

CS336 作业1 (基础) : 构建Transformer语言模型

版本1.0.4

CS336课程团
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语言模型所需的所有组件, 并训练一些模型。

您将实现的内容

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)

你将运行的内容

1. 在TinyStories数据集上训练一个BPE分词器。
2. 使用训练好的分词器处理数据集, 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语言模型生成样本并评估困惑度。
5. 在OpenWebText上训练模型, 并将获得的困惑度提交到排行榜。

你可以使用的内容我们希望您从头开始构建这些组件。具体而言, 不得使用 `torch.nn`、`torch.nn.functional` 或 `torch.optim` 中的任何定义, 除了以下例外:

- `torch.nn.Parameter`
- `torch.nn` 中的容器类 (例如, `Module`、`ModuleList`、`Sequential`等) ¹
- 基础类

您可以使用任何其他PyTorch定义。如果你想使用某个函数或类但不确定是否被允许, 请随时在Slack上询问。如有疑问, 请考虑使用它是否会损害作业的“从零开始”精神。

¹参见 [PyTorch.org/docs/stable/nn.html#containers](https://pytorch.org/docs/stable/nn.html#containers) 获取完整 列表。

关于人工智能工具的声明允许使用ChatGPT等大型语言模型进行低级编程问题或关于语言模型的高级概念性问题提示，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集成开发环境中的AI自动补全功能（例如Cursor Tab、GitHub CoPilot）（不过非AI自动补全，例如自动补全函数名称是完全没问题的）。我们发现AI自动补全会大大增加深入理解课程内容的难度。

代码结构说明所有作业代码及本文档均可在GitHub上获取：

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

请 `git clone` 该代码库。若有任何更新，我们会通知您以便 `git pull` 获取最新版本。

1. `cs336_basics/*`：这是您编写代码的区域。请注意此处没有任何预设代码——您可以完全从零开始自由发挥！2. `adapters.py`：您的代码需要实现一系列指定功能。针对每个功能模块（例如缩放点积注意力），只需通过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`的修改不应包含任何实质性逻辑，这属于粘合代码。3. `test_*.py`：这里包含所有您必须通过的测试用例（例如`test_scaled_dot_product_attention`），这些测试将调用`adapters.py`中定义的接口。请勿编辑测试文件。

如何提交请将以下文件提交至Gradescope：

- `writeup.pdf`：回答所有书面问题。请使用文本编辑器排版您的答案。
- `code.zip`：包含您编写的所有代码。

如需提交至排行榜，请向以下地址提交PR：

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

详细提交说明请参阅排行榜代码库中的 `README.md`。

数据集获取途径本作业将使用两个预处理数据集：TinyStories [Eldan与Li2023]，以及OpenWebText [Gokaslan等人 2019]。两个数据集均为单一大型纯文本文件。若您随课程完成作业，可在任意非头节点机器的 `/data` 找到这些文件。

如果您正在家中跟随操作，可以通过 `README.md`中的命令下载这些文件。

低资源/缩减规模提示：初始化

在整个课程的作业讲义中，我们将提供关于在较少或没有GPU资源的情况下完成作业各部分的建议。例如，我们有时会建议**缩减规模**你的数据集或模型大小，或者解释如何在MacOS集成GPU或中央处理器上运行训练代码。你会在蓝色框（如此处所示）中找到这些“低资源提示”。即使你是注册的斯坦福大学学生，能够使用课程机器，这些提示也可能帮助你更快地迭代并节省时间，因此我们建议你阅读它们！

低资源/缩减规模提示：在苹果芯片或中央处理器上完成作业1

使用官方解答代码，我们可以在Apple M3上训练一个语言模型，生成相当流畅的文本。配备36GB内存的Max芯片，在Metal GPU（MPS）上耗时不到5分钟，使用中央处理器约需30分钟。如果这些术语对您来说很陌生，请不必担心！只需知道，只要您拥有配置尚可的笔记本电脑，且实现正确高效，就能训练出一个小型语言模型，用以生成流畅度尚佳的简易儿童故事。

在作业的后续部分，我们将说明如果您使用CPU或MPS需要做出哪些更改。

2 字节对编码 (BPE) 分词器

在作业的第一部分，我们将训练并实现一个字节级字节对编码（BPE）分词器 [Sennrich等人，2016, Wang等人，2019]。具体来说，我们将把任意（Unicode）字符串表示为字节序列，并在此字节序列上训练我们的BPE分词器。之后，我们将使用该分词器将文本（字符串）编码为标记（整数序列）以进行语言建模。

2.1 Unicode标准

Unicode 是一种文本编码标准，它将字符映射到整型码点。截至 Unicode 16.0（2024年9月发布），该标准定义了 168 种文字体系中的 154,998 个字符。例如，字符“s”的码点为 115（通常表示为 U+0073，其中 U+ 是约定前缀，0073 是 115 的十六进制形式），字符“牛”的码点为 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形式。`chr()` 函数则将整数 Unicode 码点转换为包含对应字符的字符串。

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

问题 (unicode1): 理解 Unicode (1 分)

(a) `chr(0)` 返回什么 Unicode 字符？

可交付成果：用一句话回答。

(b) 该字符的字符串表示(`__repr__()`)与其打印表示有何不同？

可交付成果：用一句话回答。

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

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

可交付成果：用一句话回答。

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，因为词汇表会变得过于庞大（约15万个条目）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是互联网的主导编码（超过98%的网页使用）。

要将Unicode字符串编码为UTF-8，我们可以使用Python中的 `encode()` 函数。要访问Python `bytes` 对象的底层字节值，我们可以对其进行迭代（例如，调用 `list()`）。最后，我们可以使用 `decode()` 函数将UTF-8字节串解码为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'>
>>> # Get the byte values for the encoded string (integers from 0 to 255).
>>> 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]
>>> # One byte does not necessarily correspond to one Unicode character!
>>> 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），我们实质上是在将码点序列（范围在0到154,997的整数）转换为字节值序列（范围在0到255的整数）。256长度的字节词汇表更易于处理。使用字节级分词时，我们无需担心超出词汇表的词元，因为我们知道任何输入文本都可以表示为0到255的整数序列。

问题(unicode2): Unicode编码 (3分)

(a) 为什么更倾向于在UTF-8编码的字节上训练我们的分词器，而不是UTF-16或UTF-32？比较这些编码对各种输入字符串的输出可能会有所帮助。

可交付成果: 一到两句话的回复。(b) 考虑以下（错误的）函数，其目的是将UTF-8字节串解码为

一个Unicode字符串。为什么这个函数不正确？提供一个会产生不正确结果的输入字节序列的示例。

```

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: 一个示例输入字节序列，`decode_utf8_bytes_to_str_wrong` 会为其产生不正确输出，并附上一句解释说明函数为何不正确的说明。

(c) 给出一个两字节序列，它无法解码为任何Unicode字符。**可交付成果:** 一个示例，并附上一句解释说明。

2.3 子词分词

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

一个包含10个单词的句子在词级语言模型中可能只有10个标记长度，但在字符级模型中可能长达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进行更多计算。此外，对字节序列进行语言建模很困难，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折中方案。请注意，字节级分词器的词汇表包含256个条目（字节值为0到255）。子词分词器通过增大词汇表大小来换取更好的输入字节序列压缩效果。例如，如果字节序列b'the'经常出现在我们的原始文本训练数据中，将其分配为词汇表中的一个条目，就可以将这个3标记序列缩减为单个标记。

我们如何选择这些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），这是一种通过迭代方式将最频繁出现的字节对替换（“合并”）为单个未使用新索引的压缩算法。请注意，该算法通过向词汇表添加子词标记来最大化输入序列的压缩效率——如果某个词在输入文本中出现次数足够多，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，这使我们在词汇表外处理和可管理的输入序列长度方面获得了两全其美的效果。构建BPE分词器词汇表的过程被称为BPE分词器训练。

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标记到整数ID的一对一映射。由于我们正在训练一个字节级BPE分词器，我们的初始词汇表就是所有字节的集合。因为有256个可能的字节值，我们的初始词汇表大小为256。

预标记化一旦你有了一个词汇表，原则上，你可以统计字节在文本中相邻出现的频率，并从最频繁的字节对开始合并它们。然而，这在计算上相当昂贵，因为每次合并时我们都必须对整个语料库进行一次完整遍历。此外，直接跨语料库合并字节可能会导致仅在标点符号上不同的标记（例如，dog! 与 dog.）。这些标记将获得完全不同的标记ID，即使它们很可能具有高度的语义相似性（因为它们仅在标点符号上不同）。

为避免这种情况，我们对语料库进行预分词。您可以将此视为在语料库上进行的粗粒度分词，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次的预分词单元。在这种情况下，当我们统计字符't'和'e'相邻出现的频率时，会看到单词'text'中't'和'e'相邻，从而可以将它们的计数增加10，而无需遍历整个语料库。由于我们正在训练字节级字节对编码模型，每个预分词单元都表示为UTF-8字节序列。

Sennrich等人最初的字节对编码实现 [2016] 通过简单按空白字符分割（即s.split(" ")）进行预分词。相比之下，我们将使用来自 github.com/openai/tiktoken/pull/234/files 的基于正则表达式的预分词器（由GPT-2使用；Radford等人，2019）：

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

通过交互式地使用此预分词器对某些文本进行分割，可能有助于更好地理解其行为：

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

但在代码中使用时，你应该使用 `re.finditer` 来避免在构建从预分词单元到其计数的映射时存储预分词单词。

计算BPE合并规则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合并规则（即训练BPE分词器）。从高层来看，BPE算法会迭代统计每个字节对，并识别出频率最高的对（“A”、“B”）。然后，这个最频繁对（“A”、“B”）的每个出现都会被合并，即替换为新标记“AB”。这个新合并的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是初始词汇表大小

（本例中为256）加上训练期间执行的BPE合并操作数量。为了在BPE训练期间提高效率，我们不考虑跨越预标记边界的对。² 在计算合并规则时，通过优先选择字典序更大的对来确定性解决对频率相同的情况。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')
```

特殊词元通常，某些字符串（例如 `<|endoftext|>`）用于编码元数据（例如文档之间的边界）。在对文本进行编码时，通常希望将某些字符串视为“特殊标记”，这些标记永远不应拆分为多个词元（即始终保留为单个词元）。例如，序列结束字符串 `<|endoftext|>` 应始终保留为单个词元（即单个整数ID），以便我们知道何时停止从语言模型生成。这些特殊标记必须添加到词汇表中，以便它们具有对应的固定标记ID。

Sennrich等人 [2016] 的算法1包含一个低效的BPE分词器训练实现（基本上遵循我们上面概述的步骤）。作为第一个练习，实现并测试此函数可能有助于检验你的理解。

示例（`bpe_example`）：BPE训练示例

以下是来自Sennrich等人 [2016]的风格化示例。考虑一个由以下文本组成的语料库

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

且词汇表中包含一个特殊标记 `<|endoftext|>`。

词汇表 我们使用特殊标记 `<|endoftext|>` 和256个字节值来初始化词汇表。

预分词 为简化流程并专注于合并操作，本示例假设预分词仅按空白字符进行分割。经过预分词和计数后，我们得到频率表。

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

²请注意原始BPE框架 [Sennrich等人 2016] 要求包含词尾标记。在训练字节级BPE模型时我们不添加词尾标记，因为所有字节（包括空白字符和标点符号）都已包含在模型词汇表中。由于我们显式表示空格和标点符号，学习到的BPE合并规则自然会反映这些词边界。

将其表示为 `dict[tuple[bytes], int]` 十分方便，例如 `{(1,0,w):5 ...}`。请注意，即使是单个字节在 Python 中也是 `bytes` 对象。Python 中没有 `byte` 类型来表示单个字节，就像没有 `char` 类型来表示单个字符一样。

合并规则 我们首先查看每个连续的字节对，并统计它们出现的单词频率`{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')` 这两个对出现频率相同，我们选择字典序更大的对 `('st')`。然后我们会合并预标记，最终得到`{(1,0,w): 5, (1,0,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（出现次数为9），我们将合并成`{(1,0,w): 5, (1,0,w,e,r): 2, (w,i,d,est): 3, (n,e,w,est): 6}`。继续这个过程，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', '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']`，而我们的词汇表元素将是 `[<|endoftext|>, [...256 BYTE CHARS], st, est, ow, low, west, ne]`。使用这个词汇表和合并规则集，单词 `newest` 将被分词为 `[ne, west]`。

2.5 实验BPE分词器训练

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

并行化预标记化你会发现一个主要瓶颈是预标记化步骤。你可以通过使用内置库 `multiprocessing`并行化你的代码来加速预标记化。具体来说，我们建议在并行实现预标记化时，对语料库进行分块处理，同时确保数据块边界出现在特殊标记的开头。你可以直接使用以下链接中的起始代码来获取数据块边界，然后利用这些边界在进程间分配工作：

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，你始终可以按此方式拆分。无需担心接收到不包含 `<|endoftext|>` 的超大语料库这种边缘情况。

预标记化前移除特殊标记在使用正则表达式模式进行预标记化之前（使用 `re.finditer`），你应当从语料库（或数据块，若采用并行实现）中剔除所有特殊标记。确保基于特殊标记进行**分割**，以防止跨越它们所分隔的文本发生合并。例如，若你的语料库（或数据块）形如 `[Doc 1]<|endoftext|>[Doc2]`，则应在特殊标记 `<|endoftext|>`处分割，并分别对 `[Doc 1]` 和 `[Doc 2]` 进行预分词，从而避免跨文档边界的合并。这可通过

```
re.split      "|"] 实现，以作
.join(special_tokens)      re.escape      |
为分隔符（需谨慎使用，因可能出现在特殊标记中）。测试用例 test_train_bpe_special_tokens 将对此进行验证。
```

