CS336 作业 1 (基础): 构建 Transformer LM

版本 1.0.6

CS336 员工

2025 年春季

1 作业概述

在此次作业中，您将从头开始构建训练标准 Transformer 语言模型 (LM) 所需的所有组件，并训练一些模型。

您将实现的

1. 字节对编码 (BPE) 分词器 (§2)
2. Transformer 语言模型 (LM) (§3)
3. 交叉熵损失函数和 AdamW 优化器 (§4)
4. 训练循环，支持序列化和加载模型及优化器状态 (§5)

您将运行的

1. 在 TinyStories 数据集上训练一个 BPE 分词器。
2. 在数据集上运行您训练好的分词器，将其转换为整数 ID 序列。
3. 在 TinyStories 数据集上训练一个 Transformer LM。
4. 使用训练好的 Transformer LM 生成样本并评估困惑度。
5. 在 OpenWebText 上训练模型，并将您达到的困惑度提交到排行榜。

你可以使用 我们希望你从头开始构建这些组件。特别是，除了以下内容外，你不能使用 torch(nn, torch(nn)).functional 或 torch_optim 中的任何定义：

- torch(nn_PARAMETER
- torch(nn) 中的容器类（例如，Module、ModuleList、Sequential 等）[1]
- torch.optim.Optimizer 基类

你可以使用任何其他 PyTorch 定义。如果你想使用某个函数或类但不确定是否允许，请随时在 Slack 上提问。如有疑问，请考虑使用它是否会损害本次作业的“从头开始”精神。

关于 AI 工具的声明 允许使用 ChatGPT 等语言模型来回答低级编程问题或关于语言模型的高级概念性问题，但禁止直接使用它来解决问题。

我们强烈建议您在完成作业时禁用IDE中的AI自动补全功能（例如，Cursor Tab、GitHub Copilot）（尽管非AI自动补全，例如函数名自动补全是完全可以的）。我们发现AI自动补全功能会大大增加深入理解内容的难度。

代码示例：所有作业代码以及此文档都可以在GitHub上找到：

github.com/stanford-cs336/assignment1-basics

请git clone该仓库。如果有任何更新，我们会通知您，以便您git pull以获取最新版本。

1. cs336 Basics/*：这是您编写代码的地方。请注意，这里没有代码——您可以从头开始做任何您想做的事情！
2. adapters.py: 您的代码必须具备一组功能。对于每一项功能（例如，缩放点积注意力），只需调用您的代码即可完成其实现（例如，runScaled.dot_productattention）。注意：您对 adapters.py 的修改不应包含任何实质性逻辑；这是胶水代码。
3. test_*.py: 这包含了您必须通过的所有测试（例如，testScaled.dot_productattention），这些测试将调用在 adapters.py 中定义的钩子。请勿编辑测试文件。

如何提交 您将把以下文件提交到 Gradescope：

- writeup.pdf：回答所有书面问题。请排版您的答案。
- code.zip：包含您编写的所有代码。

要提交到排行榜，请提交一个 PR 到：

github.com/stanford-cs336/assignment1-basics-leaderboard

有关详细的提交说明，请参阅排行榜存储库中的 README.md。

在哪里获取数据集 本次作业将使用两个预处理过的数据集：TinyStories [Eldan and Li, 2023] 和 OpenWebText [Gokaslan et al., 2019]。这两个数据集都是单个的大型纯文本文件。如果你是和班级一起做作业，可以在任何非头节点机器的 /data 目录下找到这些文件。

如果你是在家跟着做，可以使用 README.md 中的命令下载这些文件。

低资源/缩小规模提示：初始化

在整个课程的作业讲义中，我们将提供关于如何用更少或没有 GPU 资源来完成作业部分的建议。例如，我们有时会建议缩小数据集或模型的大小，或者解释如何在 MacOS 集成 GPU 或 CPU 上运行训练代码。你会发现这些“低资源提示”在一个蓝色的框里（像这个）。即使你是斯坦福大学的学生，可以使用课程机器，这些提示也可能帮助你更快地迭代和节省时间，所以我们建议你阅读它们！

低资源/降级技巧：在 Apple Silicon 或 CPU 上进行分配 1

使用员工解决方案代码，我们可以在配备 36 GB RAM 的 Apple M3 Max 芯片上训练一个 LM，在 Metal GPU (MPS) 上不到 5 分钟即可生成相当流畅的文本，使用 CPU 大约需要 30 分钟。如果这些词对您来说意义不大，请不要担心！您只需要知道，如果您有一台相对较新的笔记本电脑，并且您的实现是正确且高效的，那么您将能够训练一个小型 LM，生成具有不错流畅度的简单儿童故事。

稍后在分配中，我们将解释如果您使用的是 CPU 或 MPS，需要进行哪些更改。

2 字节对编码 (BPE) 分词器

在本次作业的第一部分，我们将训练并实现一个字节级字节对编码（BPE）分词器 [Sennrich et al., 2016, Wang et al., 2019]。具体来说，我们将任意（Unicode）字符串表示为字节序列，并在该字节序列上训练我们的 BPE 分词器。之后，我们将使用此分词器将文本（字符串）编码为词元（整数序列）以用于语言建模。 # 2.1 Unicode 标准
Unicode 是一个文本编码标准，它将字符映射到整数代码点。截至 Unicode 16.0（于 2024 年 9 月发布），该标准定义了 168 种书写系统中的 154,998 个字符。例如，字符 "s" 的代码点是 115（通常表示为 U+0073，其中 U+ 是约定前缀，0073 是 115 的十六进制表示），字符 "牛" 的代码点是 29275。在 Python 中，您可以使用 ord() 函数将单个 Unicode 字符转换为其整数表示。chr() 函数将整数 Unicode 代码点转换为具有相应字符的字符串。

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

问题 (unicode1)：理解 Unicode（1 分）

(a) chr(0) 返回哪个 Unicode 字符?

交付物：一句话回答。

(b) 该字符的字符串表示形式（**repr**()）与其打印表示形式有何不同?

交付物：一句话回答。

(c) 当此字符出现在文本中时会发生什么？在 Python 解释器中尝试以下操作可能会有所帮助，看看它是否符合您的预期：

```
>>>chr(0)
>>>print(chr(0))
>>> "this is a test"  $^+$  chr(0)  $^+$  "string"
>>> print("this is a test"  $^+$  chr(0)  $^+$  "string")
```

交付成果：一句话回复。

2.2 Unicode 编码

虽然 Unicode 标准定义了从字符到码点（整数）的映射，但直接在 Unicode 码点上训练分词器是不切实际的，因为词汇表会过大（约 150K 个条目）且稀疏（因为许多字符非常罕见）。相反，我们将使用一种 Unicode 编码，它将 Unicode 字符转换为字节序列。Unicode 标准本身定义了三种编码：UTF-8、UTF-16 和 UTF-32，其中 UTF-8 是互联网上的主要编码（占所有网页的 98% 以上）。

要将Unicode字符串编码为UTF-8，我们可以使用Python中的encode()函数。要访问Python bytes对象的底层字节值，我们可以对其进行迭代（例如，调用list()）。最后，我们可以使用decode()函数将UTF-8字节字符串解码为Unicode字符串。

```
>>> test_string = "hello! 你怎么是"
>>> utf8-encoded = test_string.encode("utf-8")
>>> print utf8-encoded)
b'hello! \xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf!"
>>> print(typeutf8-encoded))
<class 'bytes">
>>> # 获取编码字符串的字节值（0到255的整数）。
>>> list utf8-encoded)
[104, 101, 108, 108, 111, 33, 32, 227, 129, 147, 227, 130, 147, 227, 129, 171, 227, 129, 161
>>> # 一个字节不一定对应一个Unicode字符!
>>> print(len(test_string))
13
>>> print(len utf8-encoded))
23
>>> print utf8-encodeddecode("utf-8"))
hello! 你怎么是
```

通过将我们的 Unicode 码点转换为一系列字节（例如，通过 UTF-8 编码），我们实际上是将一系列码点（0 到 154,997 范围内的整数）转换为一系列字节值（0 到 255 范围内的整数）。长度为 256 的字节词汇表更容易处理。使用字节级分词器时，我们无需担心词汇表外词元，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。

问题 (unicode2)：Unicode 编码（3 分）

(a) 与 UTF-16 或 UTF-32 相比，有哪些理由更倾向于在 UTF-8 编码的字节上训练我们的分词器？比较这些编码对各种输入字符串的输出可能有所帮助。

交付成果：一到两句话的回答。

(b) 考虑以下（不正确的）函数，它旨在将 UTF-8 字节字符串解码为 Unicode 字符串。为什么这个函数不正确？提供一个产生错误结果的输入字节字符串示例。

```
defdecodeutf8_bytes_to_str WRONG(bytes): return".join([bytes([b]).decode("utf-8")for b in by
>>>decodeutf8 bytes to str WRONG("hello".encode("utf-8")) 'hello'
```

交付物：一个示例输入字节字符串，对于该字符串 `decodeutf8_bytes_to_str WRONG` 会产生不正确的输出，并附带一句解释该函数为何不正确的说明。

(c) 给出一个不能解码为任何 Unicode 字符的两个字节序列。

交付物：一个示例，附带一句解释。

2.3 子词分词

虽然字节级分词可以缓解词级分词器面临的词汇表外问题，但将文本分词为字节会导致输入序列非常长。这会减慢模型训练速度，因为

一个包含10个单词的句子在词级别语言模型中可能只有10个词元，但在字符级别模型中可能包含50个或更多词元（取决于单词的长度）。处理这些更长的序列需要模型在每个步骤中进行更多的计算。此外，基于字节序列的语言建模很困难，因为更长的输入序列会在数据中产生长期依赖。 子词分词器是词级别分词器和字节级别分词器之间的一种折衷。请注意，字节级别分词器的词汇表有256个条目（字节值为0到225）。子词分词器通过更大的词汇表大小来换取对输入字节序列更好的压缩。例如，如果字节序列 b'the' 在我们的原始文本训练数据中经常出现，那么在词汇表中为其分配一个条目可以将这个3词元的序列减少到一个词元。
我们如何选择这些子词单元添加到我们的词汇表中？Sennrich 等人 [2016] 提出使用字节对编码 (BPE；Gage, 1994)，这是一种压缩算法，它迭代地将最常见的字节对替换（“合并”）为单个、新的未使用索引。请注意，此算法将子词词元添加到我们的词汇表中，以最大化输入序列的压缩——如果一个词在我们的输入文本中出现的次数足够多，它将被表示为单个子词单元。

通过 BPE 构建的词汇表的子词分词器通常称为 BPE 分词器。在此次作业中，我们将实现一个字节级 BPE 分词器，其中词汇表项是字节或字节的合并序列，这使我们在处理词汇表外单词和管理输入序列长度方面都能获得最佳效果。构建 BPE 分词器词汇表的过程称为“训练”BPE 分词器。

2.4 BPE 分词器训练

BPE 分词器的训练过程包含三个主要步骤。

词汇表初始化 分词器词汇表是将字节串词元映射到整数 ID 的一对一映射。由于我们训练的是字节级 BPE 分词器，因此我们的初始词汇表就是所有字节的集合。由于有 256 个可能的字节值，我们的初始词汇表大小为 256。

预分词 一旦有了词汇表，原则上，你可以计算文本中字节相邻出现的频率，并从最频繁的字节对开始合并。然而，这在计算上相当昂贵，因为每次合并都需要对语料库进行一次完整的遍历。此外，直接在语料库中合并字节可能会导致词元仅在标点符号上有所不同（例如，dog! vs. dog.）。这些词元将获得完全不同的词元 ID，尽管它们可能具有很高的语义相似性（因为它们仅在标点符号上有所不同）。

为了避免这种情况，我们对语料库进行预分词。你可以将其视为语料库上的粗粒度分词，这有助于我们计算字符对出现的频率。例如，“text”这个词可能是一个预分词，出现了10次。在这种情况下，当我们计算字符“t”和“e”相邻出现的频率时，我们会发现“text”这个词中的“t”和“e”是相邻的，我们可以将它们的计数增加10，而不是遍历整个语料库。由于我们正在训练一个字节级别的BPE模型，每个预分词都表示为UTF-8字节序列。

Sennrich 等人 [2016] 的原始BPE实现通过简单地按空格分割（即 s.split("\")）来进行预分词。相比之下，我们将使用一个基于正则表达式的预分词器（GPT-2；Radford et al., 2019 使用的），来自 github.com/openai/tiktoken/pull/234/files：

```
>>>PAT  $\equiv$  r""""?:[sdmt]ll|ve|re)| p{L}+|? p{N}+|? [^s\\p{L}\\p{N}]  $+\mid \backslash
```

为了更好地了解其行为，使用此预分词器（pre-tokenizer）交互式地分割一些文本可能会很有用：

> ## 需要 `regex` 包
>
> import regex as re re.findall(PAT, "some text that i'll pre-tokenize")

['some', 'text', 'that', 'i', "'ll", 'pre', '-', 'tokenizer']

然而，在代码中使用时，您应该使用 re.findall 来避免在构建预词元（pre-tokens）到其计数的映射时存储预分词的词语。

计算 BPE 合并 现在我们已经将输入文本转换为预分词，并将每个预分词表示为 UTF-8 字节序列，我们可以计算 BPE 合并（即训练 BPE 分词器）。总的来说，BPE 算法会迭代地计算每对字节的出现次数，并识别出频率最高的一对（“A”，“B”）。然后，将这对最频繁的字节（“A”，“B”）的每次出现进行合并，即替换为一个新的词元“AB”。这个新的合并词元被添加到我们的词汇表中；因此，BPE 训练后的最终词汇表的大小是初始词汇表的大小（在本例中为 256），加上训练过程中执行的 BPE 合并操作的数量。为了在 BPE 训练期间提高效率，我们不考虑跨越预分词边界的字节对。在计算合并时，通过优先选择字典序更大的字节对来确定性地打破字节对频率的平局。例如，如果字节对（“A”，“B”），（“A”，“C”），（“B”，“ZZ”），和（“BA”，“A”）的频率最高，我们d merge ("BA", "A"):

```
>>> max([("A", "B"), ("A", "C"), ("B", "ZZ"), ("BA", "A")])('BA', 'A')
```

特殊标记 (Special tokens) 有时，一些字符串（例如 <|endoftext|>）被用来编码元数据（例如文档之间的边界）。在编码文本时，通常希望将某些字符串视为“特殊标记”，这些标记永远不会被拆分成多个

标记（即，将始终保留为单个标记）。例如，序列结束字符串 <|endoftext|> 应始终保留为单个标记（即单个整数 ID），以便我们知道何时停止从语言模型生成。这些特殊标记必须添加到词汇表中，以便它们具有相应的固定标记 ID。

Sennrich et al. [2016] 的算法 1 包含一个低效的 BPE 分词器训练实现（基本上遵循我们上面概述的步骤）。作为第一个练习，实现并测试此函数以检验您的理解可能很有用。

示例 (bpe_example): BPE 训练示例

这是 Sennrich et al. [2016] 中的一个风格化示例。考虑一个包含以下文本的语料库：

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

并且词汇表有一个特殊词元 $ <\text{endoftext}> $。

词汇表 我们用特殊词元 $ <\text{endoftext}| $ 和 256 个字节值来初始化我们的词汇表。

预分词 为了简单起见，并专注于合并过程，在本示例中我们假设预分词只是按空格分割。当我们进行预分词和计数时，我们会得到频率表。

{low: 5, lower: 2, widest: 3, newest: 6}

将其表示为 dict[tuple[bytes], int] 会很方便，例如 $\{(1, 0, w) : 5 \ldots\}$。请注意，即使是单个字节在 Python 中也是一个字节对象。Python 中没有字节类型来表示单个字节，就像 Python 中没有字符类型来表示单个字符一样。

我们首先查看每对连续的字节，并对它们出现的单词的频率求和 $\{10 : 7, 0w : 7, we : 8, er : 2, wi : 3, id : 3, de : 3, es : 9, st : 9, ne : 6, ew : 6\}$。对 ('es') 和 ('st') 并列，所以我们取字典序更大的对 ('st')。然后我们将预词元合并，直到我们得到 $\{(1, o, w) : 5, (1, o, w, e, r) : 2, (w, i, d, e, st) : 3, (n, e, w, e, st) : 6\}$。

在第二轮中，我们看到 (e, st) 是最常见的对（计数为 9），我们将合并为 $\{(1, o, w) : 5, (1, o, w, e, r) : 2, (w, i, d, \text{est}) : 3, (n, e, w, \text{est}) : 6\}$。继续这个过程，我们最终得到的合并序列将是 ['s t', 'e st', 'o w', 'l ow', 'w est', 'n e', 'ne west', 'w i', 'wi d', 'wid est', 'low e', 'lower r']。

如果我们进行 6 次合并，我们将得到 ['s t', 'e st', 'o w', 'l ow', 'w est', 'n e']，我们的词汇表元素将是 $[< \text{endoftext}| >]$，[...256 BYTE CHARS]，st，est，ow，low，west，ne]。

使用这个词汇表和合并集，单词 newest 将被分词为 [ne, west]。

2.5 实验 BPE 分词器训练

让我们在 TinyStories 数据集上训练一个字节级 BPE 分词器。查找/下载数据集的说明可以在第 1 节中找到。开始之前，我们建议您查看 TinyStories 数据集，以了解数据的内容。

并行化预分词 您会发现一个主要的瓶颈是预分词步骤。您可以通过使用内置库 multiprocessing 来并行化代码，从而加速预分词。具体来说，我们建议在预分词的并行实现中，将语料库分块，同时确保您的块边界出现在特殊标记的开头。您可以直接使用以下链接中的入门代码来获取块边界，然后使用这些边界将工作分发到您的进程中：

https://github.com/stanford-cs336/assignment1-basics/blob/main/cs336Basics/pretokenization_example.py

