CS336 Assignment 1 (basics): Building a Transformer LM

Version 1.0.5

CS336 Staff

Spring 2025

1 Assignment Overview

In this assignment, you will build all the components needed to train a standard Transformer language model (LM) from scratch and train some models.

What you will implement

1. Byte-pair encoding (BPE) tokenizer (§2)
2. Transformer language model (LM) (§3)
3. The cross-entropy loss function and the AdamW optimizer (§4)
4. The training loop, with support for serializing and loading model and optimizer state (§5)

What you will run

1. Train a BPE tokenizer on the TinyStories dataset.
2. Run your trained tokenizer on the dataset to convert it into a sequence of integer IDs.
3. Train a Transformer LM on the TinyStories dataset.
4. Generate samples and evaluate perplexity using the trained Transformer LM.
5. Train models on OpenWebText and submit your attained perplexities to a leaderboard.

What you can use We expect you to build these components from scratch. In particular, you may *not* use any definitions from `torch.nn`, `torch.nn.functional`, or `torch.optim` except for the following:

- `torch.nn.Parameter`
- Container classes in `torch.nn` (e.g., `Module`, `ModuleList`, `Sequential`, etc.)[1]
- The `torch.optim.Optimizer` base class

You may use any other PyTorch definitions. If you would like to use a function or class and are not sure whether it is permitted, feel free to ask on Slack. When in doubt, consider if using it compromises the "from-scratch" ethos of the assignment.

[1] See PyTorch.org/docs/stable/nn.html#containers for a full list.

CS336 作业1（基础）：构建Transformer语言模型

版本 1.0.5

CS336 教师团队

2025 春季

1 作业概述

在本作业中，您将从零开始构建训练标准Transformer语言模型（LM）所需的所有组件，并训练一些模型。

您将实现的内容

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）

您将运行的内容

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对数据集进行操作，将其转换为整数ID序列。3. 在TinyStories数据集上训练一个Transformer LM。4. 使用训练好的Transformer LM生成样本并评估困惑度。5. 在OpenWebText上训练模型并将获得的困惑度提交到排行榜。

您可以使用的内容我们期望您从头开始构建这些组件。特别是，您不得使用 `torch.nn`、 `torch.nn.functional` 或 `torch.optim` 中的任何定义，除非以下内容：

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

您可以使用任何其他 PyTorch 定义。如果您想使用一个函数或类，但不确定是否允许，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“从头开始”精神。

[1] See PyTorch.org/docs/stable/nn.html#containers 完整 list.

Statement on AI tools Prompting LLMs such as ChatGPT is permitted for low-level programming questions or high-level conceptual questions about language models, but using it directly to solve the problem is prohibited.

We strongly encourage you to disable AI autocomplete (e.g., Cursor Tab, GitHub CoPilot) in your IDE when completing assignments (though non-AI autocomplete, e.g., autocompleting function names is totally fine). We have found that AI autocomplete makes it much harder to engage deeply with the content.

What the code looks like All the assignment code as well as this writeup are available on GitHub at:

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

Please `git clone` the repository. If there are any updates, we will notify you so you can `git pull` to get the latest.

1. `cs336_basics/*`: This is where you write your code. Note that there's no code in here—you can do whatever you want from scratch!
2. `adapters.py`: There is a set of functionality that your code must have. For each piece of functionality (e.g., scaled dot product attention), fill out its implementation (e.g., `run_scaled_dot_product_attention`) by simply invoking your code. Note: your changes to `adapters.py` should not contain any substantive logic; this is glue code.
3. `test_*.py`: This contains all the tests that you must pass (e.g., `test_scaled_dot_product_attention`), which will invoke the hooks defined in `adapters.py`. Don't edit the test files.

How to submit You will submit the following files to Gradescope:

- `writeup.pdf`: Answer all the written questions. Please typeset your responses.
- `code.zip`: Contains all the code you've written.

To submit to the leaderboard, submit a PR to:

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

See the `README.md` in the leaderboard repository for detailed submission instructions.

Where to get datasets This assignment will use two pre-processed datasets: TinyStories [Eldan and Li, 2023] and OpenWebText [Gokaslan et al., 2019]. Both datasets are single, large plaintext files. If you are doing the assignment with the class, you can find these files at `/data` of any non-head node machine.

If you are following along at home, you can download these files with the commands inside the `README.md`.

> **Low-Resource/Downscaling Tip: Init**
>
> Throughout the course's assignment handouts, we will give advice for working through parts of the assignment with fewer or no GPU resources. For example, we will sometimes suggest **downscaling** your dataset or model size, or explain how to run training code on a MacOS integrated GPU or CPU. You'll find these "low-resource tips" in a blue box (like this one). Even if you are an enrolled Stanford student with access to the course machines, these tips may help you iterate faster and save time, so we recommend you to read them!

AI工具声明允许使用ChatGPT等LLM进行低级编程问题或关于语言模型的高级概念性问题的提示，但直接用它解决问题是禁止的。

我们强烈建议您在完成作业时（尽管非AI自动完成，例如自动完成函数名是完全可以的）在您的IDE中禁用AI自动完成（例如，光标制表符，GitHub CoPilot）。我们发现AI自动完成使得深入内容变得非常困难。

代码看起来像什么所有作业代码以及这份文档都可以在GitHub上找到：

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

请 `git clone` 仓库。如果有任何更新，我们将通知您，以便您可以 `git pull` 以获取最新版本。

1. `cs336_basics/*`：这是您编写代码的地方。请注意，这里没有代码——您可以从头开始做任何您想做的事情！
2. `adapters.py`：您的代码必须具备一系列功能。对于每个功能（例如，缩放点积注意力），只需通过调用您的代码即可填写其实现（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；这是粘合代码。
3. `test_*.py`：这包含您必须通过的所有测试（例如，`test_scaled_dot_product_attention`），它将调用在 `adapters.py` 中定义的钩子。不要编辑测试文件。

如何提交 您需要将以下文件提交到 Gradescope：

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

要将提交到排行榜，请向以下地址提交PR：

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

请参阅排行榜仓库中的 `README.md` 以获取详细的提交说明。

数据集获取位置 本作业将使用两个预处理的语料库：TinyStories [Eldan 和 Li,2023] 以及 OpenWebText [Gokaslan 等 2019]。这两个语料库都是单个、大型纯文本文件。如果您与班级一起完成作业，您可以在任何非头节点机器的 `/data` 中找到这些文件。

如果您在家中跟随，您可以使用 `README.md`内部的命令下载这些文件。

> **低资源/降尺度提示：初始化**
>
> 在整个课程的作业手册中，我们将提供有关如何使用较少或没有GPU资源完成作业各部分的建议。例如，我们有时会建议**降维**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。您将在蓝色框（就像这个一样）中找到这些“低资源提示”。即使您是拥有课程机器访问权限的斯坦福注册学生，这些提示也可能帮助您更快迭代并节省时间，所以我们建议您阅读它们！

Low-Resource/Downscaling Tip: Assignment 1 on Apple Silicon or CPU

With the staff solution code, we can train an LM to generate reasonably fluent text on an Apple M3 Max chip with 36 GB RAM, in under 5 minutes on Metal GPU (MPS) and about 30 minutes using the CPU. If these words don't mean much to you, don't worry! Just know that if you have a reasonably up-to-date laptop and your implementation is correct and efficient, you will be able to train a small LM that generates simple children's stories with decent fluency.

Later in the assignment, we will explain what changes to make if you are on CPU or MPS.

低资源/降尺度提示：在苹果硅或CPU上完成作业1

有了员工解决方案代码，我们可以在苹果M3上训练一个LM来生成合理流畅的文本
在Metal GPU（MPS）上不到5分钟，使用CPU大约需要30分钟，配备36 GB RAM的最大芯片。如果这些话对你来说没什么意义，别担心！只需知道，如果你有一台相对较新的笔记本电脑，并且你的实现是正确且高效的，你将能够训练一个小型LM，生成简单儿童故事，并具有不错的流畅度。

在作业的后面部分，我们将解释如果你在CPU或MPS上，需要做出哪些更改。

2 Byte-Pair Encoding (BPE) Tokenizer

In the first part of the assignment, we will train and implement a byte-level byte-pair encoding (BPE) tokenizer [Sennrich et al., 2016, Wang et al., 2019]. In particular, we will represent arbitrary (Unicode) strings as a sequence of bytes and train our BPE tokenizer on this byte sequence. Later, we will use this tokenizer to encode text (a string) into tokens (a sequence of integers) for language modeling.

2.1 The Unicode Standard

Unicode is a text encoding standard that maps characters to integer *code points*. As of Unicode 16.0 (released in September 2024), the standard defines 154,998 characters across 168 scripts. For example, the character "s" has the code point 115 (typically notated as `U+0073`, where `U+` is a conventional prefix and `0073` is 115 in hexadecimal), and the character "牛" has the code point 29275. In Python, you can use the `ord()` function to convert a single Unicode character into its integer representation. The `chr()` function converts an integer Unicode code point into a string with the corresponding character.

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

Problem (`unicode1`): Understanding Unicode (1 point)

(a) What Unicode character does `chr(0)` return?

Deliverable: A one-sentence response.

(b) How does this character's string representation (`__repr__()`) differ from its printed representation?

Deliverable: A one-sentence response.

(c) What happens when this character occurs in text? It may be helpful to play around with the following in your Python interpreter and see if it matches your expectations:

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

Deliverable: A one-sentence response.

2.2 Unicode Encodings

While the Unicode standard defines a mapping from characters to code points (integers), it's impractical to train tokenizers directly on Unicode codepoints, since the vocabulary would be prohibitively large (around 150K items) and sparse (since many characters are quite rare). Instead, we'll use a Unicode encoding, which converts a Unicode character into a sequence of bytes. The Unicode standard itself defines three encodings: UTF-8, UTF-16, and UTF-32, with UTF-8 being the dominant encoding for the Internet (more than 98% of all webpages).

To encode a Unicode string into UTF-8, we can use the `encode()` function in Python. To access the underlying byte values for a Python `bytes` object, we can iterate over it (e.g., call `list()`). Finally, we can use the `decode()` function to decode a UTF-8 byte string into a Unicode string.

2 字节对编码（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字节对编码（BPE）分词器 [Sennrich 等人，2016, Wang 等人， 2019]。具体来说，我们将任意（Unicode）字符串表示为字节序列，并在该字节序列上训练我们的BPE分词器。稍后，我们将使用此分词器将文本（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，将字符映射到整数 码点。截至Unicode 16.0（于2024年9月发布），该标准定义了168个脚本中的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+` 是一个传统的前缀，0073是十六进制的115），而字符 "牛" 的码点是29275。在Python中，您可以使用 `ord()` 函数将单个Unicode字符转换为它的整数表示。 `chr()` 函数将整数Unicode码点转换为相应的字符串。

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

问题 (`unicode1`): 理解Unicode (1分)

(a) `chr(0)` 返回哪个Unicode字符?

交付物: 一句话回答。

(b) 这个字符的字符串表示形式 (`__repr__()`) 与其打印表示形式有何不同?

交付物: 一句话回答。

(c) 当此字符出现在文本中时会发生什么？您可以在 Python 解释器中尝试以下操作，看看它是否符合您的预期：

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

交付物: 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，但由于词汇表会非常大（约 150K 项）且稀疏（因为许多字符相当罕见），因此直接在 Unicode 码点上训练分词器是不切实际的。相反，我们将使用 Unicode 编码，它将 Unicode 字符转换为一系列字节。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 98% 的所有网页）。

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，我们可以遍历它（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'>
>>> # 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!
```

By converting our Unicode codepoints into a sequence of bytes (e.g., via the UTF-8 encoding), we are essentially taking a sequence of codepoints (integers in the range 0 to 154,997) and transforming it into a sequence of byte values (integers in the range 0 to 255). The 256-length byte vocabulary is *much* more manageable to deal with. When using byte-level tokenization, we do not need to worry about out-of-vocabulary tokens, since we know that *any* input text can be expressed as a sequence of integers from 0 to 255.

Problem (`unicode2`): Unicode Encodings (3 points)

(a) What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32? It may be helpful to compare the output of these encodings for various input strings.

Deliverable: A one-to-two sentence response.

(b) Consider the following (incorrect) function, which is intended to decode a UTF-8 byte string into a Unicode string. Why is this function incorrect? Provide an example of an input byte string that yields incorrect results.

```
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'
```

Deliverable: An example input byte string for which `decode_utf8_bytes_to_str_wrong` produces incorrect output, with a one-sentence explanation of why the function is incorrect.

(c) Give a two byte sequence that does not decode to any Unicode character(s).

Deliverable: An example, with a one-sentence explanation.

2.3 Subword Tokenization

While byte-level tokenization can alleviate the out-of-vocabulary issues faced by word-level tokenizers, tokenizing text into bytes results in extremely long input sequences. This slows down model training, since a


```
>>> 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 码点转换为一系列字节（例如，通过 UTF-8 编码），我们实际上是将一系列码点（范围在 0 到 154,997 之间的整数）转换为一组字节值（范围在 0 到 255 之间的整数）。256 长度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的标记，因为我们知道 任何 输入文本都可以表示为从 0 到 255 的整数序列。

问题(`unicode2`): UnicodeEncodings (3分)

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

交付物：一到两句话的回复。

(b) 考虑以下（不正确）的函数，该函数旨在将 UTF-8 字节字符串解码为一个 Unicode 字符串。为什么这个函数是不正确的？请提供一个输入字节字符串的示例，该字符串会产生不正确的结果。

```
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'
```

交付物: 一个示例输入字节字符串，其中 `decode_utf8_bytes_to_str_wrong` 产生不正确的输出，以及一个关于为什么该函数不正确的简短解释。

(c) 给出一个不解码为任何 Unicode 字符的字节序列。**交付物**: 一个示例，以及一个简短的解释。

2.3 子词标记化

虽然字节级标记化可以减轻词级标记化器面临的词汇表外问题，但将文本标记化为字节会导致输入序列极其长。这会减慢模型训练，因为一个

sentence with 10 words might only be 10 tokens long in a word-level language model, but could be 50 or more tokens long in a character-level model (depending on the length of the words). Processing these longer sequences requires more computation at each step of the model. Furthermore, language modeling on byte sequences is difficult because the longer input sequences create long-term dependencies in the data.

Subword tokenization is a midpoint between word-level tokenizers and byte-level tokenizers. Note that a byte-level tokenizer's vocabulary has 256 entries (byte values are 0 to 225). A subword tokenizer trades-off a larger vocabulary size for better compression of the input byte sequence. For example, if the byte sequence `b'the'` often occurs in our raw text training data, assigning it an entry in the vocabulary would reduce this 3-token sequence to a single token.

How do we select these subword units to add to our vocabulary? Sennrich et al. [2016] propose to use byte-pair encoding (BPE; Gage, 1994), a compression algorithm that iteratively replaces ("merges") the most frequent pair of bytes with a single, new unused index. Note that this algorithm adds subword tokens to our vocabulary to maximize the compression of our input sequences—if a word occurs in our input text enough times, it'll be represented as a single subword unit.

Subword tokenizers with vocabularies constructed via BPE are often called BPE tokenizers. In this assignment, we'll implement a byte-level BPE tokenizer, where the vocabulary items are bytes or merged sequences of bytes, which give us the best of both worlds in terms of out-of-vocabulary handling and manageable input sequence lengths. The process of constructing the BPE tokenizer vocabulary is known as "training" the BPE tokenizer.

2.4 BPE Tokenizer Training

The BPE tokenizer training procedure consists of three main steps.

Vocabulary initialization The tokenizer vocabulary is a one-to-one mapping from bytestring token to integer ID. Since we're training a byte-level BPE tokenizer, our initial vocabulary is simply the set of all bytes. Since there are 256 possible byte values, our initial vocabulary is of size 256.

Pre-tokenization Once you have a vocabulary, you could, in principle, count how often bytes occur next to each other in your text and begin merging them starting with the most frequent pair of bytes. However, this is quite computationally expensive, since we'd have to go take a full pass over the corpus each time we merge. In addition, directly merging bytes across the corpus may result in tokens that differ only in punctuation (e.g., `dog!` vs. `dog.`). These tokens would get completely different token IDs, even though they are likely to have high semantic similarity (since they differ only in punctuation).

To avoid this, we *pre-tokenize* the corpus. You can think of this as a coarse-grained tokenization over the corpus that helps us count how often pairs of characters appear. For example, the word `'text'` might be a pre-token that appears 10 times. In this case, when we count how often the characters 't' and 'e' appear next to each other, we will see that the word 'text' has 't' and 'e' adjacent and we can increment their count by 10 instead of looking through the corpus. Since we're training a byte-level BPE model, each pre-token is represented as a sequence of UTF-8 bytes.

The original BPE implementation of Sennrich et al. [2016] pre-tokenizes by simply splitting on whitespace (i.e., `s.split(" ")`). In contrast, we'll use a regex-based pre-tokenizer (used by GPT-2; Radford et al., 2019) from `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+"""
```

It may be useful to interactively split some text with this pre-tokenizer to get a better sense of its behavior:

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


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，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）。处理这些较长的序列需要模型在每一步进行更多的计算。此外，在字节序列上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。

子词标记化介于词级分词器和字节级分词器之间。请注意，字节级分词器的词汇表有256个条目（字节值是0到225）。子词分词器以更大的词汇量大小为代价，以更好地压缩输入字节序列。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，将其分配到词汇表中将把这个3标记序列缩减为一个标记。

我们如何选择这些子词单元添加到我们的词汇表中？Sennrich等人 [2016] 提出使用字节对编码（BPE；Gage，1994），这是一种迭代地用单个新未使用索引替换（“合并”）最频繁的字节对的压缩算法。请注意，此算法将子词标记添加到我们的词汇表中，以最大化我们输入序列的压缩——如果一个单词在我们的输入文本中出现的次数足够多，它将被表示为一个单独的子词单元。

使用通过BPE构建的词汇表进行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，其中词汇项是字节或字节合并序列，这使我们能够在词汇表外处理和管理可管理的输入序列长度方面达到最佳效果。构建BPE分词器词汇表的过程被称为“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。

词汇初始化分词器词汇表是从字节字符串标记到整数ID的一对一映射。由于我们正在训练一个字节级BPE分词器，我们的初始词汇只是所有字节集合。由于有256种可能的字节值，我们的初始词汇大小为256。

预分词一旦有了词汇表，原则上可以计算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。然而，这相当计算量大，因为每次合并时我们都需要对语料库进行完整遍历。此外，直接在语料库中合并字节可能会导致只有标点符号不同的标记（例如， `dog!` 与 `dog.`）。这些标记将获得完全不同的标记ID，尽管它们很可能具有很高的语义相似性（因为它们只在标点符号上有所不同）。

为了避免这种情况，我们 预分词 语料库。你可以将这视为对语料库的粗粒度分词，帮助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次的预标记。在这种情况下，当我们统计字符 ‘t’ 和 ‘e’ 相邻出现的频率时，我们会看到单词 ‘text’ 中 ‘t’ 和 ‘e’ 是相邻的，我们可以将它们的计数增加10，而不是在语料库中查找。由于我们正在训练一个字节级别的BPE模型，每个预标记都表示为UTF-8字节序列。

Sennrich 等人原始的BPE实现通过简单地根据空白字符（即 `s.split(" ")`）进行分割来进行预分词。相比之下，我们将使用基于正则表达式的预分词器（由GPT-2使用；Radford 等人，2019），来自 `github.com/openai/tiktoken/pull/234/files`：

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

使用此预分词器交互式地分割一些文本，可能会有助于更好地了解其行为：

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

When using it in your code, however, you should use `re.finditer` to avoid storing the pre-tokenized words as you construct your mapping from pre-tokens to their counts.

Compute BPE merges Now that we've converted our input text into pre-tokens and represented each pre-token as a sequence of UTF-8 bytes, we can compute the BPE merges (i.e., train the BPE tokenizer). At a high level, the BPE algorithm iteratively counts every pair of bytes and identifies the pair with the highest frequency ("A", "B"). Every occurrence of this most frequent pair ("A", "B") is then *merged*, i.e., replaced with a new token "AB". This new merged token is added to our vocabulary; as a result, the final vocabulary after BPE training is the size of the initial vocabulary (256 in our case), plus the number of BPE merge operations performed during training. For efficiency during BPE training, we do not consider pairs that cross pre-token boundaries.[2] When computing merges, deterministically break ties in pair frequency by *preferring the lexicographically greater pair*. For example, if the pairs ("A", "B"), ("A", "C"), ("B", "ZZ"), and ("BA", "A") all have the highest frequency, we'd merge ("BA", "A"):

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

Special tokens Often, some strings (e.g., <|endoftext|>) are used to encode metadata (e.g., boundaries between documents). When encoding text, it's often desirable to treat some strings as "special tokens" that should never be split into multiple tokens (i.e., will always be preserved as a single token). For example, the end-of-sequence string <|endoftext|> should always be preserved as a single token (i.e., a single integer ID), so we know when to stop generating from the language model. These special tokens must be added to the vocabulary, so they have a corresponding fixed token ID.

Algorithm 1 of Sennrich et al. [2016] contains an inefficient implementation of BPE tokenizer training (essentially following the steps that we outlined above). As a first exercise, it may be useful to implement and test this function to test your understanding.

Example (`bpe_example`): BPE training example

Here is a stylized example from Sennrich et al. [2016]. Consider a corpus consisting of the following text

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

and the vocabulary has a special token <|endoftext|>.

Vocabulary We initialize our vocabulary with our special token <|endoftext|> and the 256 byte values.

Pre-tokenization For simplicity and to focus on the merge procedure, we assume in this example that pretokenization simply splits on whitespace. When we pretokenize and count, we end up with the frequency table.

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

[2]Note that the original BPE formulation [Sennrich et al., 2016] specifies the inclusion of an end-of-word token. We do not add an end-of-word-token when training byte-level BPE models because all bytes (including whitespace and punctuation) are included in the model's vocabulary. Since we're explicitly representing spaces and punctuation, the learned BPE merges will naturally reflect these word boundaries.

当在您的代码中使用它时，然而，您应该使用 `re.finditer` 以避免在从预标记到其计数的映射过程中存储预分词

计算 BPE 合并 现在我们已经将输入文本转换为预标记，并将每个预标记表示为 UTF-8 字节序列，我们可以计算 BPE 合并（即训练 BPE 分词器）。从高层次来看，BPE 算法迭代地计算每一对字节，并识别频率最高的配对（“A”，“B”）。然后，将这个最频繁的配对（“A”，“B”）的所有出现 合并，即替换为新标记“AB”。这个新合并的标记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，我们不考虑跨越预标记边界的配对。[2] 在计算合并时，通过优先选择字典序较大的配对来确定性解决配对频率的平局。例如，如果配对（“A”，“B”）、（“A”，“C”）、（“B”，“ZZ”）和（“BA”，“A”）都具有最高的频率，我们将合并（“BA”，“A”）：

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

特殊标记通常，一些字符串（例如，<|endoftext|>）用于编码元数据（例如，文档之间的边界）。在编码文本时，通常希望将一些字符串视为“特殊标记”，这些标记不应拆分为多个标记（即，始终保留为单个标记）。例如，序列结束字符串 <|endoftext|> 应始终保留为单个标记（即，单个整数ID），这样我们就可以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，以便它们有相应的固定标记ID。

Sennrich等人算法1 [2016] 包含了一个BPE分词器训练的不高效实现（本质上遵循了我们上面概述的步骤）。作为一个初步练习，实现并测试这个函数可能有助于测试你对这些概念的理解。

示例 (`bpe_example`): BPE 训练示例

以下是一个来自 Sennrich 等人的示例 [2016]。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
```

