CS336 作业 1（基础）：构建 Transformer LM

版本 1.0.6

CS336 教职员工

2025 年春季

1 任务概述

在本作业中，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。

您将实施什么

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）

你将运行什么

1. 在 TinyStories 数据集上训练 BPE 分词器。
2. 在数据集上运行训练好的分词器，将其转换为整数 ID 序列。
3. 在 TinyStories 数据集上训练 Transformer 语言模型。
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，并将您获得的困惑度提交到排行榜。

您可以使用的内容：我们希望您从头开始构建这些组件。尤其需要注意的是，您不得使用 torch.nn、torch.nn.functional 或 torch.optim 中的任何定义，但以下定义除外：

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

您可以使用任何其他 PyTorch 定义。如果您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违背本次作业“从零开始”的理念。

[1] 有关完整列表，请参阅 PyTorch.org/docs/stable/nn.html#containers。

关于人工智能工具的声明：允许使用 ChatGPT 等语言模型提示低级编程问题或关于语言模型的高级概念性问题，但禁止直接使用此类工具解决问题。
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）（但非 AI 自动补全功能，例如自动补全函数名称，完全没问题）。我们发现 AI 自动补全功能会大大降低您深入理解学习内容的能力。

代码示例 所有作业代码以及本文说明均可在 GitHub 上找到：

github.com/stanford-cs336/assignment1-basics

请克隆此仓库。如有任何更新，我们会通知您，以便您使用 `git pull` 获取最新版本。

1. cs336_basics/*：这里是编写代码的地方。请注意，这里没有任何代码——您可以从头开始做任何您想做的事情！
2. adapters.py：你的代码必须包含一系列功能。对于每个功能（例如，缩放点积注意力机制），只需调用你的代码即可完成其实现（例如，运行 run_scaled_dot_product_attention）。注意：你对 adapters.py 的修改不应包含任何实质性逻辑；这只是粘合代码。
3. test_*.py：此文件包含所有必须通过的测试（例如 test_scaled_dot_product_attention），这些测试会调用 adapters.py 中定义的钩子。请勿编辑测试文件。

如何提交 您需要向 Gradescope 提交以下文件：

- writeup.pdf：请回答所有书面问题。请将您的答案排版好。
- code.zip：包含您编写的所有代码。

要将作品提交到排行榜，请提交 PR 至：

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

有关详细的提交说明，请参阅排行榜仓库中的 README.md 文件。

数据集获取方式：本次作业将使用两个预处理数据集：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。如果您是与班级一起完成此作业，可以在任何非主节点机器的 /data 目录下找到这些文件。
如果您在家跟着教程操作，可以使用 README.md 文件中的命令下载这些文件。

低资源/缩减规模提示：初始化

在课程作业讲义中，我们会提供一些建议，帮助您在 GPU 资源有限或没有 GPU 资源的情况下完成部分作业。例如，我们有时会建议您缩小数据集或模型规模，或者解释如何在 MacOS 集成 GPU 或 CPU 上运行训练代码。您会在蓝色方框（例如这个）中找到这些“低资源使用技巧”。即使您是斯坦福大学的注册学生，可以访问课程专用计算机，这些技巧也能帮助您更快地迭代并节省时间，因此我们建议您阅读它们！

低资源/缩减技巧：作业 1 在 Apple Silicon 或 CPU 上

利用我们提供的解决方案代码，我们可以在配备 36GB 内存的 Apple M3 Max 芯片上训练一个语言模型，使其生成相当流畅的文本；使用 Metal GPU (MPS) 时，只需不到 5 分钟；而使用 CPU 时，则大约需要 30 分钟。如果您对这些术语不太熟悉，也无需担心！您只需知道，如果您拥有一台配置较高的笔记本电脑，并且您的实现正确高效，那么您就能训练出一个小型语言模型，生成流畅度尚可的简单儿童故事。

稍后在作业中，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）字符串表示为一个字节序列，并使用该字节序列训练我们的 BPE 分词器。之后，我们将使用该分词器将文本（字符串）编码为词元（整数序列），用于语言建模。

2.1 Unicode 标准

Unicode 是一种文本编码标准，它将字符映射到整数代码点。截至 Unicode 16.0（2024 年 9 月发布），该标准定义了 168 种文字的 154,998 个字符。例如，字符“s”的代码点为 115（通常表示为 U+0073，其中 U+ 是一个约定俗成的前缀，0073 是十六进制的 115），字符“ ”的代码点为 29275。在 Python 中，可以使用 `ord()` 函数将单个 Unicode 字符转换为其整数表示形式。`chr()` 函数将整数 Unicode 代码点转换为包含对应字符的字符串。

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

问题（Unicode1）：理解 Unicode（1 分）

（a）chr(0) 返回哪个 Unicode 字符？

提交内容：一句话回复。

（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？

提交内容：一句话回复。

(c) 当这个字符出现在文本中时会发生什么？尝试一下可能会有所帮助。
请在 Python 解释器中运行以下命令，看看是否符合预期：

```
>>> chr(0)
>>> print(chr(0)) >>> "这是一个测试" + chr(0) + "字符串" >>>
print("这是一个测试" + chr(0) + "字符串")
```

提交内容：一句话回复。

2.2 Unicode 编码

虽然 Unicode 标准定义了字符到代码点（整数）的映射，但直接基于 Unicode 代码点训练分词器并不实际，因为词汇量会非常庞大（约 15 万个条目）且稀疏（因为许多字符非常罕见）。因此，我们将使用 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 是互联网上最主要的编码（超过 98% 的网页都使用 UTF-8）。

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 字节对象的底层字节值，我们可以遍历它（例如，调用 `list()` 函数）。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! !" >>> 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(type(utf8_encoded)) >>> # 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, 227, 129, 175, 33] >>> # 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_encoded.decode("utf-8")) hello! !

通过将 Unicode 码位转换为字节序列（例如，通过 UTF-8 编码），我们实际上是将码位序列（0 到 154,997 范围内的整数）转换为字节值序列（0 到 255 范围内的整数）。256 字节的词汇表更易于管理。使用字节级标记化时，我们无需担心超出词汇表范围的标记，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。问题（unicode2）：Unicode 编码（3 分）

（a）为什么我们更倾向于使用 UTF-8 编码的字节来训练分词器，而不是使用 UTF-16 或 UTF-32 编码的字节？比较这些编码对不同输入字符串的输出结果可能会有所帮助。

提交内容：一到两句话的回复。

(b) 考虑以下（错误的）函数，该函数旨在将 UTF-8 字节字符串解码为 Unicode 字符串。为什么这个函数是错误的？请提供一个会产生错误结果的输入字节字符串示例。

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    返回"".join([bytes([b]).decode("utf-8") for b in bytestring])