这种分块将始终有效，因为我们从不想跨文档边界进行合并。就本次作业而言，您可以始终这样拆分。不要担心接收到一个不包含 $\langle|\text{endoftext}|\rangle$ 的非常大的语料库的边缘情况。

在预分词前移除特殊标记 在使用正则表达式模式（使用 re.finditer）运行预分词之前，您应该从语料库（或并行实现中的分块）中剥离所有特殊标记。确保您在特殊标记处进行拆分，以免在它们分隔的文本之间发生合并。例如，如果您有一个像 [Doc 1]<|endoftext|>[Doc 2] 这样的语料库（或分块），您应该在特殊标记 <|endoftext|> 处进行拆分，并分别对 [Doc 1] 和 [Doc 2] 进行预分词，以免在文档边界处发生合并。这可以通过使用 re.split 来完成，其中 "|\"].join(special_tokens) 作为分隔符（小心使用 re.escape，因为 | 可能会出现在特殊标记中）。测试 test_train_bpe_special_tokens 将对此进行测试。

优化合并步骤

上面示例中 BPE 训练的朴素实现速度很慢，因为每次合并都需要遍历所有字节对来识别最频繁的字节对。然而，每次合并后唯一改变的字节对计数是那些与合并后的字节对重叠的计数。因此，通过索引所有字节对的计数并增量更新这些计数，而不是显式地遍历每个字节对来计算字节对频率，可以提高 BPE 训练速度。通过这种缓存过程可以显著加快速度，尽管我们注意到 BPE 训练的合并部分在 Python 中是无法并行化的。

低资源/缩小规模技巧：分析

您应该使用 cProfile 或 scalene 等分析工具来识别实现中的瓶颈，并专注于优化它们。

低资源/缩小规模技巧：“缩小规模”

与其在 TinyStories 数据集上训练分词器，我们建议你先在数据的一个小子集上进行训练：“调试数据集”。例如，你可以改用 TinyStories 验证集来训练分词器，该验证集包含 22K 个文档，而不是 2.12M 个。这说明了一种通用的策略，即尽可能缩小规模以加快开发速度：例如，使用更小的数据集、更小的模型尺寸等。选择调试数据集的大小或超参数配置需要仔细考虑：你希望调试集足够大，能够反映完整配置中的瓶颈（以便你进行的优化能够泛化），但又不能太大以至于运行时间过长。

问题（train_bpe）：BPE 分词器训练（15 分）

交付成果：编写一个函数，该函数接收输入文本文件的路径，并训练一个（字节级）BPE 分词器。你的 BPE 训练函数应至少处理以下输入参数：

input_path: str 训练 BPE 分词器数据的文本文件路径。

vocab_size: int 一个正整数，用于定义最终词汇表的总大小（包括初始字节词汇表、通过合并产生的词汇项以及任

special_tokens: list[str] 要添加到词汇表中的字符串列表。这些特殊标记不会以其他方式影响 BPE 训练。

您的 BPE 训练函数应返回生成的词汇表和合并：

vocabulary: dict[int, bytes] 分词器词汇表，一个从 int（词汇表中的词元 ID）到 bytes（词元字节）的映射

merges: list[tuple[bytes, bytes]] 训练产生的 BPE 合并列表。列表中的每个项都是一个字节元组（<词元1>

To test your BPE training function against our provided tests, you will first need to implement the test adapter at [adapters.run_train_bpe]. Then, run uv run pytest tests/test_train_bpe.py. Your implementation should be able to pass all tests. Optionally (this could be a large time-investment), you can implement the key parts of your training method using some systems language, for instance C++ (consider cppy for this) or Rust (using PyO3). If you do this, be aware of which operations require copying vs reading directly from Python memory, and make sure to leave build instructions, or make sure it builds using only pyproject.toml. Also note that the GPT-2 regex is not well-supported in most regex engines and will be too slow in most that do. We have verified that Oniguruma is reasonably fast and supports negative lookahead, but the regex package in Python is, if anything, even faster.

Problem (train_bpe_tinystories): BPE Training on TinyStories (2 points)

(a) 在 TinyStories 数据集上训练一个字节级 BPE 分词器，最大词汇表大小为 10,000。确保将 TinyStories $<\text{endoftext}|$ 特殊词元添加到词汇表中。将生成的词汇表和合并序列化到磁盘以供进一步检查。训练花费了多少小时和多少内存？词汇表中 OOV 词元最长的是什么？这有意义吗？

资源要求：≤ 30 分钟（无 GPU），≤ 30 GB RAM

提示：通过预分词过程中的多进程处理，您应该能够在 2 分钟内完成 BPE 训练，并利用以下两个事实：

(a) $<\text{endoftext}|$ 词元在数据文件中分隔文档。 (b) $<\text{endoftext}|>$ 词元在应用 BPE 合并之前被作为特殊情况处理。

交付成果：一到两句话的回答。

(b) 分析您的代码。分词器训练过程的哪个部分花费的时间最多？

交付成果：一到两句话的回答。

接下来，我们将尝试在 OpenWebText 数据集上训练一个字节级 BPE 分词器。与之前一样，我们建议您查看数据集以更好地了解其内容。

问题 (train_bpe_expts_owt): 在 OpenWebText 上训练 BPE（2 分）

(a) 在 OpenWebText 数据集上训练一个字节级 BPE 分词器，使用最大词汇表大小为 32,000。将生成的词汇表和合并保存到磁盘以供进一步检查。词汇表中存在的最长词元是什么？这是否合理？

资源需求：≤ 12 小时（无 GPU），$\leq 100\mathrm{GB}$ RAM

交付物：一到两句话的回答。

(b) 比较和对比在 TinyStories 和 OpenWebText 上训练得到的 ist。

交付物：一到两句话的回答。

2.6 BPE 分词器：编码和解码

在本次作业的前一部分，我们实现了一个函数，用于在输入文本上训练 BPE 分词器，以获得分词器词汇表和 BPE 合并列表。现在，我们将实现一个 BPE 分词器，该分词器加载提供的词汇表和合并列表，并使用它们将文本编码/解码为词元 ID。

2.6.1 编码文本

BPE 编码文本的过程与我们训练 BPE 词汇表的方式类似。有几个主要步骤。

步骤 1：预分词。我们首先对序列进行预分词，并将每个预分词表示为 UTF-8 字节序列，就像我们在 BPE 训练中所做的那样。我们将把每个预分词内的这些字节合并到词汇表元素中，独立处理每个预分词（预分词边界之间没有合并）。

步骤 2：应用合并。然后，我们采用 BPE 训练过程中创建的词汇表元素合并序列，并按创建顺序将其应用于我们的预分词。

示例 (bpe_encoding)：BPE 编码示例

例如，假设我们的输入字符串是“the cat ate”，我们的词汇表是 ParseError: KaTeX parse error: Double superscript at position 9: \{0: b' ¸ 1: b'a', 2: b...，我们学习到的合并是 [(b't', b'h'), (b' ', b'c'), (b' ', a'), (b'th', b'e'), (b'a', b't')]。首先，我们的预分词器会将此字符串拆分为 ['the', 'cat', 'ate']。然后，我们将查看每个预分词并应用 BPE 合并。

第一个预词元 'the' 最初表示为 $[b't', b'h', b'e']$。查看我们的合并列表，我们确定第一个适用的合并是 $(b't', b'h')$，并用它将预词元转换为 $[b'th', b'e']$。然后，我们回到合并列表，确定下一个适用的合并是 $(b'th', b'e')$，它将预词元转换为 $[b'the']$。最后，回顾合并列表，我们发现没有更多的合并适用于该字符串（因为整个预词元已合并为单个词元），因此我们完成了 BPE 合并的应用。相应的整数序列是 [9]。

重复此过程以处理剩余的预分词，我们看到预分词 'cat' 在应用 BPE 合并后表示为 $[b'c', b'a', b't']$，它变成整数序列 [7, 1, 5]。最终的预分词 'ate' 在应用 BPE 合并后表示为 $[b'at', b'e']$，它变成整数序列 [10, 3]。因此，编码输入字符串的最终结果是 [9, 7, 1, 5, 10, 3]。

特殊标记。您的分词器应能够正确处理用户定义的特殊标记（在构建分词器时提供）。

内存考量。假设我们要对一个无法放入内存的大文本文件进行分词。为了有效地对这个大文件（或任何其他数据流）进行分词，我们需要将其分解成可管理的数据块，并依次处理每个数据块，这样内存复杂度就是常数，而不是文本大小的线性函数。在这样做的时候，我们需要确保一个词元不会跨越数据块边界，否则我们将得到与在内存中对整个序列进行分词的朴素方法不同的分词结果。

2.6.2 解码文本

要将整数词元 ID 的序列解码回原始文本，我们可以简单地在词汇表中查找每个 ID 对应的条目（字节序列），将它们连接在一起，然后将字节解码为 Unicode 字符串。请注意，输入 ID 不保证能映射到有效的 Unicode 字符串（因为用户可以输入任何整数 ID 序列）。如果输入词元 ID 未能生成有效的 Unicode 字符串，则应将格式错误的字节替换为官方 Unicode 替换字符 U+FFFD。[3] `bytesdecode` 的 `errors` 参数控制如何处理 Unicode 解码错误，使用 `errors='replace'` 将自动用替换标记替换格式错误的字节。

问题（分词器）：实现分词器（15 分）

交付物：实现一个 Tokenizer 类，该类接收一个词汇表和一组合并，将文本编码为整数 ID，并将整数 ID 解码为文本。您的分词器还应支持用户提供的特殊标记（如果它们尚不存在，则将它们附加到词汇表中）。我们建议使用以下接口：

```
def __init__(self, vocab, merges, special_tokens=None)
```

从给定的词汇表、合并列表和（可选的）特殊标记列表构建一个分词器。此函数应接受

```
以下参数：
vocab: dict[int, bytes]
merges: list[tuple[bytes, bytes]]
special_tokens: list[str] | None = None
```

```
def from_files(cls, vocab_filepath, merges_filepath, special_tokens=None) 类方法，用于从序列化的

vocab_filepath: str
merges_filepath: str
special_tokens: list[str] | None = None


def encode(self, text: str) -> list[int] 将输入文本编码为词元ID序列。
```

问题（tokenizer_experiments）：分词器实验（4分）

```
def encode_iterable(self, iterable: Iterator[str]) -> Iterator[int] 给定一个字符串的可迭代对象

defdecode(self, ids: list[int]) ->strDecodeasequenceoftokenIDsinto text.
```

要测试您的分词器是否符合我们提供的测试，您首先需要实现 [adapters.get_tokenizer] 中的测试适配器。然后，运行 uv run pytest tests/test_tokenizer.py。您的实现应该能够通过所有测试。

2.7 实验

(a) 从 TinyStories 和 OpenWebText 中分别采样 10 个文档。使用您之前训练的 TinyStories 和 OpenWebText 分词器（词汇表大小分别为 10K 和 32K），将这些采样文档编码为整数 ID。每个分词器的压缩率（bytes/token）是多少?

交付物：一到两句话的回答。

(b) 如果您使用 TinyStories 分词器对 OpenWebText 样本进行分词，会发生什么？比较压缩率和/或定性描述发生了什么。

交付物：一到两句话的回答。

(c) 估算您的分词器的吞吐量（例如，每秒字节数）。对 Pile 数据集（825GB 文本）进行分词需要多长时间?

交付成果：一到两句话的回答。

(d) 使用您的 TinyStories 和 OpenWebText 分词器，将各自的训练和开发数据集编码为整数词元 ID 序列。我们稍后将使用它来训练我们的语言模型。我们建议将词元 ID 序列化为 uint16 数据类型的 NumPy 数组。为什么 uint16 是一个合适的选择?

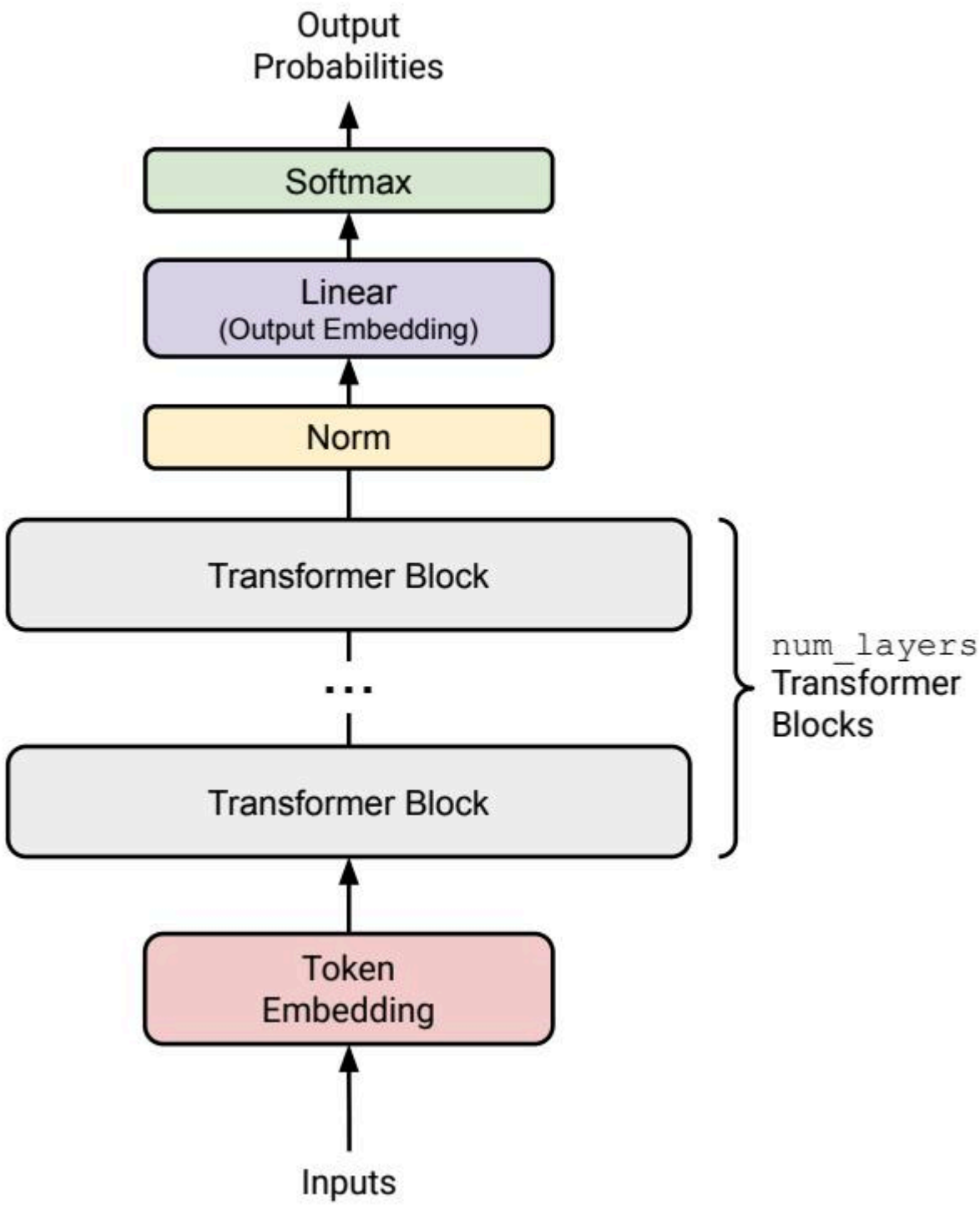

图 1：我们的 Transformer 语言模型概述。

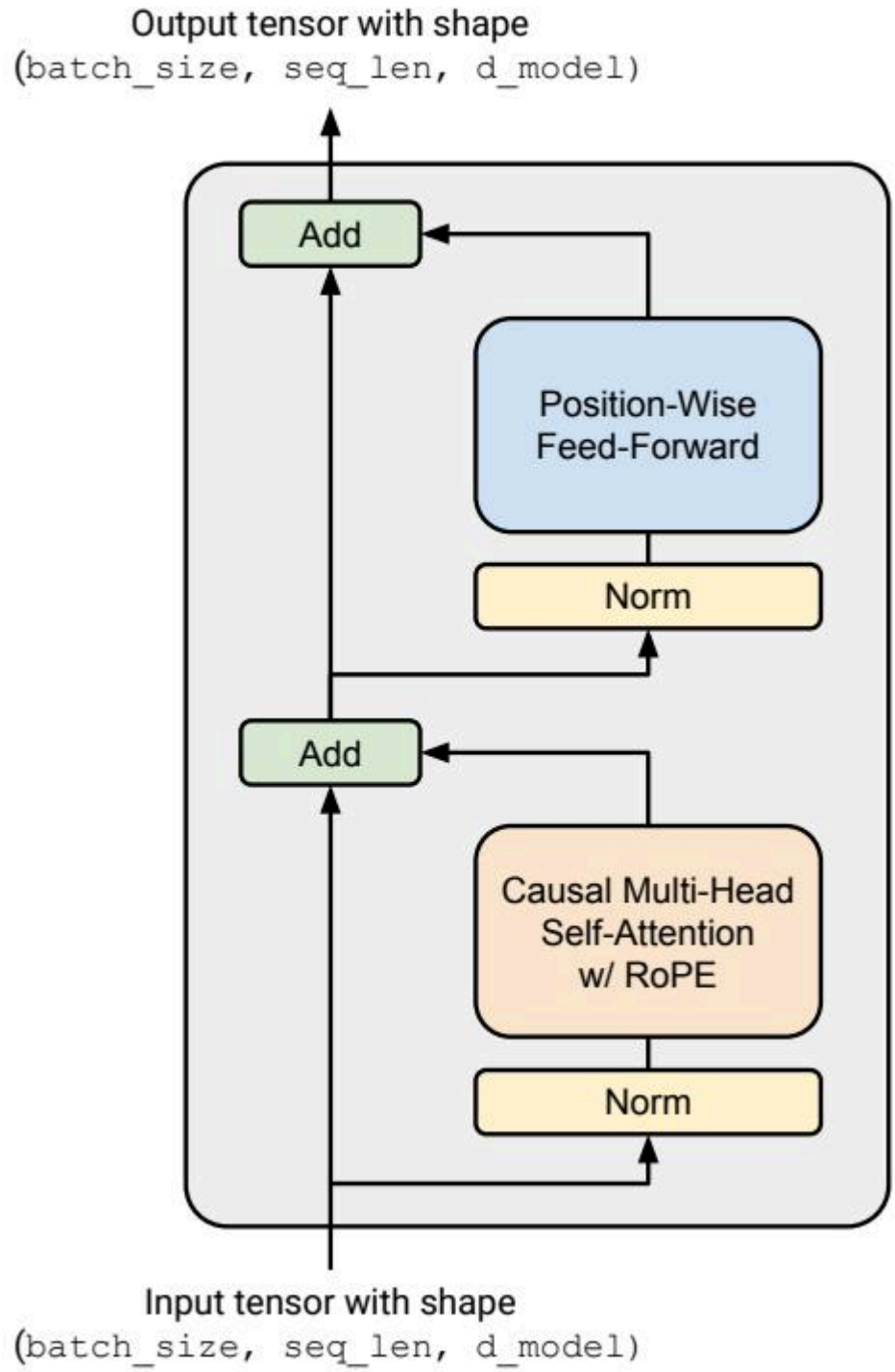

图 2：一个 Pre-norm Transformer 块。

3 Transformer 语言模型架构

语言模型接收一个整数词元 ID 的批处理序列（即，形状为 (batch_size, sequence_length) 的 torch.Tensor）作为输入，并返回一个（批处理的）词汇表上的归一化概率分布（即，形状为 (batch_size, sequence_length, vocab_size) 的 PyTorch Tensor），其中预测的分布是针对每个输入词元的下一个词。在训练语言模型时，我们使用这些下一个词的预测来计算实际下一个词与预测下一个词之间的交叉熵损失。在推理过程中从语言模型生成文本时，我们取最后一个时间步（即序列中的最后一个词元）的预测下一个词分布，以生成序列中的下一个词元（例如，通过选取概率最高的词元、从分布中采样等），将生成的词元添加到输入序列中，然后重复此过程。