并且词汇表有一个特殊标记 <|endoftext|>。

词汇表 我们使用特殊标记 <|endoftext|> 和256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，在这个例子中，我们假设预处理简单地根据空白字符进行分割。当我们进行预处理并计数时，我们最终得到频率表。

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

[2]请注意，原始的BPE公式 [Sennrich等人 2016] 指定了包含单词结束标记。在训练字节级BPE模型时，我们不添加单词结束标记，因为所有字节（包括空白字符和标点符号）都包含在模型的词汇表中。由于我们明确表示了空白字符和标点符号，因此学习到的BPE合并将自然反映这些单词边界。

It is convenient to represent this as a `dict[tuple[bytes], int]`, e.g. {(l,o,w): 5 …}. Note that even a single byte is a `bytes` object in Python. There is no `byte` type in Python to represent a single byte, just as there is no `char` type in Python to represent a single character.

Merges We first look at every successive pair of bytes and sum the frequency of the words where they appear {lo: 7, ow: 7, we: 8, er: 2, wi: 3, id: 3, de: 3, es: 9, st: 9, ne: 6, ew: 6}. The pair `('es')` and `('st')` are tied, so we take the lexicographically greater pair, `('st')`. We would then merge the pre-tokens so that we end up with {(l,o,w): 5, (l,o,w,e,r): 2, (w,i,d,e,st): 3, (n,e,w,e,st): 6}.

In the second round, we see that `(e, st)` is the most common pair (with a count of 9) and we would merge into {(l,o,w): 5, (l,o,w,e,r): 2, (w,i,d,est): 3, (n,e,w,est): 6}. Continuing this, the sequence of merges we get in the end will be `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e', 'ne west', 'w i', 'wi d', 'wid est', 'low e', 'lowe r']`.

If we take 6 merges, we have `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e']` and our vocabulary elements would be `[<|endoftext|>, [...256 BYTE CHARS], st, est, ow, low, west, ne]`.

With this vocabulary and set of merges, the word `newest` would tokenize as `[ne, west]`.

2.5 Experimenting with BPE Tokenizer Training

Let's train a byte-level BPE tokenizer on the TinyStories dataset. Instructions to find / download the dataset can be found in Section 1. Before you start, we recommend taking a look at the TinyStories dataset to get a sense of what's in the data.

Parallelizing pre-tokenization You will find that a major bottleneck is the pre-tokenization step. You can speed up pre-tokenization by parallelizing your code with the built-in library `multiprocessing`. Concretely, we recommend that in parallel implementations of pre-tokenization, you chunk the corpus while ensuring your chunk boundaries occur at the beginning of a special token. You are free to use the starter code at the following link verbatim to obtain chunk boundaries, which you can then use to distribute work across your processes:

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

This chunking will always be valid, since we never want to merge across document boundaries. For the purposes of the assignment, you can always split in this way. Don't worry about the edge case of receiving a very large corpus that does not contain `<|endoftext|>`.

Removing special tokens before pre-tokenization Before running pre-tokenization with the regex pattern (using `re.finditer`), you should strip out all special tokens from your corpus (or your chunk, if using a parallel implementation). Make sure that you **split** on your special tokens, so that no merging can occur across the text they delimit. For example, if you have a corpus (or chunk) like `[Doc 1]<|endoftext|>[Doc 2]`, you should split on the special token `<|endoftext|>`, and pre-tokenize `[Doc 1]` and `[Doc 2]` separately, so that no merging can occur across the document boundary. This can be done using `re.split` with `"|"␣.join(special_tokens)` as the delimiter (with careful use of `re.escape` since | may occur in the special tokens). The test `test_train_bpe_special_tokens` will test for this.

Optimizing the merging step The naïve implementation of BPE training in the stylized example above is slow because for every merge, it iterates over all byte pairs to identify the most frequent pair. However, the only pair counts that change after each merge are those that overlap with the merged pair. Thus, BPE training speed can be improved by indexing the counts of all pairs and incrementally updating these counts, rather than explicitly iterating over each pair of bytes to count pair frequencies. You can get significant speedups with this caching procedure, though we note that the merging part of BPE training is *not* parallelizable in Python.

表示这个作为 `dict[tuple[bytes], int]`, 例如 {(l,o,w): 5 …}。注意，即使是单个字节在 Python 中也是一个 `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。

合并 我们首先查看每对连续的字节，并计算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}。对 `('es')`和 `('st')` 是并列的，所以我们取字典序较大的对， `('st')`。然后我们会合并预标记，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}。继续这样做，我们最终得到的合并序列将是 `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e','ne west', 'w i', 'wi d', 'wid est', 'low e', 'lowe r']`。

如果我们合并6个，我们将有 `['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]`。使用这个词汇表和合并集，单词 `newest` 将分解为 `[ne, west]`。

2.5 使用BPE分词器进行实验

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，我们建议您先查看TinyStories数据集，以了解数据中包含的内容。

并行化预分词您会发现一个主要的瓶颈是预分词步骤。您可以通过使用内置库 `multiprocessing`并行化您的代码来加快预分词。具体来说，我们建议在预分词的并行实现中，您在确保块边界出现在特殊标记的开头的同时对语料库进行分块。您可以使用以下链接中的启动代码原样使用来获取块边界，然后您可以使用这些边界在您的进程之间分配工作：

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

这种分块总是有效的，因为我们永远不希望跨文档边界合并。对于作业的目的，您始终可以以这种方式进行分割。不用担心收到一个非常大的语料库，而这个语料库不包含 `<|endoftext|>`的边缘情况。

在预分词之前移除特殊标记 在运行预分词的正则表达式模式（使用 `re.finditer`）之前，您应该从您的语料库（或您的块，如果使用并行实现）中移除所有特殊标记。确保您 **在** 特殊标记上 进行分割，以便不会在它们所界定的文本之间发生合并。例如，如果您有一个语料库（或块）如下 `[Doc 1]<|endoftext|>[Doc2]`,，您应该在 `<|endoftext|>`特殊标记上 <style id='19'>进行分割，并分别进行 `[Doc 1]` 和 `[Doc 2]` 预分词，以便不会在文档边界处发生合并。这可以使用 `re.split` 完成，其中 `"|"␣.join(special_tokens)` 作为分隔符（由于 `re.escape` 可能出现在特殊标记中，因此需要谨慎使用）， | 将进行测试。

优化合并步骤 上面风格化示例中 BPE 训练的朴素实现速度较慢，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配对。然而，在每次合并后，唯一会改变的配对计数是那些与合并配对重叠的计数。因此，可以通过索引所有配对的计数并逐步更新这些计数来提高 BPE 训练速度，而不是显式地遍历每一对字节来计数配对频率。使用这种缓存过程可以获得显著的加速，尽管我们注意到 BPE 训练的合并部分在 Python 中{v3}<style id='6'>不能 并行化。

Low-Resource/Downscaling Tip: Profiling

You should use profiling tools like cProfile or scalene to identify the bottlenecks in your implementation, and focus on optimizing those.

Low-Resource/Downscaling Tip: "Downscaling"

Instead of jumping to training your tokenizer on the full TinyStories dataset, we recommend you first train on a small subset of the data: a "debug dataset". For example, you could train your tokenizer on the TinyStories validation set instead, which is 22K documents instead of 2.12M. This illustrates a general strategy of downscaling whenever possible to speed up development: for example, using smaller datasets, smaller model sizes, etc. Choosing the size of the debug dataset or hyperparameter config requires careful consideration: you want your debug set to be large enough to have the same bottlenecks as the full configuration (so that the optimizations you make will generalize), but not so big that it takes forever to run.

Problem (train_bpe): BPE Tokenizer Training (15 points)

Deliverable: Write a function that, given a path to an input text file, trains a (byte-level) BPE tokenizer. Your BPE training function should handle (at least) the following input parameters:

input_path: str Path to a text file with BPE tokenizer training data.

vocab_size: int A positive integer that defines the maximum final vocabulary size (including the initial byte vocabulary, vocabulary items produced from merging, and any special tokens).

special_tokens: list[str] A list of strings to add to the vocabulary. These special tokens do not otherwise affect BPE training.

Your BPE training function should return the resulting vocabulary and merges:

vocab: dict[int, bytes] The tokenizer vocabulary, a mapping from int (token ID in the vocabulary) to bytes (token bytes).

merges: list[tuple[bytes, bytes]] A list of BPE merges produced from training. Each list item is a tuple of bytes (<token1>, <token2>), representing that <token1> was merged with <token2>. The merges should be ordered by order of creation.

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

低资源/降维提示：性能分析

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

低资源/降维提示："降维"

与其直接在完整的 TinyStories 数据集上训练您的分词器，我们建议您首先在数据的一个小子集上训练：一个"调试数据集"。例如，您可以在 TinyStories 验证集上训练您的分词器，该验证集包含 22K 个文档，而不是 2.12M 个。这说明了在可能的情况下降维的一般策略，以加快开发速度：例如，使用较小的数据集、较小的模型大小等。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，以便具有与完整配置相同的瓶颈（这样您所做的优化才会泛化），但又不至于太大，以至于运行时间过长。

问题 (train_bpe): BPE分词器训练 (15分)

交付物：编写一个函数，该函数接受一个输入文本文件的路径，并训练一个（字节级）BPE分词器。您的BPE训练函数应处理以下输入参数：

input_path: str 包含BPE分词器训练数据的文本文件路径。

vocab_size: int 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、合并产生的词汇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训练函数应返回结果词汇表和合并信息：

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

merges: list[tuple[bytes, bytes]] 从训练中产生的BPE合并列表。列表中的每个项是一个 tuple ，表示 bytes (<token1>, <token2>)与 <token1> 合并。合并应按创建顺序排序。

为了测试您的BPE训练函数与提供的测试，您首先需要实现测试适配器在 [adapters.run_train_bpe]。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。可选的（这可能需要大量的时间投入），您可以使用某些系统语言实现训练方法的关键部分，例如 C++ (考虑 cppyy) 或 Rust (使用 PyO3)。如果您这样做，请注意哪些操作需要复制而不是直接从Python内存中读取，并确保留下构建说明，或者确保它仅使用 pyproject.toml 进行构建。此外，请注意，GPT-2正则表达式在大多数正则表达式引擎中支持不佳，并且在使用正则表达式引擎的大多数情况下会非常慢。我们已经验证，Oniguruma相当快，并支持负向前瞻，但Python中的 regex 包，如果有的话，甚至更快。

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

(a) Train a byte-level BPE tokenizer on the TinyStories dataset, using a maximum vocabulary size of 10,000. Make sure to add the TinyStories <|endoftext|> special token to the vocabulary. Serialize the resulting vocabulary and merges to disk for further inspection. How many hours and memory did training take? What is the longest token in the vocabulary? Does it make sense?

Resource requirements: $\leq$ 30 minutes (no GPUs), $\leq$ 30GB RAM

Hint You should be able to get under 2 minutes for BPE training using `multiprocessing` during pretokenization and the following two facts:

(a) The <|endoftext|> token delimits documents in the data files.

(b) The <|endoftext|> token is handled as a special case before the BPE merges are applied.

Deliverable: A one-to-two sentence response.

(b) Profile your code. What part of the tokenizer training process takes the most time?

Deliverable: A one-to-two sentence response.

Next, we'll try training a byte-level BPE tokenizer on the OpenWebText dataset. As before, we recommend taking a look at the dataset to better understand its contents.

Problem (train_bpe_expts_owt): BPE Training on OpenWebText (2 points)

(a) Train a byte-level BPE tokenizer on the OpenWebText dataset, using a maximum vocabulary size of 32,000. Serialize the resulting vocabulary and merges to disk for further inspection. What is the longest token in the vocabulary? Does it make sense?

Resource requirements: $\leq$ 12 hours (no GPUs), $\leq$ 100GB RAM

Deliverable: A one-to-two sentence response.

(b) Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText.

Deliverable: A one-to-two sentence response.

2.6 BPE Tokenizer: Encoding and Decoding

In the previous part of the assignment, we implemented a function to train a BPE tokenizer on input text to obtain a tokenizer vocabulary and a list of BPE merges. Now, we will implement a BPE tokenizer that loads a provided vocabulary and list of merges and uses them to encode and decode text to/from token IDs.

2.6.1 Encoding text

The process of encoding text by BPE mirrors how we train the BPE vocabulary. There are a few major steps.

Step 1: Pre-tokenize. We first pre-tokenize the sequence and represent each pre-token as a sequence of UTF-8 bytes, just as we did in BPE training. We will be merging these bytes within each pre-token into vocabulary elements, handling each pre-token independently (no merges across pre-token boundaries).

Step 2: Apply the merges. We then take the sequence of vocabulary element merges created during BPE training, and apply it to our pre-tokens *in the same order of creation.*

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

(a) 在 TinyStories 数据集上训练一个字节级 BPE 分词器，最大词汇量为 10,000。确保将 TinyStories <|endoftext|> 特殊标记添加到词汇表中。将生成的词汇表和合并结果序列化到磁盘以供进一步检查。训练需要多少小时和内存？词汇表中最长的标记是什么？这有意义吗？

资源需求：$\leq$ 30 分钟（无 GPU）， $\leq$ 30GB RAM

提示 您应该能够在 `multiprocessing` 期间预处理和以下两个事实的帮助下，将 BPE 训练时间控制在 2 分钟以内：

(a) <|endoftext|> 标记用于在数据文件中分隔文档。

(b) 在应用 BPE 合并之前，将 <|endoftext|> 标记处理为一个特殊情况。

交付物: 一到两句话的回复。

(b) 分析你的代码。分词器训练过程中哪个部分耗时最长？**交付物**: 一到两句话的回复。

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

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

(a) 在 OpenWebText 数据集上训练一个字节级 BPE 分词器，使用最大词汇量为 32,000。将生成的词汇和合并序列化到磁盘以供进一步检查。词汇中最长的标记是什么？它有意义吗？

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

交付物: 一到两句话的回复。

(b) 比较和对比你在TinyStories和OpenWebText上训练得到的分词器。

交付物: 一到两句话的回复。

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一个BPE分词器，以获得分词器词汇表和一组BPE合并。现在，我们将实现一个BPE分词器，该分词器加载提供的词汇表和合并列表，并使用它们将文本编码和解码为标记ID。

2.6.1 编码文本

使用BPE编码文本的过程与训练BPE词汇表的方式相似。这里有几个主要步骤。

第一步：预分词。我们首先对序列进行预分词，并将每个预标记表示为UTF-8字节序列，就像我们在BPE训练中所做的那样。我们将合并这些字节，将其合并到每个预标记的词汇元素中，独立处理每个预标记（不跨越预标记边界进行合并）。

第二步：应用合并。然后，我们将BPE训练期间创建的词汇元素合并序列应用于我们的预标记以相同的创建顺序。

Example (`bpe_encoding`): BPE encoding example

For example, suppose our input string is `'the cat ate'`, our vocabulary is `{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'}`, and our learned merges are `[(b't', b'h'), (b' ', b'c'), (b' ', 'a'), (b'th', b'e'), (b' a', b't')]`. First, our pre-tokenizer would split this string into `['the', ' cat', ' ate']`. Then, we'll look at each pre-token and apply the BPE merges.
The first pre-token `'the'` is initially represented as `[b't', b'h', b'e']`. Looking at our list of merges, we identify the first applicable merge to be `(b't', b'h')`, and use that to transform the pre-token into `[b'th', b'e']`. Then, we go back to the list of merges and identify the next applicable merge to be `(b'th', b'e')`, which transforms the pre-token into `[b'the']`. Finally, looking back at the list of merges, we see that there are no more that apply to the string (since the entire pre-token has been merged into a single token), so we are done applying the BPE merges. The corresponding integer sequence is `[9]`.
Repeating this process for the remaining pre-tokens, we see that the pre-token `' cat'` is represented as `[b' c', b'a', b't']` after applying the BPE merges, which becomes the integer sequence `[7, 1, 5]`. The final pre-token `' ate'` is `[b' at', b'e']` after applying the BPE merges, which becomes the integer sequence `[10, 3]`. Thus, the final result of encoding our input string is `[9, 7, 1, 5, 10, 3]`.

Special tokens. Your tokenizer should be able to properly handle user-defined special tokens when encoding text (provided when constructing the tokenizer).

Memory considerations. Suppose we want to tokenize a large text file that we cannot fit in memory. To efficiently tokenize this large file (or any other stream of data), we need to break it up into manageable chunks and process each chunk in-turn, so that the memory complexity is constant as opposed to linear in the size of the text. In doing so, we need to make sure that a token doesn't cross chunk boundaries, else we'll get a different tokenization than the naïve method of tokenizing the entire sequence in-memory.

2.6.2 Decoding text

To decode a sequence of integer token IDs back to raw text, we can simply look up each ID's corresponding entries in the vocabulary (a byte sequence), concatenate them together, and then decode the bytes to a Unicode string. Note that input IDs are not guaranteed to map to valid Unicode strings (since a user could input any sequence of integer IDs). In the case that the input token IDs do not produce a valid Unicode string, you should replace the malformed bytes with the official Unicode replacement character `U+FFFD`.[3] The `errors` argument of `bytes.decode` controls how Unicode decoding errors are handled, and using `errors='replace'` will automatically replace malformed data with the replacement marker.

Problem (`tokenizer`): Implementing the tokenizer (15 points)

Deliverable: Implement a `Tokenizer` class that, given a vocabulary and a list of merges, encodes text into integer IDs and decodes integer IDs into text. Your tokenizer should also support user-provided special tokens (appending them to the vocabulary if they aren't already there). We recommend the following interface:

`def __init__(self, vocab, merges, special_tokens=None)` Construct a tokenizer from a given vocabulary, list of merges, and (optionally) a list of special tokens. This function should accept

[3] See en.wikipedia.org/wiki/Specials_(Unicode_block)#Replacement_character for more information about the Unicode replacement character.