>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8")) 'hello'
```

交付成果：一个示例输入字节字符串，在该字符串中，decode_utf8_bytes_to_str_wrong 会产生不正确的输出，并用一句话解释为什么该函数不正确。

（c）给出一个不解码成任何 Unicode 字符的双字节序列。

交付成果：一个示例，并附上一句话的解释。

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，但将文本分词成字节会导致输入序列过长。这会减慢模型训练速度，因为……

在词级语言模型中，一个包含 10 个单词的句子可能只有 10 个标记，但在字符级模型中，它可能有 50 个或更多标记（取决于单词的长度）。处理这些更长的序列需要在模型的每个步骤中消耗更多的计算资源。此外，基于字节序列的语言建模也十分困难，因为更长的输入序列会在数据中产生长期依赖关系。

子词分词介于词级分词器和字节级分词器之间。需要注意的是，字节级分词器的词汇表包含 256 个条目（字节值为 0 到 225）。子词分词器牺牲了较大的词汇表容量，换取了对输入字节序列更佳的压缩效果。例如，如果字节序列 b'the' 在我们的原始文本训练数据中频繁出现，那么将其添加到词汇表中，就能将这个包含 3 个词元的序列简化为一个词元。

我们如何选择要添加到词汇表中的子词单元？Sennrich 等人 [2016] 提出使用字节对编码 (BPE; Gage, 1994)，这是一种压缩算法，它迭代地将出现频率最高的字节对替换（“合并”）为一个新的、未使用的索引。请注意，该算法将子词标记添加到词汇表中，以最大程度地压缩输入序列——如果一个词在输入文本中出现足够多次，它将被表示为一个单独的子词单元。

使用基于字节序列扩展（BPE）构建的词汇表的子词分词器通常被称为 BPE 分词器。在本作业中，我们将实现一个字节级 BPE 分词器，其中词汇表项是字节或合并的字节序列，这在处理词汇表外的词项和控制输入序列长度方面兼具两者的优势。构建 BPE 分词器词汇表的过程称为“训练”BPE 分词器。

2.4 BPE 分词器培训

BPE 分词器训练过程包括三个主要步骤。

词汇表初始化：分词器词汇表是字节串标记到整数 ID 的一一对应关系。由于我们训练的是字节级 BPE 分词器，因此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，因此初始词汇表的大小为 256。

预分词：有了词汇表之后，理论上你可以统计文本中相邻字节出现的频率，并从出现频率最高的字节对开始合并它们。然而，这种方法计算量非常大，因为每次合并都需要对整个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生成的词元仅标点符号不同（例如，dog! 与 dog.）。即使这些词元可能具有很高的语义相似度（因为它们仅标点符号不同），它们也会被赋予完全不同的词元 ID。

为了避免这种情况，我们对语料库进行预分词。您可以将其理解为对语料库进行粗粒度分词，这有助于我们统计字符对出现的频率。例如，单词“text”可能是一个出现 10 次的预分词。在这种情况下，当我们统计字符“t”和“e”相邻出现的频率时，我们会发现单词“text”中“t”和“e”是相邻的，因此我们可以直接将它们的计数加 10，而无需遍历整个语料库。由于我们训练的是字节级 BPE 模型，因此每个预分词都表示为一个 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，我们将使用基于正则表达式的预分词器（GPT-2 使用的；Radford 等人，2019）。
来自 github.com/openai/tiktoken/pull/234/files：

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

使用此预分词器交互式地拆分一些文本可能有助于更好地了解其行为：

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

但是，在代码中使用它时，应该使用 re.finditer 来避免存储预先分词的单词，因为在构建从预分词到其计数的映射时，需要这样做。

计算 BPE 合并 现在我们已经将输入文本转换为预标记，并将每个预标记表示为 UTF-8 字节序列，接下来可以计算 BPE 合并（即训练 BPE 分词器）。简而言之，BPE 算法迭代地统计每一对字节，并识别出频率最高的字节对（“A”、“B”）。然后，将这组频率最高的字节对（“A”、“B”）的每次出现进行合并，即替换为新的标记“AB”。这个合并后的新标记会被添加到词汇表中；因此，BPE 训练后的最终词汇表大小等于初始词汇表的大小（在本例中为 256）加上训练期间执行的 BPE 合并操作次数。为了提高 BPE 训练的效率，我们不考虑跨越预标记边界的字节对。在计算合并时，如果字节对出现频率相同，则确定性地选择字典序更大的字节对。例如，如果 (“A”, “B”)、(“A”, “C”)、(“B”, “ZZ”) 和 (“BA”, “A”) 这几对数字的频率最高，我们将合并 (“BA”, “A”)：

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

特殊标记 通常，一些字符串（例如，<|endoftext|>）用于编码元数据（例如，文档之间的边界）。在文本编码时，通常需要将某些字符串视为“特殊标记”，这些标记永远不应被拆分为多个标记（即，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内容。这些特殊标记必须添加到词汇表中，以便它们具有对应的固定标记 ID。

Sennrich 等人 [2016] 的算法 1 包含一种效率较低的 BPE 分词器训练实现（基本上遵循我们上面概述的步骤）。作为初步练习，实现并测试此函数可能有助于检验您对该算法的理解。

示例（bpe_example）：BPE 训练示例

以下是 Sennrich 等人 [2016] 提供的一个示例。考虑一个包含以下文本的语料库。

低 低 低 低 低 更低 更低 最宽 最宽 最宽 最新 最新 最新
最新 最新

词汇表有一个特殊标记 <|endoftext|>。

词汇表 我们使用特殊标记 <|endoftext|> 和 256 字节的值初始化词汇表。
预分词 为了简化并专注于合并过程，我们假设本例中预分词仅以空格为分割依据。预分词和计数后，即可得到频率表。

{最低：5，最低：2，最宽：3，最新：6}

[2] 需要注意的是，原始的 BPE 公式 [Sennrich et al., 2016] 规定包含词尾标记。我们在训练字节级 BPE 模型时没有添加词尾标记，因为模型词汇表中包含了所有字节（包括空格和标点符号）。由于我们显式地表示了空格和标点符号，因此学习到的 BPE 合并结果自然会反映这些词边界。

为了方便起见，我们可以将其表示为 dict[tuple[bytes], int]，例如 {(l,o,w): 5 …}。请注意，在 Python 中，即使是单个字节也是一个 bytes 对象。Python 中没有 byte 类型来表示单个字节，就像 Python 中没有 char 类型来表示单个字符一样。

合并操作：我们首先查看每一对连续的字节，并计算它们出现位置对应的单词频率之和 {lo: 7, ow: 7, we: 8, er: 2, wi: 3, id: 3, de: 3, es: 9, st: 9, ne: 6, ew: 6}。由于 ('es') 和 ('st') 的频率相同，因此我们选择字典序更大的 ('st')。然后，我们将前面的标记合并，最终得到 {(l,o,w): 5, (l,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) 是最常见的组合（出现次数为 9），因此我们将合并为 {(l,o,w): 5, (l,o,w,e,r): 2, (w,i,d,est): 3, (n,e,w,est): 6}。继续这样做，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', 'ow', 'l ow', 'west', 'ne']，我们的词汇元素将是 [<|endoftext|>, [...256 字节字符], st, est, ow, low, west, ne]。

使用此词汇表和合并集，单词 newest 将分词为 [ne, west]。

2.5 尝试使用 BPE 分词器进行训练

让我们在 TinyStories 数据集上训练一个字节级 BPE 分词器。数据集的查找/下载说明请参见第 1 节。在开始之前，我们建议您先浏览一下 TinyStories 数据集，以便了解数据内容。

并行化预分词 你会发现预分词步骤是主要的瓶颈。你可以使用内置库 multiprocessing 并行化你的代码来加速预分词。具体来说，我们建议在并行实现预分词时，将语料库分块，并确保块边界位于特定词元的开头。你可以直接使用以下链接中的初始代码来获取块边界，然后利用这些边界将工作分配到各个进程中：

https://github.com/stanford-cs336/assignment1-basics/blob/main/cs336_basics/pretokenization_example.py

这种分块方法始终有效，因为我们永远不希望跨越文档边界进行合并。就本次作业而言，您可以始终按此方式拆分。不必担心收到一个不包含 <|endoftext|> 的大型语料库这种特殊情况。

在预分词之前移除特殊标记 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（或如果您使用并行实现，则从文本块）中移除所有特殊标记。请确保以特殊标记为分割点，以避免跨特殊标记的文本合并。例如，如果您有一个类似 `[Doc 1]<|endoftext|>[Doc 2]` 的语料库（或文本块），则应以特殊标记 `<|endoftext|>` 为分割点，并分别对 `[Doc 1]` 和 `[Doc 2]` 进行预分词，以避免跨文档边界的合并。这可以通过使用 `re.split` 并以 `"|".join(special_tokens)` 作为分隔符来实现（需要谨慎使用 `re.escape`，因为特殊标记中可能包含 `|`）。测试 `test_train_bpe_special_tokens` 将对此进行测试。

优化合并步骤：上述示例中 BPE 训练的简单实现速度较慢，因为每次合并时，它都需要遍历所有字节对来识别出现频率最高的字节对。然而，每次合并后唯一会改变的字节对计数是与合并后的字节对重叠的字节对。因此，可以通过索引所有字节对的计数并逐步更新这些计数来提高 BPE 训练速度，而不是显式地遍历每个字节对来计算其频率。虽然这种缓存方法可以显著提高速度，但需要注意的是，BPE 训练的合并部分在 Python 中无法并行化。

资源不足/缩减规模技巧：性能分析

你应该使用 cProfile 或 scalene 等性能分析工具来识别实现中的瓶颈，并专注于优化这些瓶颈。

资源不足/缩减规模技巧：“缩减规模”

我们建议您不要直接使用完整的 TinyStories 数据集来训练分词器，而是先使用一小部分数据进行训练，即“调试数据集”。例如，您可以使用 TinyStories 的验证集来训练分词器，该验证集包含 2.2 万个文档，而不是 212 万个文档。这体现了一种尽可能缩小规模以加快开发的通用策略：例如，使用更小的数据集、更小的模型规模等等。选择调试数据集或超参数配置的大小需要仔细考虑：调试数据集要足够大，以便与完整配置具有相同的瓶颈（这样您所做的优化才能具有泛化性），但又不能太大，以免运行时间过长。

问题（train_bpe）：BPE 分词器训练（15 分）

交付内容：编写一个函数，给定一个输入文本文件的路径，训练一个（字节级）BPE 分词器。你的 BPE 训练函数应该能够处理（至少）以下输入参数：

input_path: str 包含 BPE 分词器训练数据的文本文件的路径。

vocab_size: int 一个正整数，定义了最大最终词汇表大小（包括初始字节词汇表、合并产生的词汇表项以及任何特殊标记）。
special_tokens: list[str] 要添加到词汇表的字符串列表。这些特殊标记不会影响 BPE 训练。

您的 BPE 训练函数应返回生成的词汇表和合并结果：

vocab: dict[int, bytes] 分词器词汇表，从 int（词汇表中的标记 ID）到 bytes（标记字节）的映射。

merges: list[tuple[bytes, bytes]] 训练过程中生成的 BPE 合并列表。每个列表项

要使用我们提供的测试用例来测试您的 BPE 训练函数，您首先需要在 [adapters.run_train_bpe] 处实现测试适配器。然后，运行 `uv run pytest tests/test_train_bpe.py`。您的实现应该能够通过所有测试。您也可以选择（这可能会耗费大量时间）使用一些系统语言来实现训练方法的关键部分，例如 C++（可以考虑使用 cppyy）或 Rust（使用 PyO3）。如果您这样做，请注意哪些操作需要复制，哪些操作可以直接从 Python 内存读取，并确保提供构建说明，或者确保仅使用 pyproject.toml 文件进行构建。另请注意，大多数正则表达式引擎对 GPT-2 正则表达式的支持并不完善，即使支持，速度也会很慢。我们已经验证 Oniguruma 的速度相当快，并且支持负向先行断言，但 Python 中的 regex 包速度更快。

问题（train_bpe_tinystories）：在 TinyStories 数据集上进行 BPE 训练（2 分）

（a）在 TinyStories 数据集上训练一个字节级 BPE 分词器，最大词汇量为 10,000。确保将 TinyStories <|endoftext|> 特殊标记添加到词汇表中。

将生成的词汇表序列化并合并到磁盘以供进一步检查。训练耗时多少小时和多少内存？词汇表中最长的词元是多少？它有意义吗？

资源要求：⩽ 30 分钟（无 GPU），⩽ 30GB 内存

提示：在预分词期间使用多进程，并结合以下两个事实，您应该能够将 BPE 训练时间缩短到 2 分钟以内：

（a）<|endoftext|>标记用于分隔数据文件中的文档。

（b）在应用 BPE 合并之前，<|endoftext|> 标记作为特殊情况进行处理。

提交内容：一到两句话的回复。

（b）分析你的代码。分词器训练过程中哪个部分耗时最长？

提交内容：一到两句话的回复。

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。和之前一样，我们建议您先查看一下该数据集，以便更好地了解其内容。

问题（train_bpe_expts_owt）：OpenWebText 上的 BPE 训练（2 分）

(a) 使用最大词汇量为 32,000 的 OpenWebText 数据集训练一个字节级 BPE 分词器。将生成的词汇表序列化并合并到磁盘以供进一步检查。词汇表中最长的词元是多少？它有意义吗？

资源要求：⩽ 12 小时（无 GPU），⩽ 100GB 内存

提交内容：一到两句话的回复。

（b）比较 TinyStories 和 OpenWebText 训练得到的分词器，并对比它们之间的区别。

提交内容：一到两句话的回复。

2.6 BPE 分词器：编码和解码

在前一部分作业中，我们实现了一个函数，用于训练一个基于输入文本的 BPE 分词器，从而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，它加载提供的词汇表和合并列表，并使用它们对文本进行编码和解码，生成相应的词元 ID。

2.6.1 文本编码

使用 BPE 进行文本编码的过程与我们训练 BPE 词汇表的过程类似。主要步骤如下：第一步：预分词。我们首先对序列进行预分词，并将每个预分词符表示为 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符内的这些字节合并成词汇元素，每个预分词符独立处理（不跨预分词符边界合并）。

步骤 2：应用合并。然后，我们将 BPE 训练期间创建的词汇元素合并序列，按照相同的创建顺序应用到我们的预标记上。

示例（bpe_encoding）：BPE 编码示例

例如，假设我们的输入字符串是“猫吃了”，我们的词汇表是{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'}，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，我们将查看每个预令牌并应用 BPE 合并。
第一个前缀“the”最初表示为[b't', b'h', b'e']。查看合并列表，我们找到第一个适用的合并是(b't', b'h')，并用它将前缀转换为[b'th', b'e']。然后，我们再次查看合并列表，找到下一个适用的合并是(b'th', b'e')，它将前缀转换为[b'the']。最后，再次查看合并列表，我们发现没有更多合并适用于该字符串（因为整个前缀已合并为一个单独的词元），因此我们完成了 BPE 合并。对应的整数序列为[9]。对剩余的前缀重复此过程，我们看到前缀“cat”在应用 BPE 合并后表示为 [b' c', b'a', b't']，即整数序列 [7, 1, 5]。最后一个前缀“ate”在应用 BPE 合并后表示为 [b' at', b'e']，即整数序列 [10, 3]。因此，输入字符串的最终编码结果为 [9, 7, 1, 5, 10, 3]。

特殊标记。您的分词器应该能够在对文本进行编码时正确处理用户定义的特殊标记（在构建分词器时提供）。

内存方面的考虑。假设我们要对一个无法完全加载到内存中的大型文本文件进行词法分析。为了高效地对这个大型文件（或任何其他数据流）进行词法分析，我们需要将其分割成易于管理的块，并依次处理每个块，从而使内存复杂度保持恒定，而不是与文本大小呈线性关系。在此过程中，我们需要确保词法单元不会跨越块边界，否则我们将得到与直接在内存中对整个序列进行词法分析的简单方法不同的结果。

2.6.2 解码文本

要将一系列整数标记 ID 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（一个字节序列），将它们连接起来，然后将这些字节解码为 Unicode 字符串。请注意，输入的 ID 不一定能映射到有效的 Unicode 字符串（因为用户可以输入任意整数 ID 序列）。如果输入的标记 ID 无法生成有效的 Unicode 字符串，则应将格式错误的字节替换为官方的 Unicode 替换字符 U+FFFD。`bytes.decode` 的 `errors` 参数控制 Unicode 解码错误的处理方式，使用 `errors='replace'` 将自动将格式错误的数据替换为替换标记。

问题（分词器）：实现分词器（15 分）

交付成果：实现一个分词器类，该类接受一个词汇表和一个合并列表，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（如果词汇表中尚不存在，则将其添加到词汇表中）。我们建议使用以下接口：

def __init__(self, vocab, merges, special_tokens=None) 从给定的词汇表构造一个分词器
词汇表、合并列表以及（可选的）特殊标记列表。此函数应接受

有关 Unicode 替换字符的更多信息，请参阅 en.wikipedia.org/wiki/Specials_(Unicode_block)#Replacement_character。

以下参数：

词汇表：dict[int, bytes]

合并：list[tuple[bytes, bytes]]

special_tokens: list[str] | None = None

def from_files(cls, vocab_filepath, merges_filepath, special_tokens=None) 类
该方法根据序列化的词汇表和合并列表（格式与您的 BPE 训练代码输出格式相同）以及（可选的）特殊标记列表构建并返回一个分词器。此方法应接受以下附加参数：

vocab_filepath: str

merges_filepath: str

special_tokens: list[str] | None = None

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

defencode_iterable(self, iterable: Iterable[str]) -> Iterator[int] 给定一个可迭代对象
字符串（例如 Python 文件句柄）返回一个生成器，该生成器会延迟生成标记 ID。这对于高效地对无法直接加载到内存中的大型文件进行标记化是必需的。

def decode(self, ids: list[int]) -> str 将一系列标记 ID 解码为文本。

要使用我们提供的测试用例来测试您的分词器，您首先需要实现测试适配器。
在 [adapters.get_tokenizer] 处。然后，运行 uv run pytest tests/test_tokenizer.py。你的实现应该能够通过所有测试。

2.7 实验

问题（tokenizer_experiments）：分词器实验（4 分）

(a) 从 TinyStories 和 OpenWebText 中分别抽取 10 个文档样本。使用你之前训练的 TinyStories 和 OpenWebText 分词器（词汇量分别为 10K 和 32K），将这些抽取的文档编码为整数 ID。每个分词器的压缩比（字节/词元）是多少？

提交内容：一到两句话的回复。

（b）如果使用 TinyStories 分词器对 OpenWebText 样本进行分词，会发生什么？比较压缩率和/或定性描述所发生的情况。

提交内容：一到两句话的回复。

(c) 估算你的分词器的吞吐量（例如，以字节/秒为单位）。对 Pile 数据集（825GB 文本）进行分词需要多长时间？

提交内容：一到两句话的回复。

(d) 使用 TinyStories 和 OpenWebText 分词器，将各自的训练数据集和开发数据集编码成一系列整数标记 ID。我们稍后将使用这些 ID 来训练语言模型。我们建议将标记 ID 序列化为 uint16 类型的 NumPy 数组。为什么 uint16 是一个合适的选择？

提交内容：一到两句话的回复。

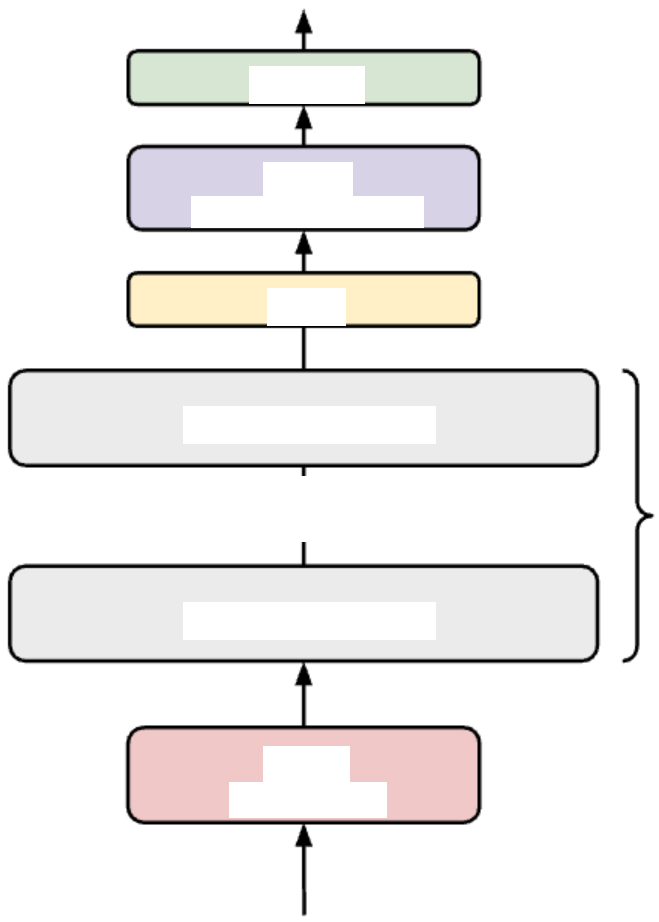

图 1：我们的 Transformer 语言模型概述。

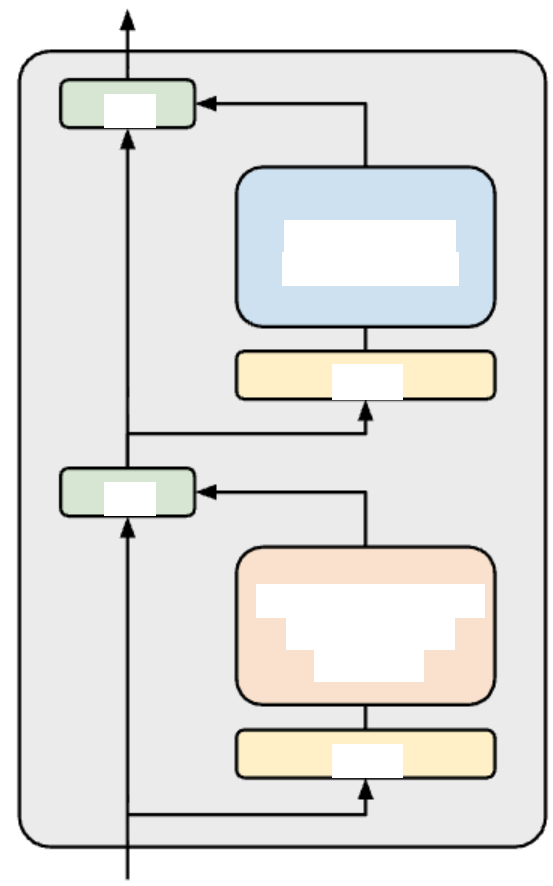

图 2：预范数 Transformer 模块。

3 Transformer 语言模型架构

语言模型以整数标记 ID 的批次序列（即形状为 (batch_size, sequence_length) 的 torch.Tensor）作为输入，并返回词汇表上的（批次）归一化概率分布（即形状为 (batch_size, sequence_length, vocab_size) 的 PyTorch Tensor），其中预测分布针对每个输入标记的下一个词。在训练语言模型时，我们使用这些下一个词的预测结果来计算实际下一个词和预测下一个词之间的交叉熵损失。在推理过程中，我们从语言模型生成文本时，使用最后一个时间步（即序列中的最后一个元素）的预测下一个词分布来生成序列中的下一个标记（例如，通过选择概率最高的标记、从分布中采样等），将生成的标记添加到输入序列中，并重复此过程。

在本部分作业中，你将从零开始构建这个 Transformer 语言模型。我们将首先对模型进行概括性的描述，然后再逐步详细介绍各个组成部分。

3.1 LM 变压器

给定一个词元 ID 序列，Transformer 语言模型使用输入嵌入将词元 ID 转换为稠密向量，然后将嵌入后的词元传递给 num_layers 个 Transformer 模块，最后应用学习到的线性投影（“输出嵌入”或“语言模型头”）生成预测的下一个词元的 logits。示意图见图 1。

3.1.1 词嵌入

在第一步中，Transformer 将（批量）标记 ID 序列嵌入到包含标记身份信息的向量序列中（图 1 中的红色方块）。

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 预规范变换器模块

嵌入之后，激活值由若干个结构相同的神经网络层进行处理。一个标准的仅解码器 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信息，并通过前馈层对其进行非线性变换。

3.2 输出归一化和嵌入

在经过 num_layers 个 Transformer 模块后，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 模块（详见第 3.5 节），该模块还需要在最后一个 Transformer 模块之后使用层归一化（详见下文），以确保其输出得到正确的缩放。

经过归一化之后，我们将使用标准的学习线性变换将 Transformer 模块的输出转换为预测的下一个标记 logits（例如，参见 Radford 等人 [2018] 公式 2）。

3.3 备注：批处理、Einsum 和高效计算

在整个 Transformer 模型中，我们将对多个批次式输入执行相同的计算。以下是一些示例：

- 批次元素：我们对每个批次元素应用相同的 Transformer 前向操作。
- 序列长度：像 RMSNorm 和前馈这样的“按位置”操作对序列的每个位置都执行相同的操作。
- 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量处理。

以符合人体工程学的方式执行此类操作非常有用，这种方式能够充分利用 GPU，并且易于阅读和理解。许多 PyTorch 操作可以在张量的开头接收额外的“批处理”维度，并高效地在这些维度上重复/广播操作。

例如，假设我们要进行一个按位置分批操作。我们有一个形状为 (batch_size, sequence_length, d_model) 的“数据张量”D，并且我们希望对形状为 (d_model, d_model) 的矩阵 A 进行分批向量-矩阵乘法运算。在这种情况下，D @ A 将执行分批矩阵乘法，这是 PyTorch 中的一个高效原语，其中 (batch_size, sequence_length) 维度会被分批处理。

因此，假设你的函数可能会被赋予额外的批处理维度，并将这些维度保留在 PyTorch 形状的开头，会很有帮助。为了组织张量以便以这种方式进行批处理，可能需要使用多次视图、重塑和转置操作来调整它们的形状。这可能会比较麻烦，而且通常难以阅读代码，也难以理解张量的形状。

更符合人体工程学的选择是在 torch.einsum 中使用 einsum 表示法，或者使用像 einops 或 einx 这样与框架无关的库。两个关键操作是 einsum 和 rearrange。einsum 可以对任意维度的输入张量进行张量收缩，而 rearrange 可以对任意维度的张量进行重新排序、连接和拆分。

并可以分割任意维度。事实上，机器学习中的几乎所有操作都是维度转换和张量收缩的某种组合，偶尔还会用到（通常是逐点的）非线性函数。这意味着，使用 einsum 符号可以使你的代码更易读、更灵活。
我们强烈建议本课程学习并使用 einsum 表示法。之前未接触过 einsum 表示法的学生应使用 einops（文档在此），而已经熟悉 einops 的学生则应学习更通用的 einx（文档在此）。我们提供的环境中已预装了这两个软件包。
这里我们提供一些如何使用 einsum 表示法的示例。这些示例是对 einops 文档的补充，您应该先阅读 einops 文档。

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 @ AT
很难分辨输入和输出的形状及其含义。
D 和 A 可以有哪些形状？这些形状中是否存在意想不到的行为？

## Einsum 具有自文档性和鲁棒性 # DA -> Y

Y = einsum(D, A, "批次序列 d_in, d_out d_in -> 批次序列 d_out")

## 或者，批量版本中 D 可以有任意主导维度，但 A 受到约束。
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，我们希望根据某个缩放因子生成 10 个变暗版本：