在此次作业的这一部分，您将从头开始构建这个 Transformer 语言模型。我们将从模型的整体描述开始，然后逐步详细介绍各个组件。

3.1 Transformer LM

给定一个词元 ID 序列，Transformer 语言模型使用输入嵌入将词元 ID 转换为密集向量，将嵌入的词元通过 num_layers 个 Transformer 块，然后应用一个学习到的线性投影（“输出嵌入”或“LM 头”）来生成预测的下一个词元 logit。有关示意图，请参见图 1。

3.1.1 词元嵌入

在第一步中，Transformer 将（批处理后的）词元 ID 序列嵌入到包含词元身份信息的向量序列中（图 1 中的红色块）。

更具体地说，给定一个词元 ID 序列，Transformer 语言模型使用词元嵌入层来生成一个向量序列。每个嵌入层接收一个形状为 (batch_size, sequence_length) 的整数张量，并生成一个形状为 (batch_size, sequence_length, d_model) 的向量序列。 # 3.1.2 Pre-norm Transformer 块 嵌入后，激活值通过几个结构相同的神经网络层进行处理。标准的仅解码器 Transformer 语言模型由 num_layers 个相同的层（通常称为 Transformer“块”）组成。每个 Transformer 块接收形状为 (batch_size, sequence_length, d_model) 的输入，并返回形状为 (batch_size, sequence_length, d_model) 的输出。每个块通过自注意力机制聚合序列中的信息，并通过前馈层进行非线性变换。 # 3.2 输出归一化和嵌入
经过 num_layers 个 Transformer 块后，我们将获取最终的激活值，并将其转换为词汇表上的分布。

我们将实现“预归一化” Transformer 块（在 §3.5 中详述），该块还需要在最后一个 Transformer 块之后使用层归一化（下文详述），以确保其输出被正确缩放。

在此归一化之后，我们将使用标准的学习线性变换将 Transformer 块的输出转换为预测的下一个词元 logit（参见，例如，Radford et al. [2018] 方程 2）。

3.3 备注：批处理、Einsum 和高效计算

在整个 Transformer 中，我们将对许多类批处理输入执行相同的计算。以下是一些示例：

- 批次中的元素：我们对每个批次元素应用相同的 Transformer 前向操作。
- 序列长度：“逐位置”操作，如 RMSNorm 和前馈网络，对序列的每个位置执行相同的操作。
- 注意力头：注意力操作在“多头”注意力操作中跨注意力头进行批处理。

以一种充分利用GPU且易于阅读和理解的方式执行此类操作非常有用。许多PyTorch操作可以接受张量开头的额外“类似批处理”的维度，并有效地跨这些维度重复/广播操作。

例如，假设我们正在执行一个逐位置的、批处理的操作。我们有一个形状为 (batch_size, sequence_length, d_model) 的“数据张量” D，我们希望与形状为 (d_model, d_model) 的矩阵 A 进

行批处理向量-矩阵乘法。在这种情况下，$D \otimes A$ 将执行批处理矩阵乘法，这是PyTorch中的一个高效的原始操作，其中 (batch_size, sequence_length) 维度被批处理。

因此，最好假设您的函数可能会获得额外的类似批处理的维度，并将这些维度保留在PyTorch形状的开头。为了以这种方式组织张量以便进行批处理，可能需要使用多个view、reshape和transpose步骤来对其进行塑形。这可能有点麻烦，而且很难看清代码的作用以及张量的形状。

一个更符合人体工程学的选择是在torch.einsum中使用einsum表示法，或者更确切地说，使用与框架无关的库，如einops或einx。两个关键操作是einsum，它可以对输入张量的任意维度进行张量收缩，以及rearrange，它可以重新排序、连接和分割任意

维度。事实证明，机器学习中的几乎所有操作都是维度调整和张量收缩的某种组合，偶尔会加上（通常是逐点）非线性函数。这意味着使用 einsum 表示法可以使您的许多代码更具可读性和灵活性。

我们强烈建议在课程中使用 einsum 表示法。以前没有接触过 einsum 表示法的学生应该使用 einops（文档在此处），而已经熟悉 einops 的学生应该学习更通用的 einx（在此处）。这两个包都已安装在我们提供的环境中。

这里我们给出了一些 einsum 表示法用法的示例。这些是对 einops 文档的补充，您应该先阅读它们。

示例（einstein_example1）：使用 einops.einsum 进行批量矩阵乘法。

```
import torch
from einops import rearrange, einsum
## 基本实现
Y = D @ A.T
# 很难说清输入和输出的形状以及它们的含义。
# D 和 A 可以有什么形状，其中是否有任何形状会产生意外行为？
## Einsum 是自文档化且健壮的
# D A -> Y
Y = einsum(D,A,"batch sequence d_in, d_out d_in -> batch sequence d_out")
## 或者，一个批处理版本，其中 D 可以有任何前导维度，但 A 是受限的。
Y = einsum(D,A,"... d_in, d_out d_in -> ... d_out")
```

示例 (einstein_example2): 使用 einops.rearrange 进行广播操作
我们有一批图像，对于每张图像，我们想根据某个缩放因子生成 10 个变暗的版本： images = torch.rand(64, 128, 128, 3) # (batch, height, width, channel) dim_by = torch.linspace(start=0.0, end=1.0, steps=10) 重塑和相乘 dim_value = rearrange(dim_by, "dim_value -> 1 dim_value 1 1 1") images_rearr = rearrange(images, "b height width channel -> b 1 height width channel") dimmed_images = images_rearr * dim_value

或者一次性完成：

dimmed_images = einsum(images, dim_by, "batch height width channel, dim_value -> batch dim_value height width channel")

示例 (einstein_example3): 使用 einops.rearrange 进行像素混合
假设我们有一个由形状为 (batch, height, width, channel) 的张量表示的图像批次，我们想要对图像的所有像素执行线性变换，但此变换应独立于每个通道进行。我们的线性变换由一个形状为 (height × width, height × width) 的矩阵 B 表示。 channels_last = torch.randn(64, 32, 32, 3) # (batch, height, width, channel) B = torch.randn(32*32, 32*32) #将图像张量重新排列以跨所有像素进行混合 channels_last_flat = channels_last.view(-1, channels_last.size(1) * channels_last.size(2), channels_last.size(3)) channels_first_flat ≡ channels_last_flat.transpose(1, 2) channels_first_flat_transformed ≡ channels_first_flat @ B.T channels_last_flat_transformed ≡ channels_first_flat_transformed.transpose(1, 2) channels_last_transformed ≡ channels_last_flat_transformed.view(*channels_last.shape)改用 einops： height = width = 32

Rearrange 取代了笨拙的 torch view + transpose

channels_first ≡ rearrange(channels_last, "batch height width channel -> batch channel (height width)") channels_firsttransformed ≡ einsum(channels_first, B, "batch channel pixel_in, pixel_out pixel_in -> batch channel pixel_out") channels_lasttransformed ≡ rearrange(channels_first_transformed, "batch channel (height width) -> batch height width channel", height=height, width=width) 或者，如果你想更进一步：使用 einx.dot（einx 相当于 einops.einsum）一次性完成 height = width = 32 channels_lasttransformed ≡ einx.dot("batch row_in col_in channel, (row_out col_out) (row_in col_in)" "-> batch row_out col_out channel", channels_last, B, col_in=width, col_out=width

这里的第一个实现可以通过在前后放置注释来指示来改进

which the input and output shapes are, but this is clunky and susceptible to bugs. With einsum notation, documentation is implementation!

Einsum notation can handle arbitrary input batching dimensions, but also has the key benefit of being self-documenting. It's much clearer what the relevant shapes of your input and output tensors are in code that uses einsum notation. For the remaining tensors, you can consider using Tensor type hints, for instance using the jaxtyping library (not specific to Jax).

We will talk more about the performance implications of using eaxsum notation in assignment 2, but for now know that they're almost always better than the alternative!

3.3.1 Mathematical Notation and Memory Ordering

Many machine learning papers use row vectors in their notation, which result in representations that mesh well with the row-major memory ordering used by default in

NumPy and PyTorch. With row vectors, a linear transformation looks like

$$y = xW^{\top}, \tag{1}$$

对于行主序 $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ 和行向量 $x \in \mathbb{R}^{1 \times d_{\text{in}}}$。

在线性代数中，使用列向量通常更常见，此时线性变换看起来像

$$y = Wx, \tag{2}$$

给定行主序 $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ 和列向量 $x \in \mathbb{R}^{d_{\text{in}}}$。在本作业中，我们将使用列向量进行数学表示，因为这样通常更容易理解数学。您应该记住，如果您想使用纯矩阵乘法表示法，您将不得不应用矩阵的行向量约定，因为 PyTorch 使用行主序内存。如果您使用 `einsum` 进行矩阵运算，这将不是问题。

3.4 基本构建块：线性（Linear）和嵌入（Embedding）模块

3.4.1 参数初始化

有效训练神经网络通常需要仔细初始化模型参数——糟糕的初始化可能导致梯度消失或爆炸等不良行为。预归一化 Transformer 对初始化异常鲁棒，但它们仍然可能对训练速度和收敛产生重大影响。由于本次作业已经很长了，我们将把细节留到作业 3，而是提供一些近似初始化方法，这些方法在大多数情况下都应该效果良好。目前，请使用：

- 线性权重：$\mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{d_{\text{in}} + d_{\text{out}}}\right)$，截断在 $[-3\sigma, 3\sigma]$。
- 嵌入：$\mathcal{N}\left(\mu = 0, \sigma^2 = 1\right)$，截断在 $[-3, 3]$。 RMSNorm：1

您应该使用 torch(nn.init.trunc_normal_) 来初始化截断正态权重。

3.4.2 线性模块

线性层是 Transformer 和神经网络的基本组成部分。首先，您将实现自己的 Linear 类，该类继承自 torch.nnModule 并执行线性变换：

$$y = Wx. \tag{3}$$

请注意，我们不包含偏置项，这与大多数现代 LLM 保持一致。

问题 (linear)：实现线性模块 (1 分)

交付成果：实现一个继承自 torch(nn.Module 并执行线性变换的 Linear 类。您的实现应遵循 PyTorch 内置 nn.Linear 模块的接口，但没有偏置参数或参数。我们推荐以下接口：

def **init**(self, in_features, out_features, device=None, dtype=None) 构建一个线性变换模块。此函数应接受以下参数：

in_features: int 输入的最终维度

out_features: int 输出的最终维度

device: torch_device | None = None 用于存储参数的设备

dtype: torch.dtype | None = None 参数的数据类型

def forward(self, x: torch.Tensor) -> torch.Tensor 对输入应用线性变换。

请确保：

- 继承 nn-module
- 调用父类构造函数
- 为了内存排序原因，将参数构建并存储为 W（而不是 $W^\top$），并将其放入 nn_PARAMETER
- 当然，不要使用 nn.Linear 或 nn.Functional-linear

对于初始化，请使用上述设置，并结合 torch(nn.init.trunc_normal_ 来初始化权重。

要测试你的 Linear 模块，请在 [adapters.runlinear] 实现测试适配器。该适配器应将给定的权重加载到你的 Linear 模块中。你可以为此目的使用 Module.load_state_dict。然后，运行 uv run pytest -k testlinear。

3.4.3 Embedding Module

如上所述，Transformer 的第一层是一个嵌入层，它将整数词元 ID 映射到一个维度为 d_model 的向量空间。我们将实现一个自定义的 Embedding 类，该类继承自 torch.nnModule（因此你不应使用 nn.Embedding）。forward 方法应通过索引形状为 (vocab_size, d_model) 的嵌入矩阵来选择每个词元 ID 的嵌入向量，其中使用形状为 (batch_size, sequence_length) 的词元 ID 的 torch.LongTensor。

问题（嵌入）：实现嵌入模块（1 分）

交付物：实现继承自 torch(nnModule 并执行嵌入查找的 Embedding 类。你的实现应遵循 PyTorch 内置 nn.Embedding 模块的接口。我们推荐以下接口：

def **init**(self, num_embeddings, embedding_dim, device=None, dtype=None) 构建一个嵌入模块。此函数应接受以下参数：

num_embedding：int 词汇表的大小

embedding_dim: int embedding向量的维度，即 d_{model}

device: torch_device | None = None 参数存储的设备

dtype: torch.dtype | None = None 参数的数据类型

def forward(self, token_ids: torch.Tensor) -> torch.Tensor 查找给定词元ID的embedding向量。

确保：

- 继承 nn-module
- 调用父类构造函数
- 将embedding矩阵初始化为 nn_PARAMETER
- 存储embedding矩阵，其中 d_model 是最终维度
- 当然，不要使用 nn.Embedding 或 nnfunctional_embedding

再次，使用上面的设置进行初始化，并使用 torch.nn.init.trunc_normal_ 来初始化权重。

为了测试您的实现，请在 [adapters.run_embedding] 中实现测试适配器。然后，运行 uv run pytest -k test_embedding。

3.5 Pre-norm Transformer 块

每个 Transformer 块包含两个子层：一个多头自注意力机制和一个逐位置前馈网络（Vaswani et al., 2017, section 3.1）。

在原始的Transformer论文中，模型在两个子层周围都使用了残差连接，然后进行层归一化。这种架构通常被称为“后归一化”（post-norm）Transformer，因为层归一化应用于子层输出。然而，大量研究发现，将层归一化从每个子层的输出移到每个子层的输入（在最后一个Transformer块之后增加一个层归一化）可以提高Transformer的训练稳定性 [Nguyen and Salazar, 2019, Xiong et al., 2020]—有关这种“前归一化”（pre-norm）Transformer块的视觉表示，请参见图 2。然后，通过残差连接将每个Transformer块子层的输出添加到子层输入（Vaswani et al., 2017, section 5.4）。前归一化的直观理解是，存在一个没有归一化的干净的“残差流”从输入嵌入流向Transformer的最终输出，这被认为可以改善梯度
flow。这种 Pre-norm Transformer 块现在是当今语言模型（例如 GPT-3、LLaMA、PaLM 等）的标准，因此我们将实现此变体。我们将逐步介绍 Pre-norm Transformer 块的每个组件，并按顺序实现它们。

3.5.1 均方根层归一化

Vaswani 等人 [2017] 的原始 Transformer 实现使用层归一化 [Ba et al., 2016] 来归一化激活。遵循 Touvron 等人 [2023]，我们将使用均方根层归一化（RMSNorm；Zhang and Senrich, 2019, 方程 4）进行层归一化。给定一个激活向量 $a \in \mathbb{R}^{d_{\text{model}}}$，RMSNorm 将按如下方式重新缩放每个激活 a_i：

$$\mathrm{RMSNorm}\left(a_i\right)=\frac{a_i}{\mathrm{RMS}(a)} g_i, \tag{4}$$

其中 $\mathrm{RMS}(a)=\sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} a_i^2+\varepsilon}$ 。这里，g_i 是一个可学习的"增益"参数（总共有 d_{model} 个这样的参数），而 ε 是一个超参数，通常固定为 1e-5。

您应该将输入上转换为 torch.float32，以防止输入平方时发生溢出。总体而言，您的 forward 方法应如下所示：

```
in dtype  $=$  x.dtype
 $\mathbf{x} = \mathbf{x}$  to(torch.float32)
```

您的代码在此处执行 RMSNorm

```
... result  $=$  ...
```

以原始 dtype 返回结果

return result.to(indtype)