示例 (`bpe_encoding`): BPE 编码示例

例如，假设我们的输入字符串是 `'the cat ate'`，我们的词汇表是 `{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'}`，并且我们学习到的合并操作是 `[(b't', b'h'), (b' ', b'c'), (b' ', 'a'), (b'th', b'e'),(b' a', b't')]`。首先，我们的预分词器会将这个字符串分割成 `['the', ' cat', ' ate']`。然后，我们将查看每个预标记并应用 BPE 合并操作。

第一个预标记 `'the'` 最初表示为 `[b't', b'h', b'e']`。查看我们的合并列表，我们确定第一个可应用的合并操作是（`b't', b'h'`），并使用它将预标记转换为 `[b'th', b'e']`。然后，我们回到合并列表，确定下一个可应用的合并操作是（`b'th', b'e'`），它将预标记转换为 `[b'the']`。最后，回顾合并列表，我们看到没有更多可应用的合并操作（因为整个预标记已经被合并成一个单独的标记），因此我们完成了 BPE 合并操作。相应的整数序列是 `[9]`。

重复此过程处理剩余的预标记，我们发现预标记 `' cat'` 在应用 BPE 合并后表示为 `[b' c', b'a', b't']`，然后成为整数序列 `[7, 1,5]`。最终预标记 `' ate'` 在应用 BPE 合并后表示为 `[b' at', b'e']`，然后成为整数序列 `[10, 3]`。因此，编码我们输入字符串的最终结果为 `[9, 7, 1, 5, 10,3]`。

特殊标记。您的分词器应该能够正确处理在编码文本时（在构建分词器时提供）用户定义的特殊标记。

内存考虑。假设我们想要标记化一个我们无法放入内存的大文本文件。为了有效地标记化这个大文件（或任何其他数据流），我们需要将其拆分成可管理的块，并逐个处理每个块，这样内存复杂度就是常数，而不是与文本大小成线性关系。在这样做的时候，我们需要确保一个标记不要跨越块边界，否则我们将得到与在内存中标记整个序列的朴素方法不同的标记化结果。

2.6.2 解码文本

要将一系列整数标记ID解码回原始文本，我们可以简单地查找词汇表中每个ID对应的条目（一个字节序列），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不一定映射到有效的Unicode字符串（因为用户可以输入任何整数ID序列）。如果输入标记ID不能生成有效的Unicode字符串，应将损坏的字节替换为官方的Unicode替换字符`U+FFFD`。[3] `errors` 参数控制如何处理Unicode解码错误，使用 `errors='replace'` 将自动将损坏的数据替换为替换标记。

问题 (`tokenizer`): 实现分词器（15分）

交付物：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，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（如果它们尚未在词汇表中，则将其添加到词汇表中）。我们建议以下接口：

`def __init__(self, vocab, merges, special_tokens=None)` 从给定的词汇表、合并列表以及（可选的）特殊标记列表构建一个分词器。此函数应接受

[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。

the following parameters:

```
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)` Class method that constructs and return a `Tokenizer` from a serialized vocabulary and list of merges (in the same format that your BPE training code output) and (optionally) a list of special tokens. This method should accept the following additional parameters:

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

`def encode(self, text: str) -> list[int]` Encode an input text into a sequence of token IDs.

`def encode_iterable(self, iterable: Iterable[str]) -> Iterator[int]` Given an iterable of strings (e.g., a Python file handle), return a generator that lazily yields token IDs. This is required for memory-efficient tokenization of large files that we cannot directly load into memory.

`def decode(self, ids: list[int]) -> str` Decode a sequence of token IDs into text.

To test your `Tokenizer` against our provided tests, you will first need to implement the test adapter at [adapters.get_tokenizer]. Then, run `uv run pytest tests/test_tokenizer.py`. Your implementation should be able to pass all tests.

2.7 Experiments

Problem (`tokenizer_experiments`): Experiments with tokenizers (4 points)

(a) Sample 10 documents from TinyStories and OpenWebText. Using your previously-trained TinyStories and OpenWebText tokenizers (10K and 32K vocabulary size, respectively), encode these sampled documents into integer IDs. What is each tokenizer's compression ratio (bytes/token)?

Deliverable: A one-to-two sentence response.

(b) What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer? Compare the compression ratio and/or qualitatively describe what happens.

Deliverable: A one-to-two sentence response.

(c) Estimate the throughput of your tokenizer (e.g., in bytes/second). How long would it take to tokenize the Pile dataset (825GB of text)?

Deliverable: A one-to-two sentence response.

(d) Using your TinyStories and OpenWebText tokenizers, encode the respective training and development datasets into a sequence of integer token IDs. We'll use this later to train our language model. We recommend serializing the token IDs as a NumPy array of datatype `uint16`. Why is `uint16` an appropriate choice?

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，用于从序列化词汇和合并列表（与您的BPE训练代码输出的格式相同）以及（可选）特殊标记列表构建并返回一个 `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]` 将输入文本编码为一系列标记ID。

`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），返回一个懒加载产生标记ID的生成器。这对于对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` 将一系列标记ID解码为文本。

为了测试您的 `Tokenizer`，您首先需要在 [adapters.get_tokenizer]实现测试适配器。 `uv run pytest tests/test_tokenizer.py`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 7 实验项目

问题 (`tokenizer_experiments`): 使用标记化器进行实验（4 分）

(a) 从 TinyStories 和 OpenWebText 中采样 10 个文档。使用您之前训练的 TinyStories 和 OpenWebText 标记化器（分别具有 10K 和 32K 的词汇量大小），将这些采样文档编码为整数 ID。每个标记化器的压缩率（字节/标记）是多少?

交付物: 一到两句话的回复。

(b) 如果您使用 TinyStories 标记化器对 OpenWebText 采样进行标记化，会发生什么？比较压缩率或定性描述发生的情况。

交付物: 一到两句话的回复。

(c) 估算您分词器的吞吐量（例如，以字节/秒为单位）。将Pile数据集（825GB的文本）分词需要多长时间?
标记Pile数据集（825GB的文本）?

交付物: 一到两句话的回复。

(d) 使用您的TinyStories和OpenWebText分词器，对相应的训练和开发文本进行编码
将开发数据集转换为一系列整数标记ID。我们稍后会用这些数据来训练我们的语言模型。我们建议将标记ID序列化为一个数据类型为 `uint16`的NumPy数组。为什么`uint16` 是一个合适的选择?

Deliverable: A one-to-two sentence response.

交付物：一到两句话的回复。

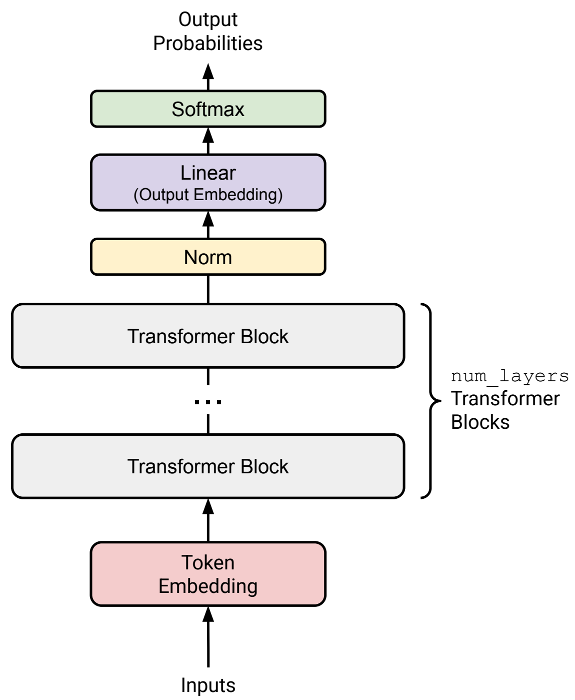

Figure 1: An overview of our Transformer language model.

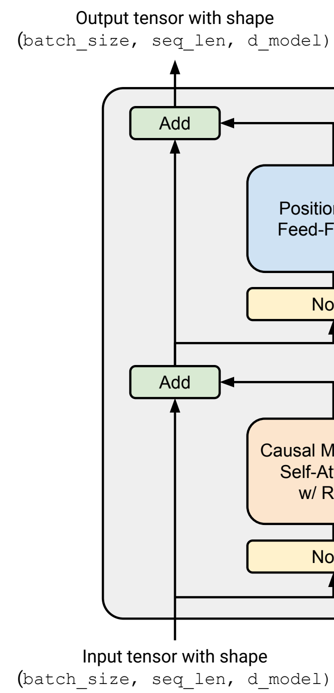

Figure 2: A pre-norm Transformer block.

3 Transformer Language Model Architecture

A language model takes as input a batched sequence of integer token IDs (i.e., `torch.Tensor` of shape `(batch_size, sequence_length)`), and returns a (batched) normalized probability distribution over the vocabulary (i.e., a PyTorch Tensor of shape `(batch_size, sequence_length, vocab_size)`), where the predicted distribution is over the next word for each input token. When training the language model, we use these next-word predictions to calculate the cross-entropy loss between the actual next word and the predicted next word. When generating text from the language model during inference, we take the predicted next-word distribution from the final time step (i.e., the last item in the sequence) to generate the next token in the sequence (e.g., by taking the token with the highest probability, sampling from the distribution, etc.), add the generated token to the input sequence, and repeat.

In this part of the assignment, you will build this Transformer language model from scratch. We will begin with a high-level description of the model before progressively detailing the individual components.

3.1 Transformer LM

Given a sequence of token IDs, the Transformer language model uses an input embedding to convert token IDs to dense vectors, passes the embedded tokens through `num_layers` Transformer blocks, and then applies a learned linear projection (the "output embedding" or "LM head") to produce the predicted next-token logits. See Figure 1 for a schematic representation.

3.1.1 Token Embeddings

In the very first step, the Transformer *embeds* the (batched) sequence of token IDs into a sequence of vectors containing information on the token identity (red blocks in Figure 1).

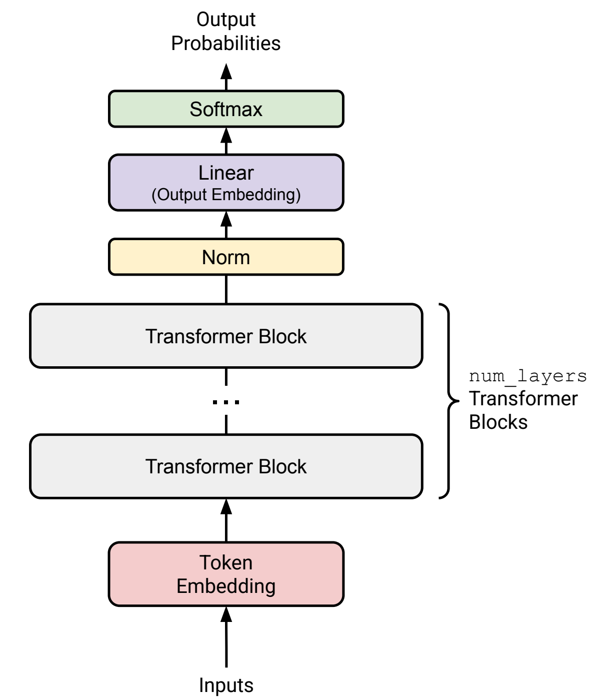

图1：我们的Transformer语言模型的概述。
模型。

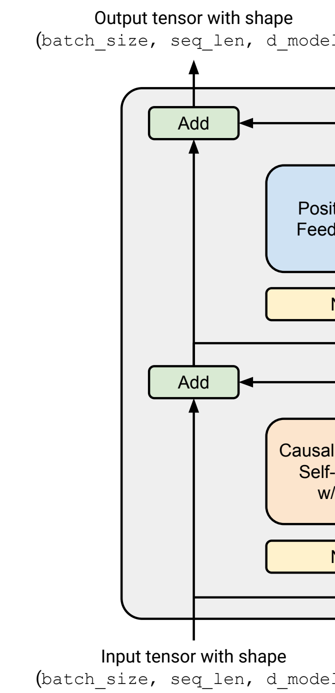

图2：预归一化Transformer块。

3 Transformer语言模型架构

语言模型将整数标记ID的批处理序列（即 `torch.Tensor` 形状为`(batch_size, sequence_length)`）作为输入，并返回一个（批处理）归一化概率分布（即形状为（`batch_size, sequence_length, vocab_size`）的PyTorch张量），其中预测分布是针对每个输入标记的下一个词。在训练语言模型时，我们使用这些下一个词预测来计算实际下一个词和预测下一个词之间的交叉熵损失。在推理期间从语言模型生成文本时，我们从最终时间步（即序列中的最后一项）的预测下一个词分布中获取预测的下一个词分布来生成序列中的下一个标记（例如，通过选择概率最高的标记，从分布中进行采样等），将生成的标记添加到输入序列中，并重复。

在本部分作业中，您将从零开始构建这个Transformer语言模型。在详细说明各个组件之前，我们将首先对模型进行高层次描述。

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转换为包含标记身份信息的向量序列（图1中的红色块），然后将嵌入的标记通过 `num_layers` Transformer块传递，并应用一个学习的线性投影（“输出嵌入”或“LM头”）来产生预测的下一个标记的对数似然。请参见图1的示意图。

3.1.1 标记嵌入

在第一步中，Transformer 嵌入（批处理）的标记ID序列到一个包含标记身份信息的向量序列中（图1中的红色块）。

More specifically, given a sequence of token IDs, the Transformer language model uses a token embedding layer to produce a sequence of vectors. Each embedding layer takes in a tensor of integers of shape `(batch_size, sequence_length)` and produces a sequence of vectors of shape `(batch_size, sequence_length, d_model)`.

3.1.2 Pre-norm Transformer Block

After embedding, the activations are processed by several identically structured neural net layers. A standard decoder-only Transformer language model consists of `num_layers` identical layers (commonly called Transformer "blocks"). Each Transformer block takes in an input of shape `(batch_size, sequence_length, d_model)` and returns an output of shape `(batch_size, sequence_length, d_model)`. Each block aggregates information across the sequence (via self-attention) and non-linearly transforms it (via the feed-forward layers).

3.2 Output Normalization and Embedding

After `num_layers` Transformer blocks, we will take the final activations and turn them into a distribution over the vocabulary.

We will implement the "pre-norm" Transformer block (detailed in §3.5), which additionally requires the use of layer normalization (detailed below) after the final Transformer block to ensure its outputs are properly scaled.

After this normalization, we will use a standard learned linear transformation to convert the output of the Transformer blocks into predicted next-token logits (see, e.g., Radford et al. [2018] equation 2).

3.3 Remark: Batching, Einsum and Efficient Computation

Throughout the Transformer, we will be performing the same computation applied to many batch-like inputs. Here are a few examples:

- **Elements of a batch**: we apply the same Transformer `forward` operation on each batch element.
- **Sequence length**: the "position-wise" operations like RMSNorm and feed-forward operate identically on each position of a sequence.
- **Attention heads**: the attention operation is batched across attention heads in a "multi-headed" attention operation.

It is useful to have an ergonomic way of performing such operations in a way that fully utilizes the GPU, and is easy to read and understand. Many PyTorch operations can take in excess "batch-like" dimensions at the start of a tensor and repeat/broadcast the operation across these dimensions efficiently.

For instance, say we are doing a position-wise, batched operation. We have a "data tensor" D of shape `(batch_size, sequence_length, d_model)`, and we would like to do a batched vector-matrix multiply against a matrix A of shape `(d_model, d_model)`. In this case, `D @ A` will do a batched matrix multiply, which is an efficient primitive in PyTorch, where the `(batch_size, sequence_length)` dimensions are batched over.

Because of this, it is helpful to assume that your functions may be given additional batch-like dimensions and to keep those dimensions at the start of the PyTorch shape. To organize tensors so they can be batched in this manner, they might need to be shaped using many steps of `view`, `reshape` and `transpose`. This can be a bit of a pain, and it often gets hard to read what the code is doing and what the shapes of your tensors are.

A more ergonomic option is to use *einsum notation* within `torch.einsum`, or rather use framework agnostic libraries like `einops` or `einx`. The two key ops are `einsum`, which can do tensor contractions with arbitrary dimensions of input tensors, and `rearrange`, which can reorder, concatenate, and split arbitrary

更具体地说，给定一个标记ID序列，Transformer语言模型使用标记嵌入层生成一个向量序列。每个嵌入层接收一个形状为 `(batch_size, sequence_length)` 的整数张量，并生成一个形状为 `(batch_size,sequence_length、 d_model)` 的向量序列。

3.1.2 预归一Transformer块

嵌入后，激活值由几个结构相同的神经网络层处理。标准的仅解码器Transformer语言模型由 `num_layers` 个相同的层（通常称为 Transformer“ 块 ” ）组成。每个 Transformer 块接收一个形状为 `(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聚合序列中的信息，并通过前馈层进行非线性转换。

3.2 输出归一化和嵌入

经过 `num_layers` 个Transformer块后，我们将最终的激活值转换为词汇表上的分布。

我们将实现“前归一化”Transformer块（详细说明见 § 3.5），该块还需要在最终的Transformer块之后使用层归一化（以下将详细说明），以确保其输出被适当缩放。

在此归一化之后，我们将使用标准的学得线性变换将Transformer块的输出转换为预测的下一个标记logits（例如，参见Radford等人 [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**：“位置感知”的操作，如RMSNorm和前馈，对序列的每个位置的操作方式相同。
- **注意力头**：在“多头”注意力操作中，注意力操作是批处理跨注意力头的。

有一个易于使用的方式来执行此类操作，充分利用GPU，并且易于阅读和理解，这很有用。许多PyTorch操作可以在张量的开头接受多余的“批次类似”维度，并有效地在这些维度上重复/广播操作。

例如，假设我们正在进行位置感知的批处理操作。我们有一个“数据张量” D 的形状 `(batch_size, sequence_length, d_model)`，并且我们想要对这个矩阵 A 的形状 `(d_model, d_model)`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，其中维度是批处理的。

因此，假设你的函数可能被赋予额外的批次类似维度，并将这些维度保留在PyTorch形状的开头，这很有帮助。为了以这种方式组织张量以便批处理，可能需要使用多个步骤的 `view`、 `reshape` 和 `transpose` 来调整形状。这可能有点麻烦，而且代码的执行内容和张量的形状往往很难以阅读。

更易用的选项是使用*einsum* 符号在 `torch.einsum` 中，或者更确切地说，使用框架无关的库如 `einops` 或 `einx`。两个关键操作是 `einsum`，它可以对任意维度的输入张量进行张量收缩，以及 `rearrange`，它可以重新排序、连接和分割任意

dimensions. It turns out almost all operations in machine learning are some combination of dimension juggling and tensor contraction with the occasional (usually pointwise) nonlinear function. This means that a lot of your code can be more readable and flexible when using einsum notation.

We **strongly** recommend learning and using einsum notation for the class. Students who have not been exposed to einsum notation before should use `einops` (docs here), and students who are already comfortable with `einops` should learn the more general `einx` (here).[4] Both packages are already installed in the environment we've supplied.

Here we give some examples of how einsum notation can be used. These are a supplement to the documentation for einops, which you should read first.

Example (`einstein_example1`): Batched matrix multiplication with `einops.einsum`

```python
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")
```

Example (`einstein_example2`): Broadcasted operations with `einops.rearrange`

We have a batch of images, and for each image we want to generate 10 dimmed versions based on some scaling factor:

```python
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] It's worth noting that while `einops` has a great amount of support, `einx` is not as battle-tested. You should feel free to fall back to using `einops` with some more plain PyTorch if you find any limitations or bugs in `einx`.

维度。结果证明，机器学习中的几乎所有操作都是维度操作和张量收缩的组合，偶尔会用到（通常是逐点）非线性函数。这意味着，当使用 einsum 符号时，你的代码可以更加易读和灵活。

我们 **强烈** 建议学习并使用 einsum 符号。之前没有接触过 einsum 符号的同学们应该使用 `einops` (文档在此)，而已经熟悉 `einops` 的同学们应该学习更通用的 `einx` (在此)。[4] 这两个包都已经安装在我们提供的环境中。

这里我们给出一些使用 einsum 符号的例子。这些是对 einops 文档的补充，你应该先阅读该文档。

Example (`einstein_example1`): Batched matrix multiplication with `einops.einsum`

```python
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")
```

Example (`einstein_example2`): Broadcasted operations with `einops.rearrange`

We have a batch of images, and for each image we want to generate 10 dimmed versions based on some scaling factor:

```python
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` 有很多支持， `einx` 还没有经过太多的实战检验。如果你在使用 `einx`时发现任何限制或错误，你可以自由地回退到使用一些更简单的 PyTorch `einops` 。

Example (`einstein_example3`): Pixel mixing with `einops.rearrange`

Suppose we have a batch of images represented as a tensor of shape (`batch, height, width, channel`), and we want to perform a linear transformation across all pixels of the image, but this transformation should happen independently for each channel. Our linear transformation is represented as a matrix B of shape (`height` × `width`, `height` × `width`).