```
images = torch.randn(64, 128, 128, 3) # (批次, 高度, 宽度, 通道) dim_by = torch.linspace(start=0.0,
end=1.0, steps=10)

## 重塑和倍增
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, "批次高度宽度通道, dim_value -> 批次 dim_value 高度宽度通道")
```

值得注意的是，虽然 einops 拥有大量的支持，但 einx 的实战经验却不如 einops 丰富。如果您在使用 einx 时遇到任何限制或错误，可以随时切换到使用 einops 和更纯粹的 PyTorch 代码。

示例（einstein_example3）：使用 einops.rearrange 进行像素混合

假设我们有一批图像，表示为形状为（批次，高度，宽度，通道）的张量，我们想要对图像的所有像素执行线性变换，但这种变换应该对每个通道独立进行。我们的线性变换表示为形状为（高度×宽度，高度×宽度）的矩阵 B。

channels_last = torch.randn(64, 32, 32, 3) # (批次, 高度, 宽度, 通道) B = torch.randn(32*32, 32*32)

重新排列图像张量，以便在所有像素上进行混合
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 @ BT

channels_last_flat_transformed = channels_first_flat_transformed.transpose(1, 2)

channels_last_transformed = channels_last_flat_transformed.view(*channels_last.shape)

相反，使用 einops：

高度 = 宽度 = 32
重新排列取代笨拙的火炬视图 + 转置
channels_first = 重新排列(
channels_last, "批次高度宽度通道 -> 批次通道（高度宽度）") channels_first_transformed
= einsum(

channels_first, B, "批通道 pixel_in, pixel_out pixel_in -> 批通道 pixel_out") channels_last_transformed =
rearrange(

channels_first_transformed, "批处理通道（高度 宽度）-> 批处理高度 宽度 通道",
height=高度, width=宽度)

或者，如果你想尝试更疯狂的方法：可以使用 einx.dot（einx 中与 einops.einsum 等效的函数）一次性完成所有操作。

height = width = 32 channels_last_transformed =
einx.dot(
"批处理 row_in col_in 通道, (row_out col_out) (row_in col_in)" "-> 批处理 row_out col_out
通道", channels_last, B, col_in=width, col_out=width)

这里的第一个实现可以通过在前后添加注释来改进，以表明……

它能告诉你输入和输出的形状，但这很笨拙，而且容易出错。使用 einsum 表示法，文档本身就是实现！

等价表示法可以处理任意输入批次维度，而且它还有一个关键优势：自文档化。使用等价表示法的代码可以更清晰地展现输入和输出张量的相关形状。对于其他类型的张量，您可以考虑使用张量类型提示，例如使用 jaxtyping 库（并非 Jax 特有）。

我们将在作业 2 中更详细地讨论使用 einsum 符号的性能影响，但现在只需知道它们几乎总是比其他方法更好！

3.3.1 数学符号和内存排序

许多机器学习论文在其表示法中使用行向量，这使得它们的表示方式与 NumPy 和 PyTorch 默认使用的行优先内存顺序非常契合。使用行向量，线性变换看起来像这样

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

对于行优先 $W \in R$ 和行向量 $x \in R$。

在线性代数中，通常更常用列向量，而线性变换看起来像这样

$$y = W x, \tag{2}$$

给定一个行优先的矩阵 $W \in R$ 和一个列向量 $x \in R$。在本作业中，我们将使用列向量进行数学表示，因为这样更容易理解数学推导。需要注意的是，如果要使用普通的矩阵乘法表示法，则必须按照行向量的约定来操作矩阵，因为 PyTorch 使用行优先的内存顺序。如果使用 einsum 进行矩阵运算，则不会出现这个问题。

3.4 基本构建模块：线性模块和嵌入模块

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——错误的初始化会导致梯度消失或梯度爆炸等不良行为。预范数 Transformer 对初始化具有异常强的鲁棒性，但初始化仍然会对训练速度和收敛性产生显著影响。由于本次作业篇幅较长，我们将把细节留到作业 3 中讲解，这里提供一些适用于大多数情况的近似初始化方法。目前，请使用：

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

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

3.4.2 线性模量

线性层是 Transformer 以及神经网络的基本组成部分。首先，你需要实现一个继承自 torch.nn.Module 并执行线性变换的 Linear 类：

$$y = W x。 \tag{3}$$

请注意，我们没有包含偏差项，这与大多数现代线性线性模型一致。

问题（线性）：实现线性模块（1 分）

交付内容：实现一个继承自 torch.nn.Module 的 Linear 类，并执行线性变换。你的实现应遵循 PyTorch 内置 nn.Linear 模块的接口，但无需设置 bias 参数。我们推荐以下接口：

def __init__(self, in_features, out_features, device=None, dtype=None) 构造一个
线性变换模块。此函数应接受以下参数：

in_features：输入的最终维度（整数）

out_features：int 表示输出的最终维度

设备：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。

务必：

- 子类 nn.Module
- 调用超类构造函数
- 出于内存排序的考虑，将参数构造并存储为 W（而非 W'），并将其放入 nn.Parameter 中。

当然，不要使用 nn.Linear 或 nn.functional.linear

初始化时，请使用上述设置以及 `torch.nn.init.trunc_normal_` 来初始化权重。要测试您的线性模块，请在 `[adapters.run_linear]` 中实现测试适配器。该适配器应将给定的权重加载到您的线性模块中。您可以使用 `Module.load_state_dict` 来实现此目的。然后，运行 `uv run pytest -k test_linear`。

3.4.3 嵌入模块

如上所述，Transformer 的第一层是嵌入层，它将整数 token ID 映射到维度为 d_model 的向量空间。我们将实现一个继承自 torch.nn.Module 的自定义 Embedding 类（因此您不应使用 nn.Embedding）。前向传播方法应通过索引形状为 (vocab_size, d_model) 的嵌入矩阵，为每个 token ID 选择嵌入向量。该嵌入矩阵使用形状为 (batch_size, sequence_length) 的 torch.LongTensor 存储 token ID。

问题（嵌入）：实现嵌入模块（1 分）

交付内容：实现继承自 torch.nn.Module 的 Embedding 类，并执行嵌入查找。您的实现应遵循 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) 构造函数
一个嵌入模块。该函数应接受以下参数：

num_embeddings：整数，词汇表的大小

embedding_dim：整数，嵌入向量的维度，即 d。

设备：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, token_ids: torch.Tensor) -> torch.Tensor 查找给定标记 ID 的嵌入向量。

务必：

- 子类 nn.Module
- 调用超类构造函数
- 将嵌入矩阵初始化为 nn.Parameter
- 存储嵌入矩阵，其中 d_model 为最终维度。

当然，不要使用 nn.Embedding 或 nn.functional.embedding。

同样，使用上述设置进行初始化，并使用 torch.nn.init.trunc_normal_ 初始化权重。要测试您的实现，请在 [adapters.run_embedding] 处实现测试适配器。然后，运行 uv run pytest -k test_embedding。

3.5 预规范变换器模块

每个 Transformer 模块有两个子层：多头自注意力机制和位置前馈网络（Vaswani 等人，2017 年，第 3.1 节）。在最初的 Transformer 论文中，模型在每个子层周围使用残差连接，然后进行层归一化。这种架构通常被称为“后归一化”Transformer，因为层归一化应用于子层的输出。然而，大量研究表明，将层归一化从每个子层的输出移到每个子层的输入（并在最后一个 Transformer 模块之后进行额外的层归一化）可以提高 Transformer 的训练稳定性[Nguyen 和 Salazar，2019；Xiong 等人，2020]——图 2 展示了这种“前归一化”Transformer 模块的可视化表示。然后，每个 Transformer 模块子层的输出通过残差连接添加到子层的输入中（Vaswani 等人，2017，第 5.4 节）。预归一化的一个直观理解是，它提供了一个干净的“残差流”，从输入嵌入到 Transformer 的最终输出之间没有任何归一化，据称这可以改善梯度流。这种预归一化 Transformer 现在是语言模型（例如 GPT-3、LLaMA、PaLM 等）的标准，因此我们将实现这个变体。我们将逐步介绍预归一化 Transformer 模块的每个组成部分，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 等人，2016] 对激活值进行归一化。根据 Touvron 等人 [2023] 的做法，我们将使用均方根层归一化（RMSNorm；Zhang 和 Sennrich，2019，公式 4）进行层归一化。给定一个激活值向量 $a \in \mathrm{R}^{d_{\text{model}}}$，RMSNorm 将按如下方式重新缩放每个激活值 a：

$$\mathrm{RMSNorm}(a) = \frac{a}{\mathrm{RMS}(a)} g, \tag{4}$$

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

为了防止在对输入进行平方运算时发生溢出，你应该将输入强制转换为 torch.float32 类型。总的来说，你的前向传播方法应该如下所示：

```
in_dtype = x.dtype x =
x.to(torch.float32)

# 这里是您执行 RMSNorm 的代码...

结果 = ...

