CS336 作业 1（基础）：构建 Transformer 语言模型

版本 1.0.4
CS336 教学团队
2025 年春季

1 作业概述

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。

您将实现什么

1. Byte-pair encoding (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)

您将运行什么

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 语言模型。
4. 使用训练好的 Transformer LM 生成样本并评估困惑度 (perplexity)。
5. 在 OpenWebText 上训练模型，并将您获得的困惑度提交到排行榜。

您可以使用什么

我们期望您从头开始构建这些组件。特别地，您不得使用 `torch.nn`、`torch.nn.functional` 或 `torch.optim` 中的任何定义，但以下情况除外：

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

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

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损害本作业“从零开始”的精神。

关于 AI 工具的声明

允许使用像 ChatGPT 这样的 LLM 来提问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 自动完成功能（例如，Cursor Tab、GitHub CoPilot）（但非 AI 自动完成，例如自动完成函数名是完全可以的）。我们发现 AI 自动完成会使深入理解内容变得更加困难。

代码结构

所有作业代码以及本说明文档都可以在 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，以便您可以通过 `git pull` 获取最新版本。

1. `cs336_basics/*`: 这是您编写代码的地方。请注意，这里没有任何代码——您可以从零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。对于每个功能（例如，scaled dot product attention），通过简单调用您的代码来填写其实现（例如，`run_scaled_dot_product_attention`）。注意：您对 `adapters.py` 的更改不应包含任何实质性逻辑；这只是粘合代码。
3. `test_*.py`: 这包含了您必须通过的所有测试（例如，`test_scaled_dot_product_attention`），这些测试将调用 `adapters.py` 中定义的钩子。请勿编辑测试文件。

如何提交

您将向 Gradescope 提交以下文件：

- **`writeup.pdf`**: 回答所有书面问题。请使用排版软件撰写您的回答。
- **`code.zip`**: 包含您编写的所有代码。

要提交到排行榜，请向以下仓库提交一个 PR：

`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`。

在哪里获取数据集

本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]。两个数据集都是单一、大型的纯文本文件。如果您是本课程的学生，您可以在任何非头节点机器的 `/data` 目录下找到这些文件。如果您在家中学习，可以使用 `README.md` 中的命令下载这些文件。

低资源/缩减提示：初始化

在整个课程的作业讲义中，我们将提供关于如何在 GPU 资源较少或没有 GPU 资源的情况下完成作业部分的建议。例如，我们有时会建议缩减您的数据集或模型大小，或者解释如何在 MacOS 集成 GPU 或 CPU 上运行训练代码。您会在一个蓝色框（如此框）中找到这些“低资源提示”。即使您是注册的斯坦福学生，可以使用课程机器，这些提示也可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/缩减提示：在 Apple Silicon 或 CPU 上的作业 1

使用我们的参考解决方案代码，我们可以在配备 36 GB RAM 的 Apple M3 Max 芯片上训练一个 LM 来生成相当流畅的文本，在 Metal GPU (MPS) 上不到 5 分钟，使用 CPU 大约需要 30 分钟。如果您对这些术语不太了解，别担心！只需知道，如果您有一台相当新的笔记本电脑，并且您的实现是正确且高效的，您将能够训练一个小型 LM 来生成具有相当流畅度的简单儿童故事。在作业的后面部分，我们将解释如果您使用 CPU 或 MPS，需要进行哪些更改。

2 Byte-Pair Encoding (BPE) 分词器

在作业的第一部分，我们将训练和实现一个字节级 (byte-level) 的 byte-pair encoding (BPE) 分词器 [Sennrich et al., 2016, Wang et al., 2019]。特别地，我们将把任意（Unicode）字符串表示为字节序列，并在这个字节序列上训练我们的 BPE 分词器。稍后，我们将使用这个分词器将文本（字符串）编码为词符（token，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码点 (code points)。截至 Unicode 16.0（2024 年 9 月发布），该标准定义了跨越 168 种文字的 154,998 个字符。例如，字符"s" 的码点是 115（通常表示为 U+0073，其中 U+ 是常规前缀，0073 是十六进制的 115），字符 "牛" 的码点是 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()` 函数将一个整数 Unicode 码点转换为带有相应字符的字符串。

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

问题 (1) unicode1: 理解 Unicode (1 分)

(a) `chr(0)` 返回哪个 Unicode 字符?
可交付成果 (1.1): 一句回答。

(b) 这个字符的字符串表示 (`__repr__()`) 与其打印表示有何不同?
可交付成果 (1.2): 一句回答。

(c) 当这个字符出现在文本中时会发生什么? 在您的 Python 解释器中尝试以下代码，看看是否符合您的预期，这可能会有所帮助:

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

```

可交付成果 (1.3): 一句回答。

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，因为词汇表会非常庞大（大约 15 万项）且稀疏（因为许多字符非常罕见）。相反，我们将使用 Unicode 编码，它将一个 Unicode 字符转换为一个字节序列。Unicode 标准本身定义了三种编码：UTF-8、UTF-16 和 UTF-32，其中 UTF-8 是互联网上占主导地位的编码（超过所有网页的 98

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，我们可以迭代它（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! こんにちは!"
>>> 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))
<class 'bytes'>
>>> # 获取编码字符串的字节值（0 到 255 之间的整数）。
>>> 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! こんにちは!
```

通过将我们的 Unicode 码点转换为字节序列（例如，通过 UTF-8 编码），我们实际上是将一个码点序列（范围在 0 到 154,997 之间的整数）转换为一个字节值序列（范围在 0 到 255 之间的整数）。长度为 256 的字节词汇表更容易处理。当使用字节级分词时，我们不需要担心词汇表外（out-of-vocabulary）的词符，因为我们知道任何输入文本都可以表示为 0 到 255 之间的整数序列。

问题 (2) unicode2: Unicode 编码 (3 分)

(a) 相比于在 UTF-16 或 UTF-32 上训练分词器，在 UTF-8 编码的字节上训练有哪些优势？比较不同输入字符串在这些编码下的输出可能会有所帮助。

可交付成果 (2.1): 一到两句回答。

(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？提供一个会导致不正确结果的输入字节字符串示例。

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

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

```
'hello'

```

可交付成果 (2.2): 一个导致 `decode_utf8_bytes_to_str_wrong` 产生不正确输出的示例输入字节字符串，并用一句话解释为什么该函数不正确。

(c) 给出一个不能解码为任何 Unicode 字符的双字节序列。
可交付成果 (2.3): 一个示例，并附带一句解释。

2.3 子词 (Subword) 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极长。这会减慢模型训练速度，因为模型的每一步都需要更多的计算。此外，字节序列上的语言建模很困难，因为较长的输入序列会在数据中产生长期依赖关系。

子词分词是词级分词器和字节级分词器之间的中间点。请注意，字节级分词器的词汇表有 256 个条目（字节值 0 到 225）。子词分词器通过牺牲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，那么为其在词汇表中分配一个条目将把这个 3 个词符的序列减少为单个词符。

我们如何选择这些子词单元添加到我们的词汇表中呢? Sennrich et al. [2016] 提出使用 byte-pair encoding (BPE; Gage, 1994)，这是一种压缩算法，它迭代地用一个新的、未使用的索引替换（“合并”）最频繁的字节对。请注意，该算法向我们的词汇表添加子词词符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足够多次，它将被表示为单个子词单元。

使用通过 BPE 构建的词汇表的子词分词器通常称为 BPE 分词器。在本作业中，我们将实现一个字节级的 BPE 分词器，其中词汇表项是字节或合并后的字节序列，这在处理词汇表外问题和可管理的输入序列长度方面为我们提供了两全其美的优势。构建 BPE 分词器词汇表的过程称为“训练” BPE 分词器。

2.4 BPE 分词器训练

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从字节串词符到整数 ID 的一对一映射。由于我们正在训练一个字节级的 BPE 分词器，我们最初的词汇表就是所有字节的集合。由于有 256 个可能的字节值，我们最初的词汇表大小为 256。

预分词 (Pre-tokenization) 一旦有了词汇表，原则上您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，这在计算上非常昂贵，因为每次合并我们都必须对语料库进行一次完整的遍历。此外，直接在整个语料库中合并字节可能会导致仅在标点符号上有所不同的词符（例如，`dog!` vs. `dog.`）。这些词符会获得完全不同的词符 ID，即使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不同）。

为了避免这种情况，我们对语料库进行预分词。您可以将其视为对语料库进行的粗粒度分词，帮助我们计算字符对出现的频率。例如，单词 `'text'` 可能是一个出现 10 次的预分词词符。在这

种情况下，当我们计算字符 't' 和 'e' 相邻出现的频率时，我们会看到单词 'text' 中 't' 和 'e' 是相邻的，我们可以将其计数增加 10，而不是遍历整个语料库。由于我们正在训练一个字节级的 BPE 模型，每个预分词词符都表示为一个 UTF-8 字节序列。