```
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)
```

Instead, using `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
)
```

Or, if you're feeling crazy: all in one go using `einx.dot` (`einx` equivalent of `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
)
```

The first implementation here could be improved by placing comments before and after to indicate

示例 (`einstein_example3`): 使用 `einops.rearrange`假设我们有一批图像，这些图像表示为形状为（`batch, height, width,channel`）的张量，并且我们想要对所有图像的像素执行线性变换，但这个变换应该独立于每个通道发生。我们的线性变换表示为形状为（`height` × `width, height` × `width`）的矩阵 B。

```
## 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，如果你感觉疯狂：一次性使用 `einx.dot` (`einx` 相当于 `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
)
```

这里的第一种实现可以通过在前后添加注释来改进，以指示

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

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

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

3.3.1 Mathematical Notation and Memory Ordering

Many machine learning papers use row vectors in their notation, which result in representations that mesh well with the row-major memory ordering used by default in NumPy and PyTorch. With row vectors, a linear transformation looks like

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

for row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and row-vector $x \in \mathbb{R}^{1 \times d_{\text{in}}}$.

In linear algebra it's generally more common to use column vectors, where linear transformations look like

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

given a row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and column-vector $x \in \mathbb{R}^{d_{\text{in}}}$. **We will use column vectors** for mathematical notation in this assignment, as it is generally easier to follow the math this way. You should keep in mind that if you want to use plain matrix multiplication notation, you will have to apply matrices using the row vector convention, since PyTorch uses row-major memory ordering. If you use `einsum` for your matrix operations, this should be a non-issue.

3.4 Basic Building Blocks: Linear and Embedding Modules

3.4.1 Parameter Initialization

Training neural networks effectively often requires careful initialization of the model parameters—bad initializations can lead to undesirable behavior such as vanishing or exploding gradients. Pre-norm transformers are unusually robust to initializations, but they can still have a siginificant impact on training speed and convergence. Since this assignment is already long, we will save the details for assignment 3, and instead give you some approximate initializations that should work well for most cases. For now, use:

- Linear weights: $\mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{d_{\text{in}} + d_{\text{out}}}\right)$ truncated at $[-3\sigma, 3\sigma]$.
- Embedding: $\mathcal{N}\left(\mu = 0, \sigma^2 = 1\right)$ truncated at $[-3, 3]$
- RMSNorm: $\mathbb{1}$

You should use `torch.nn.init.trunc_normal_` to initialize the truncated normal weights.

3.4.2 Linear Module

Linear layers are a fundamental building block of Transformers and neural nets in general. First, you will implement your own `Linear` class that inherits from `torch.nn.Module` and performs a linear transformation:

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

Note that we do not include a bias term, following most modern LLMs.

输入和输出的形状是什么，但这很笨拙且容易出错。使用 einsum 符号，文档实现！

Einsum 符号可以处理任意输入批处理维度，但还有一个关键优势是自文档。在使用 einsum 符号的代码中，可以更清楚地了解输入和输出张量的相关形状。对于剩余的张量，可以考虑使用 Tensor 类型提示，例如使用`jaxtyping` 库（不特定于 Jax）。

我们将在作业2中更详细地讨论使用 einsum 符号的性能影响，但到目前为止，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中使用行向量，这导致与 NumPy 和 PyTorch 默认使用的行主序内存排序相匹配的表示。使用行向量，线性变换看起来像

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

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

在线性代数中，通常更常见的是使用列向量，其中线性变换看起来像

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

给定行主序 $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ 和列向量 $x \in \mathbb{R}^{d_{\text{in}}}$。**我们将在这份作业中使用列向量**进行数学符号表示，因为这样通常更容易理解数学。你应该记住，如果你想使用普通的矩阵乘法符号，你必须按照行向量约定应用矩阵，因为 PyTorch 使用行主序内存排序。如果你使用 `einsum` 进行矩阵运算，这应该不会成为问题。

3.4 基本构建块：线性与嵌入模块

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——不良的初始化可能导致不希望的行为，例如梯度消失或爆炸。预归一化变换器对初始化特别稳健，但它们仍然可能对训练速度和收敛产生重大影响。由于这份作业已经很长，我们将细节留到作业3，而在这里，我们将给出一些应该适用于大多数情况的近似初始化。现在，请使用：

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

Youshould use `torch.nn.init.trunc_normal_` 来初始化截断的正态权重 s.

3.4.2 线性模块

线性层是Transformer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，我们没有包含偏置项，这遵循了大多数现代LLMs。

Problem (linear): Implementing the linear module (1 point)

Deliverable: Implement a Linear class that inherits from torch.nn.Module and performs a linear transformation. Your implementation should follow the interface of PyTorch's built-in nn.Linear module, except for not having a bias argument or parameter. We recommend the following interface:

`def __init__(self, in_features, out_features, device=None, dtype=None)` Construct a linear transformation module. This function should accept the following parameters:

- `in_features: int` final dimension of the input
- `out_features: int` final dimension of the output
- `device: torch.device | None = None` Device to store the parameters on
- `dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, x: torch.Tensor) -> torch.Tensor` Apply the linear transformation to the input.

Make sure to:

- subclass nn.Module
- call the superclass constructor
- construct and store your parameter as W (not $W^\top$) for memory ordering reasons, putting it in an nn.Parameter
- of course, don't use nn.Linear or nn.functional.linear

For initializations, use the settings from above along with torch.nn.init.trunc_normal_ to initialize the weights.
To test your Linear module, implement the test adapter at [adapters.run_linear]. The adapter should load the given weights into your Linear module. You can use Module.load_state_dict for this purpose. Then, run uv run pytest -k test_linear.

3.4.3 Embedding Module

As discussed above, the first layer of the Transformer is an embedding layer that maps integer token IDs into a vector space of dimension d_model. We will implement a custom Embedding class that inherits from torch.nn.Module (so you should not use nn.Embedding). The forward method should select the embedding vector for each token ID by indexing into an embedding matrix of shape (vocab_size, d_model) using a torch.LongTensor of token IDs with shape (batch_size, sequence_length).

Problem (embedding): Implement the embedding module (1 point)

Deliverable: Implement the Embedding class that inherits from torch.nn.Module and performs an embedding lookup. Your implementation should follow the interface of PyTorch's built-in nn.Embedding module. We recommend the following interface:

`def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)` Construct an embedding module. This function should accept the following parameters:

- `num_embeddings: int` Size of the vocabulary

问题 (linear): 实现线性模块 (1分)

交付物: 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，除了不需要 bias 参数或参数。我们建议以下接口:

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

- `in_features: int` 输入的最终维度
- `out_features: int` 输出的最终维度
- `device: torch.device | None = None` 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` 将线性变换应用于输入。

确保做到:

- 子类 nn.Module
- 调用超类构造函数
- 构造并存储你的参数作为 W (而不是 $W^\top$)，出于内存排序的原因，将其放入一个 nn.Parameter
- 当然，不要使用 nn.Linear 或 nn.functional.linear

对于初始化，请使用上面的设置以及 torch.nn.init.trunc_normal_ 来初始化权重。

为了测试你的 Linear 模块，请在 [adapters.run_linear]实现测试适配器。该适配器应将给定的权重加载到你的 Linear 模块中。你可以使用 Module.load_state_dict 来完成这个目的。然后，运行 uv run pytest -k test_linear。

3.4.3 嵌入模块

如上所述，Transformer 的第一层是一个嵌入层，它将整数标记ID映射到维度为 d_model的向量空间。我们将实现一个自定义的 Embedding 类，该类继承自torch.nn.Module (因此你不应该使用 nn.Embedding)。forward 方法应通过索引嵌入矩阵（形状为 (vocab_size, d_model)）来选择每个标记ID的嵌入向量，该矩阵的索引由形状为 (batch_size, sequence_length)的 torch.LongTensor 组成。

问题 (embedding): 实现嵌入模块 (1 分)

交付物: 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` 词汇表大小

19

`embedding_dim: int` Dimension of the embedding vectors, i.e., d_{model}

`device: torch.device | None = None` Device to store the parameters on

`dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, token_ids: torch.Tensor) -> torch.Tensor` Lookup the embedding vectors for the given token IDs.

Make sure to:

- subclass `nn.Module`
- call the superclass constructor
- initialize your embedding matrix as a `nn.Parameter`
- store the embedding matrix with the `d_model` being the final dimension
- of course, don't use `nn.Embedding` or `nn.functional.embedding`

Again, use the settings from above for initialization, and use `torch.nn.init.trunc_normal_` to initialize the weights.
To test your implementation, implement the test adapter at `[adapters.run_embedding]`. Then, run `uv run pytest -k test_embedding`.

3.5 Pre-Norm Transformer Block

Each Transformer block has two sub-layers: a multi-head self-attention mechanism and a position-wise feed-forward network (Vaswani et al., 2017, section 3.1).

In the original Transformer paper, the model uses a residual connection around each of the two sub-layers, followed by layer normalization. This architecture is commonly known as the "post-norm" Transformer, since layer normalization is applied to the sublayer output. However, a variety of work has found that moving layer normalization from the output of each sub-layer to the input of each sub-layer (with an additional layer normalization after the final Transformer block) improves Transformer training stability [Nguyen and Salazar, 2019, Xiong et al., 2020]—see Figure 2 for a visual representation of this "pre-norm" Transformer block. The output of each Transformer block sub-layer is then added to the sub-layer input via the residual connection (Vaswani et al., 2017, section 5.4). An intuition for pre-norm is that there is a clean "residual stream" without any normalization going from the input embeddings to the final output of the Transformer, which is purported to improve gradient flow. This pre-norm Transformer is now the standard used in language models today (e.g., GPT-3, LLaMA, PaLM, etc.), so we will implement this variant. We will walk through each of the components of a pre-norm Transformer block, implementing them in sequence.

3.5.1 Root Mean Square Layer Normalization

The original Transformer implementation of Vaswani et al. [2017] uses layer normalization [Ba et al., 2016] to normalize activations. Following Touvron et al. [2023], we will use root mean square layer normalization (RMSNorm; Zhang and Sennrich, 2019, equation 4) for layer normalization. Given a vector $a \in \mathbb{R}^{d_{\text{model}}}$ of activations, RMSNorm will rescale each activation a_i as follows:

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

where $\text{RMS}(a) = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} a_i^2 + \varepsilon}$. Here, g_i is a learnable "gain" parameter (there are `d_model` such parameters total), and ε is a hyperparameter that is often fixed at 1e-5.

`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tokenID的嵌入向量

确保做到：

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

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

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

为了测试您的实现，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位置感知前馈网络（Vaswani等人，2017，第3.1节）。

在原始Transformer论文中，模型在每个子层周围使用残差连接，然后进行层归一化。这种架构通常被称为"后归一化"Transformer，因为层归一化应用于子层输出。然而，许多研究发现在将层归一化从每个子层的输出移动到每个子层的输入（在最终的Transformer块之后增加一个额外的层归一化）可以提高Transformer的训练稳定性 [Nguyen和Salazar，2019，Xiong等人 2020]——请参见图2，以了解这种"预归一"Transformer块的可视化表示。然后，每个Transformer块子层的输出通过残差连接添加到子层输入（Vaswani等人，2017，第5.4节）。对于预归一，直观的想法是从输入嵌入到Transformer的最终输出的"残差流"是干净的，没有任何归一化，这据说可以改善梯度流。现在，这种预归一Transformer是今天语言模型中使用的标准（例如，GPT-3，LLaMA，PaLM等），因此我们将实现这个变体。我们将逐一介绍预归一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等人原始的Transformer实现 [2017] 使用层归一化 [Ba等人， 2016]来归一化激活。遵循Touvron等人 [2023]，，我们将使用均方根层归一化（RMSNorm；Zhang和Sennrich，2019，方程4）进行层归一化。给定一个激活 $a \in \mathbb{R}^{d_{\text{model}}}$ 向量，RMSNorm将按以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 $\text{RMS}(a) = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} a_i^2 + \varepsilon}$。在这里，$g_i$ 是一个可学习的"增益"参数（总共有 `d_model` 个这样的参数），ε 是一个通常固定为1e-5的超参数。

You should upcast your input to `torch.float32` to prevent overflow when you square the input. Overall, your `forward` method should look like:

```
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)
```

Problem (`rmsnorm`): Root Mean Square Layer Normalization (1 point)

Deliverable: Implement RMSNorm as a `torch.nn.Module`. We recommend the following interface:

`def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)`
Construct the RMSNorm module. This function should accept the following parameters:

- `d_model: int` Hidden dimension of the model
- `eps: float = 1e-5` Epsilon value for numerical stability
- `device: torch.device | None = None` Device to store the parameters on
- `dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, x: torch.Tensor) -> torch.Tensor` Process an input tensor of shape `(batch_size, sequence_length, d_model)` and return a tensor of the same shape.

Note: Remember to upcast your input to `torch.float32` before performing the normalization (and later downcast to the original dtype), as described above.
To test your implementation, implement the test adapter at [adapters.run_rmsnorm]. Then, run `uv run pytest -k test_rmsnorm`.

您应该将输入提升为 `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 (`rmsnorm`): 根均方层归一化 (1分)

交付物: 实现 RMSNorm 作为 `torch.nn.Module`。我们建议以下接口：

`def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)`
Construct the RMSNorm module. This function should accept the following parameters:

- `d_model: int` Hidden dimension of the model
- `eps: float = 1e-5` Epsilon value for numerical stability
- `device: torch.device | None = None` Device to store the parameters on
- `dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, x: torch.Tensor) -> torch.Tensor` 处理形状为 `(batch_size, sequence_length, d_model)` 的输入张量，并返回相同形状的张量。

注意： 请记住在执行归一化之前（以及稍后降级到原始数据类型）将输入提升为 `torch.float32`，如上所述。为了测试您的实现，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 Position-Wise Feed-Forward Network

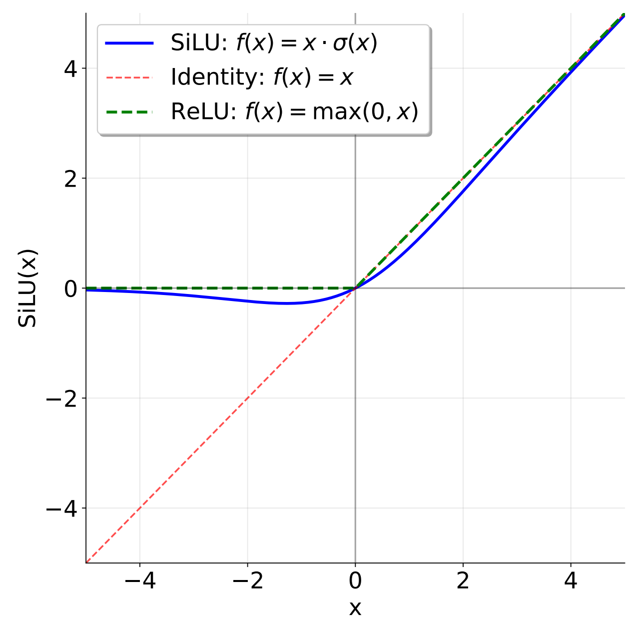

Figure 3: Comparing the SiLU (aka Swish) and ReLU activation functions.

In the original Transformer paper (section 3.3 of Vaswani et al. [2017]), the Transformer feed-forward network consists of two linear transformations with a ReLU activation ($\mathrm{ReLU}(x) = \max(0, x)$) between them. The dimensionality of the inner feed-forward layer is typically 4x the input dimensionality.

However, modern language models tend to incorporate two main changes compared to this original design: they use another activation function and employ a gating mechanism. Specifically, we will implement the "SwiGLU" activation function adopted in LLMs like Llama 3 [Grattafiori et al., 2024] and Qwen 2.5 [Yang et al., 2024], which combines the SiLU (often called Swish) activation with a gating mechanism called a Gated Linear Unit (GLU). We will also omit the bias terms sometimes used in linear layers, following most modern LLMs since PaLM [Chowdhery et al., 2022] and LLaMA [Touvron et al., 2023].

The SiLU or Swish activation function [Hendrycks and Gimpel, 2016, Elfwing et al., 2017] is defined as follows:

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

As can be seen in Figure 3, the SiLU activation function is similar to the ReLU activation function, but is smooth at zero.

Gated Linear Units (GLUs) were originally defined by Dauphin et al. [2017] as the element-wise product of a linear transformation passed through a sigmoid function and another linear transformation:

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

where $\odot$ represents element-wise multiplication. Gated Linear Units are suggested to "reduce the vanishing gradient problem for deep architectures by providing a linear path for the gradients while retaining non-linear capabilities."

Putting the SiLU/Swish and GLU together, we get the SwiGLU, which we will use for our feed-forward networks:

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

where $x \in \mathbb{R}^{d_{\text{model}}}$, $W_1, W_3 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, and canonically, $d_{\text{ff}} = \frac{8}{3} d_{\text{model}}$.

3.5.2 位置感知前馈网络

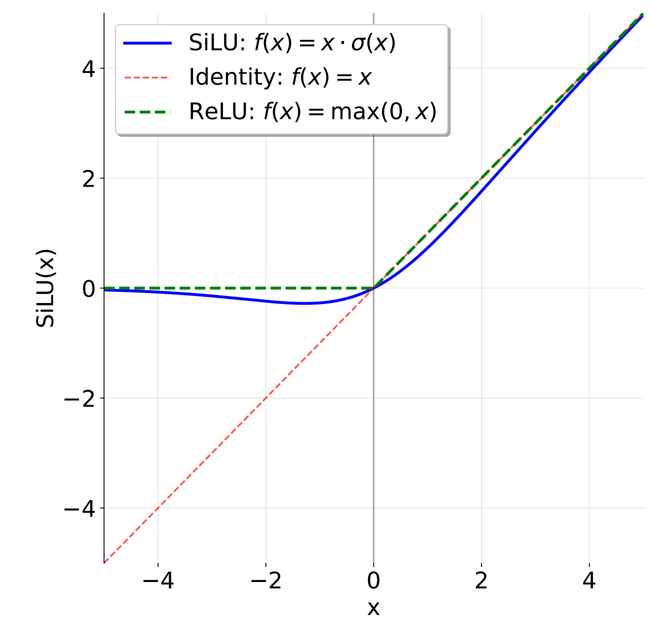

图3：比较SiLU（又称Swish）和ReLU激活函数。

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

然而，与原始设计相比，现代语言模型通常包含两个主要变化：它们使用另一种激活函数并采用门控机制。具体来说，我们将实现LLMs如Llama 3 [Grattafiori等人 2024] 和Qwen 2.5 [Yang等人 2024]，采用的"SwiGLU"激活函数，该激活函数将SiLU（通常称为Swish）激活与称为门控线性单元（GLU）的门控机制相结合。我们还将省略线性层中有时使用的偏置项，这遵循了大多数现代LLMs，如PaLM [Chowdhery等人 2022] 和LLaMA [Touvron等人 2023]。

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

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

如图 3 所示，SiLU 激活函数与 ReLU 激活函数相似，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 函数和另一个线性变换的线性变换的逐元素乘积：

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

其中 $\odot$ 表示逐元素乘法。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层架构的梯度消失问题。"

将 SiLU/Swish 和 GLU 结合起来，我们得到 SwiGLU，我们将用它来构建我们的前馈网络：

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

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

Shazeer [2020] first proposed combining the SiLU/Swish activation with GLUs and conducted experiments showing that SwiGLU outperforms baselines like ReLU and SiLU (without gating) on language modeling tasks. Later in the assignment, you will compare SwiGLU and SiLU. Though we've mentioned some heuristic arguments for these components (and the papers provide more supporting evidence), it's good to keep an empirical perspective: a now famous quote from Shazeer's paper is

> We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Problem (`positionwise_feedforward`): Implement the position-wise feed-forward network (2 points)

Deliverable: Implement the SwiGLU feed-forward network, composed of a SiLU activation function and a GLU.

Note: in this particular case, you should feel free to use `torch.sigmoid` in your implementation for numerical stability.

You should set d_{ff} to approximately $\frac{8}{3} \times d_{\text{model}}$ in your implementation, while ensuring that the dimensionality of the inner feed-forward layer is a multiple of 64 to make good use of your hardware. To test your implementation against our provided tests, you will need to implement the test adapter at `[adapters.run_swiglu]`. Then, run `uv run pytest -k test_swiglu` to test your implementation.

3.5.3 Relative Positional Embeddings

To inject positional information into the model, we will implement Rotary Position Embeddings [Su et al., 2021], often called RoPE. For a given query token $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ at token position i, we will apply a pairwise rotation matrix R^i, giving us $q'^{(i)} = R^i q^{(i)} = R^i W_q x^{(i)}$. Here, R^i will rotate pairs of embedding elements $q^{(i)}_{2k-1:2k}$ as 2d vectors by the angle $\theta_{i,k} = \frac{i}{\Theta^{(2k-1)/d}}$ for $k \in \{1, \ldots, d/2\}$ and some constant Θ. Thus, we can consider R^i to be a block-diagonal matrix of size $d \times d$, with blocks R^i_k for $k \in \{1, \ldots, d/2\}$, with

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

Thus we get the full rotation matrix

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

where 0s represent 2×2 zero matrices. While one could construct the full $d \times d$ matrix, a good solution should use the properties of this matrix to implement the transformation more efficiently. Since we only care about the relative rotation of tokens within a given sequence, we can reuse the values we compute for $\cos(\theta_{i,k})$ and $\sin(\theta_{i,k})$ across layers, and different batches. If you would like to optimize it, you may use a single RoPE module referenced by all layers, and it can have a 2d pre-computed buffer of sin and cos values created during init with `self.register_buffer(persistent=False)`, instead of a `nn.Parameter` (because we do not want to learn these fixed cosine and sine values). The exact same rotation process we did for our $q^{(i)}$ is then done for $k^{(j)}$, rotating by the corresponding R^j. Notice that this layer has no learnable parameters.

Shazeer 首次提出将 SiLU/Swish 激活与 GLU 结合，并通过实验表明 SwiGLU 在语言建模任务上优于 ReLU 和 SiLU（无门控）等基线。在作业的后续部分，你将比较 SwiGLU 和 SiLU。尽管我们提到了一些这些组件的启发式论证（并且论文提供了更多支持证据），但保持经验视角是好的：Shazeer 的论文中有一句现在广为人知的引言是

> 我们无法解释为什么这些架构似乎有效；我们将它们的成功归因于神圣的仁慈。

问题 (`positionwise_feedforward`)：实现位置感知前馈网络（2分）

交付物：实现由 SiLU 激活函数和 GLU 组成的 SwiGLU 前馈网络。

注意：在这种情况下，你可以自由地使用 `torch.sigmoid` 在你的实现中以提高数值稳定性。

你应该将 d_{ff} 设置为大约 $\frac{8}{3} \times d_{\text{model}}$ 在你的实现中，同时确保内部前馈层的维度是 64 的倍数，以充分利用你的硬件。为了测试你的实现与提供的测试，你需要在`[adapters.run_swiglu]`实现测试适配器。然后，运行 `uv run pytest -k test_swiglu` 以测试你的实现。

3.5.3 相对位置嵌入

为了将位置信息注入模型，我们将实现旋转位置嵌入 [Su 等人，2021]，通常称为 RoPE。对于给定的查询标记 $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ 在标记位置 i，我们将应用一个成对旋转矩阵 R^i，得到 $q'^{(i)} = R^i q^{(i)} = R^i W_q x^{(i)}$。在这里，$R^i$ 将旋转成对的嵌入元素 $q^{(i)}_{2k-1:2k}$ 作为 2d 向量，旋转角度为 $\theta_{i,k} = \frac{i}{\Theta^{(2k-1)/d}}$ 对于 $k \in \{1, \ldots, d/2\}$ 和某些常数 Θ。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，其中包含 R^i_k 的块，用于 $k \in \{1, \ldots, d/2\}$，

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

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 0s 代表 2×2 零矩阵。虽然可以构建完整的 $d \times d$ 矩阵，但一个好的解决方案应该利用这个矩阵的性质来更有效地实现变换。由于我们只关心给定序列中标记的相对旋转，我们可以重复使用我们在 $\cos(\theta_{i,k})$ 和 $\sin(\theta_{i,k})$ 中计算出的值，这些值跨越层和不同的批次。如果您想优化它，您可以使用一个由所有层引用的单个 RoPE 模块，并且它可以在初始化期间创建一个 2d 预计算的 sin 和 cos 值缓冲区，`self.register_buffer(persistent=False)`，而不是一个 `nn.Parameter`（因为我们不希望学习这些固定的余弦和正弦值)。然后，我们为我们的 $q^{(i)}$ 所做的完全相同的旋转过程用于 $k^{(j)}$，旋转相应的 R^j。请注意，这一层没有可学习的参数。

Problem (rope): Implement RoPE (2 points)

Deliverable: Implement a class `RotaryPositionalEmbedding` that applies RoPE to the input tensor. The following interface is recommended:

`def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)` Construct the RoPE module and create buffers if needed.

`theta: float` Θ value for the RoPE

`d_k: int` dimension of query and key vectors

`max_seq_len: int` Maximum sequence length that will be inputted

`device: torch.device | None = None` Device to store the buffer on

`def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor` Process an input tensor of shape `(..., seq_len, d_k)` and return a tensor of the same shape. Note that you should tolerate x with an arbitrary number of batch dimensions. You should assume that the token positions are a tensor of shape `(..., seq_len)` specifying the token positions of x along the sequence dimension.

You should use the token positions to slice your (possibly precomputed) cos and sin tensors along the sequence dimension.

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

3.5.4 Scaled Dot-Product Attention

We will now implement scaled dot-product attention as described in Vaswani et al. [2017] (section 3.2.1). As a preliminary step, the definition of the Attention operation will make use of softmax, an operation that takes an unnormalized vector of scores and turns it into a normalized distribution:

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

Note that $\exp(v_i)$ can become `inf` for large values (then, $\texttt{inf}/\texttt{inf} = \texttt{NaN}$). We can avoid this by noticing that the softmax operation is invariant to adding any constant c to all inputs. We can leverage this property for numerical stability—typically, we will subtract the largest entry of o_i from all elements of o_i, making the new largest entry 0. You will now implement softmax, using this trick for numerical stability.

Problem (softmax): Implement softmax (1 point)

Deliverable: Write a function to apply the softmax operation on a tensor. Your function should take two parameters: a tensor and a *dimension* i, and apply softmax to the i-th dimension of the input tensor. The output tensor should have the same shape as the input tensor, but its i-th dimension will now have a normalized probability distribution. Use the trick of subtracting the maximum value in the i-th dimension from all elements of the i-th dimension to avoid numerical stability issues.

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

We can now define the Attention operation mathematically as follows:

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

问题 (rope): 实现 RoPE (2 分)

交付物: 实现一个类 `RotaryPositionalEmbedding`，该类将 RoPE 应用到输入张量。以下是一个推荐的接口:

`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` 查询和键向量的维度

`max_seq_len: int` 将被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。请注意，您应该容忍具有任意数量批次维度的 x。您应该假设标记位置是一个形状为 `(..., seq_len)` 的张量，该张量指定了沿序列维度 x 的标记位置。

您应该使用标记位置来沿序列维度切片您的（可能已预计算的）余弦和正弦张量。

为了测试您的实现，完成 [adapters.run_rope] 并确保它通过 `uv runpytest -k test_rope`。

3.5.4 缩放点积注意力

我们现在将实现 Vaswani 等人描述的缩放点积注意力 [2017] (第 3.2.1 节)。作为初步步骤，Attentionoperation 的定义将使用 softmax，这是一个将未归一化得分向量转换为归一化分布的操作:

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

请注意，当 v_i 的值很大时，$\exp(v_i)$ 可以变成 `inf` （此时，$\texttt{inf}/\texttt{inf} = \texttt{NaN}$）。我们可以通过注意到 softmax 操作对所有输入添加任何常数 c 都是不变的来避免这种情况。我们可以利用这个属性来提高数值稳定性——通常，我们将 o_i 的最大条目从 o_i 的所有元素中减去，使新的最大条目为 0。现在，您将实现 softmax，并使用这个技巧来提高数值稳定性。

问题 (softmax): 实现 softmax (1 分)

交付物: 编写一个函数，用于对一个张量应用 softmax 操作。您的函数应接受两个参数：一个张量和一个 维度 i，并将 softmax 应用于输入张量的 i-th 维度。输出张量应与输入张量具有相同的形状，但其 i-th 维度现在将具有归一化概率分布。使用从 i-th 维度的所有元素中减去最大值的技巧来避免数值稳定性问题。

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

现在我们可以按照以下方式数学上定义注意力操作:

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

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

Masking: It is sometimes convenient to *mask* the output of an attention operation. A mask should have the shape $M \in \{\texttt{True}, \texttt{False}\}^{n \times m}$, and each row i of this boolean matrix indicates which keys the query i should attend to. Canonically (and slightly confusingly), a value of `True` at position (i, j) indicates that the query i *does* attend to the key j, and a value of `False` indicates that the query *does not* attend to the key. In other words, "information flows" at (i, j) pairs with value `True`. For example, consider a 1×3 mask matrix with entries [[`True`, `True`, `False`]]. The single query vector attends only to the first two keys.

Computationally, it will be much more efficient to use masking than to compute attention on subsequences, and we can do this by taking the pre-softmax values $\left(\frac{Q^\top K}{\sqrt{d_k}}\right)$ and adding a $-\infty$ in any entry of the mask matrix that is False.

Problem (`scaled_dot_product_attention`): Implement scaled dot-product attention (5 points)

Deliverable: Implement the scaled dot-product attention function. Your implementation should handle keys and queries of shape `(batch_size, ..., seq_len, d_k)` and values of shape `(batch_size, ..., seq_len, d_v)`, where `...` represents any number of other batch-like dimensions (if provided). The implementation should return an output with the shape `(batch_size, ..., d_v)`. See section 3.3 for a discussion on batch-like dimensions.

Your implementation should also support an optional user-provided boolean mask of shape `(seq_len, seq_len)`. The attention probabilities of positions with a mask value of `True` should collectively sum to 1, and the attention probabilities of positions with a mask value of `False` should be zero.

To test your implementation against our provided tests, you will need to implement the test adapter at `[adapters.run_scaled_dot_product_attention]`.

`uv run pytest -k test_scaled_dot_product_attention` tests your implementation on third-order input tensors, while `uv run pytest -k test_4d_scaled_dot_product_attention` tests your implementation on fourth-order input tensors.

3.5.5 Causal Multi-Head Self-Attention

We will implement multi-head self-attention as described in section 3.2.2 of Vaswani et al. [2017]. Recall that, mathematically, the operation of applying multi-head attention is defined as follows:

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

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

with Q_i, K_i, V_i being slice number $i \in \{1, \ldots, h\}$ of size d_k or d_v of the embedding dimension for Q, K, and V respectively. With Attention being the scaled dot-product attention operation defined in §3.5.4. From this we can form the multi-head *self*-attention operation:

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

Here, the learnable parameters are $W_Q \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, and $W_O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$. Since the Qs, K, and Vs are sliced in the multi-head attention operation, we can think of W_Q, W_K and W_V as being separated for each head along the output dimension. When you have this working, you should be computing the key, value, and query projections in a total of three matrix multiplies.[5]

[5] As a stretch goal, try combining the key, query, and value projections into a single weight matrix so you only need a single matrix multiply.

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

掩码： 有时对注意力操作的输出进行掩码处理是方便的。掩码应该具有形状 $M \in \{\texttt{True}, \texttt{False}\}^{n \times m}$，并且这个布尔矩阵的每一行 i 都指示查询 i 应该关注哪些键。规范地（并且稍微有些令人困惑地），位置 (i, j) 的值 `True` 表示查询 i 关注了键 j，而值 `False` 表示查询 没有关注键。换句话说，"信息流"在 (i, j) 对中与值 `True` 相匹配。例如，考虑一个具有条目 [[`True`, `True`, `False`]] 的 1×3 掩码矩阵。单个查询向量只关注前两个键。

从计算效率的角度来看，使用掩码比在子序列上计算注意力要高效得多，我们可以通过将预softmax值 $\left(\frac{Q^\top K}{\sqrt{d_k}}\right)$ 和掩码矩阵中任何为 False 的条目添加一个 $-\infty$ 来实现这一点。

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

交付物: 实现缩放点积注意力函数。您的实现应处理形状为 `(batch_size, ..., seq_len, d_k)` 的键和查询以及形状为 `(batch_size, ..., seq_len, d_v)` 的值，其中 ... 代表任何数量的其他批次类似维度（如果提供）。实现应返回形状为 `(batch_size,..., d_v)` 的输出。有关批次类似维度的讨论，请参阅第3.3节。

您的实现还应支持一个可选的用户提供的布尔掩码，形状为 `(seq_len,seq_len)`。具有掩码值为 `True` 的位置的注意力概率应总和为1，具有掩码值为 `False` 的位置的注意力概率应为零。为了测试您的实现与提供的测试，您需要实现测试适配器，位置在 `[adapters.run_scaled_dot_product_attention]`。

`uv run pytest -k test_scaled_dot_product_attention` 测试您的实现是否适用于三阶输入张量，而 `uv run pytest -k test_4d_scaled_dot_product_attention` 测试您的实现是否适用于四阶输入张量。

3.5.5 因果多头自注意力

我们将实现多头自注意力，如 Vaswani 等人 [2017] 第 3.2.2 节所述。回想一下，数学上，应用多头注意力的操作定义为以下形式:

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

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

其中 Q_i, K_i, V_i 分别是嵌入维度的切片编号 $i \in \{1, \ldots, h\}$ 的大小为 d_k 或 d_v 的 $Q, K,$ 和 V。其中 Attention 是第 3.5.4 节中定义的缩放点积注意力操作。由此我们可以形成多头 *self*-attention 操作:

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

在这里，可学习参数是 $\mathbb{R}^{d_{\text{model}} \times hd_v}$ Q K V $W_Q \in \mathbb{R}^{hd_k \times d_{\text{model}}}$ $W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}}$ $W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ $W_O \in$,,, W_Q 和 。由于 s, , 和 s 在多头注意力操作中被切片，我们可以将 ,W_K 和 W_V 视为沿着输出维度分离的每个头。当您使这工作起来时，您应该在总共三个矩阵乘法中计算键、值和查询投影。[5]

[5] 作为一个挑战目标，尝试将键、查询和值投影合并成一个单一的权重矩阵，这样你只需要进行一次矩阵乘法。

25

Causal masking. Your implementation should prevent the model from attending to future tokens in the sequence. In other words, if the model is given a token sequence $t_1, \ldots, t_n$, and we want to calculate the next-word predictions for the prefix $t_1, \ldots, t_i$ (where $i < n$), the model should *not* be able to access (attend to) the token representations at positions $t_{i+1}, \ldots, t_n$ since it will not have access to these tokens when generating text during inference (and these future tokens leak information about the identity of the true next word, trivializing the language modeling pre-training objective). For an input token sequence $t_1, \ldots, t_n$ we can naively prevent access to future tokens by running multi-head self-attention n times (for the n unique prefixes in the sequence). Instead, we'll use causal attention masking, which allows token i to attend to all positions $j \leq i$ in the sequence. You can use `torch.triu` or a broadcasted index comparison to construct this mask, and you should take advantage of the fact that your scaled dot-product attention implementation from §3.5.4 already supports attention masking.

Applying RoPE. RoPE should be applied to the query and key vectors, but not the value vectors. Also, the head dimension should be handled as a batch dimension, because in multi-head attention, attention is being applied independently for each head. This means that precisely the same RoPE rotation should be applied to the query and key vectors for each head.

Problem (`multihead_self_attention`): Implement causal multi-head self-attention (5 points)

Deliverable: Implement causal multi-head self-attention as a `torch.nn.Module`. Your implementation should accept (at least) the following parameters:

`d_model: int` Dimensionality of the Transformer block inputs.

`num_heads: int` Number of heads to use in multi-head self-attention.

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

3.6 The Full Transformer LM

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

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

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

Problem (`transformer_block`): Implement the Transformer block (3 points)

Implement the pre-norm Transformer block as described in §3.5 and illustrated in Figure 2. Your Transformer block should accept (at least) the following parameters.

`d_model: int` Dimensionality of the Transformer block inputs.

`num_heads: int` Number of heads to use in multi-head self-attention.

`d_ff: int` Dimensionality of the position-wise feed-forward inner layer.

因果掩码。您的实现应防止模型关注序列中的未来标记。换句话说，如果模型被给予一个标记序列 $t_1, \ldots,$ 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（where $i < n$），则模型应该 不能访问（关注）位置 $t_{i+1},\ldots,t_n$ 的标记表示，因为它在推理过程中生成文本时将无法访问这些标记（并且这些未来标记泄露了真正下一个词的身份信息，简化了语言建模预训练目标）。对于输入标记序列 $t_1,\ldots,t_n$，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）来天真地防止访问未来标记。相反，我们将使用因果注意力掩码，它允许标记 i 关注序列中的所有位置 $j \leq i$ 。您可以使用 `torch.triu` 或广播索引比较来构建此掩码，并且您应该利用您在 § 3.5.4 中实现的缩放点积注意力已经支持注意力掩码的事实。

应用RoPE. 应将RoPE应用于查询和键向量，但不应用于值向量。此外，头维度应作为批次维度处理，因为在多头注意力中，对每个头应用独立的注意力。这意味着应精确地应用相同的RoPE旋转到每个头的查询和键向量上。

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

交付物: 将因果多头自注意力实现为一个 `torch.nn.Module`。您的实现应接受（至少）以下参数：

`d_model: int` Transformer块输入的维度。

`num_heads: int` 在多头自注意力中使用头数。

遵循Vaswani等人 [2017], 设置 $d_k = d_v = d_{\text{model}}/h$. 要测试您的实现与提供的测试，请在 `[adapters.run_multihead_self_attention]`实现测试适配器。
`uv run pytest -k test_multihead_self_attention` 运行{v18}<style id='19'>以测试您的实现。

3.6 全Transformer LM

让我们从组装Transformer块开始（参考图2将很有帮助）。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），最后添加残差连接。

具体来说，Transformer块的前半部分（第一个‘子层’）应该实现以下一系列更新，从一个输入 x 生成一个输出 y 。

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

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

按照 § 3.5中描述并在图2中展示的方式实现预归一化Transformer块。您的Transformer块应该接受（至少）以下参数。

`d_model: int` Transformer块输入的维度。

`num_heads: int` 在多头自注意力中使用头部的数量。

`d_ff: int` 位置感知前馈内部层的维度。

To test your implementation, implement the adapter [adapters.run_transformer_block]. Then run `uv run pytest -k test_transformer_block` to test your implementation.

Deliverable: Transformer block code that passes the provided tests.

Now we put the blocks together, following the high level diagram in Figure 1. Follow our description of the embedding in Section 3.1.1, feed this into `num_layers` Transformer blocks, and then pass that into the three output layers to obtain a distribution over the vocabulary.

Problem (`transformer_lm`): Implementing the Transformer LM (3 points)

Time to put it all together! Implement the Transformer language model as described in §3.1 and illustrated in Figure 1. At minimum, your implementation should accept all the aforementioned construction parameters for the Transformer block, as well as these additional parameters:

`vocab_size: int` The size of the vocabulary, necessary for determining the dimensionality of the token embedding matrix.

`context_length: int` The maximum context length, necessary for determining the dimensionality of the position embedding matrix.

`num_layers: int` The number of Transformer blocks to use.

To test your implementation against our provided tests, you will first need to implement the test adapter at [adapters.run_transformer_lm]. Then, run `uv run pytest -k test_transformer_lm` to test your implementation.

Deliverable: A Transformer LM module that passes the above tests.

Resource accounting. It is useful to be able to understand how the various parts of the Transformer consume compute and memory. We will go through the steps to do some basic "FLOPs accounting." The *vast* majority of FLOPS in a Transformer are matrix multiplies, so our core approach is simple:

1. Write down all the matrix multiplies in a Transformer forward pass.
2. Convert each matrix multiply into FLOPs required.

For this second step, the following fact will be useful:

Rule: Given $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, the matrix-matrix product AB requires $2mnp$ FLOPs.

To see this, note that $(AB)[i,j] = A[i,:]\cdot B[:,j]$, and that this dot product requires n additions and n multiplications ($2n$ FLOPs). Then, since the matrix-matrix product AB has $m\times p$ entries, the total number of FLOPS is $(2n)(mp) = 2mnp$.

Now, before you do the next problem, it can be helpful to go through each component of your Transformer block and Transformer LM, and list out all the matrix multiplies and their associated FLOPs costs.

Problem (`transformer_accounting`): Transformer LM resource accounting (5 points)

(a) Consider GPT-2 XL, which has the following configuration:

`vocab_size` : 50,257

`context_length` : 1,024

`num_layers` : 48

`d_model` : 1,600

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

交付物: Transformer块代码，能够通过提供的测试。

现在我们将这些块组合起来，遵循图1中的高级示意图。按照第3.1.1节中对嵌入的描述，将其输入到 `num_layers` Transformer块中，然后将输出传递到三个输出层，以获得词汇表上的分布。

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

是时候将所有这些放在一起了！按照 § 3.1中描述并在图1中展示的方式实现Transformer语言模型。至少，你的实现应该接受Transformer块的所有上述构造参数，以及以下附加参数:

`vocab_size: int` 词汇表的大小，这是确定标记嵌入矩阵维度所必需的。

`context_length: int` 最大上下文长度，这是确定位置嵌入矩阵维度所必需的。

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

为了测试您的实现是否与提供的测试兼容，您首先需要在 [adapters.run_transformer_lm]实现测试适配器。然后，运行 `uv run pytest -k test_transformer_lm`以测试您的实现。

交付物：一个通过上述测试的Transformer LM模块。

资源核算。了解Transformer的各个部分如何消耗计算和内存是有用的。我们将介绍进行一些基本的“FLOPs核算”的步骤。Transformer中的大多数FLOPS都是矩阵乘法，因此我们的核心方法很简单：

1. 在Transformer的前向传递中写下所有的矩阵乘法。
2. 将每个矩阵乘法转换为所需的FLOPs。

对于这个第二步，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。

为了看到这一点，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，总的FLOPS数量是 $(2n)(mp) = 2mnp$。

现在，在你做下一个问题之前，仔细查看你的Transformer块和Transformer语言模型的每个组件，并列出所有的矩阵乘法和它们相关的FLOPs成本，这可能很有帮助。

Problem (`transformer_accounting`): Transformer语言模型资源会计（5分）　　s)

(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

Suppose we constructed our model using this configuration. How many trainable parameters would our model have? Assuming each parameter is represented using single-precision floating point, how much memory is required to just load this model?

Deliverable: A one-to-two sentence response.

(b) Identify the matrix multiplies required to complete a forward pass of our GPT-2 XL-shaped model. How many FLOPs do these matrix multiplies require in total? Assume that our input sequence has `context_length` tokens.

Deliverable: A list of matrix multiplies (with descriptions), and the total number of FLOPs required.

(c) Based on your analysis above, which parts of the model require the most FLOPs?

Deliverable: A one-to-two sentence response.

(d) Repeat your analysis with GPT-2 small (12 layers, 768 `d_model`, 12 heads), GPT-2 medium (24 layers, 1024 `d_model`, 16 heads), and GPT-2 large (36 layers, 1280 `d_model`, 20 heads). As the model size increases, which parts of the Transformer LM take up proportionally more or less of the total FLOPs?

Deliverable: For each model, provide a breakdown of model components and its associated FLOPs (as a proportion of the total FLOPs required for a forward pass). In addition, provide a one-to-two sentence description of how varying the model size changes the proportional FLOPs of each component.

(e) Take GPT-2 XL and increase the context length to 16,384. How does the total FLOPs for one forward pass change? How do the relative contribution of FLOPs of the model components change?

Deliverable: A one-to-two sentence response.

`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个可训练参数？假设每个参数都使用单精度浮点数表示，仅加载此模型需要多少内存？

交付物: 一到两句话的回复。

(b) 确定完成 GPT-2 XL形状的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` 个标记。

交付物：矩阵乘法列表（带描述），以及所需的FLOPs总数。

(c) 根据您上面的分析，模型中哪些部分需要最多的FLOPs s?

交付物：一到两句话的回复。

(d) 使用GPT-2小（12层， 768 `d_model`，12个头），GPT-2中（24层， 1024 `d_model`，16个头），和GPT-2大（36层， 1280 `d_model`，20个头）重复您的分析。随着模型大小的增加，Transformer语言模型的哪些部分在总FLOPs中占的比例更多或更少？

交付物：为每个模型提供模型组件及其相关的分解。
FLOPs（作为前向传递所需总FLOPs的比例）。此外，提供一到两句话的描述，说明模型大小变化如何改变每个组件的比例FLOPs。

(e) 使用 GPT-2 XL 并将上下文长度增加到 16,384。对于一次
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？

交付物：一到两句话的回复。

4 Training a Transformer LM

We now have the steps to preprocess the data (via tokenizer) and the model (Transformer). What remains is to build all of the code to support training. This consists of the following:

- **Loss:** we need to define the loss function (cross-entropy).
- **Optimizer:** we need to define the optimizer to minimize this loss (AdamW).
- **Training loop:** we need all the supporting infrastructure that loads data, saves checkpoints, and manages training.

4.1 Cross-entropy loss

Recall that the Transformer language model defines a distribution $p_\theta(x_{i+1} \mid x_{1:i})$ for each sequence x of length $m+1$ and $i = 1, \ldots, m$. Given a training set D consisting of sequences of length m, we define the standard cross-entropy (negative log-likelihood) loss function:

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

(Note that a single forward pass in the Transformer yields $p_\theta(x_{i+1} \mid x_{1:i})$ for *all* $i = 1, \ldots, m$.)

In particular, the Transformer computes logits $o_i \in \mathbb{R}^{\texttt{vocab\_size}}$ for each position i, which results in:[6]

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

The cross entropy loss is generally defined with respect to the vector of logits $o_i \in \mathbb{R}^{\texttt{vocab\_size}}$ and target x_{i+1}.[7]

Implementing the cross entropy loss requires some care with numerical issues, just like in the case of softmax.

> **Problem (`cross_entropy`): Implement Cross entropy**
>
> **Deliverable**: Write a function to compute the cross entropy loss, which takes in predicted logits (o_i) and targets (x_{i+1}) and computes the cross entropy $\ell_i = -\log \text{softmax}(o_i)[x_{i+1}]$. Your function should handle the following:
>
> - Subtract the largest element for numerical stability.
> - Cancel out log and exp whenever possible.
> - Handle any additional batch dimensions and return the *average* across the batch. As with section 3.3, we assume batch-like dimensions always come first, before the vocabulary size dimension.
>
> Implement `[adapters.run_cross_entropy]`, then run `uv run pytest -k test_cross_entropy` to test your implementation.

Perplexity Cross entropy suffices for training, but when we evaluate the model, we also want to report perplexity. For a sequence of length m where we suffer cross-entropy losses $\ell_1, \ldots, \ell_m$:

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

[6] Note that $o_i[k]$ refers to value at index k of the vector o_i.

[7] This corresponds to the cross entropy between the Dirac delta distribution over x_{i+1} and the predicted $\text{softmax}(o_i)$ distribution.

4 训练 TransformerLM

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以最小化这个损失（AdamW）。
- **训练循环：**我们需要所有支持的基础设施，这些基础设施负责加载数据、保存检查点和管理训练。

4.1 交叉熵损失

回想一下，Transformer语言模型为每个长度为 $m+1$ 的序列 x 定义了一个分布 $p_\theta(x_{i+1} \mid x_{1:i})$，以及 $i = 1, \ldots, m$。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：

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

(请注意，Transformer的单次前向传递为所有 $i = 1, \ldots, m$ 生成 $p_\theta(x_{i+1} \mid x_{1:i})$)。特别是，Transformer计算每个位置的logits $o_i \in \mathbb{R}^{\texttt{vocab\_size}}$ 对于每个位置 i，这导致：[6]

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

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

实现交叉熵损失需要像softmax一样注意数值问题。

> **问题 (`cross_entropy`): 实现 Cross entropy**
>
> **交付物**：编写一个函数来计算交叉熵损失，该函数接收预测的对数几率 (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内容：
>
> - 为了数值稳定性，减去最大的元素。
> - 尽可能取消对数和指数。
> - 处理任何额外的批次维度，并在批次上返回 平均值。与第 3.3 节一样，我们假设批次类似维度始终位于词汇量大小维度之前。
>
> 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 交叉熵足够用于训练，但当我们评估模型时，我们还想报告困惑度。对于一个长度为 m 的序列，我们遭受交叉熵损失 ℓ_1，…，ℓ_m：

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

[6] 注意， $o_i[k]$ of the vector . o_i 指的是索引处的值。这对应于狄拉克δ分布在 x_{i+1} 上的交叉熵与预测的[7] $\text{softmax}(o_i)$ 分布之间的交叉熵。

4.2 The SGD Optimizer

Now that we have a loss function, we will begin our exploration of optimizers. The simplest gradient-based optimizer is Stochastic Gradient Descent (SGD). We start with randomly initialized parameters θ_0. Then for each step $t = 0, \ldots, T-1$, we perform the following update:

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

where B_t is a random batch of data sampled from the dataset D, and the *learning rate* α_t and *batch size* $|B_t|$ are hyperparameters.

4.2.1 Implementing SGD in PyTorch

To implement our optimizers, we will subclass the PyTorch `torch.optim.Optimizer` class. An `Optimizer` subclass must implement two methods:

`def __init__(self, params, ...)` should initialize your optimizer. Here, `params` will be a collection of parameters to be optimized (or parameter groups, in case the user wants to use different hyperparameters, such as learning rates, for different parts of the model). Make sure to pass `params` to the `__init__` method of the base class, which will store these parameters for use in `step`. You can take additional arguments depending on the optimizer (e.g., the learning rate is a common one), and pass them to the base class constructor as a dictionary, where keys are the names (strings) you choose for these parameters.

`def step(self)` should make one update of the parameters. During the training loop, this will be called after the backward pass, so you have access to the gradients on the last batch. This method should iterate through each parameter tensor `p` and modify them *in place*, i.e. setting `p.data`, which holds the tensor associated with that parameter based on the gradient `p.grad` (if it exists), the tensor representing the gradient of the loss with respect to that parameter.

The PyTorch optimizer API has a few subtleties, so it's easier to explain it with an example. To make our example richer, we'll implement a slight variation of SGD where the learning rate decays over training, starting with an initial learning rate α and taking successively smaller steps over time:

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

Let's see how this version of SGD would be implemented as a PyTorch `Optimizer`:

```
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.
```


4.2 TheSGD优化器

现在我们有了损失函数，我们将开始探索优化器。最简单的基于梯度的优化器是随机梯度下降（SGD）。我们从随机初始化的参数 θ_0 开始。然后对于每个步骤 $t = 0, \ldots, T-1$，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 中采样的随机批次数据，并且 学习率 α_t 和 批大小$|B_t|$ 是超参数。

4.2.1 在PyTorch中实现SGD

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

`def __init__(self, params, ...)` 应该初始化你的优化器。在这里， `params` 将是一个要优化的参数集合（或者参数组，如果用户希望为模型的不同部分使用不同的超参数，例如学习率），确保将 `params` 传递给`__init__` 基类的方法，该方法将存储这些参数以供 `step` 使用。根据优化器的不同，你可以接受额外的参数（例如，学习率是一个常见的参数），并将它们作为字典传递给基类构造函数，其中键是您为这些参数选择的名称。

`def step(self)` 应更新一次参数。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的梯度。该方法应遍历每个参数张量 `p` 并就地修改它们，即设置 `p.data`，它根据梯度 `p.grad` (如果存在)以及表示相对于该参数的损失梯度的张量来保存与该参数关联的张量。

PyTorch优化器API有几个细微之处，因此用示例解释它更容易。为了使我们的示例更加丰富，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，初始学习率为 α，随着时间的推移逐步减小步长：

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

让我们看看这个版本的SGD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.
```

```python
        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
```

In `__init__`, we pass the parameters to the optimizer, as well as default hyperparameters, to the base class constructor (the parameters might come in groups, each with different hyperparameters). In case the parameters are just a single collection of `torch.nn.Parameter` objects, the base constructor will create a single group and assign it the default hyperparameters. Then, in `step`, we iterate over each parameter group, then over each parameter in that group, and apply Eq 20. Here, we keep the iteration number as a state associated with each parameter: we first read this value, use it in the gradient update, and then update it. The API specifies that the user might pass in a callable `closure` to re-compute the loss before the optimizer step. We won't need this for the optimizers we'll use, but we add it to comply with the API.

To see this working, we can use the following minimal example of a *training loop*:

```python
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.
```

This is the typical structure of a training loop: in each iteration, we will compute the loss and run a step of the optimizer. When training language models, our learnable parameters will come from the model (in PyTorch, `m.parameters()` gives us this collection). The loss will be computed over a sampled batch of data, but the basic structure of the training loop will be the same.

Problem (`learning_rate_tuning`): Tuning the learning rate (1 point)