优化合并步骤上述示例中BPE训练的朴素实现速度较慢，因为每次合并都需要迭代所有字节对以识别最频繁的对。然而，每次合并后唯一发生变化的对计数是与被合并对重叠的那些。因此，可以通过索引所有对计数并增量更新这些计数来提高BPE训练速度，而不是显式迭代每个字节对来统计对频率。通过这种缓存过程可以获得显著加速，但我们注意到BPE训练的合并部分无法在Python中并行化。

低资源/规模缩减提示：性能分析

您应该使用性能分析工具如 `cProfile` 或 `scalene` 来识别实现中的瓶颈，并专注于优化这些部分。

低资源/缩减规模提示：“缩减规模”

我们建议您不要直接在整个TinyStories数据集上训练分词器，而是先在数据的一个小子集（即“调试数据集”）上进行训练。例如，您可以在TinyStories验证集上训练分词器，该验证集包含22K个文档，而非2.12M个。这展示了尽可能通过缩减规模来加速开发的通用策略：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量：您希望调试集足够大，以保持与完整配置相同的瓶颈（这样您所做的优化才能泛化），但又不能太大以至于运行时间过长。

问题（train_bpe）：BPE分词器训练（15分）

可交付成果：编写一个函数，该函数在给定输入文本文件路径的情况下，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路径。

vocab_size: `int` 定义最终词汇表最大大小的正整数（包括初始字节词汇表、通过合并产生的词汇表项以及任何特殊标记）。

special_tokens: `list[str]` 要添加到词汇表中的字符串列表。这些特殊标记不会影响BPE训练。

您的BPE训练函数应返回最终的词汇表和合并规则：

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

merges: `list[tuple[bytes, bytes]]` 训练产生的BPE合并规则列表。每个列表项是一个 `tuple` 的 `bytes` (`<token1>`, `<token2>`)，表示 `<token1>` 与 `<token2>`进行了合并。合并规则应按创建顺序排列。

要测试你的BPE训练函数是否符合我们提供的测试，你需要先在 `[adapters.run_train_bpe]` 处实现测试适配器。然后运行 `uv run pytest tests/test_train_bpe.py`。你的实现应该能够通过所有测试。可选地（这可能耗时较长），你可以使用某些系统语言实现训练方法的关键部分，例如C++（可考虑 `cppyy`）或Rust（使用 `PyO3`）。如果这样做，请注意哪些操作需要复制数据，哪些可以直接从Python内存读取，并确保留下构建说明，或确保仅使用 `pyproject.toml`即可构建。另请注意，GPT-2正则表达式在大多数正则引擎中支持不佳，在支持的引擎中也大多运行缓慢。我们已验证 `Oniguruma`速度合理且支持负向先行断言，但Python中的 `regex` 包速度甚至更快。

问题 (train_bpe_tinystories)：在TinyStories上进行字节对编码训练 (2分)

(a) 在TinyStories数据集上训练一个字节级BPE分词器，最大词汇表大小为10,000。确保将TinyStories `<|endoftext|>` 特殊标记添加到词汇表中。将最终的词汇表和合并规则序列化到磁盘以供进一步检查。训练耗时多少小时和内存？词汇表中最长的词元是什么？这合理吗？

资源需求： ≤ 30 分钟（无需图形处理器）， ≤ 30GB内存

提示您应该能够在预分词阶段使用 `multiprocessing`，并基于以下两个事实，使字节对编码训练时间控制在2分钟以内：

(a) `<|endoftext|>` 词元用于分隔数据文件中的文档。(b) `<|endoftext|>` 词元在应用BPE合并规则前会作为特例处理。

可交付成果：一至两句话的答复。

(b) 对您的代码进行性能分析。分词器训练过程中哪个环节最耗时？**可交付成果：**一至两句话的答复。

接下来，我们将在OpenWebText数据集上尝试训练一个字节级BPE分词器。和之前一样，我们建议查看数据集以更好地理解其内容。

问题 (train_bpe_expts_owt)：在OpenWebText上进行BPE训练 (2分)

(a) 在OpenWebText数据集上训练一个字节级BPE分词器，最大词汇表大小为32,000。将生成的词汇表和合并规则序列化到磁盘以供进一步检查。词汇表中最长标记是什么？它合理吗？

资源需求： ≤ 12 小时（无需图形处理器）， ≤ 100GB内存

可交付成果：一到两句话的回复。

(b) 比较并对比在TinyStories和OpenWebText上训练得到的分词器。

可交付成果：一到两句话的回复。

2.6 BPE分词器：编码与解码

在作业的前一部分中，我们实现了一个函数来在输入文本上训练BPE分词器，以获取分词器词汇表和BPE合并规则列表。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规则列表，并使用它们将文本编码为标记ID或从标记ID解码为文本。

2.6.1 编码文本

BPE编码文本的过程与我们训练BPE词汇表的方式相似。包含几个主要步骤。

步骤1：预分词。我们首先对序列进行预分词，并将每个预分词单元表示为UTF-8字节序列，就像在BPE训练中所做的那样。我们将在每个预分词单元内将这些字节合并为词汇元素，每个预分词单元独立处理（不跨越预分词边界进行合并）。

步骤2：应用合并规则。然后我们获取BPE训练期间创建的词汇元素合并序列，并按相同的创建顺序将其应用于预分词单元。

示例 (bpe_encoding) : BPE编码示例

例如，假设我们的输入字符串是 'the cat ate'，词汇表为 {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'}，学习到的合并规则是 [(b't', b'h'), (b' ', b'c'), (b' ', 'a'), (b'th', b'e'), (b' a', b't')]. 首先，预分词器会将此字符串拆分为 ['the', ' cat', ' ate']. 然后，我们将查看每个预分词单元并应用BPE合并。第一个预分词单元 'the' 最初表示为 [b't', b'h', b'e']. 查看我们的合并规则列表，我们确定第一个适用的合并是 (b't', b'h')，并用它将预分词单元转换为 [b'th', b'e']. 然后，我们回到合并规则列表，确定下一个适用的合并是 (b'th', b'e')，它将预分词单元转换为 [b'the']. 最后，再次查看合并规则列表，我们发现没有更多适用于该字符串的合并（因为整个预分词单元已合并为单个词元），于是我们完成了BPE合并的应用。对应的整数序列是 [9]。

对其余预分词单元重复此过程，我们看到预分词单元 ' cat' 在应用BPE合并后表示为 [b' c', b'a', b't']，这变成了整数序列 [7, 1, 5]。最后一个预分词单元 ' ate' 在应用BPE合并后是 [b' at', b'e']，这变成了整数序列 [10, 3]。因此，我们输入字符串的编码最终结果是 [9, 7, 1, 5, 10, 3]。

特殊标记。 您的分词器应能够在编码文本时正确处理用户定义的特殊标记（在构建分词器时提供）。

内存考量。 假设我们需要对一个无法装入内存的大型文本文件进行分词。为了高效处理这个大文件（或任何其他数据流），我们需要将其分解为可管理的数据块并依次处理每个数据块，从而使内存复杂度保持恒定，而非与文本大小呈线性关系。在此过程中，我们需要确保标记不会跨越数据块边界，否则得到的分词结果将与在内存中直接处理整个序列的简单方法不同。

2.6.2 文本解码

要将整数标记ID序列解码回原始文本，我们只需在词汇表中查找每个ID对应的条目（字节序列），将它们连接起来，然后将字节解码为Unicode字符串。请注意，输入ID不保证能映射到有效的Unicode字符串（因为用户可以输入任意整数ID序列）。如果输入标记ID无法生成有效的Unicode字符串，您应该使用官方Unicode替换字符U+FFFD替换格式错误的字节。³ `errors` 参数的 `bytes.decode` 控制着Unicode解码错误的处理方式，使用 `errors='replace'` 会自动用替换标记替换格式错误的数据。

问题 (tokenizer) : 实现分词器 (15分)

可交付成果：实现一个 `Tokenizer` 类，该类在给定词汇表和合并规则列表的情况下，能够将文本编码为整数ID，并将整数ID解码为文本。您的分词器还应支持用户提供的特殊标记（如果它们尚未存在于词汇表中，则将其追加到词汇表）。我们推荐以下接口：

```
def __init__(self, vocab, merges, special_tokens=None) 根据给定的词汇表、合并规则列表以及（可选）特殊标记列表构建一个分词器。该函数应接受
```

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](https://en.wikipedia.org/wiki/Specials_(Unicode_block)#Replacement_character)。

以下参数：

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

`def from_files(cls, vocab_filepath, merges_filepath, special_tokens=None)` 类方法，用于从序列化的词汇表和合并规则列表（格式应与BPE训练代码输出相同）以及（可选）特殊标记列表构造并返回一个 `Tokenizer`。该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。

`def encode_iterable(self, iterable: Iterable[str]) -> Iterator[int]` 给定字符串的可迭代对象（例如Python文件句柄），返回一个惰性生成标记ID的生成器。这对于无法直接加载到内存中的大文件进行内存高效标记化是必需的。

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

要测试你的 `Tokenizer` 是否符合我们提供的测试，你需要先在 `[adapters.get_tokenizer]`处实现测试适配器。然后运行 `uv run pytest tests/test_tokenizer.py`。你的实现应能通过所有测试。

2.7 实验

问题 (tokenizer_experiments)：分词器实验（4分）

(a) 从 `TinyStories` 和 `OpenWebText` 中采样 10 个文档。使用你先前训练的 `TinyStories` 和 `OpenWebText` 分词器（词汇表大小分别为 10K 和 32K），将这些采样的文档编码为整数ID。每个分词器的压缩率（字节/标记）是多少？

可交付成果：一到两句话的回复。

(b) 如果你使用 `TinyStories` 分词器对 `OpenWebText` 样本进行分词会发生什么？比较压缩率和/或定性描述发生的情况。

可交付成果：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）。需要多长时间才能

对Pile数据集（825GB文本）进行分词？

可交付成果：一两句话的回复。(d) 使用你的 `TinyStories` 和 `OpenWebText` 分词器，将相应的训练和开发数据集编码为整数标记ID序列。

我们稍后将使用这些数据来训练我们的语言模型。建议将标记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：一到两句话的回复。

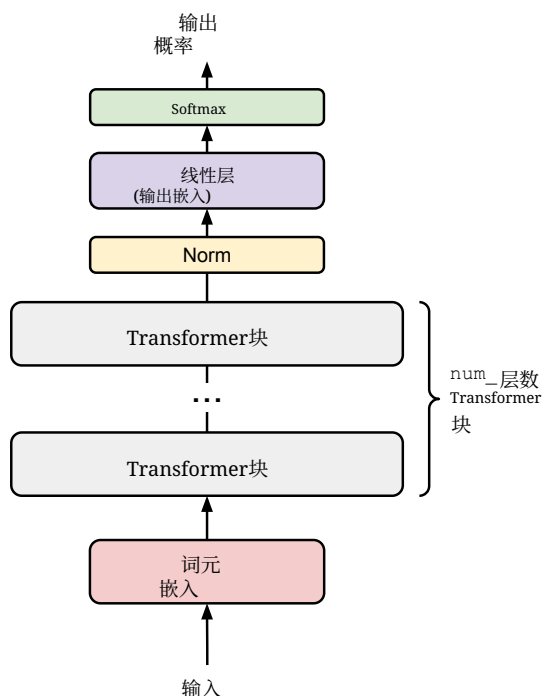


图1：我们的Transformer语言模型概览。

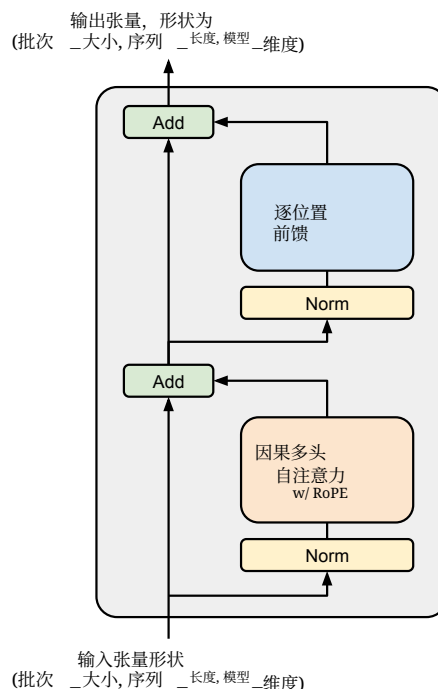


图 2：预归一化Transformer块。

3 Transformer语言模型架构

语言模型接收一个批处理的整数标记ID序列作为输入（即形状为 `torch.Tensor` 的 `(batch_size, sequence_length)`），并返回一个在词汇表上的（批处理）归一化概率分布（即一个形状为 `(batch_size, sequence_length, vocab_size)`的PyTorch张量），其中预测的分布是针对每个输入标记的下一个词。在训练语言模型时，我们使用这些下一词预测来计算实际下一个词与预测下一个词之间的交叉熵损失。在推理过程中从语言模型生成文本时，我们获取最终时间步（即序列中的最后一项）的预测下一词分布来生成序列中的下一个标记（例如，通过取概率最高的标记、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。我们将首先对模型进行高层描述，然后逐步详细介绍各个组件。

3.1 Transformer语言模型

给定一个标记ID序列，Transformer语言模型使用输入嵌入将标记ID转换为稠密向量，将嵌入后的标记通过 `num_layers` 个Transformer块传递，然后应用学习的线性投影（即“输出嵌入”或“语言模型头部”）来生成预测的下一标记逻辑值。示意图请参见图1。