Sennrich et al. [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；Radford et al., 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+"""
```

使用这个预分词器交互式地分割一些文本，以更好地了解其行为，可能会有所帮助：

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

然而，在您的代码中使用它时，您应该使用 `re.finditer` 来避免在构建从预分词词符到其计数的映射时存储预分词后的单词。

计算 BPE 合并现在我们已经将输入文本转换为预分词词符，并将每个预分词词符表示为 UTF-8 字节序列，我们可以计算 BPE 合并（即训练 BPE 分词器）。在高层次上，BPE 算法迭代地计算每对字节的计数，并识别出频率最高的一对（“A”，“B”）。这个最频繁对（“A”，“B”）的每次出现随后被合并，即替换为一个新的词符”AB”。这个新的合并词符被添加到我们的词汇表中；因此，BPE 训练后的最终词汇表大小是初始词汇表的大小（在我们的例子中是 256），加上在训练期间执行的 BPE 合并操作的数量。为了提高 BPE 训练的效率，我们不考虑跨越预分词词符边界的对。[2] 在计算合并时，通过选择字典序较大的对来确定性地解决频率并列问题。例如，如果对（“A”，“B”），（“A”，“C”），（“B”，“ZZ”），和（“BA”，“A”）都具有最高频率，我们将合并（“BA”，“A”）：

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

特殊词符 (Special tokens) 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）。在编码文本时，通常希望将某些字符串视为“特殊词符”，它们永远不应被分割成多个词符（即，将始终保留为单个词符）。例如，序列结束字符串 `<|endoftext|>` 应始终保留为单个词符（即单个整数 ID），这样我们就知道何时停止从语言模型生成。这些特殊词符必须添加到词汇表中，以便它们具有相应的固定词符 ID。

Sennrich et al. [2016] 的算法 1 包含了一个低效的 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会有所帮助。

[2]请注意，原始的 BPE 公式 [Sennrich et al., 2016] 指定了包含词尾标记。我们在训练字节级 BPE 模型时不添加词尾标记，因为所有字节（包括空格和标点符号）都包含在模型的词汇表中。由于我们明确表示空格和标点符号，学习到的 BPE 合并将自然地反映这些词边界。

示例 (1) bpe_example: BPE 训练示例 这是来自 Sennrich et al. [2016] 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符 `<|endoftext|>`。

词汇表我们用特殊词符 `<|endoftext|>` 和 256 个字节值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，我们在此示例中假设预分词只是按空格分割。当我们进行预分词和计数时，我们得到频率表。

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

将其表示为 `dict[tuple[bytes], int]` 很方便，例如 `{(l,o,w): 5 ...}`。请注意，即使是单个字节在 Python 中也是一个 `bytes` 对象。Python 中没有字节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。

合并我们首先查看每个连续的字节对，并对它们出现的单词频率求和 `{lo: 7, ow: 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，因此我们取字典序较大的对 `('st')`。然后我们将合并预分词词符，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)` 是最常见的对（计数为 9），我们将合并成 `{(l,o,w): 5, (l,o,w,e,r): 2, (w,i,d,est): 3, (n,e,w,est): 6}`。继续这样操作，我们最终得到的合并序列将是 `['st', 'e st', 'o w', 'l ow', 'west', 'n e', 'ne west', 'wi', 'wi d', 'wid est', 'low e', 'lowe 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 `[<|endoftext|>, [...256 BYTE CHARS], st, est, ow, low, west, ne]`。有了这个词汇表和合并集，单词 `newest` 将被分词为 `[ne, west]`。

2.5 使用 BPE 分词器训练进行实验

让我们在 TinyStories 数据集上训练一个字节级的 BPE 分词器。找到/下载数据集的说明可以在第 1 节中找到。在开始之前，我们建议您查看一下 TinyStories 数据集，了解数据的概况。

并行化预分词您会发现一个主要的瓶颈是预分词步骤。您可以使用内置库 `multiprocessing` 并行化您的代码来加速预分词。具体来说，我们建议在预分词的并行实现中，对语料库进行分块，同时确保您的块边界出现在特殊词符的开头。您可以自由地使用以下链接处的入门代码来获取块边界，然后您可以使用这些边界来跨进程分发工作：

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

这种分块方式始终有效，因为我们从不希望跨文档边界进行合并。就本作业而言，您总是可以这样分割。不用担心接收到一个不包含 `<|endoftext|>` 的非常大的语料库的边缘情况。

预分词前移除特殊词符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(或者您的块,如果使用并行实现)中剥离所有特殊词符。确保您在特殊词符上进行分割，这样就不会在它们界定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（或块），您应该在特殊词符 `<|endoftext|>` 上分割，并分别对 `[Doc 1]` 和 `[Doc 2]` 进行预分词，这样就不会跨文档边界发生合并。这可以使用 `re.split` 完成，使用 `"|".join(special_tokens)` 作为分隔符（注意小心使用 `re.escape`，因为 `|` 可能出现在特殊词符中）。测试 `test_train_bpe_special_tokens` 将对此进行测试。

优化合并步骤上面风格化示例中 BPE 训练的朴素实现很慢，因为对于每次合并，它都会遍历所有字节对以识别最频繁的对。然而，每次合并后唯一改变计数的对是那些与合并后的对重叠的对。因此，可以通过索引所有对的计数并增量更新这些计数，而不是显式地迭代每个字节对来计算对频率，从而提高 BPE 训练速度。您可以通过这种缓存过程获得显著的加速，尽管我们注意到 BPE 训练的合并部分在 Python 中是不可并行化的。

低资源/缩减提示：分析

您应该使用像 `cProfile` 或 `scalene` 这样的分析工具来识别您实现中的瓶颈，并专注于优化它们。

低资源/缩减提示："缩减"

我们建议您不要直接在完整的 TinyStories 数据集上训练分词器，而是首先在一个小的数据子集上训练：一个"调试数据集"。例如，您可以改为在 TinyStories 验证集上训练您的分词器，该验证集有 2.2 万个文档，而不是 212 万个。这说明了一种通用的策略，即尽可能地缩减以加速开发：例如，使用更小的数据集、更小的模型大小等。选择调试数据集的大小或超参数配置需要仔细考虑：您希望您的调试集足够大，以至于具有与完整配置相同的瓶颈（以便您所做的优化能够泛化），但又不能太大，以至于运行时间过长。

问题 (3) train_bpe: BPE 分词器训练 (15 分)

可交付成果 (3.1): 编写一个函数，给定输入文本文件的路径，训练一个（字节级）BPE 分词器。您的 BPE 训练函数应至少处理以下输入参数：

`input_path: 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，定义最终词汇表的最大大小（包括初始字节词汇表、合并产生的词汇表项和任何特殊词符）。

`special_tokens: list[str]` 要添加到词汇表中的字符串列表。这些特殊词符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：

`vocab: dict[int, bytes]` 分词器词汇表，一个从 `int`（词汇表中的词符 ID）到 `bytes`（词符字节）的映射。

`merges: list[tuple[bytes, bytes]]` BPE 合并的列表，由训练产生。每个列表项是一个字

节元组 (`<token1>`, `<token2>`)，表示 `<token1>` 与 `<token2>` 合并。合并应按创建顺序列出。

要根据我们提供的测试来测试您的 BPE 训练函数，您首先需要实现位于 `[adapters.run_train_bpe]` 的测试适配器。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。可选地（这可能是一个巨大的时间投入），您可以使用某种系统语言（例如 C++（考虑为此使用 `cppyy`）或 Rust（使用 `PyO3`））来实现训练方法的关键部分。如果您这样做，请注意哪些操作需要从 Python 内存复制与直接读取，并确保留下构建说明，或确保它仅使用 `pyproject.toml` 进行构建。另请注意，GPT-2 正则表达式在大多数正则表达式引擎中不受良好支持，并且在大多数支持它的引擎中会太慢。我们已经验证 Oniguruma 相当快并且支持负向先行断言，但 Python 中的 `regex` 包（如果有的话）甚至更快。

问题 (4) train_bpe_tinystories: 在 TinyStories 上训练 BPE (2 分)

(a) 在 TinyStories 数据集上训练一个字节级的 BPE 分词器，最大词汇表大小为 10,000。确保将 TinyStories 的特殊词符 `<|endoftext|>` 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？词汇表中最长的词符是什么？这有意义吗？**资源要求：** < 30 分钟 (无 GPU)， 30GB RAM **提示**您应该能够在预分词期间使用 `multiprocessing` 以及以下两个事实，在 2 分钟内完成 BPE 训练：

(a) `<|endoftext|>` 词符在数据文件中界定文档。

(b) 在应用 BPE 合并之前，`<|endoftext|>` 词符作为特殊情况处理。

可交付成果 (4.1): 一到两句回答。

(b) 分析您的代码。分词器训练过程的哪个部分耗时最多？

可交付成果 (4.2): 一到两句回答。

接下来，我们将尝试在 OpenWebText 数据集上训练一个字节级的 BPE 分词器。和以前一样，我们建议先查看一下数据集以更好地了解其内容。

问题 (5) 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的 BPE 分词器，最大词汇表大小为 32,000。将生成的词汇表和合并序列化到磁盘以供进一步检查。词汇表中最长的词符是什么？这有意义吗？**资源要求：** < 12 小时 (无 GPU)， 100GB RAM

可交付成果 (5.1): 一到两句回答。

(b) 比较和对比您在 TinyStories 与 OpenWebText 上训练得到的分词器。

可交付成果 (5.2): 一到两句回答。

2.6 BPE 分词器：编码和解码

在作业的前一部分，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，它加载提供的词汇表和合并列表，并使用它们将文本编码为词符 ID 以及将词符 ID 解码回文本。

2.6.1 编码文本

使用 BPE 编码文本的过程反映了我们训练 BPE 词汇表的方式。有几个主要步骤。**第 1 步：预分词**。我们首先对序列进行预分词，并将每个预分词词符表示为 UTF-8 字节序列，就像我们在 BPE 训练中所做的那样。我们将在每个预分词词符内部将这些字节合并为词汇表元素，独立处理每个预分词词符（跨预分词词符边界不进行合并）。**第 2 步：应用合并**。然后，我们采用 BPE 训练期间创建的词汇表元素合并序列，并按照创建的相同顺序将其应用于我们的预分词词符。
示例 (2) 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 `{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' ', 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分割成 `['the', ' cat', ' ate']`。然后，我们将查看每个预分词词符并应用 BPE 合并。

第一个预分词词符 `'the'` 最初表示为 `[b't', b'h', b'e']`。查看我们的合并列表，我们识别出第一个适用的合并是 `(b't', b'h')`，并使用它将预分词词符转换为 `[b'th', b'e']`。然后，我们回到合并列表并识别出下一个适用的合并是 `(b'th', b'e')`，它将预分词词符转换为 `[b'the']`。最后，再次查看合并列表，我们看到没有更多适用于该字符串的合并（因为整个预分词词符已被合并为单个词符），因此我们完成了 BPE 合并的应用。相应的整数序列是 `[9]`。

对剩余的预分词词符重复此过程，我们看到预分词词符 `' cat'` 在应用 BPE 合并后表示为 `[b' c', b'a', b't']`，它变成了整数序列 `[7, 1, 5]`。最后的预分词词符 `' ate'` 在应用 BPE 合并后是 `[b' at', b'e']`，它变成了整数序列 `[10, 3]`。因此，编码我们输入字符串的最终结果是 `[9, 7, 1, 5, 10, 3]`。

特殊词符。您的分词器应该能够在编码文本时正确处理用户定义的特殊词符（在构造分词器时提供）。

内存考虑。假设我们想要对一个无法装入内存的大型文本文件进行分词。为了高效地对这个大型文件（或任何其他数据流）进行分词，我们需要将其分解为可管理的块，并依次处理每个块，以便内存复杂度是恒定的，而不是与文本大小成线性关系。这样做时，我们需要确保词符不会跨越块边界，否则我们会得到与对整个序列进行内存中分词的朴素方法不同的分词结果。

2.6.2 解码文本

要将整数词符 ID 序列解码回原始文本，我们可以简单地查找每个 ID 在词汇表中对应的条目（一个字节序列），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）。如果输入词符 ID 不能产生有效的 Unicode 字符串，您应该用官方 Unicode 替换字符 U+FFFD 替换格式错误的字节。[3] `bytes.decode` 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数据。
问题 (6) tokenizer: 实现分词器 (15 分)

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

可交付成果 (6.1): 实现一个 `Tokenizer` 类，给定一个词汇表和合并列表，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词符（如果它们尚不存在于词汇表中，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）特殊词符列表构造一个分词器。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，从序列化的词汇表和合并列表（格式与您的 BPE 训练代码输出相同）以及（可选的）特殊词符列表构造并返回一个 `Tokenizer`。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]` 将输入文本编码为词符 ID 序列。

`def encode_iterable(self, iterable: Iterable[str]) -> Iterator[int]` 给定一个字符串的可迭代对象（例如，Python 文件句柄），返回一个惰性生成词符 ID 的生成器。这对于内存高效地对我们无法直接加载到内存中的大型文件进行分词是必需的。

`def decode(self, ids: list[int]) -> str` 将词符 ID 序列解码为文本。

要根据我们提供的测试来测试您的 `Tokenizer`，您首先需要实现位于 `[adapters.get_tokenizer]` 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

问题 (7) tokenizer_experiments: 使用分词器进行实验 (4 分)

(a) 从 TinyStories 和 OpenWebText 中各抽样 10 个文档。使用您先前训练的 TinyStories 和 OpenWebText 分词器（词汇表大小分别为 10K 和 32K），将这些抽样的文档编码为整数 ID。每个分词器的压缩率（字节/词符）是多少？
可交付成果 (7.1): 一到两句回答。

(b) 如果您使用 TinyStories 分词器对您的 OpenWebText 样本进行分词会发生什么？比较压缩率和/或定性描述发生的情况。
可交付成果 (7.2): 一到两句回答。

(c) 估计您的分词器的吞吐量（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?
可交付成果 (7.3): 一到两句回答。

(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符 ID 序列。我们稍后将使用这些来训练我们的语言模型。我们建议将词符 ID 序列化为数据类型为 `uint16` 的 NumPy 数组。为什么 `uint16` 是一个合适的选择?
可交付成果 (7.4): 一到两句回答。

3 Transformer 语言模型架构

语言模型将一个批量的整数词符 ID 序列(即形状为 `(batch_size, sequence_length)` 的 `torch.Tensor`)作为输入,并返回一个(批量的)在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符的下一个词。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。在推理期间从语言模型生成文本时,我们采用最后时间步(即序列中的最后一项)的预测下一个词分布来生成序列中的下一个词符(例如,通过取概率最高的词符、从分布中采样等),将生成的词符添加到输入序列中,然后重复。

在本作业的这一部分,您将从头开始构建这个 Transformer 语言模型。我们将从模型的高级描述开始,然后逐步详细介绍各个组件。

3.1 Transformer LM

给定一个词符 ID 序列,Transformer 语言模型使用输入嵌入将词符 ID 转换为密集向量,将嵌入后的词符通过 `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")来产生预测的下一个词符的 logits。有关示意图,请参见图 1。

3.1.1 词符嵌入 (Token Embeddings)

在第一步,Transformer 将(批量的)词符 ID 序列嵌入到一个向量序列中,该序列包含关于词符身份的信息(图 1 中的红色块)。更具体地说,给定一个词符 ID 序列,Transformer 语言模型使用词符嵌入层来产生一个向量序列。每个嵌入层接收一个形状为 `(batch_size, sequence_length)` 的整数张量,并产生一个形状为 `(batch_size, sequence_length, d_model)` 的向量序列。

3.1.2 Pre-norm Transformer 块

嵌入之后,激活通过几个结构相同的神经网络层进行处理。一个标准的仅解码器 (decoder-only) Transformer 语言模型由 `num_layers` 个相同的层(通常称为 Transformer "块")组成。每个 Transformer 块接收一个形状为 `(batch_size, sequence_length, d_model)` 的输入,并返回一个形状为 `(batch_size, sequence_length, d_model)` 的输出。每个块通过自注意力 (self-attention) 在序列上聚合信息,并通过前馈层 (feed-forward layers) 对其进行非线性转换。

3.2 输出归一化和嵌入

经过 `num_layers` 个 Transformer 块后,我们将取最终的激活,并将它们转换为词汇表上的分布。我们将实现 "pre-norm" Transformer 块(详见 §3.5),它额外要求在最后一个 Transformer 块之后使用层归一化(layer normalization,详见下文)以确保其输出被适当地缩放。在此归一化之后,我们将使用一个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符的 logits(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，我们将对许多类似批次的输入应用相同的计算。以下是一些例子：

- **批次中的元素 (Elements of a batch)**：我们对每个批次元素应用相同的 Transformer 前向操作。
- **序列长度 (Sequence length)**：像 RMSNorm 和前馈这样的“位置无关 (position-wise)”操作对序列的每个位置都进行相同的操作。
- **注意力头 (Attention heads)**：注意力操作在“多头 (multi-headed)”注意力操作中跨注意力头进行批处理。

拥有一种符合人体工程学的方式来执行此类操作，同时充分利用 GPU，并且易于阅读和理解是很有用的。许多 PyTorch 操作可以在张量的开头接受额外的“类似批次”的维度，并有效地跨这些维度重复/广播操作。

例如，假设我们正在进行一个位置无关的批处理操作。我们有一个形状为 `(batch_size, sequence_length, d_model)` 的“数据张量” D，并且我们想对一个形状为 `(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 @ A` 将执行批处理矩阵乘法，这是 PyTorch 中的一个高效原语，其中 `(batch_size, sequence_length)` 维度被批处理。

因此，假设您的函数可能会被赋予额外的类似批次的维度，并将这些维度保持在 PyTorch 张量形状的开头是很有帮助的。为了以这种方式组织张量以便进行批处理，它们可能需要使用许多 `view`, `reshape` 和 `transpose` 步骤进行整形。这可能有点痛苦，而且通常很难读懂代码在做什么以及张量的形状是什么。

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，使用像 `einops` 或 `einx` 这样的框架无关库。两个关键操作是 `einsum`，它可以对具有任意维度的输入张量进行张量收缩，以及 `rearrange`，它可以对任意维度进行重新排序、连接和分割。事实证明，机器学习中几乎所有的操作都是维度调整和张量收缩的某种组合，偶尔带有（通常是逐点的）非线性函数。这意味着当使用 einsum 表示法时，您的许多代码可以更具可读性和灵活性。

我们强烈建议学习和使用 einsum 表示法。之前没有接触过 einsum 表示法的学生应该使用 `einops`（文档在此 `https://einops.rocks/`），已经熟悉 `einops` 的学生应该学习更通用的 `einx`（文档在此 `https://github.com/arogozhnikov/einx`）。[4] 我们提供的环境中已经安装了这两个包。

这里我们给出一些如何使用 einsum 表示法的例子。这些是对 `einops` 文档的补充，您应该首先阅读。

示例 (3) einstein_example1: 使用 einops.einsum 进行批处理矩阵乘法

