。对于 1GB 的数据，CPU 需要执行大量的 ‘mov‘指令，逐字节地把数据从一块内存搬到另

一块内存。这会消耗大量的 CPU 周期，并且会严重污染 CPU 的缓存（Cache）。

- 拷贝完成后，‘read()‘系统调用返回，程序从内核态切换回用户态。现在你的程序终于可以访问这 1GB 数据了。

总结传统 I/O：数据走了‘**磁盘 -> 内核内存 -> 用户内存**‘的路径。发生了两次拷贝，其中一次是由 CPU 完成的、非常昂贵的内存拷贝。

内存映射（Memory Mapping），

就是操作系统提供的一种机制，它允许一个进程将自己的**虚拟地址空间**中的一部分，直接与一个文件或者设备进行**“链接”或“映射”**。

核心工作原理

1. **建立映射（‘mmap‘系统调用）**

- 当一个进程调用 ‘mmap()‘系统调用请求建立内存映射时，内核执行以下操作：
- 在进程的虚拟地址空间中找到一段**连续的、未被使用**的区域。
- 为这段地址区域在进程的内存描述符（如 Linux 中的 ‘vm_area_struct‘）中创建一个新的**条目**，记录下映射的起始地址、长度、权限（读/写/执行）以及映射所关联的文件和偏移量。
- **关键点**：此时，内核**并不会**立即从磁盘加载任何文件数据到物理内存中。它仅仅是建立了逻辑上的关联，即配置了相关的内核数据结构。这是一个**非常轻量级**的操作。

2. **首次访问与缺页中断 (Page Fault)**

- 当进程首次尝试访问（读或写）这段被映射的虚拟内存地址时，会发生以下一系列事件：
- **MMU（内存管理单元）** 硬件在转换虚拟地址时，会查询进程的页表。由于数据尚未加载，对应的 PTE 会标记为“不存在”（Absent）。
- MMU 无法完成地址翻译，于是触发一个硬件异常，即**缺页中断**，并将控制权交回给操作系统内核。
- 内核的缺页中断处理程序被激活。它会检查导致中断的虚拟地址，并查询进程的内存描述符，确认该地址属于一个合法的内存映射区域。

3. **数据加载与页表更新**

- 确认是合法访问后，内核会执行真正的 I/O 操作：
- 在物理内存（RAM）中分配一个**空闲的物理页帧**。
- 根据 ‘mmap‘时记录的文件信息和偏移量，计算出需要从磁盘加载的数据块位置。
- 通过**磁盘驱动程序**，将对应的一个页（通常为 4KB）的数据从文件加载到刚刚分配的物理页帧中。

- 更新进程的**页表**，将触发中断的那个虚拟页的 PTE 指向这个新加载的物理页帧，并设置好相应的权限位（如“存在”、“可读”、“可写”等）。
- 中断处理结束，控制权返回给用户进程。

4. **透明的后续访问**

- 当中断处理程序返回后，导致中断的那条指令会被重新执行。这一次，**MMU**能够成功地通过**页表**将虚拟地址翻译为物理地址，进程便可以毫无知觉地访问到数据，仿佛它从一开始就在内存里一样。后续对同一页内其他地址的访问将直接通过 MMU 快速完成，不再触发中断。

总结内存映射：数据走了‘**磁盘 -> 内核内存 (页缓存)**‘的路径。然后，你的程序被“授权”直接访问这块内核内存。从始至终，数据只从磁盘到内存拷贝了**一次**，而且**CPU 没有参与任何批量数据的搬运工作**。这就是为什么它有时被称为“零拷贝”技术（这里的“零”指的是没有发生 CPU 参与的内核态到用户态的拷贝）。

所以实际上内存映射的本质，就是通过虚拟内存机制，让用户进程能够安全、直接地访问到本属于内核空间的那块内存（页缓存），从而消除了数据从内核空间到用户空间的冗余拷贝。

Keynote 5.1

内存映射这么牛逼，这么省资源，那么干脆所有的读取方式都改成内存映射不就好了？

也不是，内存映射省资源主要省在一次 *CPU* 对文件的拷贝上，对付大文件比较合适，但对付小文件，拷贝省的资源可能还不够**建立映射的开销**和**缺页中断的开销**这两部分大，所以也是一个 *trade-off* 的问题。另外这个也会消耗虚拟地址的空间。

5.2. 模型保存 (Checkpoint)

参考资料 5.2

https://blog.csdn.net/PolarisRisingWar/article/details/145529106?utm_source=chatgpt.com
https://juejin.cn/post/7435680251878015016?utm_source=chatgpt.com

试想一下，假如我们在某次深度学习的训练中大概要训练个 1000 个 epoch，得知训练预计完成要花费将近 3 天时间，我们好不容易训练了两天 23 个小时了，这时候突然停电了，服务器/电脑关机了，一切努力付诸东流，这下要气死了。

这个时候 checkpoint 的重要性就体现出来了。

checkpoint 可以看作深度学习训练里面的**“存档点”**，它保存了深度学习某一时刻的模型权重。如果我们定时每 100 个 epoch 就保存一次模型权重，那么即使训练到第 999 个 epoch 的时候就停电了，那么我们起码还可以从第 900 个 epoch 上重新开始续训，而不用从 0 开始。