3.1.1 词元嵌入

在第一步中，Transformer嵌入（批处理）的令牌ID序列，生成包含词元身份信息的向量序列（图1中的红色块）。

更具体地说，给定一个令牌ID序列，Transformer语言模型使用词元嵌入层来生成向量序列。每个嵌入层接收形状为 $(\text{batch_size}, \text{sequence_length})$ 的整数张量，并生成形状为 $(\text{batch_size}, \text{sequence_length}, d_{\text{model}})$ 的向量序列。

3.1.2 预归一化Transformer块

经过嵌入处理后，激活值会通过多个结构相同的神经网络层进行处理。标准的仅解码器Transformer语言模型包含 num_layers 个相同层（通常称为Transformer“块”）。每个Transformer块接收形状为

$(\text{batch_size}, \text{sequence_length}, d_{\text{model}})$ 的输入，并返回形状为 $(\text{batch_size}, \text{sequence_length}, d_{\text{model}})$ 的输出。每个块通过自注意力机制聚合序列信息，并通过前馈层进行非线性变换。

3.2 输出归一化与嵌入

经过 num_layers 个Transformer块后，我们将提取最终激活值并将其转换为词汇表上的分布。

我们将实现“前归一化”Transformer块（详见§3.5），这要求在最终Transformer块之后额外使用层归一化（详见下文），以确保其输出得到适当缩放。

在此归一化之后，我们将使用标准的学习线性变换将Transformer块的输出转换为预测的下一个词元逻辑值（参见Radford等人 [2018] 的公式2）。

3.3 备注：批处理、爱因斯坦求和与高效计算

在整个Transformer中，我们将对多个批次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 forward 操作。
- **序列长度**：像RMSNorm和前馈网络这样的“逐位置”操作会对序列中的每个位置进行相同运算。
- **注意力头**：在“多头”注意力机制中，注意力操作会跨注意力头进行批处理。

我们需要一种能充分利用GPU性能、且便于阅读理解的便捷操作方式。许多PyTorch操作可以接收张量开头多余的“类批次”维度，并高效地跨这些维度重复/广播操作。

例如，假设我们正在执行一个逐位置的批处理操作。我们有一个形状为 D 的“数据张量” $(\text{batch_size}, \text{sequence_length}, d_{\text{model}})$ ，我们希望与形状为 A 的矩阵进行批处理向量-矩阵乘法。在这种情况下， $(d_{\text{model}}, d_{\text{model}})$ 将执行批处理矩阵乘法，这是PyTorch中的高效原语，其中 $D @ A$ 维度会被批处理。 $(\text{batch_size}, \text{sequence_length})$

因此，最好假设您的函数可能会接收额外的批次类维度，并将这些维度保持在PyTorch形状的头。为了以这种方式组织张量以便进行批处理，可能需要使用 `view`、`reshape` 和 `transpose` 等多个步骤来塑造它们的形状。这可能会有些麻烦，并且通常很难阅读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` 中使用爱因斯坦求和约定表示法，或者使用框架无关的库如 `einops` 或 `einx`。两个关键操作是 `einsum`（可以对输入张量的任意维度进行张量收缩）和 `rearrange`（可以重新排序、连接和拆分任意维度）。

事实证明，机器学习中几乎所有的操作都是维度操作和张量收缩的某种组合，偶尔会涉及（通常是逐点的）非线性函数。这意味着在使用爱因斯坦求和约定表示法时，你的大量代码可以变得更具可读性和灵活性。

我们**强烈**建议在本课程中学习并使用爱因斯坦求和约定表示法。之前未接触过爱因斯坦求和表示法的学生应使用 `einops`（文档在此），而已经熟悉 `einops` 的学生应学习更通用的 `einx`（此处）。⁴ 这两个软件包已在我们提供的环境中预装。

这里我们给出一些关于如何使用爱因斯坦求和约定表示法的示例。这些是对 `einops` 文档的补充，您应当先阅读该文档。

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

```
import torch
from einops import rearrange, einsum

## Basic implementation
Y = D @ A.T
# Hard to tell the input and output shapes and what they mean.
# What shapes can D and A have, and do any of these have unexpected behavior?

## Einsum is self-documenting and robust
#           D           A           ->           Y
Y = einsum(D, A, "batch sequence d_in, d_out d_in -> batch sequence d_out")

## Or, a batched version where D can have any leading dimensions but A is constrained.
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

我们有一个图像批次，对于每个图像，我们希望基于某个

缩放因子生成10个调

暗版本：

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

## Reshape and multiply
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

## Or in one go:
dimmed_images = einsum(
    images, dim_by,
    "batch height width channel, dim_value -> batch dim_value height width channel"
)
```

⁴值得注意的是，虽然 `einops` 拥有大量支持，但 `einx` 尚未经过充分实战检验。如果您发现 `einx` 存在任何限制或错误，可以随时回退到使用 `einops` 配合更基础的 PyTorch。

示例 (einstein_example3)：使用 `einops.rearrange` 进行像素混合假设我们有一个图像批次，表示为形状为 (batch, height, width, channel) 的张量，并且我们希望执行一个跨图像所有像素的线性变换，但该变换应对每个通道独立进行。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)
## Rearrange an image tensor for mixing across all pixels
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 replaces clunky 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，如果您想尝试更简洁的方式：使用 einx.dot (einx 等价于 einops.einsum)
一次性完成height = width = 32channels_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添加注释来改进，以指示
```

输入和输出形状是什么，但这很笨拙且容易出错。使用爱因斯坦求和约定表示法，文档即实现！

爱因斯坦求和约定表示法可以处理任意的输入批处理维度，同时还具有自文档化的关键优势。在使用爱因斯坦求和约定表示法的代码中，输入和输出张量的相关形状要清晰得多。对于其余的张量，你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中进一步讨论使用爱因斯坦求和约定表示法的性能影响，但目前要知道它们几乎总是比替代方案更好！

3.3.1 数学符号和内存排序

许多机器学习论文在其符号表示中使用行向量，这导致了与NumPy和PyTorch默认使用的行优先内存排序良好契合的表示形式。使用行向量时，线性变换看起来像

$$y = xW^{\top}, \quad (1)$$

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

在线性代数中，通常更常用列向量，此时线性变换的形式为

$$y = Wx, \quad (2)$$

给定行优先 $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ 和列向量 $x \in \mathbb{R}^{d_{\text{in}}}$ 。我们将使用列向量作为本作业中的数学表示法，因为这种方式通常更易于理解数学推导。您需要注意，如果想使用纯矩阵乘法表示法，则必须按照行向量约定应用矩阵，因为PyTorch采用行优先内存排序。如果使用 `einsum` 进行矩阵运算，这应该不成问题。

3.4 基本构建模块：线性和嵌入模块

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——糟糕的初始化会导致不良行为，例如梯度消失或爆炸。预归一化Transformer对初始化具有异乎寻常的鲁棒性，但初始化仍会对训练速度和收敛性产生显著影响。由于本作业已经较长，我们将把细节留到作业3中讨论，现在仅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： $\mathcal{N}(\mu = 0, \sigma^2 = \frac{2}{d + d_{\text{inout}}})$ 截断于 $[-3\sigma, 3\sigma]$ 。
- 嵌入： $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ 截断于 $[-3, 3]$
- RMSNorm: 1

You should use `torch.nn.init.trunc_normal_` 来初始化截断正态权重 s.

3.4.2 线性模块

线性层是Transformer及一般神经网络的基础构建模块。首先，您需要实现一个继承自 `torch.nn.Module` 的 `Linear` 类，并执行线性变换：

$$y = Wx. \quad (3)$$

请注意，我们遵循大多数现代大语言模型的做法，不包含偏置项。

问题 (linear) : 实现线性模块 (1分)

可交付成果: 实现一个继承自 `torch.nn.Module` 的 `Linear` 类, 并执行线性变换。您的实现应遵循 PyTorch 内置 `nn.LinearModule` 的接口, 但不包含 `bias` 参数或参数。我们推荐以下接口:

`def __init__(self, in_features, out_features, device=None, dtype=None)` 构建一个线性变换模块。该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`
- 调用父类构造函数
- 由于内存排序原因, 将参数构造并存储为 W (而非 W^T), 将其放入 `nn.Parameter`
- 当然, 不要使用 `nn.Linear` 或 `nn.functional.linear`

对于初始化, 请使用上述设置以及 `torch.nn.init.trunc_normal_` 来初始化权重。

要测试你的 `Linear Module`, 请在 [\[adapters.run_linear\]](#) 处实现测试适配器。该适配器应将给定的权重加载到你的 `Linear Module` 中。你可以使用 `Module.load_state_dict` 实现此目的。然后运行 `uv run pytest -k test_linear`。

3.4.3 嵌入模块

如前所述, Transformer 的第一层是一个嵌入层, 它将整数标记 ID 映射到维度为 `d_model` 的向量空间中。我们将实现一个继承自 `torch.nn.Module` 的自定义 `Embedding` 类 (因此不应使用 `nn.Embedding`)。 `forward` 方法应通过索引形状为 `(vocab_size, d_model)` 的嵌入矩阵, 使用形状为 `(batch_size, sequence_length)` 的令牌 ID `torch.LongTensor` 来选择每个标记 ID 对应的嵌入向量。

问题 (embedding) : 实现嵌入模块 (1分)

可交付成果: 实现继承自 `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。我们推荐以下接口:

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

`num_embeddings: int` 词汇表的大小

```
embedding_dim: int 嵌入向量的维度，即  $d_{\text{model}}$ 
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 查找给定词元ID对应的嵌入向量。
```

请确保：

- 子类化 `nn.Module`
- 调用父类构造函数
- 将嵌入矩阵初始化为 `nn.Parameter`
- 存储嵌入矩阵时使 `d_model` 作为最终维度
- 当然，不要使用 `nn.Embedding` 或 `nn.functional.embedding`

再次使用上述设置进行初始化，并使用 `torch.nn.init.trunc_normal_` 来初始化权重。

要测试你的实现，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 预归一化Transformer块

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

在原始Transformer论文中，该模型在两个子层周围都使用了残差连接，随后进行层归一化。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输出。然而，多项研究发现将层归一化从每个子层的输出移至每个子层的输入（在最终Transformer块之后增加一个额外的层归一化）能提升Transformer训练稳定性 [（Nguyen和Salazar，2019；Xiong等人，2020）——参见图2了解这种“前归一化”Transformer块的视觉呈现）。每个Transformer块子层的输出随后通过残差连接添加到子层输入中（Vaswani等人，2017，第5.4节）。前归一化的直观理解是：从输入嵌入到Transformer最终输出之间存在一个纯净的“残差流”，无需任何归一化处理，据称这能改善梯度流。这种前归一化Transformer已成为当今语言模型（如GPT-3、LLaMA、PaLM等）的标准配置，因此我们将实现这种变体。我们将逐步讲解前归一化Transformer块的各个组件，并按顺序实现它们。

3.5.1 均方根层归一化

Vaswani 等人 [2017] 的原始 Transformer 实现使用层归一化 [Ba 等人，2016] 来归一化激活值。遵循 Touvron 等人 [2023] 的方法，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 $a \in \mathbb{R}^{d_{\text{model}}}$ ，RMSNorm 将按如下方式重新缩放每个激活值 a_i ：

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

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

您应该将输入向上转换到 `torch.float32`，以防止对输入进行平方时发生溢出。总体而言，您的 `forward` 方法应如下所示：

```
in_dtype = x.dtype
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 (rmsnorm): 均方根层归一化 (1 分)

可交付成果: 将 `RMSNorm` 实现为一个 `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 Module。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` 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：请记得在执行归一化之前将输入向上转换为 `torch.float32`（之后向下转换回原始数据类型），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 逐位置前馈网络

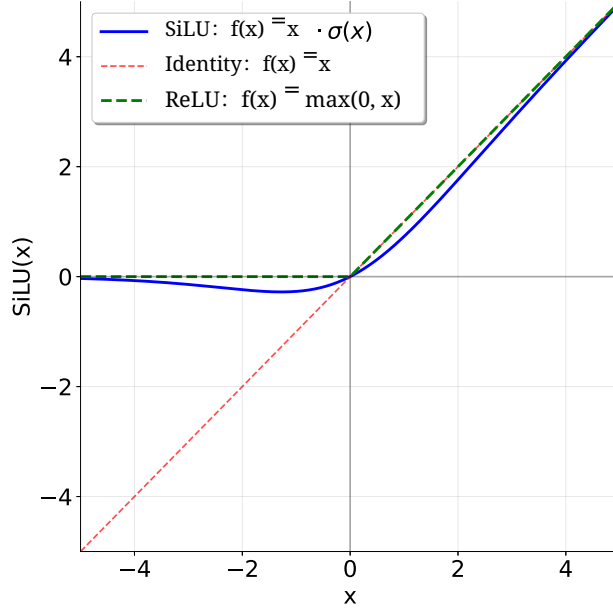


图3: SiLU (又称Swish) 与ReLU激活函数的对比。

在原始 Transformer 论文中 (Vaswani 等人 [2017] 的第 3.3 节), Transformer 前馈网络包含两个线性变换, 其间通过 ReLU 激活函数 ($\text{ReLU}(x) = \max(0, x)$) 连接。内部前馈层的维度通常是输入维度的 4 倍。

然而, 现代语言模型相比原始设计主要包含两项改进: 使用不同的激活函数并引入门控机制。具体而言, 我们将实现 Llama 3 [Grattafiori 等人, 2024] 和 Qwen 2.5 [Yang 等人, 2024], 等大型语言模型采用的“SwiGLU”激活函数, 该函数将 SiLU (常称为 Swish) 激活与名为门控线性单元 (GLU) 的门控机制相结合。同时我们将遵循自 PaLM [Chowdhery 等人, 2022] 和 LLaMA [Touvron 等人, 2023] 以来大多数现代大型语言模型的实践, 省略线性层中有时使用的偏置项。