```
import torch
from einops import rearrange, einsum
```

[4]值得注意的是，虽然 `einops` 得到了大量支持，但 `einx` 没有那么经过实战检验。如果您发现 `einx` 有任何限制或错误，可以随时回退到使用 `einops` 和一些更普通的 PyTorch 代码。

```

## 基本实现
Y = D @ A.T
# 很难判断输入和输出的形状以及它们的含义。
# D 和 A 可以有哪些形状？它们中是否有任何具有意外行为？

## Einsum 是自文档化且稳健的
#           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 是受约束的。
Y = einsum(D, A, "... d_in, d_out d_in -> ... d_out")
```

示例 (4) 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.randn(64, 128, 128, 3) # (batch, height, width, channel)
dim_by = torch.linspace(start=0.0, end=1.0, steps=10)

## Reshape 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

## 或者一步完成：
dimmed_images = einsum(
    images, dim_by,
    "batch height width channel, dim_value -> batch dim_value height width channel"
)
```

示例 (5) 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 (batch, height, width, channel) 的张量，我们想对图像的所有像素执行线性变换，但此变换应独立地对每个通道进行。我们的线性变换由形状为 (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)

## 重新排列图像张量以跨所有像素混合
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)

# 相反，使用 einops:
```

```
height = width = 32
## Rearrange 替换笨重的 torch view + transpose
channels_first = rearrange(
    channels_last,
    "batch height width channel -> batch channel (height width)"
)
channels_first_transformed = einsum(
    channels_first, B,
    "batch channel pixel_in, pixel_out pixel_in -> batch channel pixel_out"
)
channels_last_transformed = rearrange(
    channels_first_transformed,
    "batch channel (height width) -> batch height width channel",
    height=height, width=width
)

# 或者，如果你觉得疯狂：一步使用 einx.dot (einx 等价于 einops.einsum)
height = width = 32
channels_last_transformed = 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改进以指示输入和输出形状，但这很笨拙并且容易出错。使用 einsum 表示法，文档就是实现！

Einsum 表示法可以处理任意输入批处理维度，但也有自文档化的关键优势。在使用 einsum 表示法的代码中，输入和输出张量的相关形状要清晰得多。对于剩余的张量，您可以考虑使用张量类型提示，例如使用 `jaxtyping` 库（不特定于 Jax）。

我们将在作业 2 中更多地讨论使用 einsum 表示法的性能影响，但现在要知道它们几乎总是比替代方案更好！

3.3.1 数学表示法和内存顺序

许多机器学习论文在其表示法中使用行向量，这导致表示与 NumPy 和 PyTorch 中默认使用的行主序 (row-major) 内存顺序很好地吻合。对于行向量，线性变换看起来像：

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

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

在线性代数中，通常更常用列向量，其中线性变换看起来像：

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

给定行主序 $W \in \mathbb{R}^{d_{out} \times d_{in}}$ 和列向量 $x \in \mathbb{R}^{d_{in}}$。在本作业中，我们将使用列向量进行数学表示法，因为这样通常更容易理解数学。您应该记住，如果您想使用普通的矩阵乘法表示法，您将

必须使用行向量约定来应用矩阵，因为 PyTorch 使用行主序内存顺序。如果您对矩阵操作使用 `einsum`，这应该不是问题。

3.4 基本构建块：Linear 和 Embedding 模块

3.4.1 参数初始化

有效地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不希望的行为，例如梯度消失或爆炸。Pre-norm transformers 对初始化异常鲁棒，但它们仍然会对训练速度和收敛产生显著影响。由于本作业已经很长，我们将把细节留到作业 3，而是给您一些应该在大多数情况下效果良好的近似初始化。现在，请使用：

- 线性层权重 (Linear weights): $N(\mu = 0, \sigma^2 = \frac{2}{d_{in}+d_{out}})$ 截断在 $[-3\sigma, 3\sigma]$。
- 嵌入层 (Embedding): $N(\mu = 0, \sigma^2 = 1)$ 截断在 $[-3, 3]$。
- RMSNorm: 1

您应该使用 `torch.nn.init.trunc_normal_` 来初始化截断正态分布的权重。

3.4.2 Linear 模块

线性层是 Transformers 和一般神经网络的基本构建块。首先，您将实现自己的 `Linear` 类，该类继承自 `torch.nn.Module` 并执行线性变换：

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

请注意，我们不包括偏置项 (bias term)，遵循大多数现代 LLM 的做法。

问题 (8) linear: 实现线性模块 (1 分)

可交付成果 (8.1): 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，但不包含 `bias` 参数或参数。我们建议使用以下接口：

`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: int` 输入的最终维度
- `out_features: int` 输出的最终维度
- `device: torch.device | None = None` 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。