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

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

4.3 AdamW

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


```python
        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（参数可能以组的形式出现，每组具有不同的超参数）。如果参数只是一个 `torch.nn.Parameter` 对象的单个集合，则基类构造函数将创建一个组并将其分配默认超参数。然后，在 `step` 中，我们遍历每个参数组，然后遍历该组中的每个参数，并应用公式20。在这里，我们将迭代次数作为与每个参数关联的状态保持：我们首先读取此值，将其用于梯度更新，然后更新它。API指定用户可以在优化器步骤之前传递一个可调用的 `closure` 来重新计算损失。我们不需要这个功能，但为了符合API，我们添加了它。

为了看到这个功能的工作，我们可以使用以下 训练循环 的最小示例：

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

for t in range(100):
    opt.zero_grad()  # 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，我们将计算损失并运行优化器的一步。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()` 为我们提供这个集合）。损失将在采样的数据批次上计算，但训练循环的基本结构将是相同的。

问题 (`learning_rate_tuning`)：调整学习率（1分）

正如我们将看到的，影响训练的最多的超参数之一是学习率。让我们在我们的玩具示例中实际看看这一点。用上面SGD示例中的三个其他学习率值：1e1、1e2和1e3，进行仅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？它是更快地衰减、更慢地衰减，还是发散（即，在训练过程中增加）？

交付物：用一句话或两句话描述你所观察到的行为。

4.3 AdamW

现代语言模型通常使用更复杂的优化器进行训练，而不是SGD。最近使用的优化器大多是Adam优化器 [Kingma和Ba， 2015]的衍生。我们将使用AdamW[Loshchilov和Hutter， 2019], 这在最近的工作中得到了广泛的应用。AdamW对Adam进行了修改，通过添加权重衰减（在每次迭代中，我们将参数拉向0）来改进正则化。

in a way that is decoupled from the gradient update. We will implement AdamW as described in algorithm 2 of Loshchilov and Hutter [2019].

AdamW is *stateful*: for each parameter, it keeps track of a running estimate of its first and second moments. Thus, AdamW uses additional memory in exchange for improved stability and convergence. Besides the learning rate α, AdamW has a pair of hyperparameters (β_1, β_2) that control the updates to the moment estimates, and a weight decay rate λ. Typical applications set (β_1, β_2) to $(0.9, 0.999)$, but large language models like LLaMA [Touvron et al., 2023] and GPT-3 [Brown et al., 2020] are often trained with $(0.9, 0.95)$. The algorithm can be written as follows, where ϵ is a small value (e.g., 10^{-8}) used to improve numerical stability in case we get extremely small values in v:

Algorithm 1 AdamW Optimizer

init(θ) (Initialize learnable parameters)
$m \leftarrow 0$ (Initial value of the first moment vector; same shape as θ)
$v \leftarrow 0$ (Initial value of the second moment vector; same shape as θ)
for $t = 1, \ldots, T$ **do**
  Sample batch of data B_t
  $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ (Compute the gradient of the loss at the current time step)
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$ (Update the first moment estimate)
  $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (Update the second moment estimate)
  $\alpha_t \leftarrow \alpha \frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}$ (Compute adjusted α for iteration t)
  $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\epsilon}$ (Update the parameters)
  $\theta \leftarrow \theta - \alpha\lambda\theta$ (Apply weight decay)
end for

Note that t starts at 1. You will now implement this optimizer.

Problem (adamw): Implement AdamW (2 points)

Deliverable: Implement the AdamW optimizer as a subclass of `torch.optim.Optimizer`. Your class should take the learning rate α in `__init__`, as well as the β, ϵ and λ hyperparameters. To help you keep state, the base `Optimizer` class gives you a dictionary `self.state`, which maps `nn.Parameter` objects to a dictionary that stores any information you need for that parameter (for AdamW, this would be the moment estimates). Implement [adapters.get_adamw_cls] and make sure it passes `uv run pytest -k test_adamw`.

Problem (adamwAccounting): Resource accounting for training with AdamW (2 points)

Let us compute how much memory and compute running AdamW requires. Assume we are using float32 for every tensor.

(a) How much peak memory does running AdamW require? Decompose your answer based on the memory usage of the parameters, activations, gradients, and optimizer state. Express your answer in terms of the `batch_size` and the model hyperparameters (`vocab_size`, `context_length`, `num_layers`, `d_model`, `num_heads`). Assume `d_ff` $= 4 \times$ `d_model`.

For simplicity, when calculating memory usage of activations, consider only the following components:

- Transformer block
  - RMSNorm(s)

以解耦梯度更新的方式。我们将实现Loshchilov 和 Hutter算法2中描述的AdamW [2019]。

AdamW是状态化的：对于每个参数，它都会跟踪其第一和第二矩的运行估计。因此，AdamW会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还有一对控制矩估计更新的超参数 (β_1, β_2) 和一个权重衰减率 λ。典型的应用将 (β_1, β_2) 设置为 $(0.9, 0.999)$，但像LLaMA [Touvron等人 2023] 和GPT-3 [Brown等人 2020] 这样的大型语言模型通常使用 (0.9,0.95) 进行训练。该算法可以写成以下形式，其中 ϵ 是一个小的值（例如 10^{-8}），用于在 v 中得到极小值时提高数值稳定性：

算法1 AdamW 优化器

init(θ) (初始化可学习参数)
$m \leftarrow 0$ (第一动量向量的初始值；与 θ 形状相同)$v \leftarrow 0$ (第二动量向量的初始值；与 θ 形状相同)**for** $t = 1, \ldots, T$**do**

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)$m \leftarrow \beta_1 m + (1 - \beta_1) g$ (更新第一动量估计) $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (更新第二动量估计) $\alpha_t \leftarrow \alpha \sqrt{1-(\beta_2)} {}_t$ $1 - \beta_1 {}^t$ ()(计算调整后的 α for 迭代 t) $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v+\epsilon}}$ (更新参数)

$\theta \leftarrow \theta - \alpha\lambda\theta$ (应用权重衰减)**end for**

请注意，t 从 1 开始。您现在将实现此优化器。

问题 (adamw): 实现 AdamW (2 分)

交付物: 将 AdamW 优化器实现为一个子类 `torch.optim.Optimizer`。您的类应接受学习率 α，以及 `__init__`，以及 β，ϵ 和 λ 超参数。为了帮助您保持状态，基本 `Optimizer` 类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存储的任何信息（对于 AdamW，这将是对动量估计）。实现 [adapters.get_adamw_cls] 并确保它通过 `uv run pytest -k test_adamw`。

Problem (adamwAccounting): 使用 AdamW 进行训练的资源会计 (2 分 s)

让我们计算运行 AdamW 需要多少内存和计算。假设我们为每个张量使用 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`) 来表达您的答案。假设 `d_ff` $= 4 \times$ `d_model`。

为了简单起见，在计算激活的内存使用时，只需考虑以下组件：

- Transformer块
  RMSNorm(s)

– Multi-head self-attention sublayer: QKV projections, $Q^\top K$ matrix multiply, softmax, weighted sum of values, output projection.
– Position-wise feed-forward: W_1 matrix multiply, SiLU, W_2 matrix multiply

- final RMSNorm
- output embedding
- cross-entropy on logits

Deliverable: An algebraic expression for each of parameters, activations, gradients, and optimizer state, as well as the total.

(b) Instantiate your answer for a GPT-2 XL-shaped model to get an expression that only depends on the `batch_size`. What is the maximum batch size you can use and still fit within 80GB memory?

Deliverable: An expression that looks like $a \cdot \texttt{batch\_size} + b$ for numerical values a, b, and a number representing the maximum batch size.

(c) How many FLOPs does running one step of AdamW take?

Deliverable: An algebraic expression, with a brief justification.

(d) Model FLOPs utilization (MFU) is defined as the ratio of observed throughput (tokens per second) relative to the hardware's theoretical peak FLOP throughput [Chowdhery et al., 2022]. An NVIDIA A100 GPU has a theoretical peak of 19.5 teraFLOP/s for float32 operations. Assuming you are able to get 50% MFU, how long would it take to train a GPT-2 XL for 400K steps and a batch size of 1024 on a single A100? Following Kaplan et al. [2020] and Hoffmann et al. [2022], assume that the backward pass has twice the FLOPs of the forward pass.

Deliverable: The number of days training would take, with a brief justification.

4.4 Learning rate scheduling

The value for the learning rate that leads to the quickest decrease in loss often varies during training. In training Transformers, it is typical to use a learning rate *schedule*, where we start with a bigger learning rate, making quicker updates in the beginning, and slowly decay it to a smaller value as the model trains[8] In this assignment, we will implement the cosine annealing schedule used to train LLaMA [Touvron et al., 2023].

A scheduler is simply a function that takes the current step t and other relevant parameters (such as the initial and final learning rates), and returns the learning rate to use for the gradient update at step t. The simplest schedule is the constant function, which will return the same learning rate given any t.

The cosine annealing learning rate schedule takes (i) the current iteration t, (ii) the maximum learning rate $\alpha_{\max}$, (iii) the minimum (final) learning rate $\alpha_{\min}$, (iv) the number of *warm-up* iterations T_w, and (v) the number of cosine annealing iterations T_c. The learning rate at iteration t is defined as:

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

(Cosine annealing) If $T_w \le t \le T_c$, then $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t-T_w}{T_c-T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$.

(Post-annealing) If $t > T_c$, then $\alpha_t = \alpha_{\min}$.

[8] It's sometimes common to use a schedule where the learning rate rises back up (restarts) to help get past local minima.

– 多头自注意力子层： QKV 投影， $Q^\top K$ 矩阵乘法，softmax，值的加权和，输出投影。
– 位置感知前馈： W_1 矩阵乘法，SiLU， W_2 矩阵乘法

- 最终RMSNorm
- output embedding
- cross-entropy on logits

交付物：每个参数、激活、梯度和优化器状态的代数表达式，以及总和。

(b) 为GPT-2 XL形状的模型实例化你的答案，以得到仅依赖于 `batch_size`的表达式。你可以在80GB内存内使用多大的最大batch_size?

交付物：看起来像 $a \cdot \texttt{batch\_size} + b$ 的数值 a, b，以及表示最大batch_size的数字。

(c) 运行一次AdamW步骤需要多少FLOPs?

交付物：一个代数表达式，附上简要的说明。

(d) 模型FLOPs利用率（MFU）定义为观察到的吞吐量（每秒标记数）与硬件的理论峰值FLOP吞吐量 [Chowdhery等人 2022]的比率。NVIDIA A100 GPU的理论峰值是19.5 teraFLOP/s的float32操作。假设您能够达到50%的MFU，那么在单个A100上训练GPT-2 XL 400K步和批大小为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FLOPs是前向传播的两倍。

交付物：训练所需的天数，附上简要的说明。

4.4 学习率调度

导致损失最快减少的学习率值在训练过程中通常会变化。在训练Transformer时，通常使用学习率调度，我们开始时使用较大的学习率，以便在开始时进行更快的更新，然后随着模型的训练慢慢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等人2023]使用的余弦退火调度。

调度器简单来说是一个函数，它接受当前的步数 t 和其他相关参数（例如初始和最终学习率），并返回用于梯度更新的学习率 t。最简单的调度是常函数，它将返回相同的任何 t 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 $\alpha_{\max}$，（iii）最小（最终）学习率 $\alpha_{\min}$，（iv）预热迭代次数 T_w，以及（v）余弦退火迭代次数 T_c。迭代 t 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 $T_w \le t \le T_c$，则 $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t-T_w}{T_c-T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$。

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

[8] It's sometimes c通常使用一个调度，其中学习率会回升（重启），以帮助越过局部最小值。 a.

Problem (learning_rate_schedule): Implement cosine learning rate schedule with warmup

Write a function that takes t, $\alpha_{\max}$, $\alpha_{\min}$, T_w and T_c, and returns the learning rate α_t according to the scheduler defined above. Then implement [adapters.get_lr_cosine_schedule] and make sure it passes `uv run pytest -k test_get_lr_cosine_schedule`.

4.5 Gradient clipping

During training, we can sometimes hit training examples that yield large gradients, which can destabilize training. To mitigate this, one technique often employed in practice is *gradient clipping*. The idea is to enforce a limit on the norm of the gradient after each backward pass before taking an optimizer step.

Given the gradient (for all parameters) g, we compute its ℓ_2-norm $\|g\|_2$. If this norm is less than a maximum value M, then we leave g as is; otherwise, we scale g down by a factor of $\frac{M}{\|g\|_2+\epsilon}$ (where a small ϵ, like 10^{-6}, is added for numeric stability). Note that the resulting norm will be just under M.

Problem (gradient_clipping): Implement gradient clipping (1 point)

Write a function that implements gradient clipping. Your function should take a list of parameters and a maximum ℓ_2-norm. It should modify each parameter gradient in place. Use $\epsilon = 10^{-6}$ (the PyTorch default). Then, implement the adapter [adapters.run_gradient_clipping] and make sure it passes `uv run pytest -k test_gradient_clipping`.

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

编写一个函数，它接受 t, $\alpha_{\max}$, $\alpha_{\min}$, T_w 和 T_c，并返回根据上面定义的调度器计算出的学习率 α_t。然后实现 [adapters.get_lr_cosine_schedule] 并确保它通过 `uv run pytest -k test_get_lr_cosine_schedule`。

4.5 梯度裁剪

在训练过程中，我们有时会遇到产生大梯度的训练示例，这可能导致训练不稳定。为了减轻这种情况，实践中常用的一种技术是梯度裁剪。其思路是在每次反向传播后，在采取优化器步骤之前，对梯度的范数施加一个限制。

给定梯度（对于所有参数） g，我们计算其 ℓ_2-范数 $\|g\|_2$。如果这个范数小于一个最大值 M，那么我们保持 g 不变；否则，我们将 g 按因子 $\frac{M}{\|g\|_2+\epsilon}$（其中添加了一个小的 ϵ，如 10^{-6}，以增加数值稳定性）缩小。请注意，得到的范数将略小于 M。

问题 (gradient_clipping): 实现梯度裁剪 (1分)

编写一个函数，实现梯度裁剪。您的函数应接受一个参数列表和一个最大 ℓ_2-范数。它应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] 并确保它通过 `uv run pytest -k test_gradient_clipping`。

5 Training loop

We will now finally put together the major components we've built so far: the tokenized data, the model, and the optimizer.

5.1 Data Loader

The tokenized data (e.g., that you prepared in `tokenizer_experiments`) is a single sequence of tokens $x = (x_1, \ldots, x_n)$. Even though the source data might consist of separate documents (e.g., different web pages, or source code files), a common practice is to concatenate all of those into a single sequence of tokens, adding a delimiter between them (such as the `<|endoftext|>` token).

A *data loader* turns this into a stream of *batches*, where each batch consists of B sequences of length m, paired with the corresponding next tokens, also with length m. For example, for $B = 1, m = 3$, $([x_2, x_3, x_4], [x_3, x_4, x_5])$ would be one potential batch.

Loading data in this way simplifies training for a number of reasons. First, any $1 \le i < n - m$ gives a valid training sequence, so sampling sequences are trivial. Since all training sequences have the same length, there's no need to pad input sequences, which improves hardware utilization (also by increasing batch size B). Finally, we also don't need to fully load the full dataset to sample training data, making it easy to handle large datasets that might not otherwise fit in memory.

Problem (`data_loading`): Implement data loading (2 points)

Deliverable: Write a function that takes a numpy array x (integer array with token IDs), a `batch_size`, a `context_length` and a PyTorch device string (e.g., `'cpu'` or `'cuda:0'`), and returns a pair of tensors: the sampled input sequences and the corresponding next-token targets. Both tensors should have shape (`batch_size`, `context_length`) containing token IDs, and both should be placed on the requested device. To test your implementation against our provided tests, you will first need to implement the test adapter at `[adapters.run_get_batch]`. Then, run `uv run pytest -k test_get_batch` to test your implementation.

Low-Resource/Downscaling Tip: Data loading on CPU or Apple Silicon

If you are planning to train your LM on CPU or Apple Silicon, you need to move your data to the correct device (and similarly, you should use the same device for your model later on).

If you are on CPU, you can use the `'cpu'` device string, and on Apple Silicon (M* chips), you can use the `'mps'` device string.

For more on MPS, checkout these resources:

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

What if the dataset is too big to load into memory? We can use a Unix systemcall named `mmap` which maps a file on disk to virtual memory, and lazily loads the file contents when that memory location is accessed. Thus, you can "pretend" you have the entire dataset in memory. Numpy implements this through `np.memmap` (or the flag `mmap_mode='r'` to `np.load`, if you originally saved the array with `np.save`), which will return a numpy array-like object that loads the entries on-demand as you access them. **When sampling from your dataset (i.e., a numpy array) during training, be sure load the dataset in memory-mapped mode** (via `np.memmap` or the flag `mmap_mode='r'` to `np.load`, depending on how you saved the array). Make sure you also specify a `dtype` that matches the array that you're loading. It may be helpful to explicitly verify that the memory-mapped data looks correct (e.g., doesn't contain values beyond the expected vocabulary size).

5 训练循环

我们现在将最终组装到目前为止构建的主要组件：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）是一个标记的单个序列$x = (x_1, \ldots, x_n)$。尽管源数据可能由单独的文档组成（例如，不同的网页或源代码文件），但常见的做法是将所有这些连接成一个标记的单个序列，并在它们之间添加分隔符（例如， `<|endoftext|>` 标记）。

一个 数据加载器 将其转换为一系列 批次，其中每个批次由 B 个长度为m的序列组成，配对相应的下一个标记，长度也为 m。例如，对于 $B = 1, m = 3$，$([x_2, x_3, x_4], [x_3, x_4, x_5])$ 就是一个可能的批次。

以这种方式加载数据简化了训练，原因有很多。首先，任何 $1 \le i < n - m$ 都提供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，因此不需要填充输入序列，这提高了硬件利用率（通过增加批大小B）。最后，我们也不需要完全加载完整数据集来采样训练数据，这使得处理可能不适合内存的大型数据集变得容易。

问题 (`data_loading`): 实现数据加载（2分）

交付物: 编写一个函数，该函数接受一个numpy数组 x (包含标记ID的整数数组)，一个`batch_size`，一个 `context_length` 以及一个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标记目标。这两个张量都应该具有形状(`batch_size`, `context_length`) 包含标记ID，并且都应该放置在请求的设备上。为了测试您的实现与提供的测试，您首先需要在 `[adapters.run_get_batch]`实现测试适配器。然后，运行 `uv run pytest -ktest_get_batch` 以测试您的实现。

低资源/降尺度提示：在CPU或苹果硅上加载数据

如果您计划在CPU或苹果硅上训练您的LM，您需要将数据移动到正确的设备（并且同样，您以后应该使用相同的设备来训练您的模型）。

如果您使用CPU，可以使用 `'cpu'` device字符串，而在苹果硅（M*芯片）上，您可以使用 `'mps'` device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，我们可以使用名为 `mmap` which将磁盘上的文件映射到虚拟内存的Unix系统调用，并在访问该内存位置时懒加载文件内容。因此，您可以“假装”您在内存中拥有整个数据集。NumPy通过`np.memmap` (或 `mmap_mode='r'` 标志来 `np.load`，如果您最初以 `np.save`保存数组)，返回一个类似NumPy数组对象，在您访问它们时按需加载条目。当在训练期间从您的数据集（即NumPy数组）中进行采样时，请确保以内存映射模式加载数据集(通过 `np.memmap` 或 `mmap_mode='r'` 标志来 `np.load`，具体取决于您如何保存数组)。请确保您还指定了一个 `dtype` ，它与您正在加载的数组相匹配。明确验证内存映射数据是否正确（例如，不包含超出预期词汇大小的值）可能很有帮助。

5.2 Checkpointing

In addition to loading data, we will also need to save models as we train. When running jobs, we often want to be able to resume a training run that for some reason stopped midway (e.g., due to your job timing out, machine failure, etc). Even when all goes well, we might also want to later have access to intermediate models (e.g., to study training dynamics post-hoc, take samples from models at different stages of training, etc).

A checkpoint should have all the states that we need to resume training. We of course want to be able to restore model weights at a minimum. If using a stateful optimizer (such as AdamW), we will also need to save the optimizer's state (e.g., in the case of AdamW, the moment estimates). Finally, to resume the learning rate schedule, we will need to know the iteration number we stopped at. PyTorch makes it easy to save all of these: every `nn.Module` has a `state_dict()` method that returns a dictionary with all learnable weights; we can restore these weights later with the sister method `load_state_dict()`. The same goes for any `nn.optim.Optimizer`. Finally, `torch.save(obj, dest)` can dump an object (e.g., a dictionary containing tensors in some values, but also regular Python objects like integers) to a file (path) or file-like object, which can then be loaded back into memory with `torch.load(src)`.

Problem (`checkpointing`): Implement model checkpointing (1 point)

Implement the following two functions to load and save checkpoints:

`def save_checkpoint(model, optimizer, iteration, out)` should dump all the state from the first three parameters into the file-like object `out`. You can use the `state_dict` method of both the model and the optimizer to get their relevant states and use `torch.save(obj, out)` to dump `obj` into out (PyTorch supports either a path or a file-like object here). A typical choice is to have `obj` be a dictionary, but you can use whatever format you want as long as you can load your checkpoint later.

This function expects the following parameters:

`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)` should load a checkpoint from `src` (path or file-like object), and then recover the model and optimizer states from that checkpoint. Your function should return the iteration number that was saved to the checkpoint. You can use `torch.load(src)` to recover what you saved in your `save_checkpoint` implementation, and the `load_state_dict` method in both the model and optimizers to return them to their previous states.

This function expects the following parameters:

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

`model: torch.nn.Module`

`optimizer: torch.optim.Optimizer`

Implement the `[adapters.run_save_checkpoint]` and `[adapters.run_load_checkpoint]` adapters, and make sure they pass `uv run pytest -k test_checkpointing`.

5.2 检查点

除了加载数据外，我们还需要在训练过程中保存模型。当运行作业时，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，由于您的作业超时、机器故障等）。即使一切顺利，我们也可能希望在以后能够访问中间模型（例如，为了事后研究训练动态、从不同阶段的模型中采样等）。

检查点应该包含我们恢复训练所需的所有状态。我们当然希望能够至少恢复模型权重。如果使用状态化优化器（例如AdamW），我们还需要保存优化器的状态（例如，在AdamW的情况下，动量估计）。最后，为了恢复学习率调度，我们需要知道我们停止的迭代次数。PyTorch使得保存所有这些变得很容易：每个 `nn.Module` 都有一个 `state_dict()` 方法，该方法返回一个包含所有可学习权重的字典；我们可以稍后使用姐妹方法 `load_state_dict()`来恢复这些权重。对于任何 `nn.optim.Optimizer`也是如此。最后， `torch.save(obj, dest)` 可以将一个对象（例如，包含张量等值的字典，但也包含整数等常规Python对象）写入文件（路径）或文件-like对象，然后可以使用 `torch.load(src)`将其加载回内存。

问题 (`checkpointing`): 实现模型检查点（1分）

实现以下两个函数以加载和保存检查点：

`def save_checkpoint(model, optimizer, iteration, out)` 应将前三个参数的所有状态输出到文件对象中 out。您可以使用 `state_dict` 模型和优化器的 `torch.save(obj, out)` 方法来获取它们的相关状态，并使用`obj` 将其输出到out（PyTorch支持此处使用路径或文件对象）。一个典型的选择是让 `obj` 成为一个字典，但只要您可以在以后加载检查点，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)` 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]` 并且 `[adapters.run_load_checkpoint]`适配器，并确保它们通过 `uv run pytest -k test_checkpointing`。

5.3 Training loop

Now, it's finally time to put all of the components you implemented together into your main training script. It will pay off to make it easy to start training runs with different hyperparameters (e.g., by taking them as command-line arguments), since you will be doing these many times later to study how different choices impact training.

Problem (`training_together`): Put it together (4 points)

Deliverable: Write a script that runs a training loop to train your model on user-provided input. In particular, we recommend that your training script allow for (at least) the following:

- Ability to configure and control the various model and optimizer hyperparameters.
- Memory-efficient loading of training and validation large datasets with `np.memmap`.
- Serializing checkpoints to a user-provided path.
- Periodically logging training and validation performance (e.g., to console and/or an external service like Weights and Biases).[a]

[a] wandb.ai

5.3 训练循环

现在，终于到了将您所实现的所有组件整合到主训练脚本中的时候了。使训练运行能够轻松地使用不同的超参数（例如，通过将它们作为命令行参数传递）将非常有用，因为您以后将多次这样做，以研究不同的选择如何影响训练。

问题 (`training_together`): 整合 (4 分)

交付物: 编写一个脚本，运行训练循环以在用户提供的输入上训练您的模型。特别是，我们建议您的训练脚本允许（至少）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`高效加载训练和验证大型数据集。
- 将检查点序列化到用户提供的路径。
- 定期记录训练和验证性能（例如，输出到控制台和/或外部服务如Weights and Biases）。[a]

wandb.ai

6 Generating text

Now that we can train models, the last piece we need is the ability to generate text from our model. Recall that a language model takes in a (possibly batched) integer sequence of length (`sequence_length`) and produces a matrix of size (`sequence_length` $\times$ `vocab size`), where each element of the sequence is a probability distribution predicting the next word after that position. We will now write a few functions to turn this into a sampling scheme for new sequences.

Softmax By standard convention, the language model output is the output of the final linear layer (the "logits") and so we have to turn this into a normalized probability via the *softmax* operation, which we saw earlier in Eq 10.

Decoding To generate text (decode) from our model, we will provide the model with a sequence of prefix tokens (the "prompt"), and ask it to produce a probability distribution over the vocabulary that predicts the next word in the sequence. Then, we will sample from this distribution over the vocabulary items to determine the next output token.

Concretely, one step of the decoding process should take in a sequence $x_{1...t}$ and return a token x_{t+1} via the following equation,

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

where TransformerLM is our model which takes as input a sequence of `sequence_length` and produces a matrix of size (`sequence_length` $\times$ `vocab_size`), and we take the last element of this matrix, as we are looking for the next word prediction at the t-th position.

This gives us a basic decoder by repeatedly sampling from these one-step conditionals (appending our previously-generated output token to the input of the next decoding timestep) until we generate the end-of-sequence token `<|endoftext|>` (or a user-specified maximum number of tokens to generate).

Decoder tricks We will be experimenting with small models, and small models can sometimes generate very low quality texts. Two simple decoder tricks can help fix these issues. First, in *temperature scaling* we modify our softmax with a temperature parameter τ, where the new softmax is

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

Note how setting $\tau \to 0$ makes it so that the largest element of v dominates, and the output of the softmax becomes a one-hot vector concentrated at this maximal element.

Second, another trick is *nucleus* or *top-p* sampling, where we modify the sampling distribution by truncating low-probability words. Let q be a probability distribution that we get from a (temperature-scaled) softmax of size (`vocab_size`). Nucleus sampling with hyperparameter p produces the next token according to the equation

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

where $V(p)$ is the *smallest* set of indices such that $\sum_{j \in V(p)} q_j \geq p$. You can compute this quantity easily by first sorting the probability distribution q by magnitude, and selecting the largest vocabulary elements until you reach the target level of α.

6 生成文本

现在我们能够训练模型了，我们需要的最后一部分是能够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（可能批量的）整数序列，并产生一个大小为 (`sequence_length` $\times$ `vocab size`) 的矩阵，其中序列的每个元素都是一个概率分布，预测该位置之后的下一个单词。我们现在将编写几个函数，将此转换为新的序列的采样方案。

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

解码 要从我们的模型中生成文本（解码），我们将向模型提供一个前缀标记序列（即"提示"），并要求它产生一个词汇表上的概率分布，预测序列中的下一个单词。然后，我们将从这个词汇项的概率分布中进行采样，以确定下一个输出标记。

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

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

其中 TransformerLM 是我们的模型，它接收一个序列 `sequence_length` 作为输入并生成一个大小为 (`sequence_length` $\times$ `vocab_size`) 的矩阵，然后我们取这个矩阵的最后一个元素，因为我们正在寻找第 t 个位置的下个单词预测。

这使我们通过反复从这些一步条件（将我们之前生成的输出标记添加到下一个解码时间步的输入中）中进行采样，直到生成序列结束标记 `<|endoftext|>`（或用户指定的最大标记数）来得到一个基本的解码器。

解码技巧我们将尝试使用小型模型，小型模型有时会生成非常低质量的文本。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，我们通过一个温度参数 τ 修改我们的 softmax，新的 softmax 是

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

注意设置 $\tau \to 0$ 如何使 v 的最大元素占主导地位，softmax 的输出成为一个集中在最大元素上的 one-hot 向量。

其次，另一个技巧是 核采样 或 *top-p* 采样，其中我们通过截断低概率词来修改采样分布。设 q 为从大小为 (`vocab_size`) 的 (温度缩放) softmax 中得到的概率分布。使用超参数 p 的核采样根据以下方程产生下一个标记

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

其中 $V(p)$ 是满足 $_{j \in V(p)} q_j \geq p$ 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 α 的目标级别来轻松计算这个量。

Problem (`decoding`): Decoding (3 points)

Deliverable: Implement a function to decode from your language model. We recommend that you support the following features:

- Generate completions for a user-provided prompt (i.e., take in some $x_{1...t}$ and sample a completion until you hit an <|endoftext|> token).
- Allow the user to control the maximum number of generated tokens.
- Given a desired temperature value, apply softmax temperature scaling to the predicted next-word distributions before sampling.
- Top-p sampling (Holtzman et al., 2020; also referred to as nucleus sampling), given a user-specified threshold value.

问题（`decoding`）：解码（3分）交付物：实现一个从你的语言模型中解码的函数。我们建议你支持以下功能：

- 为用户提供的提示生成补全（即，接收一些 $x_{1...t}$ andsample 补全直到你遇到一个 <|endoftext|> token）。
- 允许用户控制生成的最大标记数。
- 给定一个期望的温度值，在采样之前对预测的下一个单词分布应用softmax温度缩放。
- Top-p采样（Holtzman等人，2020；也称为核采样），给定用户指定的阈值值。

7 Experiments

Now it is time to put everything together and train (small) language models on a pretaining dataset.

7.1 How to Run Experiments and Deliverables

The best way to understand the rationale behind the architectural components of a Transformer is to actually modify it and run it yourself. There is no substitute for hands-on experience.

To this end, it's important to be able to experiment **quickly, consistently, and keep records** of what you did. To experiment quickly, we will be running many experiments on a small scale model (17M parameters) and simple dataset (TinyStories). To do things consistently, you will ablate components and vary hyperparameters in a systematic way, and to keep records we will ask you to submit a log of your experiments and learning curves associated with each experiment.

To make it possible to submit loss curves, **make sure to periodically evaluate validation losses and record both the number of steps and wallclock times**. You might find logging infrastructure such as Weights and Biases helpful.

Problem (`experiment_log`): Experiment logging (3 points)

For your training and evaluation code, create experiment tracking infrastructure that allows you to track your experiments and loss curves with respect to gradient steps and wallclock time.

Deliverable: Logging infrastructure code for your experiments and an experiment log (a document of all the things you tried) for the assignment problems below in this section.

7.2 TinyStories

We are going to start with a very simple dataset (TinyStories; Eldan and Li, 2023) where models will train quickly, and we can see some interesting behaviors. The instructions for getting this dataset is at section 1. An example of what this dataset looks like is below.

Example (`tinystories_example`): One example from TinyStories

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!

Hyperparameter tuning We will tell you some very basic hyperparameters to start with and ask you to find some settings for others that work well.

`vocab_size` 10000. Typical vocabulary sizes are in the tens to hundreds of thousands. You should vary this and see how the vocabulary and model behavior changes.

`context_length` 256. Simple datasets such as TinyStories might not need long sequence lengths, but for the later OpenWebText data, you may want to vary this. Try varying this and seeing the impact on both the per-iteration runtime and the final perplexity.

7 实验项目

Now it is time 将所有内容整合在一起，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物

要真正理解Transformer架构组件背后的原理，最好的方法就是实际修改它并亲自运行。动手经验是无法替代的。

为此，能够快速、一致地进行实验并记录你所做的一切非常重要。为了快速实验，我们将在一个小型模型（17M参数）和简单数据集（TinyStories）上运行许多实验。为了保持一致性，你需要系统地移除组件并改变超参数，为了记录，我们将要求你提交实验日志以及与每个实验相关的学习曲线。

为了能够提交损失曲线，请确保定期评估验证损失并记录步骤数和墙钟时间。你可能发现Weights and Biases等日志基础设施很有帮助。

问题 (`experiment_log`): 实验日志（3分）

为您的训练和评估代码，创建实验跟踪基础设施，以便您能够跟踪您的实验和损失曲线，与梯度步数和实际时钟时间相关。

交付物：您实验的日志基础设施代码以及实验日志（记录您尝试的所有事情的文档）本节下面的作业问题。

7.2 TinyStories

我们将从一个非常简单的数据集（TinyStories；Eldan 和 Li，2023）开始，在这个数据集上模型将快速训练，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。以下是一个此数据集外观的示例。

示例 (`tinystories_example`): TinyStories中的一个示例

从前，有一个名叫本的小男孩。本喜欢探索他周围的世界。他看到了许多令人惊叹的事物，比如陈列在商店里的美丽花瓶。有一天，本在商店里散步时，发现了一个非常特别的花瓶。当本看到它时，他感到非常惊讶！他说："哇，这个花瓶真的很神奇！我可以买下它吗？"店主微笑着说："当然可以。你可以把它带回家，向你的所有朋友展示它的神奇之处！"于是本把花瓶带回了家，他对它感到非常自豪！他叫来了他的朋友们，向他们展示了这个神奇的花瓶。所有的朋友们都觉得这个花瓶很漂亮，难以相信本这么幸运。就这样，本在商店里找到了一个神奇的花瓶！

超参数调整我们将向您介绍一些非常基本的超参数以开始，并要求您为其他一些设置找到一些效果良好的参数。

`vocab_size` 10000. 典型的词汇量大小在数万到数十万之间。你应该调整这个数值，观察词汇量和模型行为的变化。

`context_length` 256. 简单数据集，如TinyStories可能不需要很长的序列长度，但对于后来的Open WebText数据，你可能需要调整这个值。尝试调整这个值，并观察它对每次迭代的运行时间和最终困惑度的影响。

`d_model` 512. This is slightly smaller than the 768 dimensions used in many small Transformer papers, but this will make things faster.

`d_ff` 1344. This is roughly $\frac{8}{3}$`d_model` while being a multiple of 64, which is good for GPU performance.

`RoPE theta parameter` Θ 10000.

`number of layers and heads` 4 layers, 16 heads. Together, this will give about 17M non-embedding parameters which is a fairly small Transformer.

`total tokens processed` 327,680,000 (your batch size $\times$ total step count $\times$ context length should equal roughly this value).

You should do some trial and error to find good defaults for the following other hyperparameters: `learning rate`, `learning rate warmup`, `other AdamW hyperparameters` $(\beta_1, \beta_2, \epsilon)$, and `weight decay`. You can find some typical choices of such hyperparameters in Kingma and Ba [2015].

Putting it together Now you can put everything together by getting a trained BPE tokenizer, tokenizing the training dataset, and running this in the training loop that you wrote. **Important note:** If your implementation is correct and efficient, the above hyperparameters should result in a roughly 30-40 minute runtime on 1 H100 GPU. If you have runtimes that are much longer, please check and make sure your dataloading, checkpointing, or validation loss code is not bottlenecking your runtimes and that your implementation is properly batched.

Tips and tricks for debugging model architectures We highly recommend getting comfortable with your IDE's built-in debugger (e.g., VSCode/PyCharm), which will save you time compared to debugging with print statements. If you use a text editor, you can use something more like `pdb`. A few other good practices when debugging model architectures are:

- A common first step when developing any neural net architecture is to overfit to a single minibatch. If your implementation is correct, you should be able to quickly drive the training loss to near-zero.
- Set debug breakpoints in various model components, and inspect the shapes of intermediate tensors to make sure they match your expectations.
- Monitor the norms of activations, model weights, and gradients to make sure they are not exploding or vanishing.

Problem (`learning_rate`): Tune the learning rate (3 points) (4 H100 hrs)

The learning rate is one of the most important hyperparameters to tune. Taking the base model you've trained, answer the following questions:

(a) Perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).

Deliverable: Learning curves associated with multiple learning rates. Explain your hyperparameter search strategy.

Deliverable: A model with validation loss (per-token) on TinyStories of at most 1.45

`d_model` 512. 这比许多小型Transformer论文中使用的768维度略小，但会使事情更快。

`d_ff` 1344. 大约是 $\frac{8}{3}$`d_model` 的倍数，同时是64的倍数，这对GPU性能很有好处。

`RoPE theta parameter` Θ 10000.

`number of layers and heads` 4 层，16 个头。总共，这将给出大约 17M 个非嵌入参数，这是一个相当小的Transformer。

`total tokens processed` 327，680,000（你的批大小 $\times$ 总步数 $\times$ 上下文长度应大致等于这个值）。

你应该做一些试验和错误来找到以下其他超参数的良好默认值：`learning rate`， `learning rate warmup`， `other AdamW hyperparameters` （$\beta_1, \beta_2, \epsilon$），以及 `weight decay`。你可以在Kingma和Ba [2015] 中找到此类超参数的一些典型选择。

将它们组合起来 现在，您可以通过获取一个训练好的BPE分词器、对训练数据集进行分词以及运行您编写的训练循环来将所有内容组合在一起。 **重要提示：** 如果您的实现正确且高效，上述超参数应该在1个H100 GPU上运行大约30-40分钟。如果您运行时间过长，请检查并确保您的数据加载、检查点或验证损失代码没有成为瓶颈，并且您的实现已经正确批处理。

调试模型架构的技巧和窍门 我们强烈建议您熟悉您的IDE内置调试器（例如，VSCode/PyCharm），这比使用print语句调试要节省时间。如果您使用的是文本编辑器，您可以使用类似 `pdb`的东西。在调试模型架构时，还有一些其他的好习惯：

- 开发任何神经网络架构的常见第一步是过度拟合到单个小批量。如果您的实现正确，您应该能够快速将训练损失驱动到接近零。
- 在各个模型组件中设置调试断点，并检查中间张量的形状，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 (`learning_rate`): 调整学习率（3分）(4 H100 小时)

学习率是调整的最重要超参数之一。以您训练的基础模型为基础，回答以下问题：

(a) 对学习率进行超参数搜索，并报告最终损失（或如果优化器发散，则记录发散情况）。
divergence if the optimizer diverges).

交付物: 与多个学习率相关的学习曲线。解释你的超参数搜索策略。
rameter search strategy.

交付物: 在 TinyStories 上的验证损失（按词）最多为 1.45 的模型。

Low-Resource/Downscaling Tip: Train for few steps on CPU or Apple Silicon

If you are running on `cpu` or `mps`, you should instead reduce the total tokens processed count to 40,000,000, which will be sufficient to produce reasonably fluent text. You may also increase the target validation loss from 1.45 to 2.00.

Running our solution code with a tuned learning rate on an M3 Max chip and 36 GB of RAM, we use batch size × total step count × context length $= 32 \times 5000 \times 256 = 40,960,000$ tokens, which takes 1 hour and 22 minutes on `cpu` and 36 minutes on `mps`. At step 5000, we achieve a validation loss of 1.80.

Some additional tips:

- When using X training steps, we suggest adjusting the cosine learning rate decay schedule to terminate its decay (i.e., reach the minimum learning rate) at precisely step X.
- When using `mps`, do **not** use TF32 kernels, i.e., do **not** set