# 以原始数据类型返回结果
返回 result.to(in_dtype)
```

问题（rmsnorm）：均方根层归一化（1 分）

交付成果：将 RMSNorm 实现为 torch.nn.Module。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。该函数应接受以下参数：

d_model：整数，模型的隐藏维度

eps：浮点数 = 1e-5，数值稳定性的ε值

设备：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（之后再向下转换为原始数据类型）之前，将输入向上转换为 torch.float32，如上所述。要测试您的实现，请在 [adapters.run_rmsnorm] 处实现测试适配器。然后，运行 uv run pytest -k test_rmsnorm。

3.5.2 位置前馈网络 4 2 0 2 4 x

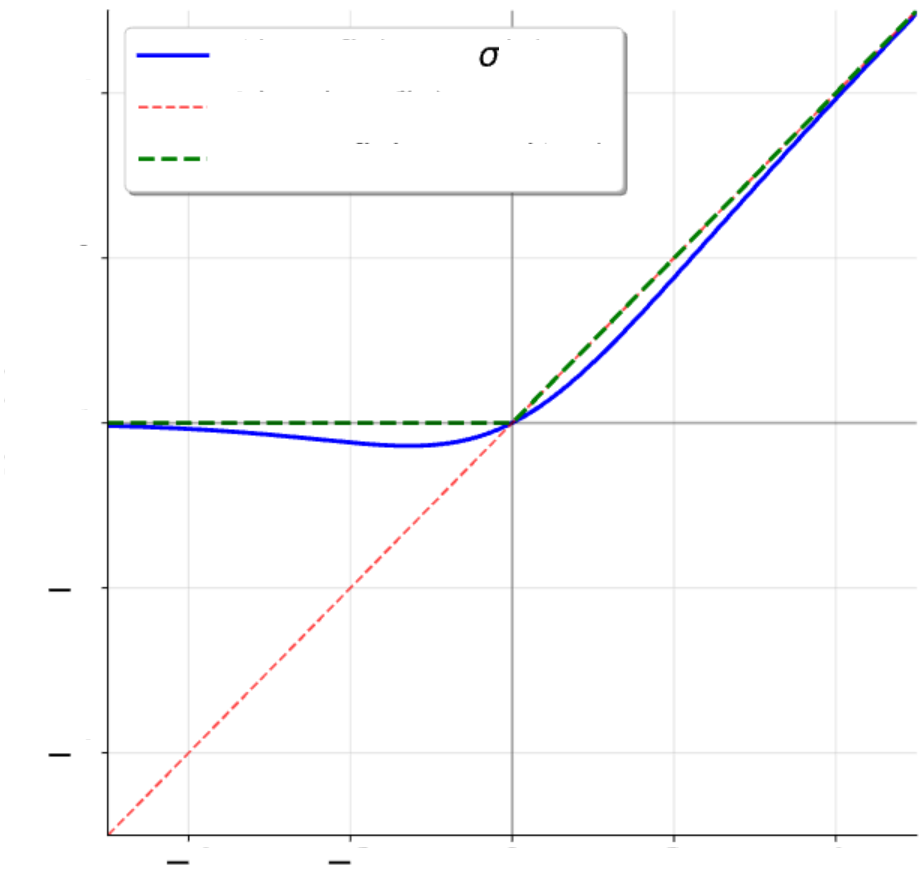

图 3：比较 SiLU（又名 Swish）和 ReLU 激活函数。

在最初的 Transformer 论文（Vaswani 等人[2017]的 3.3 节）中，Transformer 前馈网络由两个线性变换组成，中间使用 ReLU 激活函数（ReLU(x) = max(0, x)）。内部前馈层的维度通常是输入维度的 4 倍。

然而，与最初的设计相比，现代语言模型通常会进行两项主要改进：它们使用不同的激活函数并采用门控机制。具体来说，我们将实现 Llama 3 [Grattafiori et al., 2024] 和 Qwen 2.5 [Yang et al., 2024] 等语言学习模型 (LLM) 中采用的“SwiGLU”激活函数。该函数结合了 SiLU（通常称为 Swish）激活函数和一种称为门控线性单元 (GLU) 的门控机制。此外，我们将省略线性层中常用的偏置项，这与 PaLM [Chowdhery et al., 2022] 和 LLaMA [Touvron et al., 2023] 之后的大多数现代语言学习模型的做法一致。

SiLU 或 Swish 激活函数 [Hendrycks 和 Gimpel，2016，Elfwing 等人，2017] 定义如下：

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

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] 定义为经过 sigmoid 函数的线性变换与另一个线性变换的逐元素乘积：

$$\text{GLU}(x, W, W) = \sigma(Wx) \odot Wx, \tag{6}$$

其中 ⊙ 表示逐元素乘法。门控线性单元旨在“通过为梯度提供线性路径，同时保留非线性能力，来减少深度架构中的梯度消失问题”。

将 SiLU/Swish 和 GLU 结合起来，我们就得到了 SwiGLU，我们将把它用于我们的前馈网络：

$$\text{FFN}(x) = \text{SwiGLU}(x, W, W, W) = W(\text{SiLU}(Wx) \odot Wx), \tag{7}$$

其中 x ∈ R, W, W ∈ R, W ∈ R，并且规范地，d= d。

Shazeer [2020] 首先提出将 SiLU/Swish 激活函数与 GLU 函数相结合，并通过实验证明，在语言建模任务上，SwiGLU 的性能优于 ReLU 和 SiLU（无门控）等基线方法。在后续的作业中，您将比较 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（论文也提供了更多佐证），但保持实证视角仍然很重要：Shazeer 论文中有一句著名的引言是：

> 我们不解释为什么这些架构似乎有效；我们将它们的成功，如同其他一切一样，归功于神的恩惠。

问题（位置前馈网络）：实现位置前馈网络（2 分）

交付成果：实现由 SiLU 激活函数和 GLU 组成的 SwiGLU 前馈网络。
注意：在这种特定情况下，为了保证数值稳定性，您可以在实现中使用 torch.sigmoid。
在你的实现中，你应该将 dto 设置为大约 × 的值，同时确保内部前馈层的维度是 64 的倍数，以便充分利用你的硬件。为了根据我们提供的测试用例测试你的实现，你需要实现
测试适配器位于 [adapters.run_swiglu]。然后，运行 uv run pytest -k test_swiglu 来测试你的实现。

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（Rotary Position Embeddings，RoPE）[Su et al., 2021]。对于给定的查询词元 q = Wx ∈ Rat 词元位置 i，我们将应用成对旋转矩阵 R，得到 q = Rq = RWx。这里，R 将嵌入元素对 q（二维向量）旋转角度 θ =

对于 k ∈ {1, . . . , d/2} 和某个常数 Θ，我们可以将 R 视 $\frac{}{\Theta}$ $\frac{}{}$
为一个 d × d 的分块对角矩阵，其分块为 R，其中 k ∈
{1, . . . , d/2}。

$$R= \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} . \tag{8}$$

由此我们得到完整的旋转矩阵。

$$R= \begin{bmatrix} R & 0 & 0 & \dots & 0 \\ 0 & R & 0 & \dots & 0 \\ 0 & 0 & R & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & R \end{bmatrix}, \tag{9}$$

其中 0 代表 2 × 2 零矩阵。虽然可以构建完整的 d × d 矩阵，但一个好的解决方案应该利用该矩阵的性质来更高效地实现变换。由于我们只关心给定序列中标记的相对旋转，因此我们可以跨层和不同批次复用计算得到的 cos(θ) 和 sin(θ) 值。如果想要优化，可以使用所有层都引用的单个 RoPE 模块，并且该模块可以使用在初始化时通过 `self.register_buffer(persistent=False)` 创建的预计算的 2d sin 和 cos 值缓冲区，而不是使用 `nn.Parameter`（因为我们不想学习这些固定的 cos 和 sin 值）。对 qis 执行的旋转过程与对 k 执行的旋转过程完全相同，旋转角度为相应的 R。请注意，此层没有可学习的参数。

问题（绳索）：实现 RoPE 算法（2 分）

交付成果：实现一个名为 RotaryPositionalEmbedding 的类，该类将 RoPE 应用于输入张量。建议使用以下接口：

def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None) 构造 RoPE 模块，并在需要时创建缓冲区。

theta：绳索的浮点 Θ 值

d_k：查询和键向量的整数维度

max_seq_len: int 输入序列的最大长度

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

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，您应该能够容忍 x 具有任意数量的批次维度。您应该假设标记位置是一个形状为 (..., seq_len) 的张量，它指定了 x 在序列维度上的标记位置。

你应该使用标记位置来沿序列维度对（可能预先计算的）余弦和正弦张量进行切片。

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

3.5.4 缩放点积注意力

现在我们将实现 Vaswani 等人 [2017]（第 3.2.1 节）中描述的缩放点积注意力机制。作为初步步骤，注意力操作的定义将使用 softmax 函数，该函数接受一个未归一化的分数向量，并将其转换为归一化分布：

$$\text{softmax}(v)= \sum \frac{\exp(v)}{j=1\ \exp(v)} \text{——}. \tag{10}$$

注意，当 v 的值很大时，exp(v) 可能变为 inf（此时 inf/inf = NaN）。我们可以通过观察 softmax 运算在所有输入值上加上任意常数 c 不变来避免这种情况。我们可以利用这个性质来保证数值稳定性——通常，我们会从 o 的所有元素中减去 o 的最大值，从而得到新的最大值 0。现在，你将使用这个技巧来实现 softmax 函数，以确保数值稳定性。

问题（softmax）：实现 softmax（1 分）

交付内容：编写一个函数，对张量应用 softmax 操作。该函数应接受两个参数：一个张量和一个维度 i，并对输入张量的第 i 个维度应用 softmax 操作。输出张量应与输入张量具有相同的形状，但其第 i 个维度将具有归一化的概率分布。为了避免数值稳定性问题，请使用从第 i 个维度的所有元素中减去该维度最大值的技巧。

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

现在我们可以用数学方法将注意力操作定义如下：

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

其中 Q ∈ R，K ∈ R，V ∈ R。这里，Q、K 和 V 都是此操作的输入——请注意，这些不是可学习的参数。如果您想知道为什么这不是 QK，请参阅 3.3.1。掩码：有时掩码注意力操作的输出会很方便。掩码的形状应为 M ∈ {True, False}，该布尔矩阵的每一行 i 指示查询 i 应该关注哪些键。通常（但容易混淆），位置 (i, j) 的值为 True 表示查询 i 关注键 j，值为 False 表示查询不关注该键。换句话说，(i, j) 处的“信息流”与值为 True 的键成对出现。例如，考虑一个 1 × 3 的掩码矩阵，其元素为 [[True, True, False]]。单个查询向量仅关注前两个键。

从计算角度来看，使用掩码比在子序列上计算注意力要高效得多，我们可以通过获取预 softmax 值来实现这一点。

$\sqrt{\frac{}{d}}$ · 并在任何条目中添加 $-\infty$

掩码矩阵为 False。

问题（scaled_dot_product_attention）：实现缩放点积注意力机制（5 分）

交付内容：实现缩放点积注意力函数。您的实现应处理形状为 (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 ... 表示任意数量的其他类批次维度（如果提供）。实现应返回形状为 (batch_size, ..., d_v) 的输出。有关类批次维度的讨论，请参阅 3.3 节。您的实现还应支持可选的用户提供的布尔掩码，掩码形状为 (seq_len, seq_len)。掩码值为 True 的位置的注意力概率之和应为 1，掩码值为 False 的位置的注意力概率应为 0。要使用我们提供的测试用例测试您的实现，您需要实现 [adapters.run_scaled_dot_product_attention] 处的测试适配器。

uv run pytest -k test_scaled_dot_product_attention 测试你的实现对三阶输入张量的适用性，而 uv run pytest -k test_4d_scaled_dot_product_attention 测试你的实现对四阶输入张量的适用性。

3.5.5 因果多头自我注意

我们将实现 Vaswani 等人 [2017] 的 3.2.2 节中描述的多头自注意力机制。回顾一下，从数学角度来看，应用多头注意力机制的操作定义如下：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}, ..., \text{head}) \tag{12}$$

$$\text{for head} = \text{Attention}(Q, K, V) \tag{13}$$

其中 Q、K、V 分别为 Q、K 和 V 的嵌入维度，i ∈ {1, ..., h}，切片编号为 d 或 dof。Attention 是 § 3.5.4 中定义的缩放点积注意力操作。由此我们可以构建多头自注意力操作：

$$\text{MultiHeadSelfAttention}(x) = W\text{MultiHead}(Wx, Wx, Wx) \tag{14}$$

这里，可学习的参数是 W ∈ R，W ∈ R，W ∈ R 和 W ∈ R。由于 Q、K 和 V 在多头注意力操作中被切片，我们可以认为 W、W 和 Was 是沿着输出维度为每个头分离的。当此功能正常工作时，您应该总共通过三次矩阵乘法来计算键、值和查询的投影。

作为一项挑战目标，尝试将键、查询和值投影合并到一个权重矩阵中，这样就只需要一次矩阵乘法运算。

因果掩蔽。你的实现应该阻止模型关注序列中后续的词元。换句话说，如果给定一个词元序列 ti, ..., tn，并且我们想要计算前缀 ti, ..., tn（其中 i < n）的下一个词预测，那么模型不应该能够访问（关注）位置 ti, ..., tn 处的词元表示，因为在推理过程中生成文本时，模型无法访问这些词元（而且这些后续词元会泄露关于真实下一个词的信息，从而简化语言建模的预训练目标）。对于输入的词元序列 ti, ..., tn，我们可以通过运行 n 次多头自注意力机制（针对序列中的 n 个唯一前缀）来简单地阻止对后续词元的访问。但是，我们将使用因果注意力掩蔽，它允许词元 i 关注序列中所有位置 j ⩽ i。你可以使用 torch.triu 或广播索引比较来构建此掩码，并且你应该利用 § 3.5.4 中缩放点积注意力实现已经支持注意力掩码这一事实。

应用 RoPE。RoPE 应应用于查询向量和键向量，但不应应用于值向量。此外，应将头部维度视为批次维度，因为在多头注意力机制中，注意力是独立应用于每个头部的。这意味着每个头部的查询向量和键向量都应应用完全相同的 RoPE 旋转。

问题（多头自注意力机制）：实现因果多头自注意力机制（5 分）

交付成果：实现一个 torch.nn.Module 类型的因果多头自注意力机制。你的实现应该（至少）接受以下参数：

d_model：整数，Transformer 模块输入的维度。

num_heads: int 多头自注意力机制中使用的头的数量。

根据 Vaswani 等人 [2017] 的方法，设置 d= d= d/h。为了使用我们提供的测试用例来测试您的实现，请在 [adapters.run_multihead_self_attention] 处实现测试适配器。然后，运行 uv run pytest -k test_multihead_self_attention 来测试您的实现。

3.6 全变形金刚 LM

我们首先来组装 Transformer 模块（参考图 2 会很有帮助）。一个 Transformer 模块包含两个“子层”，一个用于多头自注意力机制，另一个用于前馈网络。在每个子层中，我们首先执行 RMSNorm 操作，然后执行主要操作（MHA/FF），最后添加残差连接。

具体来说，Transformer 模块的第一部分（第一个“子层”）应该实现以下更新，以根据输入 x 生成输出 y，

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

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

按照 § 3.5 中的描述和图 2 中的图示实现预范数 Transformer 模块。您的 Transformer 模块应接受（至少）以下参数。

d_model：整数，Transformer 模块输入的维度。

num_heads: int 多头自注意力机制中使用的头的数量。

d_ff：整数，位置前馈内层的维度。

要测试你的实现，请实现适配器 [adapters.run_transformer_block]。然后运行 uv run pytest -k test_transformer_block 来测试你的实现。

交付成果：通过所提供测试的 Transformer 代码块。

现在我们按照图 1 中的高级图将各个模块组合在一起。按照第 3.1.1 节中对嵌入的描述，将其输入到 num_layers Transformer 模块中，然后将其传递到三个输出层，以获得词汇表的分布。

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

是时候将所有内容整合起来了！请按照 § 3.1 中的描述和图 1 中的图示实现 Transformer 语言模型。您的实现至少应接受 Transformer 模块的所有上述构造参数，以及以下附加参数：

vocab_size: int 词汇表的大小，用于确定词嵌入矩阵的维度。

context_length: int 确定位置嵌入矩阵维度所需的最大上下文长度。

num_layers: int 要使用的 Transformer 模块的数量。

为了使用我们提供的测试用例来测试您的实现，您首先需要实现该测试用例。
适配器位于 [adapters.run_transformer_lm]。然后，运行 uv run pytest -k test_transformer_lm 来测试您的实现。

交付成果：一个通过上述测试的 Transformer LM 模块。

资源核算。了解 Transformer 的各个部分如何消耗计算资源和内存非常有用。我们将逐步介绍一些基本的“浮点运算次数核算”。Transformer 中绝大多数的浮点运算次数都是矩阵乘法，因此我们的核心方法很简单：

1. 写出 Transformer 前向传播过程中所有的矩阵乘法运算。
2. 将每个矩阵乘法运算转换为所需的浮点运算次数。

对于第二步，以下事实将很有用：规则：给定 $A \in R$ 和 $B \in R$，矩阵乘积 AB 需要 2mnp FLOPs。

为了理解这一点，注意到 $(AB)[i, j] = A[i, :] \cdot B[:, j]$，并且这个点积需要 n 次加法和 n 次乘法（2n 次浮点运算）。然后，由于矩阵乘积 AB 有 $m \times p$ 个元素，因此总浮点运算次数为 $(2n)(mp) = 2mnp$。
现在，在解决下一个问题之前，仔细检查一下 Transformer 模块和 Transformer LM 的每个组件，并列出所有矩阵乘法及其相关的 FLOPs 成本，可能会有所帮助。

问题（变压器会计）：变压器 LM 资源会计（5 分）

（a）考虑 GPT-2 XL，其配置如下：

词汇量：50,257

上下文长度：1024

层数：48

d_model：1600

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个可训练参数？假设每个参数都用单精度浮点数表示，那么仅加载此模型就需要多少内存？

提交内容：一到两句话的回复。

(b) 确定完成 GPT-2 XL 型模型前向传播所需的矩阵乘法运算。这些矩阵乘法运算总共需要多少 FLOPs？假设我们的输入序列包含 context_length 个标记。

交付成果：矩阵乘法列表（附说明）以及所需的总 FLOP 数。

（c）根据你上面的分析，模型的哪些部分需要最多的 FLOPs 运算？

提交内容：一到两句话的回复。

(d) 使用 GPT-2 small（12 层，768 d_model，12 个计算头）、GPT-2 medium（24 层，1024 d_model，16 个计算头）和 GPT-2 large（36 层，1280 d_model，20 个计算头）重复分析。随着模型规模的增加，Transformer LM 的哪些部分占用了更多或更少的总 FLOPs？

交付成果：针对每个模型，提供模型组件及其相关浮点运算次数（占前向传播所需总浮点运算次数的比例）的详细分解。此外，请用一到两句话描述模型规模变化如何影响每个组件的浮点运算次数比例。

（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 值如何变化？模型各组成部分的 FLOPs 相对贡献值如何变化？

提交内容：一到两句话的回复。

4 训练 Transformer LM

现在我们已经完成了数据预处理（通过分词器）和模型预处理（Transformer）的步骤。接下来需要编写所有支持训练的代码。这包括以下内容：

- 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：我们需要所有支持基础设施，用于加载数据、保存检查点和管理训练。

4.1 交叉熵损失

回想一下，Transformer 语言模型为每个长度为 m + 1 的序列 x 定义了一个分布 p(x| x)，其中 i = 1, . . . , m。给定一个由长度为 m 的序列组成的训练集 D，我们定义标准的交叉熵（负对数似然）损失函数：

$$\ell(\theta; D) = \frac{1}{|D|m} \sum_{x \in D} \sum_{i=1}^{m} -\log p(x|x)。 \tag{16}$$

（注意，Transformer 中的一次前向传递会得到所有 i = 1, . . . , m 的 p(x| x)。）

具体来说，Transformer 计算每个位置 i 的 logits o ∈ R，结果如下：

$$p(x| x) = \text{softmax}(o)[x] = \frac{\exp(o[x])}{\sum_{a=1}^{\text{vocab\_size}} \exp(o[a])}. \tag{17}$$

交叉熵损失通常是相对于 logits 向量 o ∈ Rand target x 定义的。
实现交叉熵损失函数需要注意一些数值问题，就像 softmax 函数的情况一样。

问题（交叉熵）：实现交叉熵

交付内容：编写一个函数来计算交叉熵损失，该函数接收预测的 logits (o) 和目标值 (x) 作为输入，并计算交叉熵 ℓ = − log softmax(o)[x]。您的函数应处理以下内容：

- 为了数值稳定性，减去最大元素。
- 尽可能抵消对数和经验值。
- 处理任何其他批次维度，并返回该批次的平均值。与第 3.3 节一样，我们假设批次维度始终优先于词汇量大小维度。

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

训练时交叉熵损失足以衡量困惑度，但评估模型时，我们也需要报告困惑度。对于长度为 m 的序列，我们承受交叉熵损失 ℓ, . . . , ℓ:

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

[6] 请注意，o[k] 指的是向量 o 中索引为 k 的值。
这对应于狄拉克δ分布在 x 上与预测的 softmax(o)分布之间的交叉熵。

4.2 随机梯度下降优化器

现在我们有了损失函数，接下来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）。我们首先随机初始化参数θ。然后，对于每个步骤 t = 0, ..., T − 1，我们执行以下更新：

$$\theta \leftarrow \theta - \alpha \nabla L(\theta; B), \tag{19}$$

其中 B 是从数据集 D 中随机抽取的一批数据，学习率 α 和批次大小 |B| 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：
`def __init__(self, params, ...)` 应该初始化你的优化器。这里，`params` 是一个集合。

待优化的参数（或参数组，如果用户希望对模型的不同部分使用不同的超参数，例如学习率）。请务必将参数传递给基类的 `__init__` 方法，该方法会将这些参数存储起来以供后续步骤使用。您可以根据优化器（例如，学习率是一个常见的参数）接受其他参数，并将它们作为字典传递给基类构造函数，其中键是您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。在训练循环期间，将调用此函数。

在反向传播之后，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，它保存基于梯度 p.grad（如果存在）与该参数关联的张量，其中 p.grad 表示损失函数相对于该参数的梯度。

PyTorch 优化器 API 有一些微妙之处，所以用例子来解释会更容易。为了使我们的例子更丰富，我们将实现一个略有不同的 SGD 算法，其中学习率在训练过程中会逐渐衰减，初始学习率 α 随时间推移而逐渐减小：

$$\theta = \theta - \frac{\alpha}{\sqrt{t+1}} \nabla L(\theta; B) \tag{20}$$

让我们看看如何将这个版本的 SGD 实现为 PyTorch 优化器：