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

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

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

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

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

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

将 SiLU/Swish 和 GLU 结合在一起, 我们得到了 SwiGLU, 我们将把它用于我们的前馈网络:

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

其中 $x \in \mathbb{R}^{d_{\text{model}}}$ 、 $W_1, W_3 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ 、 $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, 以及规范地, $d_{\text{ff}} = \frac{8}{3}d_{\text{model}}$ 。

沙泽尔 [2020] 首次提出将SiLU/Swish激活函数与门控线性单元相结合，并通过实验证明SwiGLU在语言建模任务中表现优于ReLU和SiLU（无门控）等基线方法。在本作业后续部分，您将比较SwiGLU和SiLU的性能。尽管我们已提及这些组件的启发式论证（且相关论文提供了更多佐证），但保持实证视角十分重要：沙泽尔论文中如今广为人知的名言是

我们不对这些架构为何有效提供解释；我们将它们的成功归因于神圣的恩典，如同其他一切。

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

可交付成果：实现由SiLU激活函数和GLU组成的SwiGLU前馈网络。

注意：在此特定情况下，您可随意使用 `torch.sigmoid` 以确保数值稳定性。

您应在实现中将 d_{ff} 设置为约 $3^8 \times d_{\text{model}}$ ，同时确保内部前馈层的维度是64的倍数以充分利用硬件性能。若要使用我们提供的测试验证实现，您需要在 `[adapters.run_swiglu]` 处实现测试适配器。随后运行 `uv run pytest -k test_swiglu` 来测试您的实现。

3.5.3 相对位置嵌入

为了将位置信息注入模型，我们将实现旋转位置嵌入 [苏等人,2021]，通常称为RoPE。对于给定查询词元 $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ 在词元位置 i ，我们将应用一个成对旋转矩阵 R^i ，得到 $q'^{(i)} = R^i q^{(i)} = R^i W_q x^{(i)}$ 。这里， R^i 会将嵌入元素对 $q_{2k-1}^{(i)}: 2k$ 作为二维向量旋转角度 $\theta_{i,k} = \Theta_{2k/d}^i$ 对于 $k \in \{1, \dots, d/2\}$ 和某个常数 Θ 。因此，我们可以将 R^i 视为大小为 $d \times d$ 的块对角矩阵，其中包含块 R_k^i 对于 $k \in \{1, \dots, d/2\}$ ，其中

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

因此我们得到完整的旋转矩阵

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \dots & 0 \\ 0 & R_2^i & 0 & \dots & 0 \\ 0 & 0 & R_3^i & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & R_{d/2}^i \end{bmatrix}, \quad (9)$$

其中 0 表示 2×2 零矩阵。虽然可以构建完整的 $d \times d$ 矩阵，但优秀的解决方案应利用该矩阵的特性来更高效地实现变换。由于我们只关注给定序列内标记的相对旋转，可以复用为 $\cos(\theta_{i,k})$ 和 $\sin(\theta_{i,k})$ 计算的值，这些值可跨不同层数和批次重复使用。若需优化，可使用被所有层引用的单一RoPE模块，该模块可在初始化时创建包含sin和cos值的二维预计算缓冲区（使用 `self.register_buffer(persistent=False)`），而非使用 `nn.Parameter`（因为我们不希望学习这些固定的余弦和正弦值）。我们对 $q^{(i)}$ 执行的完全相同旋转过程，也将应用于 $k^{(j)}$ ，按对应的 R^i 进行旋转。注意该层没有可学习参数。

问题 (rope) : 实现RoPE (2分)

可交付成果: 实现一个类 `RotaryPositionalEmbedding` , 用于将RoPE应用于输入张量。

推荐使用以下接口:

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

`theta: float` Θ RoPE的值

`d_k: int` 查询向量和键向量的维度

`max_seq_len: int` 将输入的最大序列长度

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

处理形状为 $(\dots, \text{seq_len}, d_k)$ 的输入张量, 并返回相同形状的张量。请注意, 您应该能够处理具有任意批次维度数量的 x 。您应假设词元位置是一个形状为 $(\dots, \text{seq_len})$ 的张量, 用于指定 x 沿序列维度的词元位置。

您应该使用词元位置沿序列维度切片您的 (可能预先计算好的) 余弦和正弦张量。

为测试您的实现, 请完成 `[adapters.run_rope]` 并确保通过 `uv runpytest -k test_rope`。

3.5.4 缩放点积注意力

我们现在将实现 Vaswani 等人 [2017] (第 3.2.1 节) 中描述的缩放点积注意力。作为预备步骤, 注意力操作的定义将使用 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:

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

请注意, 对于较大的值, $\exp(v_i)$ 可能变为 `inf` (此时, `inf/inf = NaN`)。我们可以通过注意到 `softmax` 操作对向所有输入添加任意常数 c 具有不变性来避免这种情况。我们可以利用这一特性来确保数值稳定性——通常, 我们会从 o_i 的所有元素中减去 o_i 的最大条目, 使新的最大条目变为 0。你现在将实现 `softmax`, 并使用这个技巧来确保数值稳定性。

问题 (softmax): 实现 softmax (1 分)

可交付成果: 编写一个函数以在张量上应用 `softmax` 操作。你的函数应接受两个参数: 一个张量和一个维度 i , 并将 `softmax` 应用于输入张量的第 i 个维度。输出张量应与输入张量具有相同的形状, 但其第 i 个维度现在将具有一个归一化概率分布。使用从第 i 个维度的所有元素中减去第 i 个维度中的最大值这一技巧, 以避免数值稳定性问题。

为测试你的实现, 请完成 `[adapters.run_softmax]` 并确保通过 `uv runpytest -k test_softmax_matches_pytorch`。

我们现在可以如下数学定义注意力机制:

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

其中 $Q \in \mathbb{R}^{n \times d_k}$ 、 $K \in \mathbb{R}^{m \times d_k}$ 和 $V \in \mathbb{R}^{m \times d_v}$ 。这里 Q 、 K 和 V 都是该操作的输入——请注意这些不是可学习参数。如果您想知道为什么不是 QK^\top ，请参阅 3.3.1。

掩码：有时为了方便，会对注意力机制的输出进行掩码处理。掩码应具有形状 $M \in \{\text{True}, \text{False}\}^{n \times m}$ ，这个布尔矩阵的每一行 i 表示查询向量 i 应关注哪些键向量。规范地说（可能有些令人困惑），位置 (i, j) 的值为 **True** 表示查询向量 i 确实关注键向量 j ，而值为 **False** 表示查询向量不关注该键向量。换句话说，“信息流”在值为 **True** 的 (i, j) 位置对之间流动。例如，考虑一个 1×3 掩码矩阵，其条目为 $[[\text{True}, \text{True}, \text{False}]]$ 。单个查询向量仅关注前两个键向量。

在计算上，使用掩码比计算子序列的注意力要高效得多，我们可以通过获取预softmax值 $\left(\frac{Q^\top K}{\sqrt{d_k}}\right)$ 并在掩码矩阵中任何为 **False** 的条目处添加一个 $-\infty$ 来实现这一点。——

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

可交付成果：实现缩放点积注意力函数。你的实现应能处理形状为

$(\text{batch_size}, \dots, \text{seq_len}, d_k)$ 的键和查询，以及形状为 $(\text{batch_size}, \dots, \text{seq_len}, d_v)$ 的值，其中 \dots 代表任意数量的其他批次类维度（如果提供）。该实现应返回一个形状为

$(\text{batch_size}, \dots, d_v)$ 的输出。关于批次类维度的讨论，请参见第3.3节。你的实现还应支持一个可选的、用户提供的形状为 $(\text{seq_len}, \text{seq_len})$ 的布尔掩码。掩码值为 **True** 的位置的注意力概率应总和为1，而掩码值为 **False** 的位置的注意力概率应为零。为了使用我们提供的测试来验证你的实现，你需要在 `[adapters.run_scaled_dot_product_attention]` 处实现测试适配器。

`uv run pytest -k test_scaled_dot_product_attention` 在三阶输入张量上测试你的实现，而 `uv run pytest -k test_4d_scaled_dot_product_attention` 在四阶输入张量上测试你的实现。

3.5.5 因果多头自注意力

我们将实现 Vaswani 等人 [2017] 第 3.2.2 节中描述的多头自注意力。回顾一下，在数学上，应用多头注意力的操作定义如下：

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

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

其中 Q_i 、 K_i 、 V_i 分别是 Q 、 K 和 V 的嵌入维度大小为 d_k 或 d_v 的切片编号。注意力机制是 §3.5.4 中定义的缩放点积注意力操作。由此我们可以形成多头自注意力操作：

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

这里，可学习参数是 $W_Q \in \mathbb{R}^{d_k \times d_{\text{model}}}$ 、 $W_K \in \mathbb{R}^{d_k \times d_{\text{model}}}$ 、 $W_V \in \mathbb{R}^{d_v \times d_{\text{model}}}$ 、 $W_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ 。由于在多头注意力机制中， s 、 Q 、 K 和 V 被切片处理，我们可以将 W_Q 、 W_K 、 W_V 和 W_O 视为沿输出维度为每个头部分离。当这部分

正常工作时，你应该通过总共三次矩阵乘法来计算键向量、值向量和查询投影。⁵

⁵作为进阶目标，尝试将键、查询和值投影合并为单个权重矩阵，这样您只需要进行单个矩阵乘法。

因果掩码。你的实现应阻止模型关注序列中的未来标记。换句话说，如果模型获得标记序列 t_1, \dots, t_n ，并且我们想要计算前缀 t_1, \dots, t_i 的下一词预测（其中 $i < n$ ），那么模型应该不能访问（关注）位置 t_{i+1}, \dots, t_n 处的标记表示，因为在推理期间生成文本时模型将无法访问这些标记（而这些未来标记会泄露真实下一个词的身份信息，使得语言建模预训练目标变得无关紧要）。对于输入标记序列 t_1, \dots, t_n ，我们可以通过运行多头自注意力 n 次（针对序列中的 n 个唯一前缀）来简单阻止对未来标记的访问。相反，我们将使用因果注意力掩码，它允许标记 i 关注序列中所有位置 $j \leq i$ 。你可以使用 `torch.triu` 或广播索引比较来构建此掩码，并且应该利用§3.5.4中已实现的缩放点积注意力支持注意力掩码这一特性。

应用 RoPE。RoPE 应应用于查询向量和键向量，但不应用于值向量。此外，头维度应作为批次维度处理，因为在多头注意力中，注意力是独立应用于每个注意力头的。这意味着每个注意力头的查询向量和键向量都应应用完全相同的 RoPE 旋转。

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

可交付成果：将因果多头自注意力实现为 `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` 多头自注意力中使用的注意力头数。

遵循 Vaswani 等人 [2017]，设置 $d_k = d_v = d_{\text{model}}/h$ 。要对照我们提供的测试来测试您的实现，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 完整Transformer语言模型

让我们从组装Transformer块开始（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，然后执行主要操作（MHA/FF），最后加入残差连接。

具体来说，Transformer块的前半部分（第一个“子层”）应实现以下更新集，以从输入 x 产生输出 y ，

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

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

按照§3.5所述并如图2所示，实现预归一化Transformer块。你的Transformer块应（至少）接受以下参数。

d_model: `int` Transformer块输入的维度。

num_heads: `int` 多头自注意力中使用的注意力头数。

d_ff: `int` 逐位置前馈内部层的维度。

为测试你的实现，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。

可交付成果：通过所提供测试的Transformer块代码。

现在我们按照图1中的高层示意图将这些模块组合起来。遵循第3.1.节中关于嵌入的描述，将其输入到 `num_layers` 个Transformer模块中，然后传递给三个输出层以获得词汇表上的分布。

问题 (transformer_lm)：实现Transformer语言模型 (3分)

是时候将所有内容整合起来了！请根据第3.1节的描述和图1的示意图实现Transformer语言模型。您的实现至少应接受前述Transformer模块的所有构建参数，以及以下附加参数：

`vocab_size`: `int` 词汇表的大小，用于确定词元嵌入矩阵的维度。

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

`num_layers`: `int` 要使用的Transformer块数量。

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

可交付成果：一个通过上述测试的Transformer语言模型模块。

资源核算。理解Transformer各个部分如何消耗计算和内存是非常有用的。我们将逐步进行一些基本的“浮点运算次数核算”。Transformer中绝大多数的浮点运算次数来自矩阵乘法，因此我们的核心方法很简单：

1. 列出Transformer前向传播中的所有矩阵乘法。
2. 将每个矩阵乘法转换为所需的浮点运算次数。

对于这第二个步骤，以下事实将会很有用：

规则：给定 $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ ，矩阵-矩阵乘积 AB 需要 $2mnp$ 浮点运算次数。

要理解这一点，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（ $2n$ 次浮点运算）。然后，由于矩阵-矩阵乘积 AB 有 $m \times p$ 个元素，总的浮点运算次数为 $(2n)(mp) = 2mnp$ 。

现在，在您处理下一个问题之前，逐一检查Transformer块和Transformer语言模型的每个组件，并列出所有矩阵乘法及其相关的浮点运算次数成本会很有帮助。

问题 m (transformer_accounting): Transformer语言模型资源核算 (5分)

(a) 考虑GPT-2 XL，其配置如下：

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

`num_heads : 25`

`d_ff : 6,400`

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

可交付成果：一到两句话的回复。

- (b) 识别完成GPT-2 XL形状模型前向传播所需的矩阵乘法。
这些矩阵乘法总共需要多少浮点运算次数？假设我们的输入序列有 `context_length` 个标记。