问题 (rmsnorm)：均方根层归一化（1 分）

交付成果：将 RMSNorm 实现为 torch.nnModule。我们推荐以下接口：

```
def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None) 构建 RMSNorm 模块
```

d_model: int 模型的隐藏层维度

eps: float = 1e-5 数值稳定性使用的 epsilon 值

device: torch_device | None = None 用于存储参数的设备

dtype: torch.dtype | None = None 参数的数据类型

def forward(self, x: torch.Tensor) -> torch.Tensor 处理形状为

(batch_size, sequence_length, d_model) 的输入张量并返回相同形状的张量。

注意：请记住，在执行归一化（以及稍后向下转换为原始 dtype）之前，将输入上转换为 torch.float32，如上所述。

要测试您的实现，请在 [adapters.run_rmsnorm] 中实现测试适配器。然后，运行 uv run pytest -k test_rmsnorm。

3.5.2 位置前馈网络

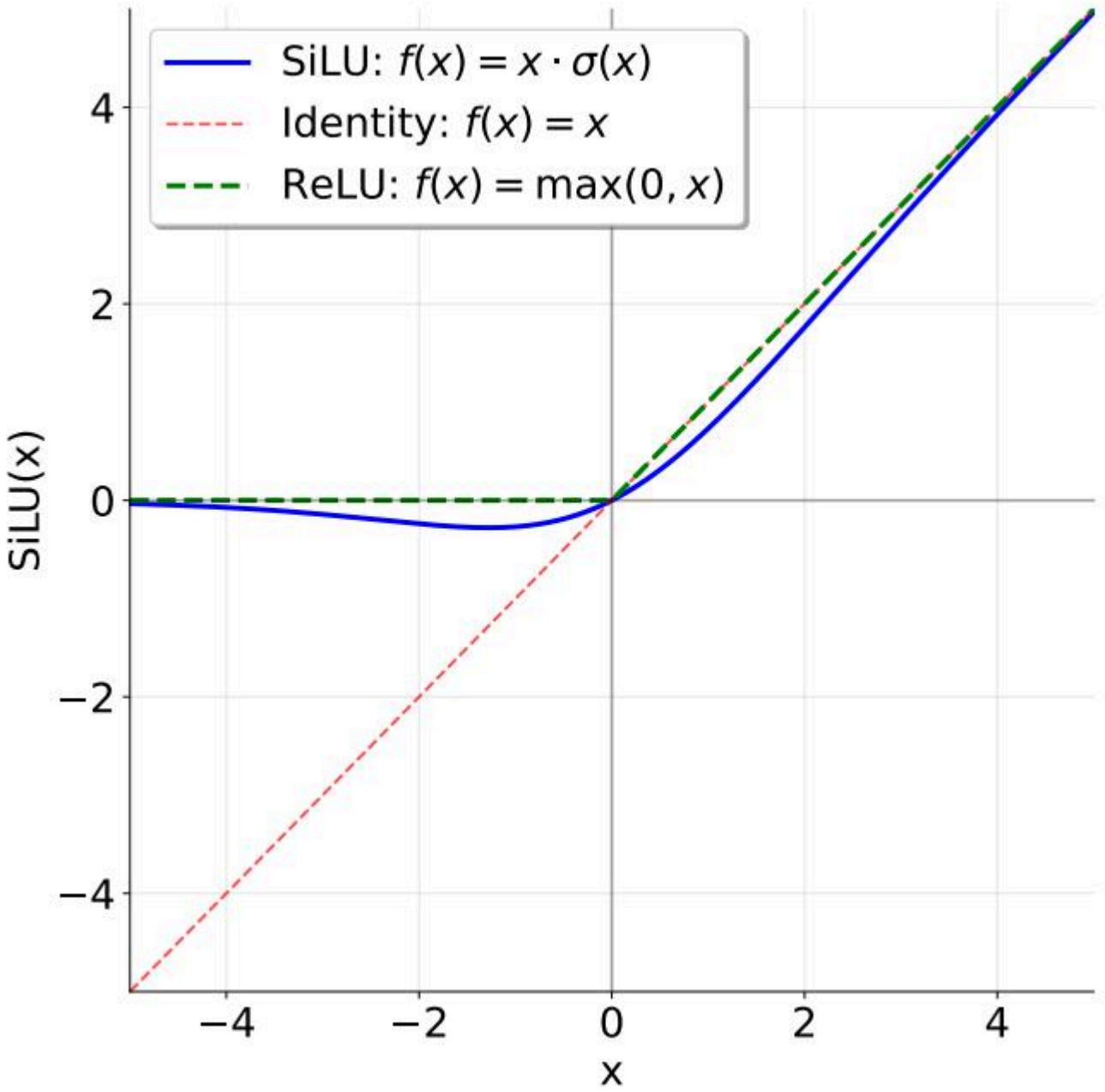

图 3：比较 SiLU（又名 Swish）和 ReLU 激活函数。

在原始的 Transformer 论文（Vaswani 等人 [2017] 的第 3.3 节）中，Transformer 前馈网络由两个线性变换组成，中间有一个 ReLU 激活函数 $(\mathrm{ReLU}(x) = \max(0, x))$。内部前馈层的维度通常是输入维度的 $4\mathrm{x}$。

然而，与原始设计相比，现代语言模型倾向于包含两个主要变化：它们使用另一种激活函数并采用门控机制。具体来说，我们将实现 LLM（如 Llama 3 [Grattafori et al., 2024] 和 Qwen 2.5 [Yang et al., 2024]）中采用的“SwiGLU”激活函数，它将 SiLU（通常称为 Swish）激活与称为门控线性单元（GLU）的门控机制相结合。我们还将省略有时在线性层中使用的偏置项，遵循自 PaLM [Chowdhery et al., 2022] 和 LLaMA [Touvron et al., 2023] 以来大多数现代 LLM 的做法。

SiLU 或 Swish 激活函数 [Hendrycks and Gimpel, 2016, Elfwing et al., 2017] 定义如下：

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{5}$$

如图 3 所示，SiLU 激活函数与 ReLU 激活函数类似，但在零点处是平滑的。

门控线性单元（GLU）最初由 Dauphin 等人 [2017] 定义，作为通过 sigmoid 函数的线性变换与另一个线性变换的逐元素乘积：

$$\text{GLU}(x, W_1, W_2) = \sigma\left(W_1 x\right) \odot W_2 x, \tag{6}$$

其中 $\odot$ 表示逐元素乘法。门控线性单元被建议“通过为梯度提供线性路径同时保留非线性能力来减少深度架构的梯度消失问题”。

将 SiLU/Swish 和 GLU 结合起来，我们得到 SwiGLU，我们将将其用于我们的前馈网络：

$$\text{FFN}(x) = \text{SwiGLU}(x, W_1, W_2, W_3) = W_2(\text{SiLU}(W_1 x) \odot W_3 x), \tag{7}$$

其中 $x \in \mathbb{R}^{d_{\text{model}}}$ ， $W_1, W_3 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ ， $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ ，并且通常情况下，$d_{\text{ff}} = \frac{8}{3} d_{\text{model}}$ 。

Shazeer [2020] 首先提出了将 SiLU/Swish 激活与 GLUs 结合，并通过实验表明 SwiGLU 在语言建模任务上优于 ReLU 和 SiLU（无门控）等基线模型。稍后，你将在作业中比较 SwiGLU 和 SiLU。尽管我们已经提到了一些关于这些组件的启发式论证（并且论文提供了更多支持性证据），但保持经验性的视角仍然很重要：Shazeer 论文中的一句名言是：

我们不解释为什么这些架构似乎有效；我们将它们的成功，以及其他一切，都归因于神圣的仁慈。

问题 (positionwise_feedforward): 实现位置前馈网络（2 分）

Deliverable: 实现 SwiGLU 前馈网络，该网络由 SiLU 激活函数和 GLU 组成。

Note: 在这种特殊情况下，为了数值稳定性，您可以随意在实现中使用 torch.sigmoid。

在您的实现中，您应该将 d_{ff} 设置为大约 $\frac{8}{3} \times d_{\text{model}}$，同时确保内部前馈层的维度是 64 的倍数，以充分利用您的硬件。要使用我们提供的测试来测试您的实现，您需要实现 [adapters.run_swiglu] 中的测试适配器。然后，运行 uv run pytest -k test_swiglu 来测试您的实现。

3.5.3 相对位置嵌入

为了将位置信息注入模型，我们将实现旋转位置嵌入 [Su et al., 2021]，通常称为 RoPE。对于给定查询词元 $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ 在词元位置 i，我们将应用一个成对旋转矩阵 R^i，得到 $q'^{(i)} = R^i q^{(i)} = R^i W_q x^{(i)}$。这里，$R^i$ 将通过角度 $\theta_{i,k} = \frac{i}{\Theta(2k-2)/d}$ 将嵌入元素的对 $q^{(i)}_{2k-1:2k}$ 作为二维向量旋转，其中 $k \in \{1, \ldots, d/2\}$ 且 Θ 为某个常数。因此，我们可以将 R^i 视为一个大小为 $d \times d$ 的块对角矩阵，其块为 R^i_k，其中 $k \in \{1, \ldots, d/2\}$，

$$R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}. \tag{8}$$

因此，我们得到完整的旋转矩阵

$$ R ^ {i} = \left[\begin{array}{c c c c c} R _ {1} ^ {i} & 0 & 0 & \dots & 0 \ 0 & R _ {2} ^ {i} & 0 & \dots & 0 \ 0 & 0 & R _ {3} ^ {i} & \dots & 0 \ \vdots & \vdots & \vdots & \ddots & \vdots \ 0 & 0 & 0 & \dots & R _ {d / 2} ^ {i} \end{array} \right], \tag {9} $$

其中 0s 表示 2×2 的零矩阵。虽然可以构造完整的 $d \times d$ 矩阵，但一个好的解决方案应该利用该矩阵的性质来更有效地实现变换。由于我们只关心给定序列中词元之间的相对旋转，因此我们可以跨层和不同的批次重用我们为 $\cos(\theta_{i,k})$ 和 $\sin(\theta_{i,k})$ 计算出的值。如果您想对其进行优化，可以使用一个由所有层引用的 RoPE 模块，并且它可以在初始化时使用 self.register_buffer(persistent=False) 创建一个 2d 预计算缓冲区（包含 sin 和 cos 值），而不是使用 nn_PARAMETER（因为我们不想学习这些固定的余弦和正弦值）。然后，我们对 $k^{(j)}$ 执行与对 $q^{(i)}$ 执行的完全相同的旋转过程，通过相应的 R^j 进行旋转。请注意，此层没有可学习的参数。

Deliverable: 实现一个 RotaryPositionalEmbedding 类，将 RoPE 应用于输入张量。推荐使用以下接口：

Problem (rope): 实现 RoPE (2 分)

```
def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)
```

构造 RoPE 模块并根据需要创建缓冲区。

theta: float RoPE 的 Θ 值

d_k: int 查询和键向量的维度

max_seq_len: int 将输入的序列的最大长度

device: torch_device | None = None 用于存储缓冲区的设备

Problem (softmax): 实现 softmax (1 分)

```
def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor
```

处理形状为 (..., seq_len, d_k) 的输入张量，并返回相同形状的张量。

请注意，您应该能够处理具有任意数量批处理维度的 x。您应该假设 token 位置是一个形状为 (..., seq_len) 的张量，指定 x 沿序列维度的 token 位置。

您应该使用 token 位置沿序列维度对（可能预先计算的）cos 和 sin 张量进行切片。

要测试您的实现，请完成 [adapters.run_ripe] 并确保它通过 uv run.pytest -k test_ripe。

3.5.4 缩放点积注意力

现在我们将实现 Vaswani et al. [2017]（第 3.2.1 节）中描述的缩放点积注意力。作为初步步骤，注意力操作的定义将使用 softmax，这是一种将未归一化的分数向量转换为归一化分布的操作：

$$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum_{j=1}^{n} \exp(v_j)}. \tag{10}$$

请注意，对于较大的值，$\exp(v_i)$ 可能会变成 inf（然后，$\text{inf} / \text{inf} = \text{NaN}$）。我们可以通过注意到 softmax 操作对于向所有输入添加任何常数 c 都是不变的来避免这种情况。我们可以利用此属性来提高数值稳定性——通常，我们会从 o_i 的所有元素中减去 o_i 的最大条目，使新的最大条目为 0。您现在将实现 softmax，使用此技巧来提高数值稳定性。

交付物：编写一个函数来对张量应用 softmax 操作。您的函数应接受两个参数：一个张量和一个维度 i，并将 softmax 应用于输入张量的第 i 维。输出张量应与输入张量具有相同的形状，但其第 i 维现在将具有归一化的概率分布。使用从第 i 维的所有元素中减去第 i 维的最大值这一技巧来避免数值稳定性问题。

To test your implementation, complete [adapters.runsoftmax] and make sure it passes uv run pytest -k testSoftmax_MATCHes_pytorch.

We can now define the Attention operation mathematically as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q^\top K}{\sqrt{d_k}}\right) V \tag{11}$$

where $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, and $V \in \mathbb{R}^{m \times d_v}$. Here, Q, K and V are all inputs to this operation—note that these are not the learnable parameters. If you're wondering why this isn't $QK^\top$, see 3.3.1.

掩码：有时将注意力操作的输出进行掩码会很方便。掩码的形状应为 $M \in \{\text{True}, \text{False}\}^{n \times m}$，此布尔矩阵的每一行 i 表示查询 i 应关注哪些键。按照惯例（并且有点令人困惑的是），位置 (i, j) 的值为 True 表示查询 i 关注键 j，值为 False 表示查询 i 不关注键 j。换句话说，“信息流”在值为 True 的 (i, j) 对处流动。例如，考虑一个条目为 [[True, True, False]] 的 1×3 掩码矩阵。单个查询向量仅关注前两个键。

在计算上，使用掩码比在子序列上计算注意力要高效得多，我们可以通过取预 softmax 值 $\left(\frac{Q^\top K}{\sqrt{d_k}}\right)$ 并将掩码矩阵中为 False 的任何条目加上 $-\infty$ 来实现这一点。

问题 (scaled.dot_productattention): 实现缩放点积注意力 (5 分)

交付物：实现缩放点积注意力函数。你的实现应该能够处理形状为 (batch_size, ..., seq_len, d_k) 的键和查询，以及形状为 (batch_size, ..., seq_len, d_v) 的值，其中 ... 代表任何数量的其他类似批次的维度

（如果提供）。实现应该返回形状为 (batch_size, ..., d_v) 的输出。有关类似批次的维度的讨论，请参阅第 3.3 节。

你的实现还应该支持一个可选的用户提供的布尔掩码，形状为 (seq_len, seq_len)。掩码值为 True 的位置的注意力概率应该加起来等于 1，而掩码值为 False 的位置的注意力概率应该为零。要针对我们提供的测试来测试你的实现，你需要实现 [adapters.runScaled.dot_productattention] 中的测试适配器。

uv run pytest -k testScaled.dot_productattention 测试您在三阶输入张量上的实现，而 uv run pytest -k test_4dScaled.dot_productattention 测试您在四阶输入张量上的实现。

3.5.5 因果多头自注意力

我们将按照 Vaswani et al. [2017] 的 3.2.2 节中的描述来实现多头自注意力。回想一下，在数学上，应用多头注意力的操作定义如下：

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right) \tag{12}$$

$$\text{for} \quad \text{head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{13}$$

其中 Q_i、 K_i、 V_i 是切片号为 $i \in \{1, \ldots, h\}$ 的 Q、 K 和 V 的大小分别为 d_k 或 d_v 的嵌入维度。Attention 是 §3.5.4 中定义的缩放点积注意力操作。由此，我们可以形成多头自注意力操作：

$$\text{MultiHeadSelfAttention}(x) = W_O \,\text{MultiHead}\left(W_Q x, W_K x, W_V x\right) \tag{14}$$

Here, the learnable parameters are $W_Q \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, and $W_O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$。由于在多头注意力操作中对 Q、K 和 V 进行了切片，我们可以认为 W_Q、W_K 和 W_V 在输出维度上是为每个头分开的。当您完成此操作后，您应该在总共三次矩阵乘法中计算键、值和查询投影。

因果掩码。你的实现应该阻止模型关注序列中的未来词元。换句话说，如果模型接收到一个词元序列 $t_1, \ldots, t_n$，并且我们想要计算前缀 $t_1, \ldots, t_i$（其中 $i < n$）的下一个词预测，模型不应该能够访问（关注）位置 $t_{i+1}, \ldots, t_n$ 处的词元表示，因为在推理过程中生成文本时，它将无法访问这些词元（并且这些未来的词元会泄露真实下一个词的身份信息，从而使语言建模预训练目标变得微不足道）。对于一个输入词元序列 $t_1, \ldots, t_n$，我们可以通过运行 n 次多头自注意力（针对序列中的 n 个唯一前缀）来粗略地阻止访问未来的词元。相反，我们将使用因果注意力掩码，它允许词元 i 关注序列中的所有位置 $j \leq i$。你可以使用 torch.triu 或广播的 index 比较来构建此掩码，并且您应该利用 §3.5.4 中提供的缩放点积注意力实现已支持注意力掩码这一事实。

应用 RoPE。RoPE 应应用于查询和键向量，但不应用于值向量。此外，头维度应被视为批次维度，因为在多头注意力中，注意力是为每个头独立应用的。这意味着应将完全相同的 RoPE 旋转应用于每个头的查询和键向量。

问题 (multihead_self_attention)：实现因果多头自注意力（5 分）

交付成果：实现因果多头自注意力作为 torch.nnModule。您的实现应接受（至少）以下参数：

d_model：int Transformer 块输入的维度。

num_heads：int 在多头自注意力中使用的头数。

Following Vaswani et al. [2017], set $d_k = d_v = d_{\text{model}}/h$. To test your implementation against our provided tests, implement the test adapter at [adapters.run-multihead_self_attention]. Then, run uv run pytest -k test-multihead_self_attention to test your implementation.