```
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):
        如果 lr < 0：
        raise ValueError(f"无效学习率： {lr}") defaults = {"lr": lr}
        super().__init__(params, defaults)


    def step(self, closure: Optional[Callable] = None):
        loss = None if closure is None else closure() for group in
        self.param_groups:
            lr = group["lr"] # 获取学习率。
```

```
for p in group["params"]:
        如果 p.grad 为 None:
            继续

        state = self.state[p] # 获取与 p 关联的状态。
        t = state.get("t", 0) # 从状态中获取迭代次数，或初始值。
        grad = p.grad.data # 获取损失函数相对于 p 的梯度。
        p.data -= lr / math.sqrt(t + 1) * grad # 就地更新权重张量。
        state["t"] = t + 1 # 递增迭代次数。

    回报损失
```

在 `__init__` 方法中，我们将参数以及默认超参数传递给基类构造函数（参数可能以组的形式出现，每组参数的超参数各不相同）。如果参数只是一个 `torch.nn.Parameter` 对象集合，基类构造函数将创建一个参数组并为其分配默认超参数。然后，在 `step` 方法中，我们遍历每个参数组，再遍历该组中的每个参数，并应用公式 20。这里，我们将迭代次数作为与每个参数关联的状态：我们首先读取该值，将其用于梯度更新，然后更新该值。API 规范允许用户传入一个可调用闭包，以便在执行优化器步骤之前重新计算损失。对于我们使用的优化器，我们不需要这样做，但为了符合 API 规范，我们仍然添加了该闭包。

为了了解其工作原理，我们可以使用以下最小训练循环示例：