可交付成果：矩阵乘法列表（含描述）及所需总浮点运算次数。

- (c) 根据你的分析，模型的哪些部分需要最多的浮点运算次数

- (d) 使用GPT-2小模型（12层， 768 `d_model`， 12个注意力头数）、GPT-2中模型（24层， 1024 `d_model`， 16个注意力头数）和GPT-2大模型（36层， 1280 `d_model`， 20个注意力头数）重复你的分析。随着模型大小的增加，Transformer语言模型的哪些部分在总浮点运算次数中所占的比例会相对增加或减少？

可交付成果：对于每个模型，提供模型组件及其相关浮点运算次数的细分（作为一次前向传播所需总浮点运算次数的比例）。此外，用一到两句话描述改变模型大小如何改变每个组件所占浮点运算次数的比例。

- (e) 以GPT-2 XL为例，将上下文长度增加到16,384。一次前向传播的总浮点运算次数会如何变化？前向传播有何变化？模型组件的浮点运算次数相对贡献如何变化？

可交付成果：一到两句话的回复。

4 训练TransformerLM

我们现在已经有了预处理数据（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：**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这个损失（AdamW）。
- **训练循环：**我们需要所有支持基础设施来加载数据、保存检查点并管理训练。

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 $p_\theta(x_{i+1} \mid x_{1:i})$ ，其中 $i = 1, \dots, m$ 。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：

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

（请注意，Transformer中的单次前向传播会为所有 $i = 1, \dots, m$ 生成 $p_\theta(x_{i+1} \mid x_{1:i})$ 。）特别地，Transformer会计算逻辑值 o

$$p(x_{i+1} \mid x_{1:i}) = \text{softmax}(o_i)[x_{i+1}] = \frac{\exp(o_i[x_{i+1}])}{\sum_{a=1}^{\text{vocab_size}} \exp(o_i[a])}. \quad (17)$$

交叉熵损失通常根据逻辑值向量 $o_i \in \mathbb{R}^{\text{vocab_size}}$ 和目标 x_{i+1} 来定义。⁷

实现交叉熵损失需要特别注意数值问题，就像在Softmax中一样。

问题(cross_entropy)：实现交叉熵

可交付成果：编写一个计算交叉熵损失的函数，该函数接收预测逻辑值(o_i)和目标(x_{i+1})，并计算交叉熵 $\ell_i = -\log \text{softmax}(o_i)[x_{i+1}]$ 。您的函数应处理以下情况：

- 为保持数值稳定性，减去最大元素。
- 尽可能对消log和exp。
- 处理所有额外的批次维度并返回批次间的平均值。如第3.3节所述，我们假定批次类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` 以测试您的实现。

困惑度 交叉熵足以用于训练，但当我们评估模型时，我们也希望报告困惑度。对于一个长度为 m 的序列，其中我们遭受交叉熵损失 ℓ_1, \dots, ℓ_m ：

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

⁶注意 $o_i[k]$ of the vector o_i 表示索引处的值。这对应于 x_{i+1} 上的狄拉克δ分布与预测 $\text{softmax}(o_i)$ 分布之间的交叉熵。⁷

4.2 随机梯度下降优化器

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

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

其中 B_t 是从数据集 D 中随机采样的数据批次，而学习率 α_t 和批次大小 $|B_t|$ 是超参数。

4.2.1 在 PyTorch 中实现随机梯度下降

为实现我们的优化器，我们将继承 PyTorch 的 `torch.optim.Optimizer` 类。一个 `Optimizer` 子类必须实现两个方法：

`def __init__(self, params, ...)` 应初始化你的优化器。此处，`params` 将是要优化的参数集合（或参数组，以防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）。确保将 `params` 传递给基类的 `__init__` 方法，该方法会存储这些参数以供 `step` 使用。你可以根据优化器需求接收额外参数（例如学习率是常见参数），并将它们以字典形式传递给基类构造函数，其中键是你为这些参数选择的名称（字符串）。

`def step(self)` 应对参数进行一次更新。在训练循环中，这将在反向传播之后被调用，因此您可以访问最后一个批次上的梯度。该方法应迭代每个参数张量 `p` 并就地修改它们 *in place*，即设置 `p.data`，该变量根据梯度 `p.grad`（如果存在）保存与该参数关联的张量，该张量表示损失相对于该参数的梯度。

PyTorch 优化器 API 有一些微妙之处，因此通过示例来解释会更容易。为了使我们的示例更丰富，我们将实现一个略有变化的 SGD，其中学习率在训练过程中会衰减，从初始学习率 α 开始，并随着时间的推移逐步减小步长：

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

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

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

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

```

for p in group["params"]:
    if p.grad is None:
        continue

    state = self.state[p] # Get state associated with p.
    t = state.get("t", 0) # Get iteration number from the state, or initial value.
    grad = p.grad.data # Get the gradient of loss with respect to p.
    p.data -= lr / math.sqrt(t + 1) * grad # Update weight tensor in-place.
    state["t"] = t + 1 # Increment iteration number.

return loss

```

在 `__init__` 中，我们将参数以及默认超参数传递给基类构造函数（参数可能以分组形式传入，每组具有不同的超参数）。如果参数只是单个 `torch.nn.Parameter` 对象集合，基类构造函数将创建单个组并为其分配默认超参数。接着在 `step` 中，我们遍历每个参数组，然后遍历该组中的每个参数，并应用公式20。此处，我们将迭代次数作为与每个参数关联的状态保存：首先读取该值，在梯度更新中使用它，然后更新该值。该应用程序接口规定用户可能传入一个可调用对象 `closure`，以便在优化器步骤之前重新计算损失。虽然我们即将使用的优化器不需要此功能，但为了符合应用程序接口规范我们仍将其添加。

为了观察其实际运行效果，我们可以使用以下最小化的训练循环示例：

```

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

for t in range(100):
    opt.zero_grad() # Reset the gradients for all learnable parameters.
    loss = (weights**2).mean() # Compute a scalar loss value.
    print(loss.cpu().item()) loss.backward() # Run backward pass,
    which computes gradients.opt.step() # Run optimizer step.

```

这是训练循环的典型结构：在每次迭代中，我们将计算损失并执行优化器的一个步骤。当训练语言模型时，我们的可学习参数将来自模型（在PyTorch中，`m.parameters()`可获取该参数集合）。损失将通过采样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)

正如我们将要看到的，对训练影响最大的超参数之一就是学习率。让我们通过这个简单示例进行实践验证。使用另外三个学习率值（`1e1`、`1e2`和`1e3`）运行上述SGD示例，仅进行10次训练迭代。这些学习率对应的损失分别有何变化？损失衰减得更快、更慢，还是出现发散（即在训练过程中持续上升）？

可交付成果：用一两句话描述你观察到的行为。

4.3 AdamW

现代语言模型通常使用更复杂的优化器进行训练，而非随机梯度下降。最近使用的大多数优化器都是Adam优化器 [Kingma和Ba, 2015]的衍生版本。我们将采用AdamW[Loshchilov和Hutter, 2019]，该方法在近期研究中被广泛使用。AdamW提出了一种对Adam的改进，通过添加权重衰减（在每次迭代时将参数向0方向收缩）来提升正则化效果，

这种方式与梯度更新解耦。我们将按照Loshchilov和Hutter [2019]算法2中的描述来实现AdamW。

AdamW优化器是有状态的：对于每个参数，它会持续跟踪其一阶矩和二阶矩的运行估计值。因此，AdamW通过消耗额外内存来换取更好的稳定性和收敛性。除了学习率 α ，AdamW还有一对控制矩估计更新的超参数 (β_1, β_2) 以及权重衰减率 λ 。典型应用将 (β_1, β_2) 设为 $(0.9, 0.999)$ ，但像LLaMA [Touvron等人, 2023] 和GPT-3 [Brown等人, 2020] 这样的大语言模型通常使用 $(0.9, 0.95)$ 进行训练。该算法可表述如下，其中 ϵ 是一个小值（例如 10^{-8} ），用于在 v 出现极小值时提升数值稳定性：

算法1 AdamW优化器

初始化 (θ) (初始化可学习参数) $m \leftarrow 0$ (一阶矩向量初始值; 与 θ 形状相同) $v \leftarrow 0$ (二阶矩向量初始值; 与 θ 形状相同) **循环** $t = 1, \dots, T$ **执行** 采样数据批次 B_t $g \leftarrow \nabla_{\theta} \ell(\theta; B_t)$ (计算当前时间步的损失梯度) $m \leftarrow \beta_1 m + (1 - \beta_1) g$ (更新一阶矩估计) $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (更新二阶矩估计) $\alpha_t \leftarrow \alpha \sqrt{1 - (\beta_2)^t} / \sqrt{v + \epsilon}$ (计算第 t 次迭代的调整后 α) $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v + \epsilon}}$ (更新参数) $\theta \leftarrow \theta - \alpha \lambda \theta$ (应用权重衰减) **结束循环**

请注意 t 从1开始。你现在将实现这个优化器。

问题(adamw): 实现AdamW (2分)

可交付成果: 将AdamW优化器实现为 `torch.optim.Optimizer` 的子类。你的类应接受学习率 α (以 `__init__` 为单位)，以及 β 、 ϵ 和 λ 超参数。为帮助你维护状态，基础 `Optimizer` 类提供了一个字典 `self.state`，它将 `nn.Parameter` 对象映射到一个存储该参数所需任何信息（对于AdamW，这将是矩估计）的字典。实现 `[adapters.get_adamw_cls]` 并确保它通过 `uv runpytest -k test_adamw`。

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

s)

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？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`。

为简化计算，在考虑激活值的内存使用时仅包含以下组件：

- Transformer块—
RMSNorm(s)

– 多头自注意力子层: QKV 投影, $Q^T K$ 矩阵乘法, Softmax, 值的加权和, 输出投影。
逐位置前馈: W_1 矩阵乘法, SiLU, W_2 矩阵乘法

- 最终RMSNorm
- 输出嵌入
- 逻辑值上的交叉熵

可交付成果: 针对参数、激活值、梯度和优化器状态的代数表达式, 以及总量。

(b) 将您的答案实例化为GPT-2 XL形状模型, 得到一个仅依赖于 `batch_size` 的表达式。在80GB内存限制下, 可使用的最大批次大小是多少?

可交付成果: 形如 $a \cdot \text{batch_size} + b$ 的表达式 (其中 a 、 b 为数值), 以及表示最大批次大小的数值。

(c) 运行一步 AdamW 需要多少浮点运算次数?

可交付成果: 一个代数表达式, 并附简要说明。

(d) 模型浮点运算利用率 (MFU) 定义为观察到的吞吐量 (每秒处理的令牌数) 与硬件理论峰值浮点运算吞吐量之比 [Chowdhery 等人, 2022]。NVIDIA A100 GPU 的 float32 操作理论峰值为 19.5 万亿次浮点运算/秒。假设您能达到 50% 的 MFU, 在单个 A100 上训练 GPT-2 XL 400K 步且批次大小为 1024 需要多长时间? 根据 Kaplan 等人 [2020] 和 Hoffmann 等人 [2022], 的假设, 反向传播的浮点运算次数是前向传播的两倍。

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

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 Transformer 模型时, 通常使用学习率计划, 即开始时使用较大的学习率以进行快速更新, 随着模型训练逐渐衰减到较小的值⁸在本作业中, 我们将实现用于训练 LLaMA [Touvron 等人, 2023] 的余弦退火计划。

调度器就是一个函数, 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 用于梯度更新的学习率。最简单的计划是常数函数, 它会在给定任何 t 时返回相同的学习率。

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

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

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

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

⁸有时c

常见的是使用一种学习率重新上升 (重启) 的计划, 以帮助越过局部极小值

a.