5.2.1. 为什么 Checkpoint 至关重要？

Checkpoint 的重要性体现在以下几个关键场景：

- **训练中断与恢复**
  - 场景: 深度学习训练非常耗时，服务器可能会意外重启、程序可能崩溃、GPU 资源可能被抢占。
  - 作用: 如果没有 Checkpoint，你需要从头开始训练，浪费大量时间和计算资源。有了 Checkpoint，你可以从最近的保存点继续训练，就像从游戏存档点复活一样，几乎无缝衔接。
- **保存最佳模型**
  - 场景: 模型在训练过程中性能会波动。通常，我们关心的是在**验证集（Validation Set）**上表现最好的那个模型，而不是训练到最后时刻的模型（后者可能已经开始过拟合）。
  - 作用: 通过在每个 Epoch（或固定步数）结束后评估模型在验证集上的性能（如准确率、损失值），我们可以只保存那个性能指标最好的模型状态。这是实际部署时最常用的策略。
- **迁移学习与微调**
  - 场景: 你想在一个新的、数据量较小的任务上训练模型。从零开始训练一个大模型（如 ResNet, BERT）既困难又低效。
  - 作用: 我们可以加载一个在大型数据集（如 ImageNet, Wikipedia）上预训练好的模型 Checkpoint，然后在这个基础上针对我们的新任务进行**微调**。这极大地加速了收敛速度并提升了模型性能。
- **模型部署与推理**
  - 场景: 模型训练完成后，你需要将它部署到生产环境中提供服务（例如，用于图像识别、文本生成）。
  - 作用: 部署时加载的就是最终选定的最佳模型的 Checkpoint 文件，用它的权重来进行预测和推理。

5.2.2. 一个 Checkpoint 文件里通常包含什么？

一个完备的 Checkpoint 文件不仅仅是模型的权重，它通常包含一个字典或对象，其中含有：

- **模型参数或权重**：这是最核心的部分，即模型学习到的所有权重（weights）和偏置（biases）。在 PyTorch 中，这通常是 ‘model.state_dict()‘。
- **优化器状态**：这对于恢复训练至关重要。像 Adam、SGD with Momentum 这类优化器，它们内部会维护一些状态（如动量、学习率适应性调整的梯度平方均值等）。如果不保存优化器状态，恢复训练时优化器会重置，相当于重新开始，会影响训练的连续性。在 PyTorch 中，这是 ‘optimizer.state_dict()‘。
- **训练元数据**：

- **当前轮次 (Epoch)**：记录训练进行到了第几轮。
- **迭代步数**：记录总共迭代了多少步。
- **损失值**：当前或历史最佳的训练/验证损失。
- **其他指标**：如准确率、F1 分数等。

如果说训练之后的结果只需要用来**推理（inference）或部署**，而不再做“从某个中断点继续训练”的话，那么只保存模型的 **state_dict**（即参数权重）就够了。例如：

```
torch.save(model.state_dict(), "model_weights.pth")
```

然后在部署或加载时：

```
model = MyModel()
model.load_state_dict(torch.load("model_weights.pth"))
model.eval()
```

但如果担心停电导致前功尽弃，需要断电续训的话，就最好把模型权重（**model.state_dict()**）、优化器状态（**optimizer.state_dict()**）——包括动量、Adam 的 m/v、学习率状态等以及当前训练轮次或步数（epoch 或 global_step）都保存下来。

```
checkpoint = {
'epoch': epoch,
'model_state_dict': model.state_dict(),
'optimizer_state_dict': optimizer.state_dict(),
'loss': some_loss_value,
# 可能也保存 scheduler 和其他东西
}
torch.save(checkpoint, "checkpoint.pth")
```

加载恢复：

```
checkpoint = torch.load("checkpoint.pth")
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
start_epoch = checkpoint['epoch'] + 1
```

5.3. 脚本化训练

等到把所有的模块都整理完毕了，到最后我们进行实验时的变量其实主要就是超参数 + 不同数据集了，这个时候使用脚本化的训练能大大方便我们的实验。

因此我们的目标：**一个 train.py，改少量参数就能换模型 / 换数据 / 换实验**。

我的经验一般是一个 config 文件 (可以是**yaml 或者 json 格式**用来保存所有的超参数，方便修改，而不是在代码里面写死导致修改极不方便。然后用**argparse**配合直接在输入命令的时候就能控制变量：

比如

```python
def get_args():
    parser = argparse.ArgumentParser()

    # 基本配置
    parser.add_argument('--exp_name', type=str, default='baseline')
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--epochs', type=int, default=20)
    parser.add_argument('--batch_size', type=int, default=64)
    parser.add_argument('--lr', type=float, default=1e-3)
    parser.add_argument('--num_workers', type=int, default=4)

    # 数据 & 模型
    parser.add_argument('--data_root', type=str, default='./data')
    parser.add_argument('--num_classes', type=int, default=10)
    parser.add_argument('--model', type=str, default='mlp')  # 预留给后面多模型选择

    # 设备 & 保存
    parser.add_argument('--device', type=str, default='cuda')
    parser.add_argument('--output_dir', type=str, default='./runs')
    parser.add_argument('--resume', type=str, default='')  # 断点续训

    args = parser.parse_args()
    return args
```