确保：

- 子类化 `nn.Module`
- 调用超类构造函数

- 出于内存顺序的原因，构造并存储您的参数为 W（而不是 W^T），将其放入 `nn.Parameter` 中
- 当然，不要使用 `nn.Linear` 或 `nn.functional.linear`

对于初始化，使用上面的设置以及 `torch.nn.init.trunc_normal_` 来初始化权重。要测试您的 `Linear` 模块，请在 `[adapters.run_linear]` 实现测试适配器。适配器应将给定的权重加载到您的 `Linear` 模块中。为此，您可以使用 `Module.load_state_dict`。然后，运行 `uv run pytest -k test_linear`。

3.4.3 Embedding 模块

如上所述，Transformer 的第一层是一个嵌入层，它将整数词符 ID 映射到维度为 `d_model` 的向量空间。我们将实现一个自定义的 `Embedding` 类，它继承自 `torch.nn.Module`（因此您不应使用 `nn.Embedding`）。`forward` 方法应通过使用形状为 `(batch_size, sequence_length)` 的 `torch.LongTensor` 词符 ID 索引到形状为 `(vocab_size, d_model)` 的嵌入矩阵中来选择每个词符 ID 的嵌入向量。

问题 (9) embedding: 实现嵌入模块 (1 分)

可交付成果 (9.1): 实现一个继承自 `torch.nn.Module` 并执行嵌入查找的 `Embedding` 类。您的实现应遵循 PyTorch 内置 `nn.Embedding` 模块的接口。我们建议使用以下接口：

`def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)` 构造一个嵌入模块。此函数应接受以下参数：

- `num_embeddings: int` 词汇表的大小
- `embedding_dim: int` 嵌入向量的维度，即 d_{model}
- `device: torch.device | None = None` 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` 查找给定词符 ID 的嵌入向量。

确保：

- 子类化 `nn.Module`
- 调用超类构造函数
- 将您的嵌入矩阵初始化为 `nn.Parameter`
- 存储嵌入矩阵时，将 `d_model` 作为最终维度
- 当然，不要使用 `nn.Embedding` 或 `nn.functional.embedding`

同样，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。要测试您的实现，请在 `[adapters.run_embedding]` 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 块有两个子层：一个多头自注意力机制和一个位置无关的前馈网络（Vaswani et al., 2017, section 3.1）。在原始的 Transformer 论文中，模型在两个子层的每一个周围使用残差连接，然后进行层归一化。这种架构通常被称为 “post-norm” Transformer，因为层归一化应用于子层的输出。然而，各种工作发现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 块之后增加一个额外的层归一化）可以提高 Transformer 训练的稳定性 [Nguyen and Salazar, 2019, Xiong et al., 2020]——有关此 “pre-norm” Transformer 块的视觉表示，请参见图 2。每个 Transformer 块子层的输出然后通过残差连接添加到子层输入中（Vaswani et al., 2017, section 5.4）。pre-norm 的一个直觉是，存在一个从输入嵌入到 Transformer 最终输出的干净 “残差流”，没有任何归一化，据称这可以改善梯度流。这种 pre-norm Transformer 现在是当今语言模型（例如 GPT-3、LLaMA、PaLM 等）使用的标准，因此我们将实现这个变体。我们将依次介绍 pre-norm Transformer 块的每个组件，并按顺序实现它们。

3.5.1 Root Mean Square Layer Normalization (RMSNorm)

Vaswani et al. [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 et al. [2023]，我们将使用 Root Mean Square Layer Normalization (RMSNorm; Zhang and Sennrich, 2019, equation 4) 进行层归一化。给定一个激活向量 $a \in \mathbb{R}^{d_{model}}$，RMSNorm 将按如下方式重新缩放每个激活 a_i：

$$\text{RMSNorm}(a)_i = \frac{a_i}{\text{RMS}(a)} g_i, \tag{4}$$

其中 $\text{RMS}(a) = \sqrt{\frac{1}{d_{model}} \sum_{i=1}^{d_{model}} a_i^2 + \epsilon}$。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），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` 以防止在平方输入时溢出。总的来说，您的 `forward` 方法应该看起来像：

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

# Your code here performing RMSNorm
...
result = ...

# Return the result in the original dtype
return result.to(in_dtype)
```

问题 (10) rmsnorm: Root Mean Square Layer Normalization (1 分)

可交付成果 (10.1): 将 RMSNorm 实现为 `torch.nn.Module`。我们建议使用以下接口：

`def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)` 构造 RMSNorm 模块。此函数应接受以下参数：

- `d_model: int` 模型的隐藏维度

- `eps: float = 1e-5` 用于数值稳定性的 Epsilon 值
- `device: torch.device | None = None` 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：如上所述，记得在执行归一化之前将输入上转换为 `torch.float32`（之后再下转换为原始 dtype）。要测试您的实现，请在 `[adapters.run_rmsnorm]` 实现测试适配器。然后，运行 `uv run pytest -k test_rmsnorm`。

3.5.2 位置无关的前馈网络 (Position-Wise Feed-Forward Network)

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

然而，现代语言模型倾向于与这种原始设计相比包含两个主要变化：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 LLM 中采用的 "SwiGLU" 激活函数，它将 SiLU（通常称为 Swish）激活与称为 Gated Linear Unit (GLU) 的门控机制相结合。我们还将省略线性层中有时使用的偏置项，遵循自 PaLM [Chowdhery et al., 2022] 和 LLaMA [Touvron et al., 2023] 以来的大多数现代 LLM。

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

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

从图 3 可以看出，SiLU 激活函数与 ReLU 激活函数相似，但在零点处是平滑的。

Gated Linear Units (GLUs) 最初由 Dauphin et al. [2017] 定义为通过 sigmoid 函数传递的线性变换与另一个线性变换的逐元素乘积：

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

其中 $\odot$ 表示逐元素乘法。Gated Linear Units 被建议用于 "通过提供梯度的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，我们得到 SwiGLU，我们将用它作为我们的前馈网络：

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

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

Shazeer [2020] 首次提出将 SiLU/Swish 激活与 GLU 结合，并进行了实验，表明 SwiGLU 在语言建模任务上的表现优于 ReLU 和 SiLU（无门控）等基线。在本作业的后面，您将比较 SwiGLU 和 SiLU。尽管我们已经为这些组件提到了一些启发式论点（并且论文提供了更多支持证据），但保持经验视角是很好的：Shazeer 论文中一句现在著名的引述是：

我们不解释为什么这些架构似乎有效；我们将其成功归因于，像其他一切一样，神圣的天意。

问题 (11) positionwise_feedforward: 实现位置无关的前馈网络 (2 分)

可交付成果 (11.1): 实现 SwiGLU 前馈网络，由 SiLU 激活函数和 GLU 组成。注意：在这种特殊情况下，为了数值稳定性，您可以在实现中使用 `torch.sigmoid`。您应该在实现中将 d_{ff} 设置为大约 $\frac{8}{3} \times d_{model}$，同时确保内部前馈层的维度是 64 的倍数，以便充分利用您的硬件。要根据我们提供的测试来测试您的实现，您需要实现位于 `[adapters.run_swiglu]` 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 (Relative Positional Embeddings)

为了向模型注入位置信息，我们将实现 Rotary Position Embeddings [Su et al., 2021]，通常称为 RoPE。对于位于词符位置 i 的给定查询词符 $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$，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，$R^i$ 将成对旋转嵌入元素 $q^{(i)}_{2k-1:2k}$ 作为 2d 向量，旋转角度为 $\theta_{i,k} = i\Theta^{-2k/d}$，其中 $k \in \{1, \ldots, d/2\}$ 且 Θ 是某个常数。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，$k \in \{1, \ldots, d/2\}$，其中

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

因此我们得到完整的旋转矩阵

$$R^i = \begin{bmatrix} R^i_1 & 0 & 0 & \ldots & 0 \\ 0 & R^i_2 & 0 & \ldots & 0 \\ 0 & 0 & R^i_3 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & R^i_{d/2} \end{bmatrix}, \tag{9}$$

其中 0 表示 2×2 的零矩阵。虽然可以构建完整的 $d \times d$ 矩阵，但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内词符的相对旋转，我们可以在层与层之间以及不同的批次之间重用我们为 $\cos(\theta_{i,k})$ 和 $\sin(\theta_{i,k})$ 计算的值。如果您想对其进行优化，可以使用一个由所有层引用的单一 ROPE 模块，并且它可以在初始化期间创建一个包含 sin 和 cos 值的 2d 预计算缓冲区，使用 `self.register_buffer(persistent=False)`，而不是 `nn.Parameter`（因为我们不想学习这些固定的余弦和正弦值）。我们对 $q^{(i)}$ 所做的完全相同的旋转过程随后也对 $k^{(i)}$ 进行，通过相应的 R^i 进行旋转。请注意，这一层没有可学习的参数。

问题 (12) rope: 实现 RoPE (2 分)

可交付成果 (12.1): 实现一个将 RoPE 应用于输入张量的 `RotaryPositionalEmbedding` 类。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: float` RoPE 的 Θ 值
- `d_k: int` 查询和键向量的维度

- `max_seq_len: int` 将输入的最大序列长度
- `device: torch.device | None = None` 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。注意，您应该容忍 `x` 具有任意数量的批处理维度。您应该假设词符位置是一个形状为 `(..., seq_len)` 的张量，指定 `x` 沿序列维度的词符位置。您应该使用词符位置来沿序列维度切片您的（可能预先计算的）cos 和 sin 张量。

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 Scaled Dot-Product Attention

我们现在将实现 Vaswani et al. [2017]（第 3.2.1 节）中描述的 scaled dot-product attention。作为预备步骤，Attention 操作的定义将使用 softmax，这是一个将未归一化的分数向量转换为归一化分布的操作：

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

注意 $\exp(v_i)$ 对于较大的值可能会变成 `inf`（然后，`inf/inf = 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 是不变的来避免这种情况。我们可以利用这个特性来实现数值稳定性——通常，我们会从 v_i 的所有元素中减去 v_i 的最大项，使得新的最大项为 0。您现在将实现 softmax，使用这个技巧来确保数值稳定性。

问题 (13) softmax: 实现 softmax (1 分)

可交付成果 (13.1): 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，并在输入张量的第 i 维上应用 softmax。输出张量应具有与输入张量相同的形状，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该维最大值的技巧来避免数值稳定性问题。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`。

我们现在可以按如下数学方式定义 Attention 操作：

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

其中 $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, 并且 $V \in \mathbb{R}^{m \times d_v}$。这里，Q、K 和 V 都是此操作的输入——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^T，请参见 3.3.1。

掩码 (Masking)：有时对注意力操作的输出进行掩码是很方便的。掩码应具有形状 $M \in \{\text{True}, \text{False}\}^{n \times m}$，此布尔矩阵的每一行 i 指示查询 i 应关注哪些键。规范地（且有点令人困惑地），位置 (i, j) 处的值为 True 表示查询 i * 确实 * 关注键 j，值为 False 表示查询 * 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 1×3 的掩码矩阵，其条目为 `[[True, True, False]]`。单个查询向量仅关注前两个键。

在计算上，使用掩码比计算子序列上的注意力要高效得多，我们可以通过取 pre-softmax 值 ($\frac{QK^T}{\sqrt{d_k}}$) 并在掩码矩阵为 False 的任何条目中添加一个 $-\infty$ 来实现。

问题 (14) scaled_dot_product_attention: 实现 scaled dot-product attention (5 分)
可交付成果 (14.1): 实现 scaled dot-product attention 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似批次的维度（如果提供）。实现应返回形状为 `(batch_size, ..., seq_len, d_v)` 的输出。有关类似批次维度的讨论，请参见第 3.3 节。您的实现还应支持可选的用户提供的形状为 `(seq_len, seq_len)` 的布尔掩码。掩码值为 True 的位置的注意力概率应总和为 1，掩码值为 False 的位置的注意力概率应为零。要根据我们提供的测试来测试您的实现，您需要实现位于 `[adapters.run_scaled_dot_product_attention]` 的测试适配器。`uv run pytest -k test_scaled_dot_product_attention` 测试您在三阶输入张量上的实现，而 `uv run pytest -k test_4d_scaled_dot_product_attention` 测试您在四阶输入张量上的实现。

3.5.5 Causal Multi-Head Self-Attention

我们将实现 Vaswani et al. [2017] 第 3.2.2 节中描述的多头自注意力。回想一下，从数学上讲，应用多头注意力的操作定义如下：

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

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

其中 Q_i, K_i, V_i 分别是 Q、K 和 V 嵌入维度的第 $i \in \{1, \ldots, h\}$ 个大小为 d_k 或 d_v 的切片。Attention 是 §3.5.4 中定义的 scaled dot-product attention 操作。由此我们可以形成多头自注意力操作：

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

这里，可学习的参数是 $W_Q \in \mathbb{R}^{hd_k \times d_{model}}$，$W_K \in \mathbb{R}^{hd_k \times d_{model}}$，$W_V \in \mathbb{R}^{hd_v \times d_{model}}$，以及 $W_o \in \mathbb{R}^{d_{model} \times hd_v}$。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_Q, W_K 和 W_V 沿着输出维度为每个头分离。当您完成此工作时，您应该总共使用三个矩阵乘法来计算键、值和查询投影。[5]

Causal masking. 您的实现应阻止模型关注序列中的未来词符。换句话说，如果模型被赋予一个词符序列 $t_1, \ldots, t_n$，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，模型不应能够访问（关注）位置 $t_{i+1}, \ldots, t_n$ 处的词符表示，因为它在推理期间生成文本时将无法访问这些词符（并且这些未来词符会泄露关于真实下一个词身份的信息，从而使语言建模预训练目标变得微不足道）。对于输入词符序列 $t_1, \ldots, t_n$，我们可以通过运行多头自注意力 n 次（对于序列中的 n 个唯一前缀）来天真地阻止访问未来词符。相反，我们将使用 causal attention masking，它允许词符 i 关注序列中所有位置 $j \leq i$。您可以使用 `torch.triu` 或广播索引比较来构建此掩码，并且您应该利用 §3.5.4 中的 scaled dot-product attention 实现已经支持注意力掩码的事实。

应用 RoPE. RoPE 应应用于查询和键向量，但不应用于值向量。此外，头维度应作为批处理维度处理，因为在多头注意力中，注意力是独立应用于每个头的。这意味着应该对每个头的查

[5]作为一个延伸目标，尝试将键、查询和值投影合并到一个权重矩阵中，这样您只需要一个矩阵乘法。

询和键向量应用完全相同的 RoPE 旋转。

问题 (15) multihead_self_attention: 实现 causal multi-head self-attention (5 分)

可交付成果 (15.1): 将 causal multi-head self-attention 实现为 `torch.nn.Module`。您的实现应至少接受以下参数：

`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。

遵循 Vaswani et al. [2017]，设置 $d_k = d_v = d_{model}/h$。要根据我们提供的测试来测试您的实现，请实现位于 `[adapters.run_multihead_self_attention]` 的测试适配器。然后，运行 `uv run pytest -k test_multihead_self_attention` 来测试您的实现。

3.6 完整的 Transformer LM

让我们开始组装 Transformer 块（参考图 2 会很有帮助）。一个 Transformer 块包含两个“子层”，一个用于多头自注意力，另一个用于前馈网络。在每个子层中，我们首先执行 RMSNorm，然后是主要操作（MHA/FF），最后在残差连接中添加回来。具体来说，Transformer 块的第一半（第一个“子层”）应实现以下更新以从输入 x 产生输出 y，

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

问题 (16) transformer_block: 实现 Transformer 块 (3 分)

可交付成果 (16.1): 按照 §3.5 中描述并在图 2 中说明的方式实现 pre-norm Transformer 块。您的 Transformer 块应至少接受以下参数。

`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。

`d_ff: int` 位置无关的前馈内部层的维度。

要测试您的实现，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。可交付成果：通过所提供测试的 Transformer 块代码。

现在我们将各个块组合在一起，遵循图 1 中的高级图示。遵循我们在 3.1.1 节中对嵌入的描述，将其输入到 `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 (17) transformer_lm: 实现 Transformer LM (3 分)

可交付成果 (17.1): 是时候把它们都组合起来了！按照 §3.1 中描述并在图 1 中说明的方式实现 Transformer 语言模型。您的实现至少应接受 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，用于确定词符嵌入矩阵的维度。

`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。

要根据我们提供的测试来测试您的实现，您首先需要实现位于 `[adapters.run_transformer_lm]` 的测试适配器。然后，运行 `uv run pytest -k test_transformer_lm` 来测试您的实现。可交付成果：通过上述测试的 Transformer LM 模块。

资源核算。能够理解 Transformer 的各个部分如何消耗计算和内存是很有用的。我们将通过一些步骤来进行一些基本的 "FLOPs 核算"。Transformer 中的绝大多数 FLOPs 来自矩阵乘法，所以我们的核心方法很简单：

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

对于这第二步，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。要看到这一点，请注意 $(AB)[i,j] = A[i,:] \cdot B[:,j]$，并且这个点积需要 n 次加法和 n 次乘法（$2n$ FLOPs）。然后，由于矩阵-矩阵乘积 AB 有 $m \times p$ 个条目，总 FLOPs 数为 $(2n)(mp) = 2mnp$。

现在，在做下一个问题之前，浏览您的 Transformer 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 (18) transformer_accounting: Transformer LM 资源核算 (5 分)

(a) 考虑 GPT-2 XL，它具有以下配置：