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

编写一个函数，接收参数 t 、 α_{\max} 、 α_{\min} 、 T_w 和 T_c ，并根据上述定义的调度器返回学习率 α_t 。然后实现 `[adapters.get_lr_cosine_schedule]` 并确保通过测试 `uv run pytest -k test_get_lr_cosine_schedule`。

4.5 梯度裁剪

在训练过程中，我们有时会遇到产生大梯度的训练样本，这可能会破坏训练稳定性。为了缓解这个问题，实践中常采用的一种技术是梯度裁剪。其核心思想是在执行优化器步骤之前，对每次反向传播后的梯度范数施加限制。

给定（所有参数的）梯度 g ，我们计算其 ℓ_2 -范数 $\|g\|_2$ 。如果该范数小于最大值 M ，则保持 g 不变；否则，将 g 按 $\frac{M}{\|g\|_2 + \epsilon}$ 的比例缩小（其中添加了极小值 ϵ ，如 10^{-6} ，以维持数值稳定性）。注意最终得到的范数将略低于 M 。

问题 (gradient_clipping)：实现梯度裁剪（1分）

编写一个实现梯度裁剪的函数。你的函数应接收一个参数列表和一个最大 ℓ_2 范数。它应该就地修改每个参数的梯度。使用 $\epsilon = 10^{-6}$ (PyTorch默认值)。然后实现适配器

`[adapters.run_gradient_clipping]` 并确保通过 `uv run pytest -k test_gradient_clipping`。

5 训练循环

我们现在终于要将迄今为止构建的主要组件整合在一起：标记化数据、模型和优化器。

5.1 数据加载器

标记化数据（例如，您在 `tokenizer_experiments` 中准备的数据）是一个单一的标记序列 $x = (x_1, \dots, x_n)$ 。尽管源数据可能包含独立的文档（例如不同的网页或源代码文件），通常的做法是将所有这些文档连接成一个单一的标记序列，并在它们之间添加分隔符（例如 `<|endoftext|>` 标记）。

数据加载器会将其转换为批次流，其中每个批次包含 B 个长度为 m 的序列，并配以对应的下一个标记（长度同样为 m ）。例如，对于 $B = 1, m = 3$ ， $([x_2, x_3, x_4], [x_3, x_4, x_5])$ 可能是一个潜在的批次。

以这种方式加载数据简化了训练过程，原因如下：首先，任何 $1 \leq i < n - m$ 都能提供有效的训练序列，因此采样序列变得非常简单。由于所有训练序列长度相同，无需对输入序列进行填充，这提高了硬件利用率（同时通过增加批次大小 B 实现）。最后，我们也不需要完全加载整个数据集来采样训练数据，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（data_loading）：实现数据加载（2分）

可交付成果：编写一个函数，该函数接收一个numpy数组 x （包含标记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：采样得到的输入序列和对应的下一词元目标。两个张量的形状都应为 $(\text{batch_size}, \text{context_length})$ ，其中包含标记 ID，且两者都应放置在请求的设备上。要对照我们提供的测试来验证你的实现，你需要先在 `[adapters.run_get_batch]` 处实现测试适配器。然后运行 `uv run pytest -ktest_get_batch` 来测试你的实现。

低资源/缩减规模提示：在中央处理器或苹果芯片上进行数据加载

如果你计划在中央处理器或苹果芯片上训练你的语言模型，你需要将数据移动到正确的设备上（类似地，之后你的模型也应使用相同的设备）。

如果您在中央处理器上，可以使用 `'cpu'` 设备字符串；在苹果芯片（M*芯片）上，则可以使用 `'mps'` 设备字符串。

有关MPS的更多信息，请查阅以下资源：

- <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惰性加载文件内容。这样，你就可以“假装”整个数据集都在内存中。NumPy通过`np.memmap`（或者如果你最初使用`np.save`保存数组，则通过`np.load`的`mmap_mode='r'`标志）实现这一功能，它将返回一个类似numpy数组的对象，在你访问时按需加载条目。在训练期间从数据集（即numpy数组）采样时，务必以内存映射模式加载数据集（通过`np.memmap`或`np.load`的`mmap_mode='r'`标志，具体取决于你保存数组的方式）。请确保同时指定与所加载数组匹配的 `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

除了加载数据，我们还需要在训练过程中保存模型。在运行作业时，我们通常希望能够恢复因某些原因中途停止的训练运行（例如，由于作业超时、机器故障等）。即使一切顺利，我们可能也希望后期能够访问中间模型（例如，用于事后研究训练动态、从不同训练阶段的模型进行采样等）。

一个检查点应包含我们恢复训练所需的所有状态。我们当然希望至少能够恢复模型权重。如果使用有状态的优化器（如AdamW），我们还需要保存优化器的状态（例如，对于AdamW来说，就是矩估计值）。最后，为了恢复学习率计划，我们需要知道停止时的迭代次数。PyTorch使得保存所有这些变得容易：每个 `nn.Module` 都有一个 `state_dict()` 方法，返回包含所有可学习权重的字典；我们之后可以通过配套方法 `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，一个在某些值中包含张量，但也包含常规Python对象（如整数）的字典）转储到文件（路径）或类文件对象中，之后可以使用 `torch.load(src)` 将其加载回内存。

问题 (checkpointing)：实现模型检查点 (1分)

实现以下两个函数来加载和保存检查点：

`def save_checkpoint(model, optimizer, iteration, out)` 应将前三个参数的所有状态转储到类文件对象 `out` 中。您可以使用模型和优化器的 `state_dict` 方法来获取它们相关的状态，并使用 `torch.save(obj, out)` 将 `obj` 转储到 `out` 中（PyTorch在此处支持路径或类文件对象）。典型的选择是让 `obj` 成为一个字典，但只要您之后能加载您的检查点，您可以使用任何格式。

此函数需要以下参数：

```
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，然后从该检查点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` 方法将它们恢复到之前的状态。

此函数需要以下参数：

```
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，终于到了将所有已实现的组件整合到主训练脚本中的时候了。通过命令行参数等方式使不同超参数的训练运行能够轻松启动将会很有帮助，因为后续你将多次进行此类操作来研究不同选择对训练的影响。

问题 (training_together): 整合实现 (4 分)

可交付成果: 编写一个运行训练循环的脚本，在用户提供的输入上训练你的模型。特别建议你的训练脚本至少支持以下功能：

- 能够配置和控制各种模型和优化器的超参数。
- 使用 `np.memmap` 高效加载大型训练和验证数据集至内存。
- 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，输出到控制台和/或外部服务如Weightsand Biases）。^a

^awandb.ai

6 生成文本

既然我们已经能够训练模型，最后需要的是从我们的模型中生成文本的能力。回想一下，语言模型接收一个长度为 `sequence_length` 的（可能经过批处理的）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预测该位置后下一个词的概率分布。我们现在将编写几个函数，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（即“逻辑值”）的输出，因此我们必须通过 *softmax* 操作将其转换为归一化概率，这在我们之前看到的公式10中已有提及。

解码 为了从我们的模型中生成文本（解码），我们将为模型提供一系列前缀标记（即“提示”），并要求其生成一个覆盖词汇表的概率分布，以预测序列中的下一个词。然后，我们将从这个词汇表项目的分布中进行采样，以确定下一个输出标记。

具体来说，解码过程的一个步骤应接收一个序列 $x_{1...t}$ ，并通过以下方程返回一个标记 x_{t+1} ，

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

$$v = \text{TransformerLM}(x_{1...t})_t \in \mathbb{R}^{\text{vocab_size}}$$

其中 `TransformerLM` 是我们的模型，它接收一个 `sequence_length` 序列作为输入，并生成一个大小为 (`sequence_length × vocab_size`) 的矩阵，我们取该矩阵的最后一个元素，因为我们要寻找第 t 个位置的下一个词预测。

这为我们提供了一个基础解码器，通过重复从这些单步条件分布中采样（将先前生成的输出标记附加到下一个解码时间步的输入中），直到我们生成序列结束标记 `<|endoftext|>`（或用户指定的最大生成标记数）。

解码技巧 我们将使用小型模型进行实验，而小型模型有时会生成质量很低的文本。两种简单的解码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 来修改 `softmax`，新的 `softmax` 为

$$\text{softmax}(v, \tau)_i = \frac{\exp(v_i/\tau)}{\sum_{j=1}^{\text{vocab_size}} \exp(v_j/\tau)}. \quad (24)$$

注意设置 $\tau \rightarrow 0$ 会使 v 的最大元素占主导地位，`softmax` 的输出会变成集中在该最大元素上的独热向量。

其次，另一个技巧是核心或 *top-p* 采样，我们通过截断低概率词来修改采样分布。令 q 为从大小为 (`vocab_size`) 的（经过温度缩放的）`softmax` 得到的概率分布。使用超参数 p 的核心采样根据以下方程生成下一个标记

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

其中 $V(p)$ 是满足 $\sum_{j \in V(p)} q_j \geq p$ 的最小索引集。您可以通过首先按大小对概率分布 q 进行排序，然后选择最大的词汇元素直到达到目标 α 水平来轻松计算此量。

问题 (decoding): 解码 (3 分) 可交付成果: 实现一个从您的语言模型进行解码的函数。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 $x_{1...t}$ 并采样补全，直到遇到 `<|endoftext|>` 词元）。
- 允许用户控制生成标记的最大数量。
- 根据给定的温度值，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 等人，2020；也称为核心采样），给定一个用户指定的阈值。

7 实验

现在是时候 将所有内容整合起来，并在预训练数据集上训练（小型）语言模型。

7.1 如何运行实验和交付成果

理解Transformer架构组件背后原理的最佳方式，就是实际修改并亲自运行它。实践经验是无可替代的。

为此，能够快速、一致地进行实验并保存记录至关重要。为实现快速实验，我们将在小型模型（1700万参数）和简单数据集（TinyStories）上运行大量实验。为确保一致性，你需要系统性地消融组件和调整超参数，而为保存记录，我们将要求你提交实验日志及每个实验对应的学习曲线。

为便于提交损失曲线，请确保定期评估验证损失并记录步骤数和挂钟时间。你可能会发现像Weights and Biases这样的日志记录基础设施很有帮助。

问题（experiment_log）：实验日志记录（3分）

为你的训练和评估代码，创建实验跟踪基础设施，使你能够跟踪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：用于实验的日志记录基础设施代码以及本节以下作业问题的实验日志（记录你尝试过的所有内容的文档）。

7.2 TinyStories

我们将从一个非常简单的数据集（TinyStories；Eldan和Li, 2023）开始，模型将在此数据集上快速训练，并且我们可以看到一些有趣的行为。获取此数据集的说明在第1节。以下是此数据集的一个示例。

示例（tinystories_example）：TinyStories中的一个示例

从前有个名叫本的小男孩。本喜欢探索周围的世界。他看到了许多令人惊叹的事物，比如商店里陈列的美丽花瓶。有一天，本在商店里散步时，遇到了一个非常特别的花瓶。当本看到它时，他惊叹不已！他说：“哇，这真是一个令人惊叹的花瓶！我可以买下它吗？”店主微笑着回答：“当然可以。你可以把它带回家，向你所有的朋友们展示它有多么神奇！”于是本把花瓶带回家，并为此感到非常自豪！他叫来朋友们，向他们展示这个令人惊叹的花瓶。所有的朋友都认为这个花瓶很漂亮，不敢相信本有多么幸运。这就是本在商店里发现神奇花瓶的故事！

超参数调优 我们将告诉你一些基础的超参数作为起点，并要求你找到其他能良好运行的参数设置。

vocab_size 10000. 典型的词汇表大小在数万到数十万之间。你应该调整这个参数，观察词汇表和模型行为的变化。

context_length 256. 像 TinyStories 这样的简单数据集可能不需要长序列长度，但对于后续的 OpenWebText 数据，您可能需要调整此参数。尝试调整此参数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. 这略小于许多小型 Transformer 论文中使用的 768 维度，但这将使运行速度更快。

`d_ff` 1344。这大约是 $\frac{8}{3}d_{\text{model}}$ ，同时是64的倍数，这对GPU性能有利。

`RoPE theta parameter` Θ 10000。

`number of layers and heads` 4 层，16个注意力头数。这样总共会产生约1700万个非嵌入参数，这是一个相当小的 Transformer。

`total tokens processed` 327,680,000（你的批次大小 \times 总步数 \times 上下文长度应大致等于该值）。

你应该通过试验和错误来为以下其他超参数找到合适的默认值：`learning rate`、`learning rate warmup`、`other AdamW hyperparameters` ($\beta_1, \beta_2, \epsilon$) 和 `weight decay`。你可以在Kingma和Ba [2015]中找到这类超参数的典型选择。

整合实现现在你可以通过获取训练好的BPE分词器，对训练数据集进行分词，并在你编写的训练循环中运行这些步骤来整合所有内容。**重要提示：**如果你的实现正确且高效，上述超参数在1个H100 GPU上的运行时间应约为30-40分钟。如果你的运行时间远长于此，请检查并确保你的数据加载、检查点或验证损失代码没有成为运行瓶颈，且你的实现已正确进行批处理。

调试模型架构的技巧与诀窍我们强烈建议您熟悉集成开发环境的内置调试器（例如VSCode/PyCharm），相比使用`print`语句进行调试，这将节省您的时间。如果您使用文本编辑器，可以更倾向于使用 `pdb`。在调试模型架构时，其他一些良好实践包括：

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

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

学习率是需要调整的最重要超参数之一。基于您训练的基础模型，请回答以下问题：

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

可交付成果：多个学习率对应的学习曲线。说明你的超参数搜索策略。

可交付成果：一个在TinyStories上验证损失（每词元）不超过1.45的模型

低资源/缩减规模提示：在中央处理器或苹果芯片上进行少量步骤的训练

如果您在 `cpu` 或 `mps` 上运行，则应将被处理的标记总数减少至 40,000,000，这足以生成相当流畅的文本。您也可以将目标验证损失从 1.45 提高到 2.00。

在 M3 Max 芯片和 36 GB 内存上运行我们的解决方案代码并采用调优后的学习率，我们使用 $\text{批次大小} \times \text{总步数} \times \text{上下文长度} = 32 \times 5000 \times 256 = 40,960,000$ 标记，这在 `cpu` 上耗时 1 小时 22 分钟，在 `mps` 上耗时 36 分钟。在第 5000 步时，我们实现了 1.80 的验证损失。

一些额外提示：

- 当使用 X 训练步骤时，我们建议调整余弦学习率衰减调度，使其恰好在第 X 步终止衰减（即达到最小学习率）。
- 当使用 `mps` 时，请**不要**使用 TF32 内核，即**不要**设置 `torch.set_float32_matmul_precision('high')`，就像在 `cuda` 设备上可能做的那样。我们尝试启用 TF32 内核与 `mps`（`torch` 版本 2.6.0），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` 对模型进行即时编译来加速训练。具体来说：— 在 `cpu` 上，使用 `model = torch.compile(model)` **编译模型**— 在 `mps` 上，您可以使用 `model = torch.compile(model, backend="aot_eager")` 在一定程度上优化反向传播。截至 `torch` 版本 2.6.0，`mps` 不支持使用 Inductor 编译器进行编译。