3.6 The Full Transformer LM

Let's begin by assembling the Transformer block (it will be helpful to refer back to Figure 2). A Transformer block contains two 'sublayers', one for the multihead self attention, and another for the feed-forward network. In each sublayer, we first perform RMSNorm, then the main operation (MHA/FF), finally adding in the residual connection.

To be concrete, the first half (the first 'sub-layer') of the Transformer block should be implementing the following set of updates to produce an output y from an input x,

$$y = x + \text{MultiHeadSelfAttention}(\text{RMSNorm}(x)). \tag{15}$$

问题 (transformer_block): 实现 Transformer 块 (3 分)

实现 §3.5 中描述并如图 2 所示的 Pre-norm Transformer 块。您的 Transformer 块应接受（至少）以下参数。

d_model: int Transformer 块输入的维度。

num_heads: int 多头自注意力中使用的头数。

d_ff: int 位置前馈内层的维度。

为了测试您的实现，请实现适配器 [adapters.run_transformer_block]。然后运行 uv run pytest -k test_transformer_block 来测试您的实现。

交付物：通过所提供测试的 Transformer 块代码。

现在我们将这些块组合在一起，遵循图 1 中的高层图。遵循我们关于嵌入的描述（第 3.1.1 节），将此输入到 num_layers 个 Transformer 块中，然后将其传递到三个输出层以获得词汇表上的分布。

问题 (transformer_lm): 实现 Transformer LM (3 分)

是时候将所有内容整合在一起了！实现 §3.1 中描述并如图 1 所示的 Transformer 语言模型。至少，您的实现应接受 Transformer 块的所有上述构造参数，以及这些附加参数：

vocab_size: int 词汇表的大小，用于确定词元嵌入矩阵的维度。

context_length: int 最大上下文长度，用于确定位置嵌入矩阵的维度。

num_layers: int 要使用的 Transformer 块的数量。

要针对我们提供的测试来测试您的实现，您首先需要实现 [adapters.run_transformer_lm] 中的测试适配器。然后，运行 uv run pytest -k test_transformer_lm 来测试您的实现。

交付物：一个通过上述测试的 Transformer LM 模块。

资源核算。能够了解 Transformer 的各个部分如何消耗计算和内存非常有用。我们将逐步进行一些基本的"FLOPs 核算"。Transformer 中绝大多数 FLOPs 是矩阵乘法，因此我们的核心方法很简单：

1. 写下 Transformer 前向传播中的所有矩阵乘法。
2. 将每个矩阵乘法转换为所需的 FLOPs。

对于第二步，以下事实将很有用：

规则：给定 $A \in \mathbb{R}^{m \times n}$ 和 $B \in \mathbb{R}^{n \times p}$，矩阵-矩阵乘积 AB 需要 2mnp FLOPs。

要理解这一点，请注意 $(AB)[i,j] = A[i,:] \cdot B(:,j]$，并且该点积需要 n 次加法和 n 次乘法（2n FLOPs）。然后，由于矩阵-矩阵乘积 AB 有 $m \times p$ 个条目，FLOPs 的总数为 $(2n)(mp) = 2mnp$ 。

现在，在进行下一个问题之前，最好逐一检查 Transformer 块和 Transformer LM 的每个组件，并列出所有的矩阵乘法及其相关的 FLOPs 成本。

问题 (transformer_accounting): Transformer LM 资源核算 (5 分)

(a) 考虑 GPT-2 XL，其配置如下：

```
vocab_size : 50,257
context_length : 1,024
num_layers : 48
d_model : 1,600
```

num_heads : 25

d_ff:6,400

假设我们使用此配置构建模型。我们的模型将有多少可训练参数？假设每个参数都使用单精度浮点数表示，仅加载此模型需要多少内存？

交付成果：一到两句话的回答。

(b) 确定完成我们 GPT-2 XL 模型前向传播所需的矩阵乘法。这些矩阵乘法总共需要多少 FLOPs？假设我们的输入序列有 context_length 个词元。

交付成果：矩阵乘法列表（附说明）以及所需的总 FLOPs。 (c) 根据您上面的分析，模型的哪些部分需要最多的 FLOPs？ 交付成果：一到两句话的回答。 (d) 使用 GPT-2 small（12 层，768 d_model，12 头）、GPT-2 medium（24 层，1024 d_model，16 头）和 GPT-2 large（36 层，1280 d_model，20 头）重复您的分析。随着模型尺寸的增加，Transformer LM 的哪些部分占总 FLOPs 的比例会增加或减少？ 交付成果：对于每个模型，提供模型组件及其相关 FLOPs 的明细（作为前向传播所需总 FLOPs 的比例）。此外，提供一到两句话的描述，说明改变模型尺寸如何改变每个组件的比例 FLOPs。 (e) 将 GPT-2 XL 的上下文长度增加到 16,384。一次前向传播的总 FLOPs 如何变化？模型组件的 FLOPs 的相对贡献如何变化？

交付物：一到两句话的回答。

4 训练 Transformer LM

我们现在有了预处理数据（通过分词器）和模型（Transformer）的步骤。剩下的就是构建支持训练的所有代码。这包括：

- 损失：我们需要定义损失函数（交叉熵）。
- 优化器：我们需要定义优化器来最小化此损失（AdamW）。
- 训练循环：我们需要所有支持性基础设施来加载数据、保存检查点和管理训练。

4.1 交叉熵损失

我们回顾一下 Transformer 语言模型为长度为 $m+1$ 的每个序列 x 和 $i=1,\ldots,m$ 定义了一个分布 $p_\theta(x_{i+1} \mid x_{1:i})$。给定一个由长度为 m 的序列组成的训练集 D，我们定义了标准的交叉熵（负对数似然）损失函数：

$$\ell(\theta; D) = \frac{1}{|D|m} \sum_{x \in D} \sum_{i=1}^{m} -\log p_\theta \left(x_{i+1} \mid x_{1:i}\right). \tag{16}$$

（请注意，Transformer 中的一次前向传播会为所有 $i=1,\ldots,m$ 产生 $p_\theta(x_{i+1} \mid x_{1:i})$。）

特别是，Transformer 为每个位置 i 计算 logits $o_i \in \mathbb{R}^{\text{vocab-size}}$，这导致：

$$

p \left(x _ {i + 1} \mid x _ {1: i}\right) = \operatorname {softmax} \left(o _ {i}\right) \left[x _ {i + 1} \right] = \frac {\exp \left(o _ {i} \left[x _ {i + 1} \right]\right)}{\sum_ {a = 1} ^ {\text {v o c a b - s i z e}} \exp \left(o _ {i} [a]\right)}. \tag {17} $$

交叉熵损失通常是相对于 logits 向量 $o_i \in \mathbb{R}^{\text{vocab\_size}}$ 和目标 x_{i+1} 来定义的。[7]

实现交叉熵损失需要像 softmax 一样，在数值问题上小心处理。

问题 (cross_entropy): 实现交叉熵

交付物：编写一个函数来计算交叉熵损失，该函数接收预测的 logits (o_i) 和目标 (x_{i+1})，并计算交叉熵 $\ell_i = -\log \text{softmax}(o_i)[x_{i+1}]$。你的函数应该处理以下问题：

- 减去最大元素以获得数值稳定性。
- 在可能的情况下抵消 log 和 exp。
- 处理任何额外的批次维度，并在批次上返回平均值。与第 3.3 节一样，我们假设批次类维度始终排在词汇表大小维度之前。

实现 [adapters.run CROSS_entropy]，然后运行 uv run pytest -k test CROSS_entropy 来测试你的实现。

困惑度交叉熵足以用于训练，但当我们评估模型时，我们也想报告困惑度。对于长度为 m 的序列，我们遭受交叉熵损失 $\ell_1, \ldots, \ell_m$：

$$\text{困惑度} = \exp\left(\frac{1}{m}\sum_{i=1}^{m}\ell_i\right). \tag{18}$$

4.2 SGD 优化器

现在我们有了损失函数，我们将开始探索优化器。最简单的基于梯度的优化器是随机梯度下降 (SGD)。我们从随机初始化的参数 θ_0 开始。然后对于每一步 $t = 0, \ldots, T-1$，我们执行以下更新：

$$

$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L\left(\theta_t; B_t\right)$, (19)

其中 B_t 是从数据集 D 中采样的一个随机数据批次，学习率 α_t 和批次大小 $|B_t|$ 是超参数。

4.2.1 在 PyTorch 中实现 SGD

为了实现我们的优化器，我们将继承 PyTorch 的 torch.optim.Optimizer 类。Optimizer 子类必须实现两个方法：

`__init__` 方法的 `params` 参数应该初始化你的优化器。这里，`params` 将是待优化的参数集合（或者参数组，如果用户希望为模型的不同部分使用不同的超参数，例如学习率）。确保将 `params` 传递给基类的 `__init__` 方法，它将存储这些参数以供 `step` 方法使用。你可以根据优化器接受额外的参数（例如，学习率是一个常见的参数），并将它们作为字典传递给基类构造函数，其中键是你为这些参数选择的名称（字符串）。
def step(self) 应该对参数进行一次更新。在训练循环中，这将在反向传播后调用，因此您可以访问最后一个批次的梯度。此方法应遍历每个参数张量 p 并就地修改它们，即设置 p.data，它根据梯度 p.grad（如果存在）保存与该参数关联的张量，该张量表示损失相对于该参数的梯度。

PyTorch 优化器 API 有一些细微之处，因此最好通过示例来解释。为了使我们的示例更丰富，我们将实现 SGD 的一个细微变体，其中学习率在训练过程中衰减，从初始学习率 α 开始，并随着时间的推移逐渐减小步长：

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{t+1}} \nabla L\left(\theta_t; B_t\right) \tag{20}$$

让我们看看这个版本的 SGD 如何作为 PyTorch 优化器实现：

```
from collections.abc import Callable, Iterable
from typing import Optional
import torch
import math

class SGD(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3):
        if lr < 0:
            raise ValueError(f"Invalid learning rate:{lr}")
        defaults = {"lr": lr}
        super(SGD, self).__init__(params, defaults)

    def step(self, closure: Optional[Callable] = None):
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            lr = group["lr"]  # 获取学习率

            for p in group["params"]:
                if p.grad is None:
                    continue
                state = self.state[p]  # 获取与 p 关联的状态
                # 从状态中获取迭代次数，或使用初始值
                t = state.get("t", 0)
                grad = p.grad.data  # 获取损失相对于 p 的梯度
                # 更新权重张量
                p.data -= lr / math.sqrt(t + 1) * grad
                state["t"] = t + 1  # 增加迭代次数

        return loss


for p in group["params"]:
    if p.grad is None:
        continue
    state = self.state[p]  # Get state associated with p
    t = state.get("t", 0)  # Get iteration number from the state, or initial value.
    grad = p.grad.data  # Get the gradient of loss with respect to p.
    p.data -= lr / math.sqrt(t + 1) * grad # Update weight tensor in-place.
```

```
        state["t"] = t + 1  # Increment iteration number.

    return loss
```

在 **init** 中，我们将参数和默认超参数传递给基类构造函数（参数可能成组，每组有不同的超参数）。如果参数只是一个 torch.nn_PARAMETER 对象的集合，基类构造函数将创建一个组并为其分配默认超参数。然后，在 step 中，我们遍历每个参数组，然后遍历该组中的每个参数，并应用公式 20。在这里，我们将迭代次数作为与每个参数关联的状态：我们首先读取此值，在梯度更新中使用它，然后更新它。API 指定用户可以传入一个可调用闭包，在优化器步之前重新计算损失。我们使用的优化器不需要这个，但为了符合 API，我们添加了它。

为了看到这个工作过程，我们可以使用以下训练循环的最小示例：

```
weights  $=$  torch.nn.Parameters(5 \* torch.random((10, 10))) opt  $=$  SGD([weights],lr=1)


for t in range(100):
    opt.zero_grad() # 重置所有可学习参数的梯度。
    loss = (weights**2).mean() # 计算标量损失值。
    print(loss.cpu().item())
    loss.backup() # 执行反向传播，计算梯度。
    opt.step() # 执行优化器步。
```

这是典型的训练循环结构：在每次迭代中，我们将计算损失并执行一次优化器步。在训练语言模型时，我们的可学习参数将来自模型（在PyTorch中，m.params()为我们提供了这个集合）。损失将在采样的数据批次上计算，但训练循环的基本结构将保持不变。

问题 (学习率调整): 调整学习率 (1 分)

正如我们将看到的，对训练影响最大的超参数之一是学习率。让我们在我们的玩具示例中实际看看。使用学习率的三个其他值：1e1、1e2 和 1e3，运行上面的 SGD 示例，仅进行 10 次训练迭代。这些学习率的损失会发生什么？它衰减得更快、更慢，还是发散（即在训练过程中增加）？

交付物：一两句话的响应，描述您观察到的行为。

4.3 AdamW

现代语言模型通常使用更复杂的优化器进行训练，而不是 SGD。最近使用的大多数优化器都是 Adam 优化器 [Kingma and Ba, 2015] 的派生。我们将使用 AdamW [Loshchilov and Hutter, 2019]，它在最近的工作中被广泛使用。AdamW 提出了一种对 Adam 的修改，通过添加权重衰减（在每次迭代中，我们将参数拉向 0）来改进正则化，

AdamW 是一种优化器，它将权重衰减与梯度更新解耦。我们将按照 Loshchilov 和 Hutter [2019] 的算法 2 中的描述来实现 AdamW。

AdamW 是有状态的：对于每个参数，它都会跟踪其一阶矩和二阶矩的运行估计。因此，AdamW 使用额外的内存来换取改进的稳定性和收敛性。除了学习率 α 之外，AdamW 还拥有一对超参数 (β_1, β_2)，用于控制矩估计的更新，以及一个权重衰减率 λ。典型应用将 (β_1, β_2) 设置为 (0.9, 0.999)，但像 LLaMA [Touvron et al., 2023] 和 GPT-3 [Brown et al., 2020] 这样的大型语言模型通常使用 (0.9, 0.95) 进行训练。该算法可以写成如下形式，其中 ϵ 是一个很小的值（例如，10^{-8}），用于在 v 中出现极小值时提高数值稳定性：

算法 1 AdamW 优化器 问题 (adamw)：实现 AdamW（2 分）
问题 (adamwAccounting): 使用 AdamW 进行训练的资源核算（2 分）

```
init $(\theta)$（初始化可学习参数）
$m\gets 0$（一阶矩向量的初始值；形状与 $\theta$ 相同 $v\gets 0$（二阶矩向量的初始值；形状与 $\thet
for $t = 1,\ldots ,T$ do 采样数据批次 $B_{t}$ $g\gets \nabla_{\theta}\ell (\theta ;B_t)$（计算
end for
```

请注意，t 从 1 开始。你现在将实现这个优化器。

交付物：实现 AdamW 优化器，作为 torch.optim.Optimizer 的子类。你的类应该在 **init** 中接收学习率 α，以及 β、ϵ 和 λ 超参数。为了帮助你保持状态，基类 Optimizer 提供了一个字典 self.state，它将 nn.Parameters 对象映射到一个字典，该字典存储了你需要的关于该参数的任何信息（对于 AdamW，这将是动量估计）。实现 [adapters.get_adamw_cls]，并确保它通过 uv_run.pytest -k test_adamw。

让我们计算一下运行 AdamW 所需的内存和计算量。假设我们对每个张量都使用 float32。

(a) 运行 AdamW 需要多少峰值内存？根据参数、激活值、梯度和优化器状态的内存使用情况分解您的答案。用 batch_size 和模型超参数（vocab_size、context_length、num_layers、d_model、num_heads）表示您的答案。假设 $\mathrm{d_{ff}} = 4 \times \mathrm{d_{model}}$。

为简单起见，在计算激活值的内存使用时，仅考虑以下组件：

- Transformer 块

RMSNorm(s)

- 多头自注意力子层： QKV 投影， $Q^\top K$ 矩阵乘法，softmax，值的加权和，输出投影。
- 按位置的馈forward： W_1 矩阵乘法，SiLU， W_2 矩阵乘法
- 最终 RMSNorm 输出嵌入
- logits 上的交叉熵

交付成果：参数、激活值、梯度和优化器状态以及总计的代数表达式。

(b) 为 GPT-2 XL 模型实例化您的答案，得到一个仅取决于 batch_size 的表达式。您能使用的最大 batch size 是多少，才能仍然适合 80GB 内存？

交付物：一个形如 $a \cdot \text{batch\_size} + b$ 的表达式，其中 a, b 为数值，以及一个代表最大 batch size 的数字。

(c) 运行一步 AdamW 需要多少 FLOPs?