```
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了我们的模型。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表示，仅加载此模型需要多少内存？
可交付成果 (18.1): 一到两句回答。

(b) 识别完成我们 GPT-2 XL 形状模型的一次前向传递所需的矩阵乘法。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` 个词符。
可交付成果 (18.2): 一个矩阵乘法列表（带描述），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？
可交付成果 (18.3): 一到两句回答。

(d) 对 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。随着模型大小的增加，Transformer LM 的哪些部分占总 FLOPs 的比例或多或少？

可交付成果 (18.4): 对于每个模型，提供模型组件及其相关 FLOPs 的细分（作为一次前向传递所需总 FLOPs 的比例）。此外，提供一到两句话描述模型大小的变化如何改变每个组件的 FLOPs 比例。

(e) 取 GPT-2 XL 并将上下文长度增加到 16,384。一次前向传递的总 FLOPs 如何变化？模型组件的 FLOPs 相对贡献如何变化？
可交付成果 (18.5): 一到两句回答。

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。剩下的就是构建所有支持训练的代码。这包括以下内容：

- **损失 (Loss)**: 我们需要定义损失函数（交叉熵）。
- **优化器 (Optimizer)**: 我们需要定义优化器来最小化这个损失（AdamW）。
- **训练循环 (Training loop)**: 我们需要所有加载数据、保存检查点和管理训练的支持基础设施。

4.1 交叉熵损失 (Cross-entropy loss)

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

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

（注意，Transformer 的单次前向传递会产生所有 $i = 1, \ldots, m$ 的 $p_\theta(x_{i+1}|x_{1:i})$。）特别地，Transformer 为每个位置 i 计算 logits $o_i \in \mathbb{R}^{\text{vocab\_size}}$，这导致：[6]

$$p(x_{i+1}|x_{1:i}) = \text{softmax}(o_i)[x_{i+1}] = \frac{\exp(o_i[x_{i+1}])}{\sum_{a=1}^{\text{vocab\_size}} \exp(o_i[a])} \tag{17}$$

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} 定义的。[7]

实现交叉熵损失需要小心处理数值问题，就像 softmax 的情况一样。
问题 (19) cross_entropy: 实现交叉熵
可交付成果 (19.1): 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}) 并计算交叉熵损失 $\ell_i = -\log \text{softmax}(o_i)[x_{i+1}]$。您的函数应处理以下情况：

- 减去最大元素以确保数值稳定性。
- 尽可能抵消 log 和 exp。

[6] 注意 $o_i[k]$ 指的是向量 o_i 在索引 k 处的值。

[7] 这对应于 x_{i+1} 上的 Dirac delta 分布与预测的 softmax(o_i) 分布之间的交叉熵。

- 处理任何附加的批处理维度并返回批次上的平均值。与第 3.3 节一样，我们假设批处理维度始终位于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` 来测试您的实现。

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

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

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 (Stochastic Gradient Descent, SGD)。我们从随机初始化的参数 θ_0 开始。然后对于每个步骤 $t = 0, \ldots, T-1$，我们执行以下更新：

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

其中 B_t 是从数据集 D 中随机抽样的一个数据批次，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，我们将子类化 PyTorch 的 `torch.optim.Optimizer` 类。一个 `Optimizer` 子类必须实现两个方法：

`def __init__(self, params, ...)` 应该初始化您的优化器。这里，`params` 将是要优化的参数集合（或参数组，以防用户想要对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在 `step` 中使用。您可以根据优化器接受额外的参数（例如，学习率是常见的），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。在训练循环中，这将在后向传递之后调用，因此您可以访问上一个批次的梯度。此方法应迭代每个参数张量 `p` 并就地修改它们，即设置 `p.data`，它持有与该参数关联的张量，基于梯度 `p.grad`（如果存在），即表示损失相对于该参数的梯度的张量。

PyTorch 优化器 API 有一些微妙之处，所以通过一个例子来解释它更容易。为了使我们的例子更丰富，我们将实现 SGD 的一个轻微变体，其中学习率在训练过程中衰减，从初始学习率 α 开始，并随着时间的推移采取越来越小的步长：