(b) 传统经验认为最佳学习率位于“稳定性边缘”。请研究学习率开始发散的点与您最佳学习率之间的关系。

可交付成果：包含至少一次发散运行的递增学习率学习曲线，以及关于此现象与收敛速率关系的分析。

现在让我们改变批次大小，观察训练会发生什么变化。批次大小很重要——通过执行更大的矩阵乘法，它们能让我们从图形处理器获得更高效率，但批次大小是否总是越大越好？让我们通过实验来寻找答案。

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

将您的批次大小从 1 一直调整到图形处理器内存极限。在中间至少尝试几种批次大小，包括典型大小如 64 和 128。

可交付成果：不同批次大小运行的学习曲线。如有必要，应重新优化学习率。

可交付成果：用几句话讨论您关于批次大小及其对训练影响的发现。

有了手中的解码器，我们现在可以生成文本了！我们将从模型中生成并查看其质量。作为参考，您应该获得至少与以下示例一样好的输出。

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

从前，有个名叫莉莉的漂亮女孩。她特别喜欢吃口香糖，尤其是那种大块的黑色口香糖。有一天，莉莉的妈妈请她帮忙做晚饭。莉莉非常兴奋！她很喜欢帮妈妈做事。莉莉的妈妈为晚餐煮了一大锅汤。莉莉开心地说：“谢谢你，妈妈！我爱你。”她帮妈妈把汤倒进一个大碗里。晚餐后，莉莉的妈妈又做了些美味的汤。莉莉非常喜欢！她说：“谢谢你，妈妈！这汤真好喝！”妈妈笑着说：“很高兴你喜欢，莉莉。”她们做完饭后又继续一起烹饪。故事结束。

低资源/缩减规模提示：在中央处理器或苹果芯片上生成文本

如果您改用处理了4000万词元的低资源配置，您应该会看到仍然类似英语但不如上述流畅的生成结果。例如，我们在4000万词元上训练的TinyStories语言模型的示例输出如下：

从前，有个名叫苏的小女孩。苏有一颗她非常喜爱的牙齿。那是他最好的脑袋。有一天，苏去散步时遇到了一只瓢虫！他们成了好朋友，一起在小路上玩耍。

“嘿，波莉！我们出去吧！”蒂姆说。苏望着天空，发现很难找到跳舞闪耀的方式。她微笑着同意帮助说话！”当苏看着天空移动时，它是什么。她

以下是精确的问题陈述及我们的要求：

问题(generate)：生成文本（1分）

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

可交付成果：至少256个标记的文本转储（或直到第一个 `<|endoftext|>` 标记），以及对此输出流畅度的简要评论，以及影响此输出质量好坏的至少两个因素。

7.3 消融实验和架构修改

理解Transformer的最佳方式实际上是修改它并观察其行为。我们现在将进行一些简单的消融实验和修改。

消融实验 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块中移除RMSNorm，看看会发生什么。

问题 (layer_norm_ablation)：移除RMSNorm并训练（1分）（1 H100小时）

从你的Transformer中移除所有RMSNorm并进行训练。在之前的最优学习率下会发生什么？通过使用较低的学习率能否获得稳定性？

可交付成果：移除RMSNorm并训练时的学习曲线，以及最佳学习率对应的学习曲线。

可交付成果：关于RMSNorm影响的简要评述。

现在让我们研究另一种初看似乎任意的层归一化选择。前归一化Transformer块定义为

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

这是对原始Transformer架构为数不多的'共识'修改之一，该架构采用了后归一化方法。

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

让我们恢复到后归一化方法，看看会发生什么。

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

将你的前归一化Transformer实现修改为后归一化版本。使用后归一化模型进行训练并观察结果。
可交付成果：后归一化Transformer与前归一化模型对比的学习曲线。

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

消融实验2：位置嵌入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的模型（NoPE）。事实证明，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]。我们现在将通过实证测试来比较NoPE与RoPE的表现。

问题 (no_pos_emb)：实现无位置嵌入 (1分) (1 H100 小时)

修改你带有 RoPE 的 Transformer 实现，完全移除位置嵌入信息，观察会发生什么。

可交付成果：比较 RoPE 和无位置嵌入性能的学习曲线。

消融实验 3：SwiGLU 对比 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 激活但无门控线性单元（GLU）的前馈网络的性能，来测试前馈网络中门控机制的重要性：

$$\text{FFN}_{\text{SiLU}}(x) = W_2 \text{SiLU}(W_1 x). \quad (25)$$

回忆在我们的 SwiGLU 实现中，我们将内部前馈层的维度设置为大约 $d_{\text{ff}} = \frac{8}{3} d_{\text{model}}$ （同时确保 $d_{\text{ff}} \bmod 64 = 0$ ，以利用 GPU 张量核心）。在你的 FFN_{SiLU} 实现中，应设置 $d_{\text{ff}} = 4 \times d_{\text{model}}$ ，以大致匹配 SwiGLU 前馈网络的参数量（后者具有三个而非两个权重矩阵）。

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

可交付成果：一条比较 SwiGLU 和 SiLU 前馈网络性能的学习曲线，且两者的参数数量大致匹配。

可交付成果：用几句话讨论你的发现。

低资源/缩减规模提示：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、噪声更多的网络数据集（Open-WebText），进行架构修改实验，并（可选地）向课程排行榜提交结果。

在OpenWebText上将语言模型训练到流畅需要很长时间，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预训练数据集，该数据集来自网络爬取。OpenWebText的一个小样本 [Gokaslan等人, 2019] 也以单个文本文件的形式提供：请参阅第1节了解如何访问此文件。

这是来自OpenWebText的一个示例。请注意文本如何更加真实、复杂和多样化。你可能需要浏览训练数据集，以了解网络爬取语料库的训练数据是什么样的。

示例 (owt_example)：来自OWT的一个示例

《棒球展望》技术总监哈里·帕夫拉迪斯在聘用乔纳森·贾奇时承担了风险。帕夫拉迪斯深知，正如艾伦·施瓦茨在《数字游戏》中所写：“美国文化中没有任何领域比棒球运动员的表现被更精确地计数、更热情地量化。”通过随处点击几下，你就能发现诺阿·辛德加德的快球在飞向本垒板时每分钟旋转超过2,100次，纳尔逊·克鲁兹在2016年合格击球手中拥有比赛最高的平均出射速度，以及无数其他仿佛从电子游戏或科幻小说中截取的趣闻。不断增长的数据海洋赋予了棒球文化中日益重要的角色——分析型业余爱好者——更多力量。

这种赋权也带来了更严格的审视——不仅针对测量数据，也针对背后的人员和出版物。帕夫拉迪斯深知《棒球展望》伴随定量不完美性而来的反弹。他也清楚该网站的接球指标需要重新设计，并且需要一位博学的专业人士——能够解决复杂统计建模问题的人——来完成这项工作。

“他让我们感到害怕。”哈里·帕夫拉迪斯

帕夫拉迪斯有种直觉，认为贾奇“理解了这一点”，这是基于后者的文章以及他们在公司赞助的棒球场活动中的互动。不久之后，两人边喝酒边交谈。帕夫拉迪斯的直觉得到了验证。贾奇很适合这个职位——更妙的是，他是自愿适合的。“我和很多人谈过，”帕夫拉迪斯说，“他是唯一有勇气接受这个职位的人。” [...]

注意：您可能需要为此实验重新调整超参数，例如学习率或批次大小。

问题 (main_experiment)：在OWT上的实验（2分）（3 H100小时）

在OpenWebText上使用与TinyStories相同的模型架构和总训练迭代次数来训练您的语言模型。该模型表现如何？

可交付成果：你的语言模型在OpenWebText上的学习曲线。描述与TinyStories的损失值差异——我们应如何解读这些损失？

可交付成果：来自OpenWebText语言模型的生成文本，格式与TinyStories输出相同。这段文本的流畅度如何？尽管我们使用与TinyStories相同的模型和计算预算，为何输出质量反而更差？

7.5 自定义修改 + 排行榜

恭喜您成功抵达这一阶段。您即将完成所有任务！现在请尝试改进Transformer架构，并观察您的超参数和架构在班级中与其他同学的对比表现。

排行榜规则除以下规定外无其他限制：

Runtime您的提交在H100上的运行时间不得超过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。

Data 您只能使用我们提供的OpenWebText训练数据集。

除此之外，您可以自由实现任何创意。若需要实现方向的灵感，可参考以下资源：

- 最先进的开源大语言模型系列，例如 Llama 3 [Grattafiori 等人, 2024] 或 Qwen 2.5 [杨等人, 2024]。
- NanoGPT 速通代码库 (<https://github.com/KellerJordan/modded-nanogpt>)，社区成员在此发布许多有趣的“速通”小规模语言模型预训练的修改方案。例如，可追溯至原始 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 等人 [2017]（第 3.4 节）和 Chowdhery 等人 [2022]（第 2 节））。若尝试权重绑定，可能需要降低嵌入/语言模型头部初始化的标准差。

在尝试完整的 1.5 小时运行前，建议在 OpenWebText 的小型子集或 TinyStories 上测试这些修改。

需要提醒的是，此排行榜中某些有效的修改方案可能无法推广至更大规模的预训练。我们将在课程的缩放定律单元进一步探讨这一概念。

问题（leaderboard）：排行榜（6分）（10 H100小时）

您将根据上述排行榜规则训练一个模型，目标是在1.5 H100小时内将语言模型的验证损失降至最低。

可交付成果：最终记录的验证损失、一条明确显示挂钟时间x轴小于1.5小时的相关学习曲线，以及对你所做工作的描述。我们期望排行榜提交至少能超越损失值为5.0的朴素基线。请在此处提交至排行榜：<https://github.com/stanford-cs336/assignment1-basics-leaderboard>。

参考文献

罗南·埃尔丹和李元智。TinyStories：语言模型需要多小才能说出连贯的英语？，2023年。arXiv:2305.07759。亚伦·戈卡斯兰、瓦尼亚·科恩、艾莉·帕夫利克和斯特凡妮·特莱克斯。开放网络文本语料库。http://Skylion007.github.io/OpenWebTextCorpus，2019年。里科·森里希、巴里·哈多和亚历山德拉·伯奇。基于子词单元的稀有词神经机器翻译。载于ACL会议录，2016年。王昌汉、赵灵珙和顾家涛。基于字节级子词的神经机器翻译，2019年。arXiv:1909.03341。菲利普·盖奇。一种新的数据压缩算法。C用户期刊，12(2):23–38，1994年2月。ISSN 0898-9788。亚历克·拉德福德、吴杰夫、雷文·柴尔德、栾大卫、达里奥·阿莫代伊和伊利亚·苏茨克弗。语言模型是无监督多任务学习器，2019年。亚历克·拉德福德、卡尔提克·纳拉辛汉、蒂姆·萨利曼斯和伊利亚·苏茨克弗。通过生成式预训练提升语言理解能力，2018年。

阿希什·瓦斯瓦尼、诺姆·沙泽尔、尼基·帕尔马、雅各布·乌兹科雷特、利翁·琼斯、艾丹·N·戈麦斯、Łukasz Kaiser 和伊利亚·波洛苏金。注意力就是一切。载于神经信息处理系统大会论文集，2017年。

阮全和朱利安·萨拉萨尔。无泪Transformer：改进自注意力机制的归一化。载于IWSWLT论文集，2019年。

熊瑞斌、杨运昌、何迪、郑凯、郑书新、邢晨、张会帅、蓝艳艳、王立威和刘铁岩。论Transformer架构中的层归一化。载于ICML论文集，2020年。

吉米·雷·巴、杰米·瑞安·基罗斯和杰弗里·E·辛顿。层归一化，2016年。arXiv:1607.06450。雨果·图夫龙、蒂博·拉夫里尔、高蒂埃·伊扎卡、泽维尔·马蒂内、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·阿兹哈尔、奥雷利安·罗德里格斯、阿尔芒·朱林、爱德华·格拉夫和纪尧姆·朗普勒。Llama：开放高效的基础语言模型，2023年。arXiv:2302.13971。

张彪和里科·森里希。均方根层归一化。载于神经信息处理系统大会论文集，2019年。

亚伦·格拉塔菲奥里、阿比曼纽·杜贝、阿布希纳夫·乔赫里、阿布希纳夫·潘迪、阿布舍克·卡迪安、艾哈迈德·阿尔·达赫莱、艾莎·莱特曼、阿基尔·马图尔、艾伦·谢尔顿、亚历克斯·沃恩、艾米·杨、安吉拉·范、阿尼鲁德·戈亚尔、安东尼·哈茨霍恩、奥博·杨、阿尔奇·米特拉、阿尔奇·斯里瓦库马尔、阿尔乔姆·科列涅夫、亚瑟·辛斯瓦克、阿伦·拉奥、阿斯顿·张、奥雷利安·罗德里格斯、奥斯汀·格雷格森、艾娃·斯帕塔鲁、巴蒂斯特·罗齐埃、贝瑟尼·比龙、宾·唐、博比·陈、夏洛特·科什特、查亚·纳亚克、克洛伊·毕、克里斯·马拉、克里斯·麦康奈尔、克里斯蒂安·凯勒、克里斯托夫·图雷、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、丹妮尔·平茨、丹尼·利夫希茨、丹尼·怀亚特、大卫·埃西奥布、德鲁夫·乔杜里、德鲁夫·马哈詹、迭戈·加西亚·奥拉诺、迭戈·佩里诺、迪乌克·胡普克斯、埃戈尔·拉科姆金、埃哈布·阿尔巴达维、艾琳娜·洛巴诺娃、艾米丽·迪南、埃里克·迈克尔·史密斯、菲利普·拉登诺维奇、弗朗西斯科·古兹曼、弗兰克·张、加布里埃尔·辛纳夫、加布里埃尔·李、乔治亚·刘易斯·安德森、戈文德·塔泰、格雷姆·奈尔、格雷瓜尔·米亚隆、庞关、吉列姆·库库雷尔、海莉·阮、汉娜·科瓦尔、许虎、雨果·图夫龙、伊利亚恩·扎罗夫、伊马诺尔·阿里埃塔·伊巴拉、伊莎贝尔·克洛曼、伊山·米斯拉、伊万·埃夫蒂莫夫、杰克·张、杰德·科佩特、李在元、扬·格费尔特、亚娜·弗拉内斯、杰森·帕克、杰·马哈德奥卡、吉特·沙阿、耶尔默·范德林德、詹妮弗·比洛克、珍妮·洪、珍妮·李、杰里米·傅、纪建峰、贾祖

