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。