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

让我们看看这个版本的 SGD 如何作为 PyTorch Optimizer 来实现：

```python
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().__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"]:
                if p.grad is None:
                    continue

                state = self.state[p] # 获取与 p 关联的状态。
                # 如果状态字典中没有 't'，则 get 返回默认值 0。
                t = state.get("t", 0) # 从状态获取迭代次数，或初始值。
                grad = p.grad.data # 获取损失相对于 p 的梯度。

                # 更新权重张量 p.data。就地修改 p.data!
                # 注意：我们使用 p.data 而不是 p 来避免跟踪此操作以进行自动微分。
                p.data.add_(grad, alpha=-lr / math.sqrt(t + 1)) # 就地更新权重张量。

                state["t"] = t + 1 # 增加迭代次数。
        return loss
```

在 `__init__` 中，我们将参数以及默认超参数传递给基类构造函数（参数可能成组出现，每组具有不同的超参数）。如果参数只是 `torch.nn.Parameter` 对象的单个集合，基类构造函数将创建一个单独的组并为其分配默认超参数。然后，在 `step` 中，我们迭代每个参数组，然后迭代该组中的每个参数，并应用方程 20。在这里，我们将迭代次数作为与每个参数关联的状态来保持：我们首先读取这个值，在梯度更新中使用它，然后更新它。API 指定用户可能传入一个可调用闭包 `closure` 以在优化器步骤之前重新计算损失。我们使用的优化器不需要这个，但我们添加它以符合 API。

要看到这个工作，我们可以使用以下最小的训练循环示例：

```python
weights = torch.nn.Parameter(5 * torch.randn((10, 10)))
opt = SGD([weights], lr=1)

for t in range(100):
    opt.zero_grad() # 为所有可学习参数重置梯度。
```

```
    loss = (weights**2).mean() # 计算标量损失值。
    print(loss.cpu().item())
    loss.backward() # 运行后向传递，计算梯度。
    opt.step() # 运行优化器步骤。
```

这是训练循环的典型结构：在每次迭代中，我们将计算损失并运行优化器的一个步骤。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.parameters()` 为我们提供了这个集合）。损失将通过抽样的数据批次计算，但训练循环的基本结构将是相同的。

问题 (20) learning_rate_tuning: 调整学习率 (1 分)

可交付成果 (20.1): 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践一下。使用另外三个学习率值：1e1、1e2 和 1e3 运行上面的 SGD 示例，仅进行 10 次训练迭代。对于这些学习率中的每一个，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）？可交付成果：对您观察到的行为的一到两句回答。

4.3 AdamW

现代语言模型通常使用比 SGD 更复杂的优化器进行训练。最近使用的大多数优化器都是 Adam 优化器 [Kingma and Ba, 2015] 的衍生物。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）来改善正则化，这种方式与梯度更新解耦。我们将实现 Loshchilov and Hutter [2019] 算法 2 中描述的 AdamW。

AdamW 是有状态的：对于每个参数，它跟踪其一阶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 还有一对超参数 (β_1, β_2) 控制矩估计的更新，以及一个权重衰减率 λ。典型的应用设置 (β_1, β_2) 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 是一个小的数值（例如，10^{-8}）用于在 v 中得到极小值时提高数值稳定性：

请注意 t 从 1 开始。您现在将实现这个优化器。

问题 (21) adamw: 实现 AdamW (2 分)

可交付成果 (21.1): 将 AdamW 优化器实现为 `torch.optim.Optimizer` 的子类。您的类应在 `__init__` 中接受学习率 α，以及 β、ϵ 和 λ 超参数。为了帮助您保持状态，基类 `Optimizer` 为您提供了一个字典 `self.state`，它将 `nn.Parameter` 对象映射到一个字典，该字典存储您需要为该参数提供的任何信息（对于 AdamW，这将是矩估计）。实现 `[adapters.get_adamw_cls]` 并确保它通过 `uv run pytest -k test_adamw`。

问题 (22) adamwAccounting: 使用 AdamW 进行训练的资源核算 (2 分)

可交付成果 (22.1): 让我们计算运行 AdamW 需要多少内存和计算量。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、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`）来表示您的答案。假设 $d_{ff} = 4 \times d_{model}$。为简单起见，在计算激活

Algorithm 1 AdamW 优化器

init(θ) (初始化可学习参数)
$m \leftarrow 0$ (一阶矩向量的初始值；形状与 θ 相同)
$v \leftarrow 0$ (二阶矩向量的初始值；形状与 θ 相同)
for $t = 1, \ldots, T$ **do**
    抽样数据批次 B_t
    $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ (计算当前时间步损失的梯度)
    $m \leftarrow \beta_1 m + (1 - \beta_1) g$ (更新一阶矩估计)
    $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (更新二阶矩估计)
    $\hat{m} \leftarrow m / (1 - \beta_1^t)$ (计算偏差校正后的一阶矩估计)
    $\hat{v} \leftarrow v / (1 - \beta_2^t)$ (计算偏差校正后的二阶矩估计)
    $\theta \leftarrow \theta - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$ (更新参数)
    $\theta \leftarrow \theta - \alpha \lambda \theta$ (应用权重衰减)
end for

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，QK^T 矩阵乘法，softmax，值的加权和，输出投影。
  - 位置无关的前馈：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

可交付成果：参数、激活、梯度和优化器状态各自的代数表达式，以及总和。

(b) 将您的答案实例化为一个 GPT-2 XL 形状的模型，以获得一个仅依赖于 `batch_size` 的表达式。您可以使用的最大批次大小是多少，并且仍然适合 80GB 内存？可交付成果：一个形如 $a \cdot \text{batch\_size} + b$ 的表达式，其中 a, b 是数值，以及一个表示最大批次大小的数字。

(c) 运行一步 AdamW 需要多少 FLOPs？可交付成果：一个代数表达式，并附简要说明。

(d) 模型 FLOPs 利用率 (MFU) 定义为观察到的吞吐量（每秒词符数）与硬件理论峰值 FLOP 吞吐量的比率 [Chowdhery et al., 2022]。NVIDIA A100 GPU 对于 float32 操作的理论峰值为 19.5 teraFLOP/s。假设您能够获得 50 可交付成果：训练所需的天数，并附简要说明。

4.4 学习率调度 (Learning rate scheduling)

导致损失最快下降的学习率值在训练过程中通常会变化。在训练 Transformers 时，通常使用学习率调度，我们从较大的学习率开始，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较小的值。[8] 在本作业中，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，它接收当前步骤 t 和其他相关参数（例如初始和最终学习率），并返回在步骤 t 进行梯度更新时使用的学习率。最简单的调度是常数函数，它对任何给定的 t 返回相同的学习率。

余弦退火学习率调度接受 (i) 当前迭代 t，(ii) 最大学习率 α_{max}，(iii) 最小（最终）学习率 α_{min}，(iv) 预热 (warm-up) 迭代次数 T_w，以及 (v) 余弦退火迭代次数 T_c。迭代 t 时的学习率 α_t 定义为：

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

(Cosine annealing) 如果 $T_w \leq t \leq T_c$，则 $\alpha_t = \alpha_{min} + \frac{1}{2}(1 + \cos(\frac{t-T_w}{T_c-T_w}\pi))(\alpha_{max} - \alpha_{min})$。

(Post-annealing) 如果 $t > T_c$，则 $\alpha_t = \alpha_{min}$。

问题 (23) learning_rate_schedule: 实现带预热的余弦学习率调度

可交付成果 (23.1): 编写一个函数，接收 $t, \alpha_{max}, \alpha_{min}, T_w$ 和 T_c，并根据上面定义的调度器返回学习率 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 (Gradient clipping)

在训练期间，我们有时会遇到导致大梯度的训练样本，这会破坏训练的稳定性。为了缓解这种情况，实践中常用的一种技术是梯度裁剪。其思想是在进行优化器步骤之前，在每次后向传递之后对梯度的范数强制施加一个限制。

给定（所有参数的）梯度 g，我们计算其 l_2-范数 $\|g\|_2$。如果此范数小于最大值 M，则我们保持 g 不变；否则，我们将 g 按因子 $\frac{M}{\|g\|_2+\epsilon}$ 缩小（其中添加了一个小的 ϵ，例如 10^{-6}，用于数值稳定性）。请注意，结果范数将略低于 M。

问题 (24) gradient_clipping: 实现梯度裁剪 (1 分)

可交付成果 (24.1): 编写一个实现梯度裁剪的函数。您的函数应接受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_gradient_clipping]` 并确保它通过 `uv run pytest -k test_gradient_clipping`。

5 训练循环

我们现在将最终组合我们构建的主要组件：分词后的数据、模型和优化器。

[8] 有时使用学习率再次上升（重启）的调度以帮助跳出局部最小值也很常见。

5.1 数据加载器 (Data Loader)

分词后的数据（例如，您在 `tokenizer_experiments` 中准备的数据）是单个词符序列 $x = (x_1, \ldots, x_n)$。即使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），一种常见的做法是将所有这些连接成单个词符序列，并在它们之间添加一个分隔符（例如 `<|endoftext|>` 词符）。

数据加载器将其转换为批次流，其中每个批次由 B 个长度为 m 的序列组成，并与相应的下一个词符配对，长度也为 m。例如，对于 $B = 1, m = 3$，一个潜在的批次将是 $([x_2, x_3, x_4], [x_3, x_4, x_5])$。

以这种方式加载数据简化了训练，原因有几个。首先，任何 $1 \le i \le n - m$ 都给出了一个有效的训练序列，因此采样序列是微不足道的。由于所有训练序列都具有相同的长度，因此无需填充输入序列，这提高了硬件利用率（也通过增加批次大小 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装入内存的大型数据集变得容易。

问题 (25) data_loading: 实现数据加载 (2 分)

可交付成果 (25.1): 编写一个函数，接收一个 numpy 数组 `x`（包含词符 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：采样的输入序列和相应的下一个词符目标。两个张量都应具有形状 `(batch_size, context_length)`，包含词符 ID，并且都应放置在请求的设备上。要根据我们提供的测试来测试您的实现，您将首先需要实现位于 `[adapters.run_get_batch]` 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（类似地，您稍后应该对您的模型使用相同的设备）。如果您在 CPU 上，可以使用 `'cpu'` 设备字符串，在 Apple Silicon（M* 芯片）上，可以使用 `'mps'` 设备字符串。有关 MPS 的更多信息，请查看这些资源：

- `https://developer.apple.com/metal/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加载文件内容。因此，您可以“假装”您拥有整个数据集在内存中。Numpy 通过 `np.memmap`（或者如果您最初使用 `np.save` 保存数组，则使用 `np.load` 的标志 `mmap_mode='r'`）实现了这一点，它将返回一个类似 numpy 数组的对象，该对象在您访问条目时按需加载它们。在训练期间从您的数据集（即 numpy 数组）采样时，请确保以内存映射模式加载数据集（通过 `np.memmap` 或标志 `mmap_mode='r'` 到 `np.load`，取决于您如何保存数组）。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看起来正确（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 (Checkpointing)

除了加载数据，我们还需要在训练时保存模型。在运行作业时，我们经常希望能够恢复由于某种原因中途停止的训练运行（例如，由于您的作业超时、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、在不同训练阶段从模型中抽样等）。