```
weights = torch.nn.Parameter(5 * torch.randn((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.backward() # 运行反向传播，计算梯度。
        opt.step() # 运行优化器步骤。
```

这是典型的训练循环结构：在每次迭代中，我们会计算损失并运行优化器的一个步骤。在训练语言模型时，可学习参数将来自模型本身（在 PyTorch 中，`m.parameters()` 会返回该参数集合）。损失将基于采样后的数据批次进行计算，但训练循环的基本结构保持不变。

问题（learning_rate_tuning）：调整学习率（1 分）

As we will see, one of the hyperparameters that affects training the most is the learning rate. Let's see that in practice in our toy example. Run the SGD example above with three other values for the learning rate: 1e1, 1e2, and 1e3, for just 10 training iterations. What happens with the loss for each of these learning rates? Does it decay faster, slower, or does it diverge (i.e., increase over the course of training)? ___ _____

Deliverable: A one-two sentence response with the behaviors you observed. ___ _____

4.3 AdamW

Modern language models are typically trained with more sophisticated optimizers, instead of SGD. Most optimizers used recently are derivatives of the Adam optimizer [Kingma and Ba, 2015]. We will use AdamW [Loshchilov and Hutter, 2019], which is in wide use in recent work. AdamW proposes a modification to Adam that improves regularization by adding weight decay (at each iteration, we pull the parameters towards 0), ___ _____

in a way that is decoupled from the gradient update. We will implement AdamW as described in algorithm 2 of Loshchilov and Hutter [2019]. AdamW is stateful: for each parameter, it keeps track of a running estimate of its first and second moments. Thus, AdamW uses additional memory in exchange for improved stability and convergence. Besides the learning rate α, AdamW has a pair of hyperparameters (β, β) that control the updates to the moment estimates, and a weight decay rate λ. Typical applications set (β, β) to (0.9, 0.999), but large language models like LLaMA [Touvron et al., 2023] and GPT-3 [Brown et al., 2020] are often trained with (0.9, 0.95). The algorithm can be written as follows, where ϵ is a small value (e.g., 10) used to improve numerical stability in case we get extremely small values in v:

算法 1 AdamW 优化器 init(θ)（初始化可学习参数）m ← 0（一阶矩向量的初始值；形状与 θ 相同）v ← 0（二阶矩向量的初始值；形状与 θ 相同）

对于 t = 1, . . . , T 执行

样本数据批次 B g ← ∇ℓ(θ; B)（计算当前时间步损失的梯度） m ← βm + (1 − β)g（更新一阶矩估计） v ← βv + (1 − β)g（更新二阶矩估计）

（计算迭代 t 的调整 α） θ ← θ − α（更新参数） θ ← θ − αλθ（应用权重衰减） end for

注意 t 从 1 开始。现在你将实现这个优化器。

问题（AdamW）：实现 AdamW（2 分）

交付内容：实现 AdamW 优化器，作为 torch.optim.Optimizer 的子类。你的类应该在 `__init__` 方法中接收学习率 α，以及超参数 β、ϵ 和 λ。为了帮助你维护状态，基类 Optimizer 提供了一个字典 `self.state`，它将 `nn.Parameter` 对象映射到一个字典，该字典存储了你需要的参数信息（对于 AdamW 来说，这指的是矩估计值）。实现 `[adapters.get_adamw_cls]` 并确保它能通过 `uv run pytest -k test_adamw` 的测试。

问题（adamwAccounting）：使用 AdamW 进行培训的资源会计核算（2 分）

让我们计算一下运行 AdamW 需要多少内存和计算资源。假设我们对每个张量都使用 float32 类型。

(a) 运行 AdamW 需要多少峰值内存？请根据参数、激活值、梯度和优化器状态的内存使用情况分解你的答案。用 batch_size 和模型超参数（vocab_size、context_length、num_layers、d_model、num_heads）表示你的答案。假设 d_ff = 4 × d_model。

为简化起见，在计算激活的内存使用量时，仅考虑以下组成部分：

- 变压器模块 – RMSNorm(s)

– 多头自注意力子层：QKV 投影、QK 矩阵乘法、softmax、加权值求和、输出投影。 – 位置前馈：W 矩阵乘法、SiLU、W 矩阵乘法

- 最终 RMSNorm
- 输出嵌入
- 对数坐标的交叉熵

交付成果：每个参数、激活值、梯度和优化器状态的代数表达式，以及总和的代数表达式。

(b) 将你的答案实例化为 GPT-2 XL 形状的模型，得到一个仅依赖于 batch_size 的表达式。在 80GB 内存范围内，你能使用的最大 batch_size 是多少？

交付物：一个类似于 a · batch_size + b 的表达式，其中 a、b 为数值，b 为表示最大批次大小的数字。

(c) AdamW 运行一步需要多少 FLOPs？

交付成果：一个代数表达式，并附上简要的解释说明。

(d) 模型浮点运算利用率 (MFU) 定义为观测吞吐量（每秒令牌数）与硬件理论峰值浮点运算吞吐量的比值 [Chowdhery et al., 2022]。NVIDIA A100 GPU 的理论峰值浮点运算吞吐量为 19.5 teraFLOP/s（针对 float32 运算）。假设 MFU 为 50%，那么在单个 A100 上训练 GPT-2 XL 模型，训练 40 万步，批大小为 1024，需要多长时间？参考 Kaplan et al. [2020] 和 Hoffmann et al. [2022] 的研究，假设反向传播的浮点运算量是正向传播的两倍。

交付成果：培训所需天数，并简要说明理由。

4.4 学习率调度

训练过程中，能够使损失下降最快的学习率值通常会发生变化。在训练 Transformer 模型时，通常会使用学习率递减策略，即从较大的学习率开始，在初期进行较快的更新，然后随着模型的训练逐渐将其衰减到较小的值。在本作业中，我们将实现用于训练 LLaMA 模型的余弦退火策略 [Touvron et al., 2023]。

调度器就是一个函数，它接受当前步骤 t 和其他相关参数（例如初始学习率和最终学习率），并返回在步骤 t 进行梯度更新时使用的学习率。最简单的调度器是常数函数，它对于任何 t 值都返回相同的学习率。

余弦退火学习率调度算法考虑以下参数：(i) 当前迭代次数 t，(ii) 最大学习率 α，(iii) 最小（最终）学习率 α，(iv) 预热迭代次数 T，以及 (v) 余弦退火迭代次数 T。第 t 次迭代的学习率定义为：

（热身）如果 $t < T$，则 $\alpha = \alpha$。

（余弦退火）若 $T \leqslant t \leqslant T$，则 $\alpha = \alpha + \frac{1}{2}\left(1 + \cos\left(\frac{t-T}{T-T}\pi\right)\right)(\alpha - \alpha)$。

（退火后）如果 $t > T$，则 $\alpha = \alpha$。

[8] 有时，为了帮助学生突破局部最小值，会采用一种让学习率逐步提高（重新启动）的计划。

问题（学习率调度）：实现带预热的余弦学习率调度

编写一个函数，该函数接受 t、α、α、T 和 T 作为参数，并根据上面定义的调度器返回学习率 α。然后实现 [adapters.get_lr_cosine_schedule] 并确保它通过 uv run pytest -k test_get_lr_cosine_schedule 测试。

4.5 渐变裁剪

在训练过程中，我们有时会遇到梯度过大的训练样本，这会导致训练不稳定。为了缓解这个问题，实践中常用的一种技术是梯度裁剪。其基本思想是在每次反向传播之后、进行优化器迭代之前，对梯度的范数进行限制。

给定梯度（对所有参数）g，我们计算其ℓ范数 ∥ g ∥ 。如果该范数小于最大值 M，则 g 保持不变；否则，我们将 g 按 ∥ 因子缩小（此处添加一个较小的ϵ，例如 10，是为了保证数值稳定性）。注意，最终得到的范数将略小于 M。

问题（梯度裁剪）：实现梯度裁剪（1 分）

编写一个实现梯度裁剪的函数。该函数应接受一个参数列表和一个最大ℓ范数。它应该对每个参数的梯度进行原地修改。使用ϵ = 10（PyTorch 的默认值）。然后，实现适配器[adapters.run_gradient_clipping]，并确保它能通过 uv run pytest -k test_gradient_clipping 的测试。

5 训练循环

现在，我们将最终把目前为止构建的主要组件组合在一起：标记化数据、模型和优化器。

5.1 数据加载器

分词后的数据（例如，您在 tokenizer_experiments 中准备的数据）是一个标记序列 $x = (x, \ldots, x)$。即使源数据可能包含多个单独的文档（例如，不同的网页或源代码文件），常见的做法是将所有这些文档连接成一个标记序列，并在它们之间添加分隔符（例如 <|endoftext|> 标记）。

数据加载器将其转换为批次流，其中每个批次包含 B 个长度为 m 的序列，每个序列与其对应的长度也为 m 的下一个标记配对。例如，当 B = 1，m = 3 时，([x, x, x], [x, x, x]) 就是一个可能的批次。

以这种方式加载数据可以简化训练过程，原因有以下几点。首先，任何 $1 \leqslant i < n - m$ 都构成一个有效的训练序列，因此序列采样非常简单。由于所有训练序列的长度相同，无需填充输入序列，从而提高了硬件利用率（同时增大了批大小 B）。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完全加载到内存中的大型数据集变得容易。

问题（数据加载）：实现数据加载功能（2 分）

交付内容：编写一个函数，该函数接受一个 NumPy 数组 x（包含 token ID 的整数数组）、一个 batch_size、一个 context_length 和一个 PyTorch 设备字符串（例如，'cpu' 或 'cuda:0'），并返回一对张量：采样后的输入序列和对应的下一个 token 的目标值。两个张量的形状都应为 (batch_size, context_length)，包含 token ID，并且都应放置在指定的设备上。为了使用我们提供的测试用例测试您的实现，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/缩减配置技巧：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，则需要将数据移动到正确的设备（同样，您以后也应该使用同一设备来训练您的模型）。
如果是 CPU，可以使用“cpu”设备字符串；如果是 Apple Silicon（M* 芯片），可以使用“mps”设备字符串。

有关 MPS 的更多信息，请查看以下资源：

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

如果数据集太大，无法加载到内存中怎么办？我们可以使用名为 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延迟加载文件内容。这样，您就可以“假装”整个数据集都已加载到内存中。NumPy 通过 np.memmap（或者如果您最初使用 np.save 保存了数组，则可以使用 np.load 的 mmap_mode='r' 标志）实现了此功能，它会返回一个类似 NumPy 数组的对象，该对象会在您访问时按需加载条目。在训练期间从数据集（即 NumPy 数组）中采样时，请确保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，具体取决于您保存数组的方式）。同时，请确保指定与要加载的数组匹配的 dtype。明确验证内存映射数据是否正确（例如，不包含超出预期词汇表大小的值）可能会有所帮助。

5.2 检查点

除了加载数据之外，我们还需要保存训练过程中的模型。运行作业时，我们通常希望能够恢复因某种原因（例如作业超时、机器故障等）中途停止的训练。即使一切顺利，我们之后也可能需要访问中间模型（例如，事后研究训练动态、从不同训练阶段的模型中抽取样本等）。

检查点应该包含恢复训练所需的所有状态。我们当然至少需要能够恢复模型权重。如果使用有状态优化器（例如 AdamW），我们还需要保存优化器的状态（例如，对于 AdamW，需要保存矩估计值）。最后，为了恢复学习率调度，我们需要知道停止训练时的迭代次数。PyTorch 可以轻松保存所有这些信息：每个 `nn.Module` 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；我们可以使用其对应的 `load_state_dict()` 方法稍后恢复这些权重。任何 `nn.optim.Optimizer` 也同样如此。最后，`torch.save(obj, dest)` 可以将一个对象（例如，包含某些值的张量的字典，以及像整数这样的常规 Python 对象）保存到文件（路径）或类文件对象中，然后可以使用 `torch.load(src)` 将其加载回内存。

问题（检查点机制）：实现模型检查点机制（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) 应该将所有状态从模型中导出

将前三个参数写入类似文件的对象 out。你可以使用模型和优化器的 state_dict 方法获取它们的相应状态，然后使用 torch.save(obj, out) 将 obj 保存到 out 中（PyTorch 支持路径或类似文件的对象）。通常的做法是将 obj 设置为字典，但只要之后可以加载检查点，你可以使用任何你想要的格式。

此函数需要以下参数：

模型：torch.nn.Module

优化器：torch.optim.Optimizer

迭代次数：整数

输出：字符串 | os.PathLike | typing.BinaryIO | typing.IO[bytes]

def load_checkpoint(src, model, optimizer) 应该从 src（路径或文件）加载检查点

（例如对象），然后从该检查点恢复模型和优化器的状态。你的函数应该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，并使用模型和优化器中的 `load_state_dict` 方法将它们恢复到之前的状态。

此函数需要以下参数：

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

模型：torch.nn.Module

优化器：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，终于到了将你实现的所有组件整合到主训练脚本中的时候了。这样做的好处在于，你可以很方便地使用不同的超参数启动训练（例如，通过命令行参数传递参数），因为之后你会多次执行这些操作来研究不同的选择如何影响训练结果。

问题（training_together）：将它们整合起来（4 分）

交付成果：编写一个脚本，运行训练循环，使用用户提供的输入来训练模型。我们特别建议训练脚本至少应包含以下内容：

- 能够配置和控制各种模型和优化器的超参数。
- 使用 np.memmap 高效加载训练和验证的大型数据集。

- 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] wandb.ai

6 生成文本

现在我们已经可以训练模型了，最后一步就是让模型生成文本。回想一下，语言模型接收一个长度为 (sequence_length) 的（可能是分批处理的）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，用于预测该位置之后的下一个词。接下来，我们将编写一些函数，将其转换为新序列的采样方案。

Softmax 按照标准惯例，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，我们在前面的公式 10 中看到了这一点。

解码：为了从我们的模型生成文本（解码），我们将向模型提供一个前缀标记序列（“提示”），并要求它生成一个词汇表概率分布，用于预测序列中的下一个词。然后，我们将从该词汇表分布中进行采样，以确定下一个输出标记。
具体来说，解码过程的一个步骤应该接收一个序列 x，并通过以下等式返回一个标记 x：

$$P(x= i \mid x) = \sum \frac{\exp(v)}{j \exp(v)}$$

$$v = \text{TransformerLM}(x) \in R$$

其中 TransformerLM 是我们的模型，它以 sequence_length 的序列作为输入，并生成大小为 (sequence_length × vocab_size) 的矩阵，我们取该矩阵的最后一个元素，因为我们正在寻找第 t 个位置的下一个单词预测。

这为我们提供了一个基本的解码器，通过反复从这些一步条件中采样（将我们先前生成的输出标记附加到下一个解码时间步的输入），直到我们生成序列结束标记 <|endoftext|>（或用户指定的最大生成标记数）。

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很低的文本。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，我们使用温度参数 τ 修改 softmax 函数，其中新的 softmax 函数是

$$\text{softmax}(v, \tau) = \sum_{j=1}^{\text{vocab\_size}} \frac{\exp(v/\tau)}{\exp(v/\tau)}. \tag{24}$$

注意，当 $\tau \to 0$ 时，v 的最大元素将占主导地位，softmax 的输出将变成一个集中于该最大元素的独热向量。

其次，另一种技巧是核心采样或 top-p 采样，它通过截断低概率词来修改采样分布。设 q 为一个概率分布，该分布由大小为(vocab_size)的（温度缩放的）softmax 函数得到。使用超参数 p 的核心采样根据以下公式生成下一个词元。

$$P(x= i \mid q) = \begin{cases} \sum_{j \in V(p)} q & \text{if } i \in V(p) \\ 0 & \text{否则} \end{cases}$$