交付物：一个代数表达式，附带简要说明。

(d) 模型 FLOPs 利用率 (MFU) 定义为观察到的吞吐量（每秒词元数）相对于硬件理论峰值 FLOPs 吞吐量的比率 [Chowdhery et al., 2022]。一块英伟达 A100 GPU 的 float32 操作理论峰值为 19.5 teraFLOP/s。假设你能获得 50% 的 MFU，在单块 A100 上训练 GPT-2 XL 400K 步，批次大小为 1024，需要多长时间？根据 Kaplan et al. [2020] 和 Hoffmann et al. [2022] 的假设，反向传播的 FLOPs 是前向传播的两倍。

交付成果：训练所需天数，并附简要理由。

4.4 学习率调度

在训练过程中，导致损失下降最快的学习率值通常会发生变化。在训练 Transformer 时，通常会使用学习率调度，即我们从一个较大的学习率开始，在开始时进行更快的更新，并随着模型的训练逐渐将其衰减到一个较小的值[8]。在此次作业中，我们将实现用于训练 LLaMA [Touvron et al., 2023] 的余弦退火调度。

调度器只是一个函数，它接收当前步数 t 和其他相关参数（例如初始和最终学习率），并返回在步数 t 的梯度更新中使用的学习率。最简单的调度器是常数函数，它对于任何 t 都会返回相同的学习率。

余弦退火学习率调度器需要 (i) 当前迭代次数 t、(ii) 最大学习率 $\alpha_{\max}$、(iii) 最小（最终）学习率 $\alpha_{\min}$、(iv) 热身迭代次数 T_w 和 (v) 余弦退火迭代次数 T_c。迭代次数 t 时的学习率定义为：

(热身) 如果 $t < T_w$，则 $\alpha_t = \frac{t}{T_w}\alpha_{\max}$。

(余弦退火) 如果 $T_w \leq t \leq T_c$，则 $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t-T_w}{T_c-T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$。

(退火后) 如果 $t > T_c$，则 $\alpha_t = \alpha_{\min}$。

问题 (learning_rate_schedule): 实现带热身的余弦学习率调度器

编写一个函数，该函数接受 t、$\alpha_{\max}$、$\alpha_{\min}$、T_w 和 T_c，并根据上述调度器返回学习率 α_t。然后实现 [adapters.get_lr_cosine_schedule]，并确保它通过 uv run pytest -k test_get_lr_cosine_schedule。

4.5 梯度裁剪

在训练过程中，我们有时会遇到产生大梯度的训练样本，这会使训练不稳定。为了缓解这种情况，实践中常用的技术是梯度裁剪。其思想是在每次反向传播后，在执行优化器步之前，对梯度的范数施加一个限制。

给定（所有参数的）梯度 g，我们计算其 ℓ_2 范数 $\|g\|_2$。如果该范数小于最大值 M，则保持 g 不变；否则，将 g 按因子 $\frac{M}{\|g\|_2+\epsilon}$ 进行缩放（其中添加了一个小的 ϵ，例如 10^{-6}，以提高数值稳定性）。请注意，结果范数将略小于 M。

问题 (gradient_clipping): 实现梯度裁剪 (1 分)

编写一个实现梯度裁剪的函数。您的函数应接收一个参数列表和一个最大 ℓ_2 范数。它应该就地修改每个参数梯度。使用 $\epsilon = 10^{-6}$（PyTorch 默认值）。然后，实现适配器 [adapters.run_gradment_clipping]，并确保它通过 uv run pytest -k test_gradment_clipping。

5 训练循环

现在我们将把我们迄今为止构建的主要组件放在一起：分词数据、模型和优化器。

5.1 数据加载器

token=>词元 tokenizer=>分词器 <|endoftext|>=><|endoftext|>

分词后的数据（例如，您在 tokenizer_experiments 中准备的数据）是单个词元序列 $x = (x_1, \ldots, x_n)$。尽管源数据可能由单独的文档组成（例如，不同的网页或源代码文件），但一种常见的做法是将所有这些文档连接成一个单一的词元序列，并在它们之间添加一个分隔符（例如 <|endoftext|> 词元）。

数据加载器将此转换为一批一批的流，其中每批包含 m 长度的 B 个序列，并配有相应的下一个词元，长度也为 m。例如，对于 $B = 1, m = 3$，$([x_2, x_3, x_4], [x_3, x_4, x_5])$ 将是一个可能的批次。

以这种方式加载数据有几个原因可以简化训练。首先，任何 $1 \leq i < n - m$ 都可以得到一个有效的训练序列，因此采样序列是微不足道的。由于所有训练序列的长度都相同，因此无需填充输入序列，这可以提高硬件利用率（也通过增加批次大小 B 来实现）。最后，我们也不需要完全加载整个数据集来采样训练数据，这使得处理可能无法放入内存的大型数据集变得容易。

问题（数据加载）：实现数据加载（2分）

Deliverable：编写一个函数，该函数接受一个 NumPy 数组 x（整数数组，包含词元 ID）、batch_size、context_length 和一个 PyTorch 设备字符串（例如 'cpu' 或 'CUDA:0'），并返回一个张量对：采样的输入序列和相应的下一个词元目标。两个张量都应具有形状 (batch_size, context_length)，包含词元 ID，并且都应放置在请求的设备上。要使用我们提供的测试来测试您的实

现，您首先需要实现 [adapters.run_get_batch] 中的测试适配器。然后，运行 uv run pytest -k test_get_batch 来测试您的实现。

低资源/降级提示：在 CPU 或 Apple Silicon 上加载数据

如果您计划在 CPU 或 Apple Silicon 上训练您的 LM，您需要将数据移动到正确的设备（同样，稍后您应该为您的模型使用相同的设备）。

如果您使用的是 CPU，可以使用 'cpu' 设备字符串；如果您使用的是 Apple Silicon（M*芯片），可以使用 'mps' 设备字符串。

有关 MPS 的更多信息，请参阅以下资源：

- https://developer.apple.com/metals/pytorch/
- https://pytorch.org/docs/main/notes/mps.html

如果数据集太大而无法加载到内存中怎么办？我们可以使用一个名为 mmap 的 Unix 系统调用，它将磁盘上的文件映射到虚拟内存，并在访问该内存位置时惰性地加载文件内容。因此，您可以“假装”整个数据集都在内存中。Numpy 通过 np.memmap（或者在 np.load 中使用标志 mmap_mode='r'，如果您最初使用 np.save 保存了数组）来实现这一点，它将返回一个类似 numpy 数组的对象，该对象会在您访问它们时按需加载条目。在训练期间从数据集中采样（即 numpy 数组）时，请确保以内存映射模式加载数据集（通过 np.memmap 或在 np.load 中使用标志 mmap_mode='r'，具体取决于您保存数组的方式）。确保您还指定一个与您正在加载的数组匹配的 dtype。显式验证内存映射数据是否正确可能很有帮助（例如，不包含超出预期词汇量大小的值）。

5.2 检查点

除了加载数据，我们还需要在训练过程中保存模型。在运行作业时，我们通常希望能够恢复因某种原因中途停止的训练运行（例如，由于作业超时、机器故障等）。即使一切顺利，我们也可能希望稍后能够访问中间模型（例如，事后研究训练动态、从不同训练阶段的模型中采样等）。

一个检查点应该包含我们恢复训练所需的所有状态。我们当然希望至少能够恢复模型的权重。如果使用有状态的优化器（例如 AdamW），我们还需要保存优化器的状态（例如，对于 AdamW，是动量估计）。最后，要恢复学习率调度器，我们需要知道停止时的迭代次数。PyTorch 可以轻松保存所有这些：每个 nnModule 都有一个 state_dict() 方法，该方法返回一个包含所有可学习权重的字典；我们可以稍后使用其对应的 load_state_dict() 方法恢复这些权重。对于任何 nn_optimOptimizer 也是如此。最后，torch.save(obj, dest) 可以将一个对象（例如，一个包含张量值或普通 Python 对象（如整数）的字典）转储到一个文件（路径）或类文件对象，然后可以使用 torch.load(src) 将其加载回内存。

问题（检查点）：实现模型检查点（1 分）

实现以下两个函数来加载和保存检查点：

def save_checkpoint(model, optimizer, iteration, out) 应将前三个参数的所有状态转储到类文件对象 out 中。您可以使用模型和优化器各自的 state_dict 方法来获取它们的相关状态，并使用 torch.save(obj, out) 将 obj 转储到 out 中（PyTorch 支持路径或类文件对象）。通常选择 obj 为字典，但只要您以后能够加载检查点，您就可以使用任何格式。

此函数需要以下参数：

model: torch(nnModule

optimizer: torch.optim.Optimizer

iteration: int

out: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]

def load_checkpoint(src, model, optimizer) 应从 src（路径或类文件对象）加载检查点，然后从该检查点恢复模型和优化器状态。你的函数应返回保存到检查点的迭代次数。你可以使用 torch.load(src) 来恢复你在 save_checkpoint 实现中保存的内容，并使用 model 和 optimizers 中的 load_state_dict 方法将它们恢复到之前的状态。

此函数期望以下参数：

src: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]

model: torch(nnModule

optimizer: torch.optim.Optimizer

实现 [adapters.run_save_checkpoint] 和 [adapters.run_load_checkpoint] 适配器，并确保它们通过 uv run pytest -k test_checkpointing。

5.3 训练循环

现在，是时候将你实现的所有组件整合到主训练脚本中了。让不同超参数的训练运行（例如，通过将它们作为命令行参数传入）更容易启动将会带来回报，因为之后你将多次进行这些操作来研究不同选择如何影响训练。

问题 (training_together)：整合 (4 分)

交付成果：编写一个脚本，该脚本运行一个训练循环，在用户提供的输入上训练你的模型。特别是，我们建议你的训练脚本至少允许：

- 配置和控制各种模型及优化器超参数的能力。
- 使用 np.memmap 内存高效地加载训练和验证大型数据集。

- 将检查点序列化到用户提供的路径。
- 定期记录训练和验证性能（例如，记录到控制台和/或外部服务，如 Weights and Biases）。[a]

6 生成文本

现在我们可以训练模型了，我们需要的最后一块就是从模型生成文本的能力。回想一下，语言模型接收一个长度为 (sequence_length) 的（可能分批的）整数序列，并生成一个大小为 (sequence_length × vocab_size) 的矩阵，其中序列的每个元素都是一个概率分布，预测该位置之后的下一个词。我们现在将编写几个函数来实现这种新序列的采样方案。

Softmax 按照标准约定，语言模型的输出是最后一个线性层（“logits”）的输出，因此我们必须通过 softmax 操作将其转换为归一化概率，我们在公式10中已经见过。

解码

为了从我们的模型中生成文本（解码），我们将向模型提供一个前缀词元序列（“提示”），并要求它生成一个词汇表上的概率分布，以预测序列中的下一个词。然后，我们将从词汇表项上的这个分布中采样，以确定下一个输出词元。

具体来说，解码过程的一个步骤应该接收一个序列 $x_{1\ldots t}$，并通过以下方程返回一个词元 x_{t+1}：

$$P(x_{t+1}=i \mid x_{1\ldots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)}$$
$$v = \mathrm{TransformerLM}(x_{1\ldots t})_t \in \mathbb{R}^{\mathrm{vocab\text{-}size}}$$

其中 TransformerLM 是我们的模型，它接收一个 sequence_length 的序列作为输入，并产生一个 (sequence_length × vocab_size) 大小的矩阵，我们取这个矩阵的最后一个元素，因为我们正在寻找第 t 个位置的下一个词预测。

通过重复地从这些单步条件中采样（将我们先前生成的输出词元附加到下一个解码时间步的输入中），直到我们生成序列结束词元 $<|$ endoftext $|>$（或用户指定的要生成的词元的最大数量），我们可以得到一个基本的解码器。

解码器技巧 我们将尝试使用小型模型，而小型模型有时会生成质量非常低的文本。两个简单的解码器技巧可以帮助解决这些问题。首先，在温度缩放中，我们用温度参数 τ 修改我们的 softmax，其中新的 softmax 是

$$ ParseError: KaTeX parse error: \tag works only in display equations

注意，将 $\tau \to 0$ 设置为使 v 的最大元素占主导地位，并且 softmax 的输出成为集中在该最大元素上的独热向量。

其次，另一个技巧是核采样或 top- p 采样，我们通过截断低概率词来修改采样分布。令 q 为我们从大小为 (vocab_size) 的（温度缩放的）softmax 得到的概率分布。具有超参数 p 的核采样根据以下方程产生下一个词元：

$$P(x_{t+1} = i|q) = \begin{cases} \frac{q_i}{\sum_{j \in V(p)} q_j} & \text{if } i \in V(p) \\ 0 & \text{otherwise} \end{cases}$$

其中 $V(p)$ 是最小的索引集，使得 $\sum_{j \in V(p)} q_j \geq p$。你可以通过首先按大小对概率分布 q 进行排序，然后选择最大的词汇元素直到达到目标水平 α 来轻松计算此数量。

问题（解码）：解码（3分）

交付成果：实现一个从语言模型解码的函数。我们建议您支持以下功能：

- 为用户提供的提示生成补全（即，输入一些 $x_{1...t}$ 并采样补全，直到遇到一个 $<|\text{endoftext}|>$ 词元）。
- 允许用户控制生成的词元的最大数量。
- 给定所需的温度值，在采样之前将预测的下一个词分布应用 softmax 温度缩放。
- Top-p 采样（Holtzman 等人，2020；也称为 nucleus 采样），给定用户指定的阈值。

7 实验

现在是时候将所有内容整合起来，并在预训练数据集上训练（小型）语言模型了。

7.1 如何运行实验和交付成果

理解 Transformer 架构组件背后原理的最佳方法是亲自修改和运行它。没有替代动手实践的经验。

为此，能够快速、一致地进行实验并记录所做的事情非常重要。为了快速进行实验，我们将在小型模型（1700 万参数）和简单数据集（TinyStories）上运行许多实验。为了保持一致性，您将以系统化的方式剥离组件并改变超参数，为了记录，我们将要求您提交实验日志以及与每个实验相关的学习曲线。

为了能够提交损失曲线，请确保定期评估验证损失，并记录步数和挂钟时间。您可能会发现 Weights and Biases 等日志记录基础设施很有帮助。

问题 (experiment_log): 实验日志 (3 分)

对于您的训练和评估代码，请创建实验跟踪基础设施，使您能够跟踪实验和损失曲线相对于梯度步数和挂钟时间。

交付物：实验日志记录基础设施代码以及本节下面分配问题的实验日志（您尝试过的所有内容的文档）。

7.2 TinyStories

我们将从一个非常简单的数据集（TinyStories；Eldan and Li, 2023）开始，模型将在此数据集上快速训练，并且我们可以看到一些有趣的行为。获取此数据集的说明在第 1 节。下面是该数据集外观的示例。

示例 (tinystories_example): TinyStories 的一个示例 从前有个小男孩叫蒂姆。蒂姆喜欢探索周围的世界。他看到了许多令人惊叹的东西，比如商店里陈列的美丽花瓶。有一天，蒂姆在商店里散步时，发现了一个非常特别的花瓶。当蒂姆看到它时，他惊呆了！他说：“哇，这真是一个令人惊叹的花瓶！我能买下它吗？”店主笑了笑说：“当然可以。你可以把它带回家，向所有朋友展示它有多么令人惊叹！”于是蒂姆把花瓶带回了家，他为此感到非常自豪！他叫来朋友们，向他们展示了这个令人惊叹的花瓶。他的朋友们都觉得这个花瓶很漂亮，简直不敢相信蒂姆有多幸运。这就是蒂姆在商店里找到一个令人惊叹的花瓶的故事！ 超参数调整 我们将为您提供一些非常基础的超参数作为起点，并要求您找到一些适用于其他参数的有效设置。
vocab_size 10000。典型的词汇表大小在十万到数十万之间。你应该改变这个值，看看词汇表和模型行为如何变化。

context_length 256。像TinyStories这样简单的数据集可能不需要很长的序列长度，但对于后面的OpenWebText数据，你可能需要改变它。尝试改变它，看看对每次迭代的运行时间和最终困惑度的影响。

d_model 512。这比许多小型Transformer论文中使用的768个维度要小一些，但这会使事情更快。

d_ff 1344。这大约是 $\frac{8}{3}$ d_model，同时是64的倍数，这有利于GPU性能。

RoPE theta参数 Θ 10000。

层数和头数 4层，16头。总的来说，这将提供大约17M个非嵌入参数，这是一个相当小的Transformer。

处理的总词元数 327,680,000（你的批次大小 $\times$ 总步数 $\times$ 上下文长度应大致等于此值）。

你需要进行一些试错来找到以下其他超参数的良好默认值：学习率、学习率预热、其他 AdamW 超参数 $(\beta_1, \beta_2, \epsilon)$ 和权重衰减。你可以在 Kingma 和 Ba [2015] 中找到这些超参数的一些典型选择。

整合起来 现在你可以将所有内容整合起来，获取一个训练好的 BPE 分词器，对训练数据集进行分词，并在你编写的训练循环中运行它。重要提示：如果你的实现是正确且高效的，上述超参数应该能在 1 块 H100 GPU 上实现大约 30-40 分钟的运行时间。如果你的运行时间长得多，请检查并确保你的数据加载、检查点或验证损失代码没有成为你运行时间的瓶颈，并且你的实现已正确分批。

调试模型架构的技巧和窍门 我们强烈建议您熟悉 IDE 内置的调试器（例如 VSCode/PyCharm），与使用 print 语句进行调试相比，这可以节省您的时间。如果您使用文本编辑器，可以使用类似 pdb 的工具。调试模型架构时，还有一些其他好的做法：

- 开发任何神经网络架构时，一个常见的首要步骤是使其过拟合到单个小批量。如果您的实现正确，您应该能够快速将训练损失降至接近零。
- 在模型的各个组件中设置调试断点，并检查中间张量的形状，以确保它们符合您的预期。
- 监控激活、模型权重和梯度的范数，以确保它们不会爆炸或消失。

问题（learning_rate）：调整学习率（3 分）（4 H100 小时）

学习率是最重要的超参数之一。以你训练的基础模型为例，回答以下问题：

(a) 对学习率进行超参数扫描，并报告最终损失（如果优化器发散，请注明发散情况）。

交付物：与多个学习率相关的学习曲线。解释你的超参数搜索策略。

交付物：一个在 TinyStories 上验证损失（每词元）不超过 1.45 的模型。

低资源/降级技巧：在 CPU 或 Apple Silicon 上训练较少步数

如果你在 CPU 或 MPS 上运行，你应该将处理的总词元数减少到 40,000,000，这足以生成相当流畅的文本。你也可以将目标验证损失从 1.45 提高到 2.00。

使用经过优化的学习率，在 M3 Max 芯片和 36 GB RAM 上运行我们的解决方案代码，我们使用的批次大小 $\times$ 总步数 $\times$ 上下文长度 $= 32 \times 5000 \times 256 = 40,960,000$ 个词元，在 CPU 上耗时 1 小时 22 分钟，在 MPS 上耗时 36 分钟。在第 5000 步时，我们达到了 1.80 的验证损失。

一些额外的建议：

- 当使用 X 训练步数时，我们建议调整余弦学习率衰减计划，使其在第 X 步精确地终止衰减（即达到最小学习率）。
- 使用 MPS 时，不要使用 TF32 核心，即不要设置 `torch.set_float32_matmul_precision('high')`，这与使用 CUDA 设备时不同。我们尝试在 MPS（torch 版本 2.6.0）上启用 TF32 核心，发现后端会默默地使用损坏的核心，导致训练不稳定。
- 您可以通过使用 `torch.compile` 对模型进行 JIT 编译来加速训练。具体来说：
- 在 CPU 上，使用以下命令编译您的模型

$$

模型 = torch.compile(model)

- 在 mps 上，您可以通过以下方式在一定程度上优化反向传播：

模型 = torch.compile(model, backend="aot_eager")

截至 torch 版本 2.6.0，在 mps 上不支持使用 Inductor 进行编译。

(b) 普遍的看法是，最佳学习率是“处于稳定性边缘”。研究学习率发散点与您的最佳学习率之间的关系。

交付成果：学习曲线，其中包含递增的学习率，至少包含一次发散运行，并分析其与收敛率的关系。

现在，让我们改变批次大小，看看训练会发生什么。批次大小很重要——它们通过执行更大的矩阵乘法使我们的 GPU 获得更高的效率，但我们是否总是希望批次大小很大呢？让我们进行一些实验来找出答案。

问题（batch_size 实验）：Batch size 变化（1 个点）（2 个 H100 小时）

将 batch size 从 1 调整到 GPU 内存限制。尝试几个中间的 batch size，包括 64 和 128 等典型大小。

交付物：不同 batch size 运行的学习曲线。如有必要，应重新优化学习率。

交付物：几句话讨论您关于 batch size 及其对训练影响的发现。

有了 decoder，我们现在可以生成文本了！我们将从模型中生成并查看其效果。作为参考，您的输出应该至少和下面的示例一样好。

示例（ts_generate_example）：TinyStories 语言模型的示例输出

Once upon a time, there was a pretty girl named Lily. She loved to eat gum, especially the big black one. One day, Lily's mom asked her to help cook dinner. Lily was so excited! She loved to help her mom. Lily's mom made a big pot of soup for dinner. Lily was so happy and said, "Thank you, Mommy! I love you." She helped her mom pour the soup into a big bowl. After dinner, Lily's mom made some yummy soup. Lily loved it! She said, "Thank you, Mommy! This soup is so yummy!" Her mom smiled and said, "I'm glad you like it, Lily." They finished cooking and continued to cook together. The end.