检查点应包含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，在 AdamW 的情况下，是矩估计）。最后，要恢复学习率调度，我们需要知道我们停止的迭代次数。PyTorch 使保存所有这些变得容易：每个 `nn.Module` 都有一个 `state_dict()` 方法，返回一个包含所有可学习权重的字典；我们可以稍后使用姊妹方法 `load_state_dict()` 恢复这些权重。任何 `nn.optim.Optimizer` 也是如此。最后，`torch.save(obj, dest)` 可以将一个对象（例如，一个字典，其中一些值是张量，但也包含像整数这样的常规 Python 对象）转储到一个文件（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 (26) checkpointing: 实现模型检查点 (1 分)

可交付成果 (26.1): 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 在这里支持路径或类文件对象）。一个典型的选择是让 `obj` 成为一个字典，但只要您以后可以加载您的检查点，您可以使用任何您想要的格式。此函数期望以下参数：

- `model: torch.nn.Module`
- `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，然后从中恢复模型和优化器状态。您的函数应返回保存到检查点的迭代次数。您可以使用 `torch.load(src)` 来恢复您在 `save_checkpoint` 实现中保存的内容，并使用模型和优化器中的 `load_state_dict` 方法将它们恢复到以前的状态。此函数期望以下参数：

- `src: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`
- `model: torch.nn.Module`
- `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，是时候将您实现的所有组件组合到您的主训练脚本中了。使使用不同超参数（例如，通过将它们作为命令行参数）轻松启动训练运行是值得的，因为您稍后将多次执行此操作以研究不同选择如何影响训练。

问题 (27) training_together: 组合起来 (4 分)

可交付成果 (27.1): 编写一个运行训练循环以在用户提供的输入上训练您的模型的脚本。特别地，我们建议您的训练脚本至少允许以下内容：

- 能够配置和控制各种模型和优化器超参数。
- 使用 `np.memmap` 对大型训练和验证数据集进行内存高效加载。
- 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，到控制台和/或像 Weights and Biases[a] 这样的外部服务）。

[a] `wandb.ai`

6 生成文本

既然我们可以训练模型，我们需要的最后一部分是从我们的模型生成文本的能力。回想一下，语言模型接收一个（可能批处理的）长度为 `sequence_length` 的整数序列，并产生一个大小为 `(sequence_length × vocab_size)` 的矩阵，其中序列的每个元素都是一个预测该位置之后下一个词的概率分布。我们现在将编写一些函数，将其转化为新序列的采样方案。

Softmax 按照标准惯例，语言模型输出是最终线性层的输出（“logits”），因此我们必须通过 softmax 操作将其转换为归一化概率，我们之前在公式 10 中看到过。

解码要从我们的模型生成文本（解码），我们将为模型提供一个前缀词符序列（“prompt”），并要求它产生一个预测序列中下一个词的词汇表上的概率分布。然后，我们将从此分布中采样词汇表项以确定下一个输出词符。

具体来说，解码过程的一步应该接收一个序列 $x_{1\ldots t}$ 并通过以下方程返回一个词符 x_{t+1}，

$$P(x_{t+1} = i | x_{1\ldots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)}$$

$$v = \text{TransformerLM}(x_{1\ldots t})_t \in \mathbb{R}^{\text{vocab\_size}}$$

其中 TransformerLM 是我们的模型，它接收一个 `sequence_length` 的序列作为输入，并产生一个大小为 `(sequence_length × vocab_size)` 的矩阵，我们取此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。

这为我们提供了一个基本的解码器，通过从这些单步条件中重复采样（将我们先前生成的输出词符附加到下一个解码时间步的输入）直到我们生成序列结束词符 `<|endoftext|>`（或用户指定的最大词符数）。

解码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。两个简单的解码技巧可以帮助解决这些问题。首先，在温度缩放 (temperature scaling) 中，我们使用温度参数 τ 修改我们的 softmax，其中新的 softmax 是

$$\text{softmax}(v, \tau)_i = \frac{\exp(v_i/\tau)}{\sum_{j=1}^{\text{vocab\_size}} \exp(v_j/\tau)} \tag{21}$$

注意设置 $\tau \to 0$ 如何使得 v 的最大元素占主导地位，并且 softmax 的输出变成一个集中在该最大元素上的 one-hot 向量。

其次，另一个技巧是 nucleus 或 top-p 采样，我们通过截断低概率词来修改采样分布。设 q 是我们从大小为 `vocab_size` 的（温度缩放的）softmax 得到的概率分布。使用超参数 p 的 Nucleus 采样根据以下方程产生下一个词符

$$P(x_{t+1} = i|q) = \begin{cases} \frac{q_i}{\sum_{j \in V(p)} q_j} & \text{if } i \in V(p) \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

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

问题 (28) decoding: 解码 (3 分)

可交付成果 (28.1): 实现一个从您的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，接收一些 $x_{1...t}$ 并采样一个补全，直到遇到 `<|endoftext|>` 词符）。
- 允许用户控制生成的最大词符数。
- 给定所需的温度值，在采样前对预测的下一个词分布应用 softmax 温度缩放。
- Top-p 采样（Holtzman et al., 2020；也称为 nucleus sampling），给定用户指定的阈值。

7 实验

现在是时候将所有内容组合在一起，并在预训练数据集上训练（小型）语言模型了。

7.1 如何运行实验和可交付成果

理解 Transformer 架构组件背后原理的最佳方法是实际修改它并亲自运行。没有什么能替代动手实践经验。为此，能够**快速**、**一致地**进行实验并**记录**您所做的事情非常重要。为了快速实验，我们将在小型模型（17M 参数）和简单数据集（TinyStories）上运行许多实验。为了保持一致性，您将系统地消融组件并改变超参数，为了记录，我们将要求您提交您的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**记录步骤数和挂钟时间**。您可能会发现像 Weights and Biases 这样的日志记录基础设施很有帮助。

问题 (29) experiment_log: 实验日志记录 (3 分)

可交付成果 (29.1): 对于您的训练和评估代码，创建实验跟踪基础设施，使您能够跟踪您的实验以及相对于梯度步数和挂钟时间的损失曲线。可交付成果：用于您的实验的日志记录基础设施代码和一个实验日志（记录您在本节下面的作业问题中尝试的所有事情的文档）。

7.2 TinyStories

我们将从一个非常简单的数据集（TinyStories; Eldan and Li, 2023）开始，模型将在其上快速训练，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。该数据集的一个示例如下所示。

示例 (6) tinystories_example: 来自 TinyStories 的一个示例

```
Once upon a time there was a little boy named Ben. Ben loved to explore the world around him. He saw many amazing things, like beautiful vases that were on display in a store. One day, Ben was walking through the store when he came across a very special vase. When Ben saw it he was amazed! He said, "Wow, that is a really amazing vase! Can I buy it?" The shopkeeper smiled and said, "Of course you can. You can take it home and show all your friends how amazing it is!" So Ben took the vase home and he was so proud of it! He called his friends over and showed them the amazing vase. All his friends thought the vase was beautiful and couldn't believe how lucky Ben was. And that's how Ben found an amazing vase in the store!
```

超参数调整我们将告诉您一些非常基本的超参数开始，并要求您找到一些其他有效的设置。

`vocab_size` 10000。典型的词汇表大小在数万到数十万之间。您应该改变这个值，看看词汇表和模型行为如何变化。

`context_length` 256。像 TinyStories 这样的简单数据集可能不需要很长的序列长度，但对于稍后的 OpenWebText 数据，您可能想要改变这个值。尝试改变这个值，看看它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 维略小，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，这对于 GPU 性能很好。

`ROPE theta` 参数 $\Theta = 10000$。

层数和头数 4 层，16 头。总的来说，这将给出大约 17M 的非嵌入参数，这是一个相当小的 Transformer。

处理的总词符数 327,680,000（您的批次大小 × 总步数 × 上下文长度应大致等于此值）。

您应该进行一些试验和错误，为以下其他超参数找到好的默认值：学习率、学习率预热、其他 AdamW 超参数 $(\beta_1, \beta_2, \epsilon)$ 和权重衰减。您可以在 Kingma and Ba [2015] 中找到此类超参数的一些典型选择。

组合起来现在您可以将所有内容组合起来，通过获取训练好的 BPE 分词器，对训练数据集进行分词，并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

数应导致在 1 个 H100 GPU 上的运行时间大约为 30-40 分钟。如果您的运行时间长得多，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时间的瓶颈，并且您的实现已正确批处理。

调试模型架构的提示和技巧我们强烈建议您熟悉 IDE 的内置调试器（例如 VSCode/PyCharm），与使用 print 语句进行调试相比，这将节省您的时间。如果您使用文本编辑器，可以使用像 `pdb` 这样的工具。调试模型架构时其他一些好的做法是：

- 开发任何神经网络架构时，常见的第一个步骤是在单个小批次上过拟合。如果您的实现正确，您应该能够快速将训练损失驱动到接近零。
- 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、模型权重和梯度的范数，以确保它们没有爆炸或消失。

问题 (30) learning_rate: 调整学习率 (3 分) (4 H100 小时)

可交付成果 (30.1): 学习率是要调整的最重要的超参数之一。以您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，则注意发散）。可交付成果：与多个学习率相关的学习曲线。解释您的超参数搜索策略。可交付成果：在 TinyStories 上验证损失（每个词符）最多为 1.45 的模型。

低资源/缩减提示：在 CPU 或 Apple Silicon 上训练少量步骤

如果您在 cpu 或 mps 上运行，您应该改为将处理的总词符数减少到 40,000,000，这将足以产生相当流畅的文本。您也可以将目标验证损失从 1.45 增加到 2.00。在 M3 Max 芯片和 36 GB RAM 上运行我们的解决方案代码，使用调整后的学习率，我们使用批次大小 × 总步数 × 上下文长度 = 32 × 5000 × 256 = 40,960,000 个词符，这在 cpu 上需要 1 小时 22 分钟，在 mps 上需要 36 分钟。在第 5000 步，我们达到了 1.80 的验证损失。一些额外的提示：

- 当使用 X 训练步骤时，我们建议调整余弦学习率衰减计划，使其恰好在步骤 X 结束其衰减（即达到最小学习率）。
- 使用 mps 时，请勿使用 TF32 内核，即不要设置 `torch.set_float32_matmul_precision('high')` 就像您可能对 cuda 设备所做的那样。我们尝试在 mps (torch 版本 2.6.0) 上启用 TF32 内核，发现后端会静默使用损坏的内核，导致训练不稳定。
- 您可以通过使用 `torch.compile` JIT 编译您的模型来加速训练。具体来说：
  - 在 cpu 上，使用以下命令编译您的模型 `model = torch.compile(model)`
  - 在 mps 上，您可以使用以下命令在一定程度上优化后向传递 `model = torch.compile(model, backend="aot_eager")` 截至 torch 版本 2.6.0，mps 不支持使用 Inductor 进行编译。