其中 V(p) 是满足以下条件的最小词汇子集：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，即可轻松计算此量。

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 并采样补全，直到遇到 <|endoftext|> 标记）。
- 允许用户控制生成的令牌的最大数量。
- 给定一个期望的温度值，在采样之前对预测的下一个词分布应用 softmax 温度缩放。
- Top-p 抽样（Holtzman 等人，2020；也称为核抽样），给定用户指定的阈值。

7 个实验

现在是时候将所有内容整合起来，并在相关的数据集上训练（小型）语言模型了。

7.1 如何开展实验和交付成果

要理解 Transformer 架构组件背后的原理，最好的方法就是实际修改并运行它。实践经验无可替代。

为此，能够快速、持续地进行实验并记录实验过程至关重要。为了快速进行实验，我们将使用小型模型（1700 万个参数）和简单的数据集（TinyStories）进行大量实验。为了确保实验的一致性，您需要系统地消融模型组件并调整超参数。为了记录实验过程，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，请务必定期评估验证损失，并记录步数和实际运行时间。您可能会发现 Weights and Biases 等日志记录工具很有帮助。

问题（实验日志）：实验日志记录（3 分）

对于您的训练和评估代码，创建实验跟踪基础架构，以便您可以跟踪实验和损失曲线与梯度步长和实际运行时间的关系。

交付成果：记录实验的基础设施代码，以及本节以下作业问题的实验日志（记录您尝试过的所有操作的文档）。

7.2 微型故事

我们将从一个非常简单的数据集（TinyStories；Eldan 和 Li，2023）开始，模型可以快速训练，并且我们可以观察到一些有趣的现象。获取此数据集的说明请参见第 1 节。

下面展示了该数据集的一个示例。

示例（tinystories_example）：来自 TinyStories 的一个示例

从前有个小男孩名叫本。本喜欢探索周围的世界。

他看到了许多令人惊叹的东西，比如商店里陈列的漂亮花瓶。一天，本在商店里闲逛时，偶然发现了一个非常特别的花瓶。本看到它时，简直惊呆了！他说：“哇，这花瓶真漂亮！我可以买吗？”店主笑着说：“当然可以。你可以把它带回家，让你的朋友们都看看它有多漂亮！”于是，本把花瓶带回了家，他为此感到非常自豪！他叫来朋友们，向他们展示这个漂亮的花瓶。他的朋友们都觉得花瓶很漂亮，并且难以置信本竟然如此幸运。这就是本在商店里发现一个漂亮花瓶的故事！

超参数调优 我们将告诉你一些非常基本的超参数，并请你找到一些其他效果良好的设置。

vocab_size 设置为 10000。典型的词汇量在数万到数十万之间。你应该改变这个值，观察词汇量和模型行为的变化。

context_length 256。像 TinyStories 这样的简单数据集可能不需要很长的序列长度，但对于

对于后续的 OpenWebText 数据，您可能需要调整此值。尝试调整此值，并观察其对每次迭代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 个维度略小，但这将使速度更快。

d_ff 1344。这大致等于 d_model，同时也是 64 的倍数，这对 GPU 性能有好处。

RoPE theta 参数 Θ 10000。

层数和磁头数分别为 4 层和 16 个磁头。加起来，这将产生大约 1700 万个非嵌入参数，这是一个相当小的 Transformer 模型。

总共处理了 327,680,000 个令牌（您的批次大小 × 总步数 × 上下文长度应该大致等于此值）。
你应该进行一些尝试和错误，以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（β、β、ϵ）和权重衰减。

您可以在 Kingma 和 Ba [2015] 中找到一些此类超参数的典型选择。

现在，您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，然后在您编写的训练循环中运行它。重要提示：如果您的实现正确且高效，上述超参数在 1 个 H100 GPU 上应该可以运行大约 30-40 分钟。如果您的运行时间更长，请检查并确保您的数据加载、检查点或验证损失代码没有成为运行瓶颈，并确保您的实现已正确批处理。

模型架构调试技巧 我们强烈建议您熟练使用 IDE 的内置调试器（例如 VSCode/PyCharm），与使用 print 语句调试相比，这将节省您的时间。如果您使用文本编辑器，可以使用类似 pdb 的工具。调试模型架构的其他一些良好实践包括：

- 开发任何神经网络架构时，常见的第一步是对单个小批量数据进行过拟合。如果你的实现正确，你应该能够迅速将训练损失降至接近于零。
- 在各种模型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。
- 监控激活值、模型权重和梯度的规范，确保它们不会爆炸或消失。

问题（学习率）：调整学习率（3 分）（4 小时 100 小时）

学习率是需要调整的最重要的超参数之一。基于你训练好的基础模型，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，则记录发散情况）。

交付成果：与多个学习率相关的学习曲线。解释您的超参数搜索策略。

交付成果：一个在 TinyStories 数据集上验证损失（每个 token）至多为 1.45 的模型

低资源/降级训练技巧：先在 CPU 或 Apple Silicon 上进行几步训练。

如果您使用的是 CPU 或 MPS，则应将处理的总标记数减少到 40,000,000，这足以生成较为流畅的文本。您还可以将目标验证损失从 1.45 增加到 2.00。
在配备 36 GB 内存的 M3 Max 芯片上运行我们优化后的学习率解决方案代码时，我们使用的批大小 × 总步数 × 上下文长度 = 32 × 5000 × 256 = 40,960,000 个 token，这在 CPU 上耗时 1 小时 22 分钟，在 MPS 上耗时 36 分钟。在第 5000 步，我们获得了 1.80 的验证损失。

一些补充建议：

- 当使用 X 个训练步骤时，我们建议调整余弦学习率衰减方案，使其在第 X 步恰好终止衰减（即达到最小学习率）。

- 使用 mps 时，不要使用 TF32 内核，即不要设置 torch.set_float32_matmul_precision('high')。

  就像使用 CUDA 设备一样。我们尝试在 mps（torch 版本 2.6.0）中启用 TF32 内核，发现后端会使用一些存在缺陷但未被发现的内核，导致训练不稳定。

- 您可以使用 torch.compile 对模型进行即时编译 (JIT) 来加快训练速度。具体来说，ical：

  - 在 CPU 上，使用以下命令编译您的模型

    model = torch.compile(model)

  在 mps 中，你可以使用以下方法对反向传播进行一定程度的优化：

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

  从 torch 版本 2.6.0 开始，mps 不支持使用 Inductor 进行编译。

（b）民间智慧认为，最佳学习速度处于“稳定边缘”。探究学习速度开始发散的点与你的最佳学习速度之间的关系。

交付成果：学习率递增的学习曲线，其中至少包含一次发散运行，并分析这与收敛率的关系。

45 现在我们来改变批次大小，看看训练会发生什么变化。批次大小很重要——它能让我们通过执行更大的矩阵乘法来提高 GPU 的效率，但我们真的总是需要很大的批次大小吗？让我们做一些实验来找出答案。

问题（批次大小实验）：批次大小变化（1 分）（2 小时 100 小时）

将批处理大小从 1 变化到 GPU 内存限制。尝试几个中间的批处理大小，包括 64 和 128 等常见大小。
交付成果：不同批次大小运行的学习曲线。如有必要，应再次优化学习率。
交付成果：用几句话讨论一下你对批次规模及其对培训的影响的研究结果。

有了解码器，我们现在就可以生成文本了！我们将使用模型生成文本，看看效果如何。作为参考，您应该得到至少与以下示例一样好的输出结果。

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

从前，有一个漂亮的小女孩名叫莉莉。她喜欢吃口香糖，尤其是那种又大又黑的。有一天，莉莉的妈妈让她帮忙做晚饭。莉莉兴奋极了！她喜欢帮妈妈的忙。莉莉的妈妈煮了一大锅汤。莉莉高兴极了，说：“谢谢妈妈！我爱你。”她帮妈妈把汤倒进一个大碗里。晚饭后，莉莉的妈妈又做了一些美味的汤。莉莉很喜欢！她说：“谢谢妈妈！这汤真好喝！”妈妈笑着说：“莉莉，你喜欢就好。”她们一起做完了饭，然后继续一起做饭。故事结束。

低资源/缩减空间技巧：在 CPU 或 Apple Silicon 上生成文本

如果使用低资源配置，处理 4000 万个词元，则生成的文本仍然类似于英语，但不如上述流畅。例如，我们使用 4000 万个词元训练的 TinyStories 语言模型的示例输出如下：

从前，有个小女孩名叫苏。苏有一颗她非常珍爱的牙齿，那是她最喜欢的牙齿。一天，苏出去散步，遇到了一只瓢虫！她们成了好朋友，一起在小路上玩耍。
“嘿，波莉！我们出去吧！”蒂姆说。苏抬头望向天空，发现很难找到一条闪耀的舞步。她笑了笑，答应帮忙说话！

苏看着天空移动，想知道那是什么。

以下是问题的具体描述以及我们的要求：

问题（生成）：生成文本（1 分）

使用解码器和训练好的检查点，报告模型生成的文本。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。
交付内容：至少 256 个文本标记（或直到第一个 <|endoftext|> 标记）的文本转储，以及对该输出流畅度的简要评论，并至少说明影响该输出好坏的两个因素。

7.3 消融术和结构改造

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

消融步骤 1：层归一化 人们常说层归一化对于 Transformer 训练的稳定性至关重要。但或许我们想冒险一试。让我们从每个 Transformer 模块中移除 RMSNorm，看看会发生什么。

问题（layer_norm_ablation）：移除 RMSNorm 并进行训练（1 分）（1 小时 100 小时）

从 Transformer 模型中移除所有 RMSNorm，然后重新训练。在之前的最佳学习率下会发生什么？使用更低的学习率能否获得稳定性？
交付成果：移除 RMSNorms 并进行训练时的学习曲线，以及最佳学习率的学习曲线。

交付成果：对 RMSNorm 的影响进行几句话的评论。

现在我们来探讨另一种乍看之下似乎任意的层归一化选择。预归一化 Transformer 模块定义为：

$$z = x + \text{MultiHeadedSelfAttention}(\text{RMSNorm}(x)) \; y = z + \text{FFN}(\text{RMSNorm}(z))。$$

这是对原始 Transformer 架构为数不多的“共识性”修改之一，该架构采用了一种后范数方法。

$$z = \text{RMSNorm}(x + \text{MultiHeadedSelfAttention}(x)) \; y = \text{RMSNorm}(z + \text{FFN}(z))。$$

让我们回到后规范时代，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 小时 100 分钟）

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

交付成果：后规范时代 Transformer 的学习曲线与前规范时代 Transformer 的学习曲线对比。

我们发现，层归一化对转换器的行为有重大影响，甚至层归一化的位置也很重要。

消融步骤 2：位置嵌入 接下来，我们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（包含 RoPE）和完全不包含位置嵌入的模型（NoPE）。结果表明，仅解码器 Transformer（即我们实现的带有因果掩码的模型）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]。现在，我们将通过实证测试 NoPE 与 RoPE 的性能对比。

问题（no_pos_emb）：实现 NoPE（1 分）（1 小时 100 分钟）

使用 RoPE 修改你的 Transformer 实现，完全移除位置嵌入信息，看看会发生什么。

交付成果：比较 RoPE 和 NoPE 性能的学习曲线。

消融 3：SwiGLU 与 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：

$$\text{FFN}(x) = W\text{SiLU}(Wx)。 \tag{25}$$

回想一下，在我们的 SwiGLU 实现中，我们将内部前馈层的维度设置为大致为 $d = d$（同时确保 $d \bmod 64 = 0$，以便利用 GPU 张量核心）。在您的 FFN 实现中，您应该将 d 设置为 $4 \times d$，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 与 SiLU 的比较（1 分）（1 小时 100 小时）

交付成果：比较 SwiGLU 和 SiLU 前馈网络性能的学习曲线，参数数量大致匹配。

交付成果：用几句话阐述你的发现。

资源不足/降级提示：GPU 资源有限的在线学生应在 TinyStories 上测试修改。

在本作业的剩余部分，我们将转向更大规模、噪声更大的网络数据集（OpenWebText），尝试架构修改，并（可选地）提交到课程排行榜。

在 OpenWebText 上训练语言模型达到流畅水平需要很长时间，因此我们建议 GPU 访问有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 运行于 OpenWebText

接下来，我们将使用通过网络爬虫创建的更标准的预训练数据集。我们还提供了一个 OpenWebText [Gokaslan et al., 2019] 的小样本，它以单个文本文件的形式提供：有关如何访问此文件，请参见第 1 节。

以下是 OpenWebText 中的一个示例。请注意，文本更加真实、复杂和多样化。您可能需要查看训练数据集，以了解网络抓取语料库的训练数据是什么样的。

示例（owt_example）：OWT 的一个示例

《棒球展望》（Baseball Prospectus）的技术总监哈里·帕夫利迪斯（Harry Pavlidis）聘请乔纳森·贾奇（Jonathan Judge）可谓冒了很大的风险。帕夫利迪斯深知，正如艾伦·施瓦茨（Alan Schwarz）在《数字游戏》（The Numbers Game）一书中写道：“在美国文化中，没有哪个领域比棒球运动员的表现更能被精确计算、更被热情地量化。”只需轻点几下鼠标，你就能发现诺亚·辛德加德（Noah Syndergaard）的快速球在飞向本垒板的过程中每分钟旋转超过 2100 圈，尼尔森·克鲁兹（Nelson Cruz）在 2016 年拥有所有合格击球手中最高的平均击球初速，以及其他无数看似来自电子游戏或科幻小说的奇闻轶事。海量数据的涌现赋予了棒球文化中一个日益重要的角色——数据分析爱好者——以力量。

这种权力赋予了他们更多关注——不仅关注数据本身，也关注数据背后的团队和出版物。帕夫利迪斯在创办《棒球展望》（Baseball Prospectus）时，深知量化数据不完善会招致怎样的强烈反对。他也明白，网站的接球数据需要重新调整，而这项工作需要一位博学之士——一位能够解决复杂统计建模问题的人——才能完成。

“他让我们感到不安。”哈里·帕夫利迪斯（Harry Pavlidis）根据贾奇（Judge）的文章以及两人在一次由网站赞助的棒球场活动上的互动，预感到贾奇“明白了”这个职位。此后不久，两人边喝边聊。帕夫利迪斯的直觉得到了证实。贾奇非常适合这个职位——更重要的是，他欣然接受了这份工作。“我和很多人谈过，”帕夫利迪斯说，“他是唯一一个有勇气接受这份工作的人。” [...]

注意：您可能需要针对此实验重新调整超参数，例如学习率或批次大小。

问题（主实验）：OWT 实验（2 分）（3 小时 100 小时）

使用与 TinyStories 相同的模型架构和总训练迭代次数，在 OpenWebText 上训练您的语言模型。该模型的表现如何？

交付成果：你的语言模型在 OpenWebText 上的学习曲线。描述它与 TinyStories 的损失差异——我们应该如何解读这些损失？

交付成果：使用 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？为什么我们使用了与 TinyStories 相同的模型和计算资源，输出质量却更差？

7.5 您自己的修改 + 排行榜

恭喜你走到这一步！你快要完成了！现在，你将尝试改进 Transformer 架构，并看看你的超参数和架构与班上其他同学相比如何。

排行榜规则：除以下限制外，没有其他限制：

运行时间：您的提交在 H100 上最多可以运行 1.5 小时。您可以通过在 slurm 提交脚本中设置 --time=01:30:00 来强制执行此限制。

数据 您只能使用我们提供的 OpenWebText 训练数据集。

除此之外，你可以随心所欲地做任何你想做的事。

如果您正在寻找一些实施方案方面的想法，可以参考以下资源：

- 最先进的开源 LLM 系列，例如 Llama 3 [Grattafiori 等人，2024] 或 Qwen 2.5 [Yang 等人，2024]。
- NanoGPT 速成代码库 (https://github.com/KellerJordan/modded-nanogpt) 中，社区成员发布了许多有趣的修改方案，用于“速成”小规模语言模型预训练。例如，一个可以追溯到 Transformer 原始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 等人 [2017]（第 3.4 节）和 Chowdhery 等人 [2022]（第 2 节））。如果您尝试绑定权重，则可能需要降低嵌入/语言模型头初始化的标准差。

在尝试完整的 1.5 小时运行之前，您需要在 OpenWebText 的一小部分或 TinyStories 上测试这些功能。

需要注意的是，您在此排行榜中发现的一些有效修改方法可能并不适用于更大规模的预训练。我们将在课程的“扩展规律”单元中进一步探讨这一问题。

问题（排行榜）：排行榜（6 分）（10 小时 100 小时）

您将按照上述排行榜规则训练一个模型，目标是在 1.5 小时 100 小时内最大限度地减少语言模型的验证损失。

提交内容：记录的最终验证损失、清晰显示实际运行时间（x 轴）小于 1.5 小时的相关学习曲线，以及您的操作说明。我们希望排行榜提交结果至少能超过 5.0 损失的基准值。请在此处提交至排行榜：https://github.com/stanford-cs336/assignment1-basics-leaderboard。

参考

Ronen Eldan 和 Yuanzhi Li. TinyStories：语言模型可以有多小，才能说连贯的英语？，2023 年。arXiv:2305.07759。

Aaron Gokaslan、Vanya Cohen、Ellie Pavlick 和 Stefanie Tellex。OpenWebText 语料库。http://Skylion007.github.io/OpenWebTextCorpus，2019 年。

Rico Sennrich、Barry Haddow 和 Alexandra Birch。“基于子词单元的罕见词神经机器翻译”。载于 ACL 会议论文集，2016 年。

Changhan Wang、Kyunghyun Cho 和 Jiatao Gu。基于字节级子词的神经机器翻译，2019 年。arXiv:1909.03341。

Philip Gage. 一种新的数据压缩算法。《C 用户杂志》，12(2):23-38，1994 年 2 月。ISSN 0898-9788.

Alec Radford、Jeff Wu、Rewon Child、David Luan、Dario Amodei 和 Ilya Sutskever。语言模型是无监督多任务学习器，2019 年。

Alec Radford、Karthik Narasimhan、Tim Salimans 和 Ilya Sutskever。通过生成式预训练提高语言理解能力，2018 年。

Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Ł ukasz Kaiser 和 Illia Polosukhin。您所需要的就是关注。在过程中。 NeurIPS，2017 年。