Low-Resource/Downscaling Tip: Generate text on CPU or Apple Silicon

If instead you used the low-resource configuration with 40M tokens processed, you should see generations that still resemble English but are not as fluent as above. For example, our sample output from a TinyStories language model trained on 40M tokens is below:

从前，有一个叫苏的小女孩。苏有一颗她非常喜欢的牙齿。这是他最好的头。有一天，苏出去散步，遇到了一只瓢虫！他们成了好朋友，一起在小路上玩耍。

“嘿，波莉！我们出去吧！”蒂姆说。苏看着天空，发现很难找到跳舞闪耀的方法。她笑了，同意帮助说话！

当苏看着天空移动时，它是什么。她

这是精确的问题陈述以及我们的要求：

问题（生成）：生成文本（1 分）

使用您的解码器和训练好的检查点，报告您的模型生成的文本。您可能需要调整解码器参数（温度、top-p 等）以获得流畅的输出。

可交付成果：文本转储，至少包含 256 个词元（或直到第一个 $<$ |endoftext|> 词元），以及关于此输出流畅度的简短评论，以及至少两个影响此输出好坏的因素。

7.3 消融和架构修改

理解 Transformer 的最佳方法是实际修改它并观察其行为。我们现在将进行一些简单的消融和修改。

消融 1：层归一化 有时人们说层归一化对于 Transformer 训练的稳定性很重要。但也许我们想冒险一搏。让我们移除我们每个 Transformer 块中的 RMSNorm，看看会发生什么。

问题 (layer_norm_ablation)：移除 RMSNorm 并进行训练（1 分）（1 H100 小时）

移除你 Transformer 中的所有 RMSNorm 并进行训练。在之前的最优学习率下会发生什么？你能通过使用较低的学习率来获得稳定性吗？

交付物：移除 RMSNorm 并进行训练时的学习曲线，以及最佳学习率的学习曲线。

交付物：几句话评论 RMSNorm 的影响。

现在让我们研究另一个乍一看似乎是任意选择的层归一化。Pre-norm Transformer 块定义为

$$ z = x + \text {M u l t i H e a d e d S e l f A t t e n t i o n} (\operatorname {R M S N o r m} (x))

$$y = z + \mathrm{FFN}(\mathrm{RMSNorm}(z)).$$

这是对原始 Transformer 架构的少数“共识”修改之一，它使用了后归一化方法，如

$$z = \mathrm{RMSNorm}(x + \mathrm{M u l t i H e a d e d S e l f A t t e n t i o n}(x))$$

$$y = \mathrm{RMSNorm}(z + \mathrm{FFN}(z)).$$

让我们回到后归一化方法，看看会发生什么。

问题 (pre_norm_ablation): 实现后归一化并进行训练 (1 分) (1 H100 小时)

将您的预归一化 Transformer 实现修改为后归一化。使用后归一化模型进行训练，看看会发生什么。

交付成果：后归一化 Transformer 的学习曲线，与预归一化 Transformer 进行比较。

我们看到层归一化对 Transformer 的行为有重大影响，甚至层归一化的位置也很重要。

消融实验 2：位置嵌入 我们接下来将研究位置嵌入对模型性能的影响。具体来说，我们将比较我们的基础模型（带有 RoPE）与完全不包含位置嵌入（NoPE）的模型。事实证明，仅解码器的 Transformer，即那些具有我们已实现的因果掩码的 Transformer，理论上可以在不显式提供位置嵌入的情况下推断相对或绝对位置信息 [Tsai et al., 2019, Kazemnejad et al., 2023]。我们现在将通过实验测试 NoPE 与 RoPE 相比的性能。

问题 (no_pos_emb)：实现 NoPE（1 分）（1 H100 小时）

修改您带有 RoPE 的 Transformer 实现，完全移除位置嵌入信息，看看会发生什么。

交付物：一个学习曲线，比较 RoPE 和 NoPE 的性能。

消融实验 3：SwiGLU vs. SiLU 接下来，我们将遵循 Shazeer [2020] 的方法，通过比较 SwiGLU 前馈网络与使用 SiLU 激活但没有门控线性单元 (GLU) 的前馈网络的性能，来测试门控在前馈网络中的重要性：

$$\mathrm{FFN}_{\mathrm{SiLU}}(x) = W_2 \mathrm{SiLU}\left(W_1 x\right). \tag{25}$$

回想一下，在我们 SwiGLU 的实现中，我们将内部前馈层的维度设置为大约 $d_{\mathrm{ff}} = \frac{8}{3} d_{\mathrm{model}}$ （同时确保 $d_{\mathrm{ff}} \mod 64 = 0$，以利用 GPU 张量核心）。在你的 $\mathrm{FFN}_{\mathrm{SiLU}}$ 实现中，你应该将 $d_{\mathrm{ff}} = 4 \times d_{\mathrm{model}}$ ，以大致匹配 SwiGLU 前馈网络的参数数量（它有三个权重矩阵而不是两个）。

问题 (swiglu_ablation)：SwiGLU vs. SiLU (1 分) (1 H100 小时)

交付成果：一个学习曲线，比较 SwiGLU 和 SiLU 前馈网络的性能，参数数量大致匹配。 # 低资源/降尺度技巧：GPU 资源有限的在线学生应在 TinyStories 上测试修改 在本次作业的剩余部分，我们将转向更大规模、更嘈杂的网络数据集（Open-WebText），尝试架构修改，并（可选地）提交到课程排行榜。 在 OpenWebText 上将一个 LM 训练到流利需要很长时间，因此我们建议 GPU 访问受限的在线学生继续在 TinyStories 上测试修改（使用验证损失作为评估性能的指标）。 # 7.4 在 OpenWebText 上运行 我们现在将转向一个由网络爬取创建的更标准的预训练数据集。Open-WebText [Gokaslan et al., 2019] 的一小部分也作为单个文本文件提供：有关如何访问此文件的信息，请参阅第 1 节。

Here is an example from OpenWebText. Note how the text is much more realistic, complex, and varied. You may want to look through the training dataset to get a sense of what training data looks like for a webscraped corpus.

Example (owt_example): One example from OWT

棒球前景技术总监哈里·帕夫利迪斯在雇佣乔纳森·贾奇时冒了风险。

帕夫利迪斯知道，正如艾伦·施瓦茨在《数字游戏》中所写的那样，“美国文化中没有哪个角落比棒球运动员的表现更能被精确计算、更被热情量化。” 经过几次点击，你就可以发现诺亚·辛德加德的快球在飞向本垒的途中每分钟旋转超过 2100 次，纳尔逊·克鲁兹在 2016 年的合格击球手中拥有最高的比赛平均击球速度，以及无数其他似乎摘自电子游戏或科幻小说的小细节。不断增长的数据海洋赋予了棒球文化中一个日益重要的参与者力量：分析爱好者。

伴随这种赋权而来的是更多的审视——不仅是对测量方法，也是对它们背后的人员和出版物。通过《棒球前景》，帕夫利迪斯深知量化不完美所带来的负面影响。他也知道该网站的接球指标需要重新调整，并且需要一个有学识的人——一个能够解决复杂统计建模问题的人——来完成这项工作。

“他让我们大吃一惊。”哈里·帕夫利迪斯

帕夫利迪斯凭直觉认为，根据后者（Judge）的写作和他俩在一个网站赞助的球场活动中的互动，Judge“领会了”其中的要点。此后不久，两人在一次饮酒时进行了交谈。帕夫利迪斯的直觉得到了证实。Judge非常适合这个职位——更确切地说，他是一个乐意接受这个职位的人。“我问了很多人，”帕夫利迪斯说，“他是唯一一个敢于承担这项工作的人。” [...]

注意：您可能需要为本次实验重新调整超参数，例如学习率或批次大小。

问题 (mainExperiment): 在 OWT 上进行实验 (2 分) (3 H100 小时)

在 OpenWebText 上训练你的语言模型，模型架构和总训练迭代次数与 TinyStories 相同。这个模型表现如何?

交付物：你的语言模型在 OpenWebText 上的学习曲线。描述与 TinyStories 的损失差异——我们应该如何解释这些损失?

交付物：从 OpenWebText LM 生成的文本，格式与 TinyStories 的输出相同。这段文本的流畅度如何？为什么即使我们拥有与 TinyStories 相同的模型和计算预算，输出质量却更差?

7.5 你自己的修改 + 排行榜

恭喜你走到这一步。你快完成了！你现在将尝试改进 Transformer 架构，并看看你的超参数和架构与其他同学相比如何。

排行榜规则 除了以下几点外，没有其他限制：

运行时 您的提交最多可以在 H100 上运行 1.5 小时。您可以通过在 slurm 提交脚本中设置 --time=01:30:00 来强制执行此操作。 数据 您只能使用我们提供的 OpenWebText 训练数据集。 否则，您可以随心所欲。 如果您正在寻找一些实现想法，可以查看以下资源： - 最先进的开源 LM 系列，例如 Llama 3 [Grattafori et al., 2024] 或 Qwen 2.5 [Yang et al., 2024]。
- NanoGPT 速度运行仓库 (https://github.com/KellerJordan/modded-nanogpt)，社区成员在此发布了许多用于"速度运行"小型语言模型预训练的有趣修改。例如，一个可以追溯到原始 Transformer 论文的常见修改是将输入和输出嵌入的权重绑定在一起（参见 Vaswani et al. [2017]（第 3.4 节）和 Chowdhery et al. [2022]（第 2 节））。如果您尝试权重绑定，可能需要减小嵌入/LM 头初始化的标准差。

您可能希望在 OpenWebText 的一小部分数据或 TinyStories 上测试这些修改，然后再尝试完整的 1.5 小时运行。

需要注意的是，我们确实注意到，您可能会发现在此排行榜中运行良好的某些修改可能无法推广到更大规模的预训练。我们将在课程的缩放定律单元中进一步探讨这个想法。

问题（排行榜）：排行榜（6 分）（10 H100 小时）

您将在上述排行榜规则下训练一个模型，目标是在 1.5 个 H100 小时内最小化您的语言模型的验证损失。

交付物：记录的最终验证损失，一个清晰显示时钟时间 x 轴（小于 1.5 小时）的相关学习曲线，以及对您所做工作的描述。我们期望排行榜提交的成绩至少要优于 5.0 损失的朴素基线。在此处提交到排行榜：https://github.com/stanford-cs336/assignment1-basics-leaderboard。

References

Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent English?, 2023. arXiv:2305.07759. Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. OpenWebText corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019. Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proc. of ACL, 2016.
王昌汉、赵景贤和顾嘉涛。Neural machine translation with byte-level subwords, 2019. arXiv:1909.03341. 菲利普·盖奇。A new algorithm for data compression. C Users Journal, 12(2):23-38, February 1994. ISSN 0898-9788. 亚历克·拉德福德、吴杰、雷翁·柴尔德、David Luan、达里奥·阿莫代和伊利亚·苏茨克维。Language models are unsupervised multitask learners, 2019. 亚历克·拉德福德、卡尔蒂克·纳拉辛汉、蒂姆·萨利曼斯和伊利亚·苏茨克维。Improving

language understanding by generative pre-training, 2018. 阿希什·瓦斯瓦尼、诺姆·沙泽尔、妮基·帕玛、雅各布·乌什科雷特、里昂·琼斯、艾丹·N·戈麦斯、卢卡斯·凯泽和伊利亚·波洛苏欣。Attention is all you need. In Proc. of NeurIPS, 2017. 阮Q. 阮和朱利安·萨拉查。Transformers without tears: Improving the normalization of self-attention. In Proc. of IWSWLT, 2019.
熊瑞宾、杨云昌、何迪、郑凯、郑树新、邢晨、张辉帅、兰艳艳、王利伟和刘铁岩。On layer normalization in the Transformer architecture。In Proc. of ICML, 2020。 鲍继荣、杰米·瑞安·基罗斯和杰弗里·E·辛顿。Layer normalization, 2016。arXiv:1607.06450。 雨果·图夫龙、蒂博·拉夫里尔、戈蒂尔·伊扎卡德、泽维尔·马蒂内、玛丽-安妮·拉肖、蒂莫泰·拉科鲁瓦、巴蒂斯特·罗齐耶尔、纳曼·戈亚尔、埃里克·汉布罗、费萨尔·阿兹哈尔、奥雷利安·罗德里格斯、阿曼德·朱林、爱德华·格雷夫和纪尧姆·兰普尔。Llama: Open and efficient foundation language models, 2023。arXiv:2302.13971。 张彪和里科·森里奇。Root mean square layer normalization。In Proc. of NeurIPS, 2019。
亚伦·格拉塔夫蒂奥里、阿比曼尤·杜贝、阿比纳夫·乔里、阿比纳夫·潘迪、阿布舍克·卡迪安、艾哈迈德·阿尔-达赫勒、艾莎·莱特曼、阿基尔·马图尔、艾伦·谢尔滕、亚历克斯·沃恩、艾米·杨、安吉拉·范、阿尼鲁德·戈亚尔、安东尼·哈特肖恩、奥博·杨、阿尔奇·米特拉、阿奇·斯劳万库马尔、阿尔テム·科列涅夫、亚瑟·欣斯瓦克、阿伦·拉奥、阿斯顿·张、奥雷利安·罗德里格斯、奥斯滕·格雷格森、艾娃·斯帕塔鲁、巴蒂斯特·罗齐耶尔、贝丝妮·比隆、宾·唐、鲍比·切恩、夏洛特·考谢特、查雅·纳亚克、克洛伊·比、克里斯·马拉、克里斯·麦康奈尔、克里斯蒂安·凯勒、克里斯托夫·图雷特、春阳·吴、科琳·黄、克里斯蒂安·坎通·费雷尔、赛勒斯·尼古拉迪斯、达米安·阿隆修斯、丹尼尔·宋、丹妮尔·品茨、丹尼·利夫希茨、丹尼·怀亚特、大卫·埃西奥布、德鲁夫·乔杜里、德鲁夫·马哈詹、迭戈·加西亚-奥拉诺、迭戈·佩里诺、迪乌克·胡普克斯、叶戈尔·拉科姆金、埃哈布·阿尔巴达维、埃琳娜·洛巴诺娃、艾米丽·迪南、埃里克·迈克尔·史密斯、菲利普·拉德诺维奇、弗朗西斯科·古兹曼、弗兰克·张、加布里埃尔·辛纳夫、加布里埃尔·李、乔治亚·刘易斯·安德森、戈文德·塔塔伊、格雷姆·奈尔、格雷瓜尔米亚隆, 关鹏, 吉列姆·库库雷尔, 海莉·阮, 汉娜·科雷瓦尔, 胡旭, 雨果·图夫龙, 伊利扬·扎罗夫, 伊马诺尔·阿列塔·伊巴拉, 伊莎贝尔·克劳曼, 伊尚·米斯拉, 伊万·埃夫季莫夫, 杰克·张, 杰德·科佩特, 在元·李, 扬·格费特, 雅娜·弗兰斯, 杰森·朴, 杰·马哈德奥卡, 吉特·沙阿, 杰尔默·范德林德, 詹妮弗·比洛克, 珍妮·洪, 叶夫根尼·李, 杰里米·傅, 建峰·池, 建宇