黄、刘嘉文、王杰、于杰操、乔安娜·比顿、乔·斯皮萨克、郑秀公园、约瑟夫·罗卡、约书亚·约翰斯顿、约书亚·萨克斯、贾俊腾、卡尔扬·瓦苏登·阿尔瓦拉、卡蒂克·普拉萨德、卡蒂凯亚·乌帕萨尼、凯特·普拉维亚克、柯力、肯尼斯·希菲尔德、凯文·斯通、哈立德·埃尔·阿里尼、克里西卡·艾耶、克什蒂兹·马利克、昆利·邱、库纳尔·巴拉、库沙尔·拉科蒂亚、劳伦·兰塔拉·耶里、劳伦斯·范德马滕、劳伦斯·陈、谭亮、莉兹·詹金斯、路易斯·马丁、洛维什·马达安、卢博·马洛、卢卡斯·布莱彻、卢卡斯·兰德扎特、卢克·德奥利维拉、玛德琳·穆齐、马赫什·帕苏普莱蒂、曼纳特·辛格、马诺哈尔·帕卢里、马尔钦·卡达斯、玛丽亚·辛普凯利、马修·奥尔德姆、马蒂厄·丽塔、玛雅·帕夫洛娃、梅兰妮·坎巴杜尔、迈克·刘易斯、闵思、米特什·库马尔·辛格、莫娜·哈桑、纳曼·戈亚尔、纳尔杰斯·托拉比、尼古拉·巴什利科夫、尼古拉·博戈伊切夫、尼拉德里·查特吉、宁张、奥利维尔·杜谢纳、奥努尔·切莱比、帕特里克·阿尔拉西、张鹏川、李鹏伟、佩塔尔·瓦西奇、彼得·温、普拉杰瓦尔·巴尔加瓦、普拉蒂克·杜巴尔、普拉文·克里希南、普尼特·辛格·科拉、徐普新、何清、董庆晓、拉加万·斯里尼瓦桑、拉吉·加纳帕西、拉蒙·卡尔德里、里卡多·西尔维拉·卡布拉尔、罗伯特·斯托伊尼克、罗伯特·雷勒努、罗汉·马赫斯瓦里、罗希特·吉尔达尔、罗希特·帕特尔、罗曼·索维斯特雷、罗尼·波利多罗、罗shan·桑巴利、罗斯·泰勒、鲁安·席尔瓦、侯瑞、王瑞、萨加尔·霍塞尼、萨哈娜·琴纳巴索帕、桑杰·辛格、肖恩·贝尔、金瑞贤、谢尔盖·埃杜诺夫、聂少良、沙兰·纳兰、沙拉思·拉帕西、沈升、万胜业、舒鲁蒂·博萨莱、张顺、西蒙·范登亨德、索米亚·巴特拉、斯宾塞·惠特曼、斯滕·苏特拉、斯特凡·科洛、苏钦·古鲁甘、悉尼·博罗丁斯基、塔玛·赫尔曼、塔拉·福勒、塔雷克·谢沙、托马斯·乔治乌、托马斯·西亚洛姆、托比亚斯·斯佩克巴赫、托多尔·米哈伊洛夫、肖彤、乌杰瓦尔·卡恩、韦达努吉·戈萨米、维博尔·古普塔、维格内什·拉马纳坦、维克托·凯尔凯兹、文森特·贡古埃、维吉妮·杜、维什·沃格蒂、维托尔·阿尔比罗、弗拉丹·彼得罗维奇、褚伟伟、熊文翰、傅文寅、惠特尼·米尔斯、泽维尔·马蒂内、王小东、王晓芳、谭晓青、夏西德、谢新峰、贾旭超、王学伟、耶尔·戈德施拉格、亚什斯·高尔、亚斯明·巴巴伊、易雯、宋艺文、张宇晨、李悦、毛宇宁、扎卡里·德尔皮埃尔·库德尔、严正、陈正兴、佐伊·帕帕基波斯、阿迪亚·辛格、阿尤希·斯里瓦斯塔瓦、阿巴·贾恩、亚当·凯尔西、亚当·沙恩菲尔德、阿迪亚·甘吉迪、阿道弗·维多利亚、阿胡瓦·戈德斯坦德、阿贾伊·梅农、阿贾伊·夏尔马、亚历克斯·博森伯格、阿列克谢·巴耶夫斯基、艾莉·范斯坦、阿曼达·卡莱特、阿米特·桑加尼、阿莫斯·特奥、阿纳姆·尤努斯、安德烈·卢普、安德烈斯·阿尔瓦拉多、安德鲁·卡普尔斯、安德鲁·顾、安德鲁·何、安德鲁·波尔顿、安德鲁·瑞安、安基特·拉姆钱达尼、安妮·董、安妮·弗朗哥、阿努吉·戈亚尔、阿帕拉吉塔·萨拉夫、阿尔卡班杜·乔杜里、阿什利·加布里埃尔、阿什温·巴拉姆贝、阿萨夫·艾森曼、阿扎德·亚兹丹、博·詹姆斯、本·莫勒、本杰明·莱昂哈迪、伯尼·黄、贝丝·劳埃德、贝托·德保拉、巴尔加维·帕兰贾佩、刘兵、吴波、倪博宇、布拉登·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·蒙塔沃、卡尔·帕克、卡莉·伯顿、卡塔利娜·梅希亚、刘策、王昌汉、金昌圭、周超、切斯特·胡、朱敬祥、克里斯·蔡、克里斯·廷达尔、克里斯托夫·费希滕霍费尔、高辛西娅、达蒙·西文、达娜·贝蒂、丹尼尔·克雷默、丹尼尔·李、戴维·阿德金斯、徐大卫、达维德·泰斯托金、迪莉娅·戴维、德维·帕里克、戴安娜·利斯科维奇、迪德姆·福斯、王定康、黎德、达斯汀·霍兰德、爱德华·道林、艾萨·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、埃里克·团·黎、埃里克·布林克曼、埃斯特班·阿考特、埃文·邓巴、埃文·斯马瑟斯、孙飞、费利克斯·克罗伊克、田峰、菲利波斯·科基诺斯、菲拉特·奥兹格内尔、弗朗切斯科·卡焦尼、弗兰克·卡纳耶特、弗兰克·塞德、加布里埃拉·梅迪纳·弗洛雷斯、加布里埃拉·施瓦茨、加达·巴迪尔、乔治亚·斯威、吉尔·哈尔彭、格兰特·赫尔曼、格里戈里·西佐夫、张光义、古纳·拉克什米纳拉亚南、哈坎·伊南、哈密德·肖贾纳泽里、邹涵、汉娜·王、查汉文、哈龙·哈比卜、哈里森·鲁道夫、海伦·苏克、亨利·阿斯佩格伦、亨特·高盛、詹鸿元、易卜拉欣·达姆拉吉、伊戈尔·莫利博格、伊戈尔·图法诺夫、伊利亚斯·莱昂蒂亚迪斯、伊琳娜·埃琳娜·韦利切、伊泰·加特、杰克·韦斯曼、詹姆斯·格博斯基、詹姆斯·科利、珍妮丝·拉姆、贾菲特·阿舍、让·巴蒂斯特·加亚、杰夫·马库斯、杰夫·唐、詹妮弗·陈、詹妮·甄、杰里米·赖岑斯坦、杰里米·特布尔、杰西卡·钟、金健、杨静怡、乔·卡明斯、乔恩·卡维尔、乔恩·谢泼德、乔纳森·麦克菲、乔纳森·托雷斯、乔希·金斯堡、王俊杰、吴凯、U Kam Hou、卡兰·萨克塞纳、卡蒂凯·坎德瓦尔、卡塔云·赞德、凯西·马托西奇、考希克·维拉加拉万、凯利·米切莱纳、李可谦、基兰·贾加迪什、黄坤、库纳尔·乔拉、凯尔·黄、陈来林、加格·拉克什亚、薰衣草A、莱安德罗·席尔瓦、李·贝尔、张磊、郭良鹏、于立成、利龙·莫什科维奇、卢卡·韦尔施泰特、马迪安·哈布萨、马纳

mad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, 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 Ratanachandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Rutu 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 Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, 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 Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, 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, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao 和 Zhiyu Ma. Llama 3 模型群, 2024 年。网址 <https://arxiv.org/abs/2407.21783>。

杨安、杨宝松、张北辰、惠斌元、郑博、于博文、李承源、刘大一恒、黄斐、魏浩然、林欢、杨健、涂建宏、张建伟、杨建新、杨佳熹、周靖人、林俊暘、党凯、卢柯明、鲍克钦、杨可欣、余乐、李梅、薛明峰、张培、朱钦、门锐、林润基、李天浩、夏廷宇、任行章、任宣成、范阳、苏阳、张艺昌、万宇、刘玉琼、崔泽宇、张振如、邱子涵。Qwen2.5技术报告。*arXiv*预印本 *arXiv:2412.15115*, 2024年。

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, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayanan Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, Noah Fiedel. PaLM: 基于Pathways扩展语言建模, 2022年。arXiv:2204.02311。

Dan Hendrycks 和 Kevin Gimpel。通过高斯误差线性单元桥接非线性与随机正则化器, 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S型加权线性单元, 2017年。网址 <https://arxiv.org/abs/1702.03118>。

扬·N·多芬、安吉拉·范、迈克尔·奥利和大卫·格兰吉尔。使用门控卷积网络进行语言建模, 2017年。网址 <https://arxiv.org/abs/1612.08083>。

诺姆·沙泽尔。GLU变体改进Transformer, 2020年。arXiv:2002.05202。

苏剑林、陆宇、潘胜锋、文博和刘云峰。RoFormer: 使用旋转位置嵌入增强的Transformer, 2021年。

迪德里克·P·金马 和吉米·巴。Adam：一种随机优化方法。载于国际学习表征会议论文集，2015年。

伊利亚·洛什奇洛夫和弗兰克·胡特。解耦权重衰减正则化。载于国际学习表征会议论文集，2019年。

汤姆·B·布朗、本杰明·曼、尼克·赖德、梅兰妮·苏比亚、贾里德·卡普兰、普拉富拉·达里瓦尔、阿尔温德·尼拉坎坦、普拉纳夫·希亚姆、吉里什·萨斯特里、阿曼达·阿斯科尔、桑迪尼·阿加瓦尔、阿里尔·赫伯特·沃斯、格雷琴·克鲁格、汤姆·赫尼根、雷文·柴尔德、阿迪蒂亚·拉梅什、丹尼尔·M·齐格勒、杰弗里·吴、克莱门斯·温特、克里斯托弗·赫西、马克·陈、埃里克·西格勒、马特乌什·利特温、斯科特·格雷、本杰明·切斯、杰克·克拉克、克里斯托弗·伯纳、萨姆·麦坎德利什、亚历克·拉德福德、伊利亚·苏茨克弗和达里奥·阿莫代伊。语言模型是小样本学习器。载于神经信息处理系统大会论文集，2020年。

贾里德·卡普兰、萨姆·麦坎德利什、汤姆·赫尼根、汤姆·B·布朗、本杰明·切斯、雷文·柴尔德、斯科特·格雷、亚历克·拉德福德、杰弗里·吴和达里奥·阿莫代伊。神经语言模型的缩放定律，2020年。arXiv:2001.08361。

乔丹·霍夫曼、塞巴斯蒂安·博尔戈、亚瑟·门施、叶莲娜·布哈茨卡娅、特雷弗·蔡、伊莱扎·卢瑟福、迭戈·德拉斯卡萨斯、丽莎·安妮·亨德里克斯、约翰内斯·韦尔布尔、艾丹·克拉克、汤姆·亨尼根、埃里克·诺兰、凯蒂·米利肯、乔治·范登德里斯切、博格丹·达莫克、奥瑞莉亚·盖伊、西蒙·奥辛德罗、卡伦·西蒙尼安、埃里希·埃尔森、杰克·W·雷、奥里奥尔·比尼亚尔斯和洛朗·西弗雷。训练计算最优的大型语言模型，2022年。arXiv:2203.15556。

阿里·霍尔茨曼、扬·拜斯、杜丽、麦克斯韦·福布斯和崔艺珍。神经文本退化的奇特案例。发表于国际学习表征会议论文集，2020年。

蔡耀鸿、白少杰、山田诚、路易-菲利普·莫伦西和鲁斯兰·萨拉赫丁诺夫。Transformer剖析：通过核视角统一理解Transformer的注意力机制。见乾健太郎、蒋静、黄永辉和万小军编辑的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，第4344–4353页，中国香港，2019年11月。计算语言学协会。数字对象标识符：10.18653/v1/D19-1443。网址 <https://aclanthology.org/D19-1443/>。

阿米尔侯赛因·卡泽姆内贾德、因基特·帕迪、卡尔蒂基扬·纳特桑、帕耶尔·达斯和希瓦·雷迪。位置编码对Transformer模型长度泛化能力的影响。见第三十七届神经信息处理系统大会，2023年。网址<https://openreview.net/forum?id=Drrl2gcjzl>。