Toan Q. Nguyen 和 Julian Salazar. 无泪 Transformers：改进自注意力机制的归一化。载于 IWSWLT 会议论文集，2019。

熊瑞斌、杨云昌、何迪、郑凯、郑树新、邢晨、张惠帅、兰岩岩、王立伟和刘铁岩。 Transformer 架构中的层标准化。在过程中。 ICML，2020 年。

Jimmy Lei Ba、Jamie Ryan Kiros 和 Geoffrey E. Hinton。层归一化，2016 年。arXiv:1607.06450。

雨果·图夫龙、蒂博·拉夫里尔、戈蒂埃·伊扎卡尔、泽维尔·马丁内特、玛丽-安妮·拉肖、蒂莫西·拉克鲁瓦、巴蒂斯特·罗齐埃、纳曼·戈亚尔、埃里克·汉布罗、费萨尔·阿扎尔、奥雷利安·罗德里格斯、阿曼德·朱兰、爱德华·格雷夫和纪尧姆·兰普尔。 Llama：开放高效的基础语言模型，2023 年。arXiv:2302.13971。

张彪和 Rico Sennrich. 均方根层归一化. 2019 年 NeurIPS 会议论文集。

Aaron Grattafiori、Abhimanyu Dubey、Abhinav Jauhri、Abhinav Pandey、Abhishek Kadian、Ahmad AlDahle、Aiesha Letman、Akhil Mathur、Alan Schelten、Alex Vaughan、Amy Yang、Angela Fan、Anirudh Goyal、Anthony Hartshorn、Aobo Yang、Archi Mitra、Archie Sravankumar、Artem Korenev、Arthur Hinsvark、阿伦·拉奥、张阿斯顿、奥瑞利安·罗德里格兹、奥斯汀·格雷格森、艾娃·斯帕塔鲁、巴蒂斯特·罗齐尔、贝瑟尼·拜伦、平·唐、鲍比·陈、夏洛特·考切特、查亚·纳亚克、Chloe Bi、克里斯·马拉、克里斯·麦康奈尔、克里斯蒂安·凯勒、克里斯托弗·图雷、吴春阳、科琳·王、克里斯蒂安·坎顿·费雷尔、赛勒斯尼古拉迪斯、达米安·阿隆修斯、丹尼尔·宋、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·哈普克斯、叶戈尔·拉科姆金、埃哈卜·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·辛奈夫、加布里埃尔 Lee、Georgia Lewis Anderson、Govind Thattai、Graeme Nail、Gregoire Mialon、Guan Pang、Guillem Cucurell、Hailey Nguyen、Hannah Korevaar、胡旭、Hugo Touvron、Iliyan Zarov、Imanol Arrieta Ibarra、Isabel Kloumann、Ishan Misra、Ivan Evtimov、Jack Chang、Jade Copet、Jaewon Lee、Jan Geffert、Jana Vranes、贾森·帕克、杰伊·玛哈多卡、吉特 Shah、Jelmer van der Linde、Jennifer Billock、Jenny Hong、Jenya Lee、Jeremy Fu、Jianfeng Chi、Jianyu

黄建宇、刘嘉文、王杰、于杰曹、乔安娜·比顿、Joe Spisak、Jongsoo Park、Joseph Rocca、Joshua Johnstun、Joshua Saxe、Junteng Jia、Kalyan Vasuden Alwala、Karthik Prasad、Kartikeya Upasani、Kate Plawiak、Ke Li、Kenneth Heafield、Kevin Stone、Khalid El-Arini、Krithika Iyer、 Kshitiz Malik、Kuenley Chiu、Kunal Bhalla、Kushal Lakhotia、Lauren Rantala-Yeary、Laurens van der Maaten、Lawrence Chen、Liang Tan、Liz Jenkins、Louis Martin、Lovish Madaan、Lubo Malo、Lukas Blecher、Lukas Landzaat、Luke de Oliveira、Madeline Muzzi、Mahesh Pasupuleti、Mannat Singh、Manohar 帕鲁里、马辛·卡达斯、玛丽亚 Tsimpoukelli、Mathew Oldham、Mathieu Rita、Maya Pavlova、Melanie Kambadur、Mike Lewis、Min Si、Mitesh Kumar Singh、Mona Hassan、Naman Goyal、Narjes Torabi、Nikolay Bashlykov、Nikolay Bogoychev、Niladri Chatterji、Ning Chang、Olivier Duchenne、Onur Çelebi、Patrick Alrassy、Pengchuan 张,李鹏伟、Petar Vasic、Peter Weng、Prajjwal Bhargava、Pratik Dubal、Praveen Krishnan、Punit Singh Koura、徐朴新、何清、董清晓、Ragavan Srinivasan、Raj Ganapathy、Ramon Calderer、Ricardo Silveira Cabral、Robert Stojnic、Roberta Raileanu、Rohan Maheswari、Rohit Girdhar、Rohit 帕特尔、罗曼·索维斯特、罗尼 Polidoro、Roshan Sumbaly、Ross Taylor、Ruan Silva、Rui Hou、Rui Wang、Saghar Hosseini、Sahana Chennabasappa、Sanjay Singh、Sean Bell、Seohyun Sonia Kim、Sergey Edunov、少良聂、Sharan Narang、Sharath Raparthy、Sheng Shen、Shengye Wan、Shruti Bhosale、Shun 张、Simon Vandenhende、Soumya Batra、斯宾塞·惠特曼, 斯坦·索特拉, 史蒂芬·科洛特, 苏金·古鲁兰甘, 西德尼·博罗金斯基, 塔玛·赫尔曼, 塔拉·福勒, 塔里克·谢沙, 托马斯·乔治欧, 托马斯·夏洛姆, 托比亚斯·斯佩克巴赫, 托多尔·米哈伊洛夫, 肖童, 乌吉瓦尔·卡恩, 维达努吉·戈斯瓦米, 维博·古普塔, 维涅什·拉马纳坦, 维克托·科尔克兹, 文森特·贡盖特、维吉妮·多、维什·沃盖蒂、维托尔·阿尔比罗、弗拉丹·彼得罗维奇、褚伟伟、熊文汉、付文银、惠特尼·梅尔斯、泽维尔·马丁内特、王晓东、王晓芳、谭小青、夏希德、谢新峰、贾旭超、王雪伟、Yaelle Goldschlag、Yashesh Gaur、Yasmine Babaei、Yi Wen、Yismine Babaei、Yi Wen、Yiwen Song、Yuchen 张，李跃，毛宇宁，扎查里·德尔皮埃尔·考德特，严正，陈正兴，佐伊·帕帕基波斯，阿迪亚·辛格，阿尤什·斯里瓦斯塔瓦，阿卜哈·杰因，亚当·凯尔西，亚当·沙恩菲尔德，阿迪蒂亚·甘吉迪，阿道夫·维多利亚，阿胡瓦·金斯坦德，阿杰·梅农，阿杰·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，阿曼达·卡莱特，阿米特·桑加尼，阿莫斯 Teo、阿南·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·古、安德鲁·何、安德鲁·波尔顿、安德鲁·瑞安、安基特·拉姆昌达尼、安妮·东、安妮·弗兰科、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·巴兰贝、阿萨夫·艾森曼、阿扎德·亚兹丹、博·詹姆斯、本·毛雷尔、本杰明·莱昂哈迪、伯尼·黄、贝丝·洛伊德、贝托·德·保拉、巴尔加维·帕兰贾普、刘冰、吴波、倪博宇、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、卡塔琳娜·梅希亚、刘策、王长瀚、金长奎、周潮、胡志明、Ching-Hsiang Chu、Chris Cai、Chris Tindal、Christoph 费希滕霍夫、辛西娅·高、达蒙·西文、达纳·贝蒂、丹尼尔·克雷默、丹尼尔·李、大卫·阿金斯、大卫·徐、大卫·泰图金、迪莉娅·大卫、德维·帕里克、戴安娜·利斯科维奇、迪德姆·福斯、王定康、杜克·勒、达斯汀·霍兰、爱德华·道林、艾莎·贾米尔、伊莱恩·蒙哥马利、埃莱奥诺拉·普雷萨尼、艾米丽·哈恩、艾米丽·伍德、 Eric-Tuan Le、埃里克·布林克曼、埃斯特班·阿尔科特、埃文·邓巴、埃文·斯莫瑟斯、孙飞、菲利克斯·克鲁克、冯田、菲利波斯·科基诺斯、菲拉特·奥兹格内尔、弗朗西斯科·卡吉奥尼、弗兰克·卡纳耶特、弗兰克·塞德、 Gabriela Medina Florez、Gabriella Schwarz、Gada Badeer、Georgia Swee、Gil Halpern、Grant Herman、Grigory Sizov、Guangyi、Zhang、Guna Lakshminarayanan、Hakan Inan、Hamid Shojanazeri、Han Zou、Hannah Wang、Hanwen Zha、Haroun Habeeb、Harrison Rudolph、Helen Suk、Henry Aspegren、Hunter Goldman、詹宏源、 Ibrahim Damlaj、Igor Molybog、Igor Tufanov、Ilias Leontiadis、Irina-Elena Veliche、Itai Gat、Jake Weissman、James Geboski、James Kohli、Janice Lam、Japhet Asher、Jean-Baptiste Gaya、Jeff Marcus、Jeff Tang、Jennifer Chan、Jennyzhen、Jeremy Reizenstein、Jeremy Teboul、Jessica Chung、Jian Jin、杨静怡、乔·卡明斯、乔恩卡维尔、乔恩·夏普德、乔纳森·麦菲、乔纳森·托雷斯、乔什·金斯伯格、王俊杰、吴凯、Kam Hou 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·维拉拉哈万、凯莉·米歇尔娜、李克谦、基兰·贾加德什、黄昆、昆纳尔·查瓦拉、凯尔·黄、陈来琳、拉克希亚·加尔格、 Lavender A、莱安德罗·席尔瓦、李·贝尔、张磊、郭良鹏、于立成、利隆·莫什科维奇、卢卡·韦尔斯特、马迪安·哈布萨、马纳夫·阿瓦拉尼、曼尼什·巴特、马丁纳斯·曼库斯、马坦·哈森、马修·伦尼、马蒂亚斯·雷索、马克西姆·格罗舍夫、马克西姆·瑙莫夫、玛雅·拉蒂、梅根·肯尼利、Miao Liu、Michael L. Seltzer、米哈尔·瓦尔科、米歇尔·雷斯特雷波、米希尔·帕特尔、米克·维亚茨科夫、米卡耶尔·萨姆维利安、迈克·克拉克、迈克·梅西、迈克·王、米克尔·朱伯特·赫莫索、莫·梅塔纳特、穆罕默德

mad Rastegari、Munish Bansal、Nandhini Santhanam、Natascha Parks、Natasha White、Navyata Bawa、Nayan Singhal、Nick Egebo、Nicolas Usunier、Nikhil Mehta、Nikolay Pavlovich Laptev、宁东、Norman Cheng、Oleg Chernoguz、Olivia Hart、Omkar Salpekar、Ozlem Kalinli、Parkin Kent、Parth Parekh、 Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, 钱亮, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li,丽贝卡·霍根、罗宾·巴蒂、 Rocky Wang、Russ Howes、Ruty Rinott、Sachin Mehta、Sachin Siby、Sai Jayesh Bondu、Samyak Datta、Sara Chugh、Sara Hunt、Sargun Dhillon、Sasha Sidorov、Satadru Pan、Saurabh Mahajan、Saurabh Verma、Seiji Yamamoto、Sharadh Ramaswamy、Shaun Lindsay、Shaun Lindsay、Sheng Feng、Shenghao Lin、Shengxin Cindy 查、Shishir Patil、Shiva Shankar、张书强、张书强、Sinong Wang、Sneha Agarwal、Soji Sajuyigbe、Soumith Chintala、Stephanie Max、Stephen Chen、Steve Kehoe、Steve Satterfield、Sudarshan Govindaprasad、Sumit Gupta、Summer Deng、Sungmin Cho、Sunny Virk、Suraj Subramanian、Sy Choudhury、Sydney Goldman、Tal 塔玛拉·雷梅兹、塔玛·格拉泽 Best、Thilo Koehler、Thomas Robinson、李天河、张天军、Tim Matthews、Timothy Chou、Tzook Shaked、Varun Vontimitta、Victoria Ajayi、Victoria Montanez、Vijai Mohan、Vinay Satish Kumar、Vishal Mangla、Vlad Ionescu、Vlad Poenaru、Vlad Tiberiu Mihailescu、Vladimir Ivanov、Wei Li、Wenchen Wang、Wenwen Jiang、 Wes Bouaziz、Will Constable、唐晓成、吴小建、王晓兰、吴希伦、高新波、Yaniv Kleinman、陈彦君、胡野、佳佳、叶琪、李彦达、张一琳、张英、Yossi Adi、Youngjin Nam、Yu、Wang、Yu Zhu、Yuchenhao, 钱云迪、李云禄、何雨子、Zach Rait、Zachary DeVito、Zef Rosnbrick、温兆铎、杨振宇、赵志伟和马志宇。美洲驼 3 群模型，2024 年。URL https://arxiv.org/abs/2407.21783。

杨安安、杨宝松、张北辰、惠斌源、郑博、余博文、李成远、刘大一恒、黄飞、魏浩然、林焕、杨建、涂建宏、张建伟、杨建新、杨家喜、周静仁、林俊阳、党凯、卢克明、包克勤、杨可欣、于乐、李美、薛明峰、张培、朱勤、门锐、林润吉、天浩李、夏廷玉、任兴章、任宣成、杨帆、杨苏、张一昌、万宇、刘玉琼、崔泽宇、张振如和邱子涵。Qwen2.5 技术报告。 arXiv 预印本 arXiv:2412.15115, 2024。

Aakanksha Chowdhery、Sharan Narang、Jacob Devlin、Maarten Bosma、Gaurav Mishra、Adam Roberts、Paul Barham、Hyung Won Chung、Charles Sutton、Sebastian Gehrmann、Parker Schuh、Kensen Shi、Sasha Tsvyashchenko、Joshua Maynez、Abhishek Rao、Parker Barnes、Yi Tay、Noam Shazeer、Vinodkumar Prabhakaran、艾米莉·莱夫、南杜、本·哈钦森、雷纳·波普、詹姆斯·布拉德伯里、雅各布·奥斯汀、迈克尔·伊萨德、盖伊·古尔-阿里、尹鹏程、Toju Duke、安塞姆·列夫斯卡娅、桑杰·格马瓦特、苏尼帕·戴夫、亨利克·米查勒夫斯基、泽维尔·加西亚、韦丹特·米斯拉、凯文·罗宾逊、利亚姆·费杜斯、丹尼·周、达芙妮·伊波利托、大卫·栾、贤泽林、巴雷特·佐夫、亚历山大 Spiridonov、Ryan Sepassi、David Dohan、Shivani Agrawal、Mark Omernick、Andrew M. Dai、Thanumalayan Sankaranarayana Pillai、Marie Pellat、Aitor Lewkowycz、Erica Moreira、Rewon Child、Oleksandr Polozov、Katherine Lee、周宗伟、王学智、Brennan Saeta、Mark Diaz、Orhan Firat、Michele Catasta、Jason 魏、凯西·梅尔-赫尔斯特恩、道格拉斯·艾克、杰夫·迪恩、斯拉夫·彼得罗夫和诺亚·菲德尔。 PaLM：通过路径扩展语言模型，2022 年。

arXiv:2204.02311。

Dan Hendrycks 和 Kevin Gimpel。利用高斯误差线性单元连接非线性和随机正则化器，2016 年。arXiv:1606.08415。

Stefan Elfwing、Eiji Uchibe 和 Kenji Doya。“用于强化学习中神经网络函数逼近的 Sigmoid 加权线性单元”，2017 年。URL：https://arxiv.org/abs/1702.03118。

Yann N. Dauphin、Angela Fan、Michael Auli 和 David Grangier。“使用门控卷积网络进行语言建模”，2017 年。URL：https://arxiv.org/abs/1612.08083。

Noam Shazeer. GLU 变体改进 transformer，2020。arXiv:2002.05202。

苏建林，卢宇，潘胜峰，文博，刘云峰。 Roformer：具有旋转位置嵌入的增强型变压器，2021 年。

Diederik P. Kingma 和 Jimmy Ba. Adam：一种随机优化方法。ICLR 会议论文集，2015 年。

Ilya Loshchilov 和 Frank Hutter. 解耦权重衰减正则化. 2019 年 ICLR 会议论文集。

Tom B. Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel M. Ziegler、Jeffrey Wu、Clemens Winter、Christopher Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。“语言模型是少样本学习器”。载于 NeurIPS 会议论文集，2020 年。

Jared Kaplan、Sam McCandlish、Tom Henighan、Tom B. Brown、Benjamin Chess、Rewon Child、Scott Gray、Alec Radford、Jeffrey Wu 和 Dario Amodei。神经语言模型的缩放定律，2020 年。
arXiv:2001.08361。

Jordan Hoffmann、Sebastian Borgeaud、Arthur Mensch、Elena Buchatskaya、Trevor Cai、Eliza Rutherford、Diego de Las Casas、Lisa Anne Hendricks、Johannes Welbl、Aidan Clark、Tom Hennigan、Eric Noland、Katie Millican、George van den Driessche、Bogdan Damoc、Aurelia Guy、Simon Osindero、Karen Simonyan、Erich Elsen、Jack W. Rae、Oriol Vinyals 和 Laurent Sifre。“训练计算最优的大型语言模型”，2022 年。arXiv:2203.15556。

Ari Holtzman、Jan Buys、Li Du、Maxwell Forbes 和 Yejin Choi。“神经文本退化的奇特案例”。载于 ICLR 会议论文集，2020 年。

蔡耀宏、白少杰、山田诚、路易斯-菲利普·莫伦西和鲁斯兰·萨拉胡迪诺夫。
Transformer 剖析：通过内核视角对 Transformer 的注意力进行统一理解。载于 Kentaro Inui、Jing Jiang、Vincent Ng 和 Xiaojun Wan 编辑的《2019 年自然语言处理实证方法会议暨第九届自然语言处理国际联合会议论文集 (EMNLP-IJCNLP)》，第 4344 – 4353 页，中国香港，2019 年 11 月。

计算语言学协会。doi：10.18653/v1/D19-1443。URL https://aclanthology.org/D19-1443/。

Amirhossein Kazemnejad、Inkit Padhi、Karthikeyan Natesan、Payel Das 和 Siva Reddy。“位置编码对 Transformer 中长度泛化的影响”。载于 2023 年第三十七届神经信息处理系统会议。网址：https://openreview.net/forum?id=Drrl2gcjzl。