黄嘉文 刘杰 王杰草 余乔安娜 比顿·乔 斯皮萨克·钟洙 朴·约瑟夫 罗卡·约书亚 约翰斯顿·约书亚 萨克斯·俊腾 贾·卡利安 瓦苏登·阿尔瓦拉·卡尔蒂克 普拉萨德·卡尔蒂克亚 乌帕萨尼·凯特 普拉维亚克·科 李·肯尼思 希菲尔德·凯文 斯通·哈立德 埃尔-阿里尼·克里西卡 艾耶尔·克希蒂兹 马利克·奎恩利 邱·库纳尔 巴拉·库沙尔 拉霍蒂亚·劳伦 兰塔拉-耶里·劳伦斯 范德马滕·劳伦斯 陈亮 谭·丽兹 詹金斯·路易斯 马丁·洛维什 马丹·卢博 马洛·卢卡斯 布莱彻·卢卡斯 兰德扎特·卢克 德奥利维拉·玛德琳 穆齐·马赫什 帕苏普莱蒂·曼纳特 辛格·马诺哈尔 帕卢里·马尔钦 卡尔达斯·玛丽亚 齐姆普凯利·马修 奥尔德姆·马修 里塔·玛雅 帕夫洛娃·梅兰妮 坎巴杜尔·迈克 刘易斯·敏 西·米特什 库马尔·辛格·莫娜 哈桑·纳曼 戈亚尔·纳尔吉斯 托拉比·尼古拉 巴什利科夫·尼古拉 博戈伊切夫·尼拉德里 查特吉·宁 张·奥利维尔 杜申·奥努尔 切莱比·帕特里克 阿尔拉西·彭川 张鹏伟 李·佩塔尔 瓦西奇·彼得 翁·普拉吉瓦尔 巴尔加瓦·普拉蒂克 杜巴尔·普拉文 克里希南·普尼特 辛格·库拉·普克斯在 Xu, 清河, 董庆晓, 拉加万·斯里尼瓦桑, 拉杰·加纳帕西, 拉蒙·卡尔德勒, 里卡多·席尔维拉·卡布拉尔, 罗伯特·斯托伊奇尼克, 罗伯塔·莱莱努, 罗汉·马赫什瓦里, 罗希特·吉尔达尔, 罗希特·帕特尔, 罗曼·索维斯特, 罗尼·波利多罗, 罗珊·桑巴利, 罗斯·泰勒, 阮·席尔瓦, 侯锐, 王锐, 萨加尔·侯赛尼, 萨哈纳·切纳巴萨帕, 桑杰·辛格, 肖恩·贝尔, 徐贤·索尼娅·金, 谢尔盖·埃杜诺夫, 聂少良, Sharan Narang, 沙拉思·拉帕西, 盛申, 万圣业, 斯鲁蒂·布萨莱, 张顺, 西蒙·范登亨德, 索姆亚·巴特拉, 斯宾塞·惠特曼, 斯滕·苏特拉, 斯特凡·科洛特, 苏钦·古鲁兰甘, 悉尼·博罗丁斯基, 塔玛尔·赫尔曼, 塔拉·福勒, 塔里克·谢沙, 托马斯·乔治乌, 托马斯·西亚洛姆, 托比亚斯·斯佩克巴赫, 托多尔·米哈伊洛夫, 童晓, 乌贾瓦尔·卡恩, 韦达努吉·戈斯瓦米, 维布霍尔·古普塔, 维格内什·拉马纳森, 维克多·克尔克兹, 文森特·贡古埃, 维尔吉妮·多, 维什·沃格蒂, 维托尔·阿尔比耶罗, 弗拉丹·彼得罗维奇, 楚伟伟, 熊文瀚, 傅文印, 惠特尼·米尔斯, 泽维尔·马蒂内, 王晓东, 王晓芳, 谭晓庆·艾伦, 夏希德, 谢新峰, 徐查 的论文中贾学伟 王亚埃勒 戈尔德施拉格,亚什·高尔,亚斯明·巴巴埃伊

易文易文 宋宇晨 张悦 李宇宁 毛扎卡里 德尔皮埃尔·库德尔特,郑岩 郑兴·陈 佐伊·帕帕基波斯 阿迪亚·辛格,阿尤什·斯里瓦斯塔瓦 阿芭·贾恩 亚当·凯尔西 亚当·沙因费尔德 阿迪亚·甘迪迪 阿道夫·维多利亚,阿胡瓦·戈尔德斯坦德 阿贾伊·梅农 阿贾伊·夏尔马 亚历克斯·博森伯格 阿列克谢·巴耶夫斯基 艾莉·范斯坦 阿米特·桑加尼 阿莫斯·特奥 阿纳姆·尤努斯 安德烈·卢普 安德烈斯·阿尔瓦拉多 安德鲁·卡普尔斯 安德鲁·顾 安德鲁·波尔顿 安德鲁·瑞安 安基特·拉姆昌达尼 安妮·董 安妮·弗朗哥 阿努吉·戈亚尔 阿帕拉吉塔·萨拉夫 阿尔卡班杜·乔杜里 阿什利·加布里埃尔 阿什温·巴拉姆贝 阿萨夫·艾森曼 阿扎德·亚兹丹 博·詹姆斯 本·莫瑞尔 本杰明·莱昂哈迪 伯尼·黄 贝丝·劳埃德 贝托·德·保拉 巴尔加维·帕兰加佩 刘冰 吴波 博宇倪 布雷登·汉考克 布拉姆·瓦斯蒂 布兰登·斯宾塞 布拉尼·斯托伊科维奇 布莱恩·加米多 布里特·蒙塔尔沃 卡尔·帕克 卡莉·伯顿 卡塔利娜·梅希亚 刘策 王昌汉 金昌奎 周超 切斯特·胡 朱庆祥 克里斯·蔡 克里斯·廷达尔 克里斯托夫·费希滕霍费尔 辛西娅·高 达蒙·西文 达娜比蒂·丹尼尔·克雷默 丹尼尔·李 大卫·阿德金斯 大卫·徐 大卫德·泰斯图吉内 德利娅·大卫 黛维·帕里赫 戴安娜·利斯科维奇 迪德姆·福斯 丁康·王 杜克·勒,达斯汀·荷兰 爱德华·唐林 艾萨·贾米尔 伊莱恩·蒙哥马利 埃莉诺拉·普雷萨尼 艾米丽·哈恩 艾米丽·伍德 埃里克-图安·勒 埃里克·布林克曼 埃斯特班·阿尔考特 埃文·邓巴 埃文·斯莫瑟斯 飞·孙 费利克斯·克鲁克 峰·田 菲利波斯·科基诺斯 菲拉特·奥兹格内尔 弗朗切斯科·卡乔尼 弗兰克·卡纳耶特 弗兰克·赛德 加布里埃拉·梅迪纳·弗洛雷斯,加布里埃拉·施瓦茨 加达·巴迪尔 乔治亚·斯威 吉尔·哈尔珀恩 格兰特·赫尔曼 格里戈里·西佐夫 光义·张古纳·拉克什米纳拉亚南 哈坎·伊南 哈米德·肖贾纳泽里 汉·邹 汉娜·王汉文·赵 哈伦·哈比卜 哈里森·鲁道夫 海伦·苏克 亨利·阿斯佩格伦 亨特·戈德曼 宏远·展 易卜拉欣·达姆拉吉 伊戈尔·莫利博格 伊戈尔·图法诺夫 伊利亚斯·莱昂蒂亚迪斯 伊琳娜-埃琳娜·韦利切 伊泰·加特 杰克·韦斯曼詹姆斯·格博斯基 詹姆斯·科利 珍妮丝·兰迦弗·亚设 让-巴蒂斯特·盖亚 杰夫·马库斯杰夫·唐 詹妮弗·陈 珍妮·甄 杰里米·赖岑斯坦 杰里米·特布尔 杰西卡·钟 建·金 静怡·杨 乔·卡明斯 J卡维尔·乔恩·谢泼德 乔纳森·麦克菲 乔纳森·托雷斯 乔什·金斯伯格 王俊杰 吴凯 坎·侯·U 卡兰·萨克塞纳 卡蒂凯·坎德尔瓦尔 卡塔尤恩·赞德 凯西·马托西奇 考希克·维拉拉加万 凯利·米歇尔纳 李克谦 基兰·贾加迪什 黄坤 库纳尔·乔拉 凯尔·黄 陈来林 拉克沙·加格 薰衣草A 莱昂德罗·席尔瓦 李·贝尔 张磊 郭良鹏 于立诚 利隆·莫什科维奇 卢卡·韦尔施泰特 马迪安·哈卜萨 马纳夫·阿瓦拉尼 马尼什·巴特 马丁纳斯·曼库斯 马坦·哈森 马修·莱尼 马蒂亚斯·雷索 马克西姆·格罗舍夫 马克西姆·瑙莫夫 玛雅·拉西 梅根·肯尼利 刘淼 迈克尔·L·塞尔泽 米哈尔·瓦尔科 米歇尔·雷斯特雷波 米希尔·帕特尔 米克·维亚特斯科夫 米卡埃尔·萨姆维良 迈克·克拉克 迈克·梅西 王迈克 米克尔·胡贝特·埃尔莫索 莫·梅塔纳特 穆罕默德

马德·拉斯特加里, 穆尼什·班萨尔, 南迪尼·桑塔纳姆, 娜塔莎·帕克斯, 娜塔莎·怀特, 纳夫亚塔·巴瓦, 纳扬·辛格尔, 尼克·埃格博, 尼古拉斯·尤苏尼尔, 尼基尔·梅塔, 尼古拉·巴甫洛维奇·拉普捷夫, 宁东, 诺曼·程, 奥列格·切尔诺古兹, 奥利维亚·哈特, 奥姆卡尔·萨尔佩卡尔, 厄兹莱姆·卡林利, 帕金·肯特, 帕斯·帕雷克, 保罗·萨阿布, 帕万·巴拉吉, 佩德罗·里特纳, 菲利普·邦特拉格, 皮埃尔·鲁, 皮奥特尔·多拉尔, 波琳娜·兹维亚吉娜, 普拉尚特·拉坦昌达尼, 普里蒂什·尤夫拉吉, 钱亮, 拉沙德·阿拉奥, 瑞秋·罗德里格斯, 拉菲·阿尤布, 拉格托汉·穆尔蒂, 拉古·纳亚尼, 拉胡尔·米特拉, 兰加普拉布·帕塔萨拉西, 雷蒙德·李, 丽贝卡·霍根, 罗宾·巴蒂, 洛奇·王, 拉斯·豪斯, 鲁迪·里诺特, 萨钦·梅塔, 萨钦·西比, 赛·贾耶什·邦杜, 萨米亚克·达塔, 萨拉·丘格, 萨拉·亨特, 萨尔贡·迪隆, 萨沙·西多罗夫, 萨塔德鲁·潘, 索拉布·马哈詹, 索拉布·维尔马, 清治·山本, 沙拉德·拉马斯瓦米, 肖恩·林赛, 肖恩·林赛, 盛峰, 盛浩·林, 盛欣·辛迪·赵, 希希尔·帕蒂尔, 希瓦·香卡尔, 书强·张, 书强·张, 思农·王, 斯内哈·阿格拉瓦尔, 索吉萨朱伊格贝、苏密特·钦塔拉、斯蒂芬妮·马克思、斯蒂芬·陈、史蒂夫·基霍、史蒂夫·萨特菲尔德、苏达申·戈文达普拉萨德、苏米特·古普塔、萨默·邓、成民·赵、桑尼·维尔克、苏拉杰·苏布拉马尼安、赛·乔杜里、悉尼·戈德曼、塔尔·雷梅兹、塔玛尔·格拉泽、塔玛拉·贝斯特、蒂洛·科勒、托马斯·罗宾逊、李天河、张天俊、蒂姆·马修斯、蒂莫西·周、祖克·沙凯德、瓦伦·冯蒂米塔、维多利亚·阿贾伊、维多利亚·蒙塔内兹、维杰·莫汉、维奈·萨蒂什·库马尔、维沙尔·曼格拉、弗拉德·伊万内斯库、弗拉德·波埃纳鲁、弗拉德·蒂贝里乌·米哈莱斯库、弗拉基米尔·伊万诺夫、李伟、王文臣、蒋文文、韦斯·布阿齐兹、威尔·康斯特布尔、唐晓成、吴晓健、王晓兰、吴希伦、高新波、亚尼夫·克莱因曼、陈延军、胡叶、贾叶、齐叶、李燕达、张一林、张颖、约西·阿迪、永镇·南、余、王、赵宇、郝宇辰、钱

韵迪、李云露、何雨子、扎克·赖特、扎卡里·德维托、泽夫·罗森布里克、文兆多、杨振宇、赵志伟和马志宇。The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

安阳, 宝松杨, 北辰张, 宾远惠, 博政, 博文宇, 成远李, 大一恒刘, 飞黄, 浩然魏, 欢林, 建杨, 建红图, 建伟张, 建新杨, 嘉喜杨, 敬人周, 俊阳林, 凯当, 克明陆, 克勤鲍, 科信杨, 乐宇, 梅李, 明峰薛, 佩张, 琴朱, 瑞门, 润吉林, 天浩李, 廷宇夏, 兴章任, 宣成任, 杨帆, 杨苏, 益昌张, 宇万, 玉琼刘, 泽宇崔, 真如张, 和子涵邱. Qwen2.5 技术报告. arXiv preprint arXiv:2412.15115, 2024.
Aakanksha Chowdhery, Sharan Narang, 雅各布·德夫林, Maarten Bosma, Gaurav Mishra, 亚当·罗伯茨, 保罗·巴勒姆, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, 诺姆·沙泽尔, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, 詹姆斯·布拉德伯里, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, 桑杰·格马瓦特, Sunipa Dev, Henryk Michalewski, 哈维尔·加西亚, Vedant Misra, 凯文·罗宾逊, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, 雷翁·柴尔德, Oleksandr Polozov, 凯瑟琳·李, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, 和 Noah Fiedel。PaLM: Scaling language modeling with pathways, 2022. arXiv:2204.02311。 Dan Hendrycks 和 Kevin Gimpel。Bridging nonlinearities and stochastic regularizers with gaussian error linear units, 2016. arXiv:1606.08415。 Stefan Elfwing, Eiji Uchibe, 和 Kenji Doya。Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017. URL https://arxiv.org/abs/1702.03118。 Yann N. Dauphin, Angela Fan, Michael Auli, 和 David Grangier。Language modeling with gated convolutional networks, 2017. URL https://arxiv.org/abs/1612.08083。 Noam Shazeer。GLU variants improve transformer, 2020. arXiv:2002.05202。 Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, 和 Yunfeng Liu。Roformer: Enhanced transformer with rotary position embedding, 2021。

Diederik P. Kingma 和 Jimmy Ba。Adam: A method for stochastic optimization. In Proc. of ICLR, 2015。 Ilya Loshchilov 和 Frank Hutter。Decoupled weight decay regularization. In Proc. of ICLR, 2019。
汤姆·B·布朗、本杰明·曼恩、尼克·莱德、梅兰妮·苏比亚、贾里德·卡普兰、普拉富拉·达里瓦尔、阿尔温德·尼拉克坦坦、普拉纳夫·夏姆、吉里什·萨斯特里、阿曼达·阿斯凯尔、桑迪尼·阿格拉瓦尔、阿里尔·赫伯特-沃斯、格蕾琴·克鲁格、汤姆·亨尼根、雷翁·柴尔德、阿迪亚·拉梅什、丹尼尔·M·齐格勒、杰弗里·吴、克莱门斯·温特、克里斯托弗·赫塞、马克·陈、埃里克·西格勒、马特乌什·利特温、斯科特·格雷、本杰明·切斯、杰克·克拉克、克里斯托弗·伯纳、萨姆·麦坎德利什、亚历克·拉德福德、伊利亚·苏茨克维和达里奥·阿莫代。Language models are few-shot learners. In Proc. of NeurIPS, 2020. 贾里德·卡普兰、萨姆·麦坎德利什、汤姆·亨尼根、汤姆·B·布朗、本杰明·切斯、雷翁·柴尔德、斯科特·格雷、亚历克·拉德福德、杰弗里·吴和达里奥·阿莫代。Scaling laws for neural language models, 2020. arXiv:2001.08361.
乔丹·霍夫曼，塞巴斯蒂安·博尔若，亚瑟·门施，埃琳娜·布查茨卡娅，特雷弗·蔡，伊丽莎·卢瑟福，迭戈·德·拉斯·卡萨斯，丽莎·安妮·亨德里克斯，约翰内斯·韦尔布尔，艾丹·克拉克，汤姆·亨尼根，埃里克·诺兰德，凯蒂·米利坎，乔治·范登德里谢，博格丹·达莫克，奥蕾莉亚·盖伊，西蒙·奥辛德罗，凯伦·西蒙扬，埃里希·埃尔森，杰克·W·雷，奥里奥尔·维尼亚尔斯，和洛朗·西弗雷。Training compute-optimal large

language models, 2022. arXiv:2203.15556. 阿里·霍尔茨曼，扬·拜斯，李杜，麦克斯韦尔·福布斯，和崔艺珍。The curious case of neural text degeneration. In Proc. of ICLR, 2020.
姚宏胡伯特蔡、邵杰白、山田诚、路易-菲利普·莫伦西和鲁斯兰·萨拉赫特迪诺夫。Transformer dissection: An unified understanding for transformer's attention via the lens of kernel。在 Kentaro Inui、Jing Jiang、Vincent Ng 和 Xiaojun Wan 编辑的《Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)》中，第 4344-4353 页，香港，中国，2019 年 11 月。Association for Computational Linguistics。doi: 10.18653/v1/D19-1443。URL https://aclanthology.org/D19-1443/。 Amirhossein Kazemnejad、Inkit Padhi、Karthikeyan Natesan、Payel Das 和 Siva Reddy。The impact of positional encoding on length generalization in transformers。在《Thirty-seventh Conference on Neural Information Processing Systems》中，2023 年。URL https://openreview.net/forum?id=Drrl2gcjzl。