问题 (31): (c) 民间智慧认为最佳学习率处于“稳定性的边缘”。调查学习率发散点与您的最佳学习率有何关系。可交付成果：增加学习率的学习曲线，其中至少包括一次发散运行，

以及关于这与收敛率关系的分析。

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——它们让我们通过进行更大的矩阵乘法来从 GPU 获得更高的效率，但是否总是希望批次大小越大越好？让我们运行一些实验来找出答案。

问题 (32) batch_size_experiment: 批次大小变化 (1 分) (2 H100 小时)

可交付成果 (32.1): 将您的批次大小从 1 一直改变到 GPU 内存限制。至少尝试几个中间的批次大小，包括像 64 和 128 这样的典型大小。可交付成果：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。可交付成果：几句话讨论您关于批次大小及其对训练影响的发现。

有了您的解码器，我们现在可以生成文本了！我们将从模型生成，看看它有多好。作为参考，您应该得到至少与下面示例一样好的输出。

示例 (7) ts_generate_example: 来自 TinyStories 语言模型的样本输出

```
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.
```

低资源/缩减提示：在 CPU 或 Apple Silicon 上生成文本

如果您改用了处理 40M 词符的低资源配置，您应该会看到仍然类似于英语但不如上面流畅的生成结果。例如，我们从在 40M 词符上训练的 TinyStories 语言模型得到的样本输出如下：

```
Once upon a time, there was a little girl named Sue. Sue had a tooth that she
loved very much. It was his best head. One day, Sue went for a walk and met a
ladybug! They became good friends and played on the path together.
"Hey, Polly! Let's go out!" said Tim. Sue looked at the sky and saw that it was
difficult to find a way to dance shining. She smiled and agreed to help the
talking!"
As Sue watched the sky moved, what it was. She
```

这是精确的问题陈述和我们要求的内容：

问题 (33) generate: 生成文本 (1 分)

可交付成果 (33.1): 使用您的解码器和训练好的检查点，报告由您的模型生成的文本。您可能需要操纵解码器参数（温度、top-p 等）以获得流畅的输出。可交付成果：至少 256 个词符的文本转储（或直到第一个 `<|endoftext|>` 词符），以及对此输出流畅性的简要评论，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它的行为。我们现在将进行一些简单的消融和修改。

消融 1：层归一化人们常说层归一化对于 Transformer 训练的稳定性很重要。但也许我们想冒险一下。让我们从我们的 Transformer 块中移除 RMSNorm，看看会发生什么。
问题 (34) layer_norm_ablation: 移除 RMSNorm 并训练 (1 分) (1 H100 小时)
可交付成果 (34.1): 从您的 Transformer 中移除所有的 RMSNorms 并进行训练。在之前的最优学习率下会发生什么？您可以通过使用较低的学习率来获得稳定性吗？可交付成果：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。可交付成果：关于 RMSNorm 影响的几句评论。

让我们现在研究另一个乍一看似乎随意的层归一化选择。Pre-norm Transformer 块定义为

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

这是对原始 Transformer 架构的少数几个“共识”修改之一，原始架构使用了 post-norm 方法：

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

让我们回到 post-norm 方法，看看会发生什么。
问题 (35) pre_norm_ablation: 实现 post-norm 并训练 (1 分) (1 H100 小时)
可交付成果 (35.1): 将您的 pre-norm Transformer 实现修改为 post-norm。使用 post-norm 模型进行训练，看看会发生什么。可交付成果：与 pre-norm 相比的 post-norm transformer 的学习曲线。

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 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（使用 RoPE）与完全不包含位置嵌入（NoPE）的情况。事实证明，仅解码器 transformers，即像我们实现的那样具有 causal mask 的 transformers，理论上可以在没有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]。我们现在将凭经验测试 NoPE 与 RoPE 的表现如何。
问题 (36) no_pos_emb: 实现 NoPE (1 分) (1 H100 小时)
可交付成果 (36.1): 修改您使用 RoPE 的 Transformer 实现，以完全移除位置嵌入信息，看看会发生什么。可交付成果：比较 RoPE 和 NoPE 性能的学习曲线。

消融 3：SwiGLU vs. SiLU 接下来，我们将遵循 Shazeer [2020] 并测试门控在前馈网络中的重要性，方法是比较 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：

$$\text{FFN}_{\text{SiLU}}(x) = W_2\text{SiLU}(W_1 x). \tag{23}$$

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 $d_{ff} = \frac{8}{3}d_{model}$（同时确保 $d_{ff} \pmod{64} = 0$，以利用 GPU 张量核心）。在您的 FFN_{SiLU} 实现中，您应该设置 $d_{ff} = 4 \times d_{model}$，以大致匹配 SwiGLU 前馈网络的参数数量（它有三个而不是两个权重矩阵）。

问题 (37) swiglu_ablation: SwiGLU vs. SiLU (1 分) (1 H100 小时)

可交付成果 (37.1): 比较 SwiGLU 和 SiLU 前馈网络性能的学习曲线，参数数量大致匹配。可交付成果：几句话讨论您的发现。

低资源/缩减提示：GPU 资源有限的在线学生应在 TinyStories 上测试修改

在作业的剩余部分，我们将转向一个更大规模、更嘈杂的网络数据集（OpenWebText），尝试架构修改并（可选地）向课程排行榜提交。在 OpenWebText 上训练一个 LM 达到流畅程度需要很长时间，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更标准的从网络爬取创建的预训练数据集。OpenWebText [Gokaslan et al., 2019] 的一个小样本也作为单个文本文件提供：请参阅第 1 节了解如何访问此文件。这是 OpenWebText 的一个示例。注意文本如何更加真实、复杂和多样化。您可能想浏览训练数据集，以了解网络爬取语料库的训练数据是什么样的。

示例 (8) owt_example: 来自 OWT 的一个示例

```
Baseball Prospectus director of technology Harry Pavlidis took a risk when he
hired Jonathan Judge. Pavlidis knew that, as Alan Schwarz wrote in The Numbers
Game, "no corner of American culture is more precisely counted, more passionately
quantified, than performances of baseball players." With a few clicks here and
there, you can findout that Noah Syndergaard's fastball revolves more than 2,100
times per minute on its way to the plate, that Nelson Cruz had the game's highest
average exit velocity among qualified hitters in 2016 and myriad other tidbits that
seem ripped from a video game or science fiction novel. The rising ocean of data
has empowered an increasingly important actor in baseball's culture: the
analytical hobbyist.
That empowerment comes with added scrutiny on the measurements, but also on the
people and publications behind them. With Baseball Prospectus, Pavlidis knew all
about the backlash that accompanies quantitative imperfection. He also knew the
site's catching metrics needed to be reworked, and that it would take a learned
mind someone who could tackle complex statistical modeling problems to complete
the job.
"He freaks us out." Harry Pavlidis
Pavlidis had a hunch that Judge "got it" based on the latter's writing and their
interaction at a site-sponsored ballpark event. Soon thereafter, the two talked
over drinks. Pavlidis' intuition was validated. Judge was a fit for the position –
```

`better yet, he was a willing fit. "I spoke to a lot of people," Pavlidis said, "he was the only one brave enough to take it on." [...]`

注意：对于此实验，您可能需要重新调整超参数，例如学习率或批次大小。

问题 (38) main_experiment: 在 OWT 上进行实验 (2 分) (3 H100 小时)

可交付成果 (38.1): 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。这个模型表现如何？可交付成果：您在 OpenWebText 上的语言模型的学习曲线。描述与 TinyStories 损失的差异 - 我们应该如何解释这些损失？可交付成果：从 OpenWebText LM 生成的文本，格式与 TinyStories 输出相同。此文本的流畅性如何？为什么即使我们拥有与 TinyStories 相同的模型和计算预算，输出质量也更差？

7.5 您自己的修改 + 排行榜

恭喜您到达了这一点。您快完成了！您现在将尝试改进 Transformer 架构，并看看您的超参数和架构与班上其他学生相比如何。

排行榜规则除了以下内容外，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，您可以自由地做任何您想做的事情。如果您正在寻找一些关于实现什么的想法，您可以查看这些资源：

- 最先进的开源 LLM 系列，例如 Llama 3 [Grattafiori et al., 2024] 或 Qwen 2.5 [Yang et al., 2024]。
- NanoGPT speedrun 仓库 (`https://github.com/KellerJordan/modded-nanogpt`)，社区成员发布了许多有趣的修改，用于“速通”小型语言模型预训练。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））。如果您确实尝试权重绑定，您可能需要减小嵌入/LM 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 的小子集或 TinyStories 上测试这些。需要注意的是，您在此排行榜中发现的一些有效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 (39) leaderboard: 排行榜 (6 分) (10 H100 小时)

可交付成果 (39.1): 您将在上面的排行榜规则下训练一个模型，目标是在 1.5 H100 小时内最小化您的语言模型的验证损失。可交付成果：记录的最终验证损失，一个相关的学习曲线，清楚地显示小于 1.5 小时的挂钟时间 x 轴，以及您所做工作的描述。我们期望排行榜提交至少击败 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`。

参考文献

参考文献

[1] Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent English?, 2023. arXiv:2305.07759.

[2] Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. OpenWebText corpus. `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

[3] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proc. of ACL, 2016.

[4] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords, 2019. arXiv:1909.03341.

[5] Philip Gage. A new algorithm for data compression. C Users Journal, 12(2):23-38, February 1994. ISSN 0898-9788.

[6] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[7] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Proc. of NeurIPS, 2017.

[9] Toan Q. Nguyen and Julian Salazar. Transformers without tears: Improving the normalization of self-attention. In Proc. of IWSWLT, 2019.

[10] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the Transformer architecture. In Proc. of ICML, 2020.

[11] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. arXiv:1607.06450.

[12] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. arXiv:2302.13971.

[13] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In Proc. of NeurIPS, 2019.

[14] Aaron Grattafiori, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[15] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units, 2016. arXiv:1606.08415.

[16] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017. URL https://arxiv.org/abs/1702.03118.

[17] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks, 2017. URL https://arxiv.org/abs/1612.08083.

[18] Noam Shazeer. GLU variants improve transformer, 2020. arXiv:2002.05202.

[19] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021. arXiv:2104.09864.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Proc. of ICLR, 2015.

[21] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In Proc. of ICLR, 2019.

[22] Tom B. Brown, et al. Language models are few-shot learners. In Proc. of NeurIPS, 2020.

[23] Aakanksha Chowdhery, et al. PaLM: Scaling language modeling with pathways, 2022. arXiv:2204.02311.

[24] Jared Kaplan, et al. Scaling laws for neural language models, 2020. arXiv:2001.08361.

[25] Jordan Hoffmann, et al. Training compute-optimal large language models, 2022. arXiv:2203.15556.

[26] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In Proc. of ICLR, 2020.

[27] Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. In EMNLP-IJCNLP, 2019.

[28] Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. In NeurIPS, 2023.

[29] An Yang, et al. Qwen2.5 technical report. arXiv preprint arXiv:2412.15115, 2024.