```
torch.set_float32_matmul_precision('high')
```

  as you might with `cuda` devices. We tried enabling TF32 kernels with `mps` (`torch` version 2.6.0) and found the backend will use silently broken kernels that cause unstable training.
- You can speed up training by JIT-compiling your model with `torch.compile`. Specifically:
  - On `cpu`, compile your model with

```
model = torch.compile(model)
```

  - On `mps`, you can somewhat optimize the backward pass using

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

    Compilation with Inductor is not supported on `mps` as of `torch` version 2.6.0.

(b) Folk wisdom is that the best learning rate is "at the edge of stability." Investigate how the point at which learning rates diverge is related to your best learning rate.

Deliverable: Learning curves of increasing learning rate which include at least one divergent run and an analysis of how this relates to convergence rates.

Now let's vary the batch size and see what happens to training. Batch sizes are important – they let us get higher efficiency from our GPUs by doing larger matrix multiplies, but is it true that we always want batch sizes to be large? Let's run some experiments to find out.

Problem (`batch_size_experiment`): Batch size variations (1 point) (2 H100 hrs)

Vary your batch size all the way from 1 to the GPU memory limit. Try at least a few batch sizes in between, including typical sizes like 64 and 128.

Deliverable: Learning curves for runs with different batch sizes. The learning rates should be optimized again if necessary.

Deliverable: A few sentences discussing of your findings on batch sizes and their impacts on training.

With your decoder in hand, we can now generate text! We will generate from the model and see how good it is. As a reference, you should get outputs that look at least as good as the example below.

低资源/降尺度提示：在CPU或苹果硅上训练几步

如果您正在使用 `cpu` 或 `mps`，您应该将总处理令牌数减少到 40,000,000，这将足以生成合理流畅的文本。您还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 的 RAM 上使用调整后的学习率运行我们的解决方案代码，我们使用批大小 × 总步数 × 上下文长度 $= 32 \times 5000 \times 256 = 40{,}960{,}000$ 令牌，这在 `cpu` 上需要 1 小时 22 分钟，在 `mps`上需要 36 分钟。在第 5000 步时，我们实现了 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，以便在精确的步数 X时终止其衰减（即达到最小学习率）。
- 当使用 `mps`时，**不要** 使用 TF32 内核，即不要 设置

```
torch.set_float32_matmul_precision('high')
```

  就像您可能在 `cuda` 设备上做的那样。我们尝试启用 TF32 内核 `mps` (`torch`版本 2.6.0)，发现后端会使用静默损坏的内核，这会导致训练不稳定。
- 您可以通过使用 `torch.compile`即时编译您的模型来加速训练。具体来说：
  - 在 `cpu`上，使用以下方式编译您的模型：

```
model = torch.compile(model)
```

  - 在 `mps`上，您可以使用以下方法在一定程度上优化反向传播：

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

    带电感器的编译在 `mps` 上不支持，截至 `torch` 版本 2.6.0。

(b) 民间智慧认为最佳学习率是“处于稳定边缘”。调查学习率发散的点与最佳学习率之间的关系。

交付物：包含至少一个发散运行的学习曲线，以及如何将其与收敛率相关联的分析。

现在让我们改变批大小，看看训练会发生什么。批大小很重要 - 它们让我们通过执行更大的矩阵乘法来从我们的 GPU 获取更高的效率，但真的是我们总是希望批大小很大吗？让我们进行一些实验来找出答案。

问题 (`batch_size_experiment`)：批大小变化（1分）（2 H100 小时）

将批大小从1调整到GPU内存限制。尝试至少几个中间批大小，包括典型的64和128等大小。

交付物：不同批大小运行的 学习曲线。如有必要，应再次优化学习率。

交付物：讨论批大小及其对训练影响的几句话。

现在我们有了解码器，我们可以生成文本了！我们将从模型生成文本，看看它的效果如何。作为参考，你应该得到至少与以下示例一样好的输出。

Example (`ts_generate_example`): Sample output from a TinyStories language model

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

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

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

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

Here is the precise problem statement and what we ask for:

Problem (`generate`): Generate text (1 point)

Using your decoder and your trained checkpoint, report the text generated by your model. You may need to manipulate decoder parameters (temperature, top-p, etc.) to get fluent outputs.

Deliverable: Text dump of at least 256 tokens of text (or until the first <|endoftext|> token), and a brief comment on the fluency of this output and at least two factors which affect how good or bad this output is.

7.3 Ablations and architecture modification

The best way to understand the Transformer is to actually modify it and see how it behaves. We will now do a few simple ablations and modifications.

Ablation 1: layer normalization It is often said that layer normalization is important for the stability of Transformer training. But perhaps we want to live dangerously. Let's remove RMSNorm from each of our Transformer blocks and see what happens.

Problem (`layer_norm_ablation`): Remove RMSNorm and train (1 point) (1 H100 hr)

Remove all of the RMSNorms from your Transformer and train. What happens at the previous optimal learning rate? Can you get stability by using a lower learning rate?

Deliverable: A learning curve for when you remove RMSNorms and train, as well as a learning curve for the best learning rate.

Deliverable: A few sentence commentary on the impact of RMSNorm.

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

从前，有一个名叫莉莉的漂亮女孩。她喜欢吃口香糖，尤其是那种大黑块的。有一天，莉莉的妈妈让她帮忙做饭。莉莉非常兴奋！她喜欢帮助她的妈妈。莉莉的妈妈为晚餐做了一大锅汤。莉莉非常开心，说："谢谢你，妈妈！我爱你。"她帮助妈妈把汤倒进一个大碗里。晚饭后，莉莉的妈妈又做了一些美味的汤。莉莉非常喜欢！她说："谢谢你，妈妈！这个汤真好吃！"她的妈妈微笑着说："我很高兴你喜欢它，莉莉。"他们完成了烹饪，继续一起做饭。故事结束。

低资源/降尺度提示：在CPU或苹果硅上生成文本

如果你使用了带有40M个标记的低资源配置，你应该看到生成的文本仍然类似于英语，但不如上面流畅。例如，我们使用在40M个标记上训练的TinyStories语言模型的样本输出如下：

从前，有一个名叫苏的小女孩。苏非常喜欢她的牙齿。这是他最好的头。有一天，苏去散步，遇到了一只瓢虫！他们成为了好朋友，一起在路上玩耍。

"嘿，波莉！我们去外面玩吧！"蒂姆说。苏看着天空，发现很难找到一种方式来跳舞闪耀。她笑了笑，同意去帮忙！"

当苏看着天空移动时，她看到了它是什么。她

以下是精确的问题陈述和我们要求的内容:

问题(`generate`): Generatetext (1分)

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

交付物: 至少256个文本标记的文本转储（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至少两个影响此输出好坏的因素。

7.3 消融和架构修改

了解Transformer的最好方法就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。让我们从我们的每个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 hr)

从您的Transformer中移除所有RMSNorm并训练。在之前的最佳学习率下会发生什么？您能否通过使用更低的学习率来获得稳定性?

交付物：当您移除RMSNorm并训练时的学习曲线，以及最佳学习率的学习曲线。

交付物：关于RMSNorm影响的几句评论。

Let's now investigate another layer normalization choice that seems arbitrary at first glance. *Pre-norm* Transformer blocks are defined as

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

This is one of the few 'consensus' modifications to the original Transformer architecture, which used a *post-norm* approach as

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

Let's revert back to the *post-norm* approach and see what happens.

Problem (`pre_norm_ablation`): Implement post-norm and train (1 point) (1 H100 hr)

Modify your pre-norm Transformer implementation into a post-norm one. Train with the post-norm model and see what happens.

Deliverable: A learning curve for a post-norm transformer, compared to the pre-norm one.

We see that layer normalization has a major impact on the behavior of the transformer, and that even the position of the layer normalization is important.

Ablation 2: position embeddings We will next investigate the impact of the position embeddings on the performance of the model. Specifically, we will compare our base model (with RoPE) with not including position embeddings at all (NoPE). It turns out that decoder-only transformers, i.e., those with a causal mask as we have implemented, can in theory infer relative or absolute position information without being provided with position embeddings explicitly [Tsai et al., 2019, Kazemnejad et al., 2023]. We will now test empirically how NoPE performs compare to RoPE.

Problem (`no_pos_emb`): Implement NoPE (1 point) (1 H100 hr)

Modify your Transformer implementation with RoPE to remove the position embedding information entirely, and see what happens.

Deliverable: A learning curve comparing the performance of RoPE and NoPE.

Ablation 3: SwiGLU vs. SiLU Next, we will follow Shazeer [2020] and test the importance of gating in the feed-forward network, by comparing the performance of SwiGLU feed-forward networks versus feed-forward networks using SiLU activations but no gated linear unit (GLU):

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

Recall that in our SwiGLU implementation, we set the dimensionality of the inner feed-forward layer to be roughly $d_{\text{ff}} = \frac{8}{3} d_{\text{model}}$ (while ensuring that $d_{\text{ff}} \mod 64 = 0$, to make use of GPU tensor cores). In your FFN_{SiLU} implementation you should set $d_{\text{ff}} = 4 \times d_{\text{model}}$, to approximately match the parameter count of the SwiGLU feed-forward network (which has three instead of two weight matrices).

Problem (`swiglu_ablation`): SwiGLU vs. SiLU (1 point) (1 H100 hr)

Deliverable: A learning curve comparing the performance of SwiGLU and SiLU feed-forward networks, with approximately matched parameter counts.

现在让我们调查另一种看起来似乎随意的层归一化选择。 前归一化Transformer块被定义为

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

这是对原始Transformer架构的少数几个‘共识’修改之一，该架构使用了一种后归一化方法作为

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

让我们回到后归一化方法，看看会发生什么。

问题 (`pre_norm_ablation`): 实现后归一化和训练（1分）(1 H100 hr)

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

交付物：与预归一化相比的后归一化Transformer的学习曲线。

我们发现层归一化对transformer的行为有重大影响，并且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（带有RoPE）与完全不包含位置嵌入的情况（NoPE）。结果表明，仅解码器的transformer，即那些具有因果掩码的transformer，在理论上可以推断相对或绝对位置信息，而无需显式地提供位置嵌入 [Tsai等人，2019年，Kazemnejad等人， 2023]。现在，我们将通过实验来测试NoPE与RoPE的性能表现。

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

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

交付物：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，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：

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

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

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

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

44

Deliverable: A few sentences discussing your findings.

Low-Resource/Downscaling Tip: Online students with limited GPU resources should test modifications on TinyStories

In the remainder of the assignment, we will move to a larger-scale, noisier web dataset (OpenWebText), experimenting with architecture modifications and (optionally) making a submission to the course leaderboard.

It takes a long time to train an LM to fluency on OpenWebText, so we suggest that online students with limited GPU access continue testing modifications on TinyStories (using validation loss as a metric to evaluate performance).

7.4 Running on OpenWebText

We will now move to a more standard pretraining dataset created from a webcrawl. A small sample of OpenWebText [Gokaslan et al., 2019] is also provided as a single text file: see section 1 for how to access this file.

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

Example (`owt_example`): One example from OWT

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." [...]

Note: You may have to re-tune your hyperparameters such as learning rate or batch size for this experiment.

Problem (`main_experiment`): Experiment on OWT (2 points) (3 H100 hrs)

Train your language model on OpenWebText with the same model architecture and total training iterations as TinyStories. How well does this model do?

Deliverable: A learning curve of your language model on OpenWebText. Describe the difference in losses from TinyStories – how should we interpret these losses?

交付物: 讨论您发现的一些句子。

低资源/降尺度提示：有限 GPU 资源的在线学生应在 TinyStories 上测试修改。

在本作业的剩余部分，我们将转向更大规模、更嘈杂的网页数据集（Open-WebText），尝试架构修改，并（可选）向课程排行榜提交作品。

在 OpenWebText 上训练一个流畅的语言模型需要很长时间，因此我们建议有限 GPU 访问的在线学生继续在 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，该数据集由网络爬取创建。OpenWebText 的一个小型样本 [Gokaslan 等 2019] 也提供了一个单独的文本文件：请参阅第1节了解如何访问此文件。

这是一个来自OpenWebText的示例。注意文本的逼真度、复杂性和多样性。您可能想浏览一下训练数据集，以了解网络爬取语料库的训练数据是什么样的。

示例 (`owt_example`)：OWT 中的一个示例

棒球预测总监哈里·帕夫利迪斯在雇佣乔纳森·贾奇时承担了风险。帕夫利迪斯知道，正如艾伦·施瓦茨在《数字游戏》中所写，"没有哪个美国文化的角落比棒球运动员的表现更精确地计数，更热情地量化。"只需几点击，你就可以发现，诺亚·辛德加德在击球过程中，他的快速球每分钟旋转超过2,100次，纳尔逊·克鲁兹在2016年成为合格击球手中游戏平均出球速度最高的球员，以及其他无数看似来自电子游戏或科幻小说的趣闻轶事。数据的海洋不断上升，赋予了棒球文化中一个越来越重要的角色：分析爱好者。

这种赋权伴随着额外的审查——不仅是对测量方法的审查，还包括对背后的人和出版物的审查。在Baseball Prospectus，Pavlidis深知量化不完美带来的反冲。他也知道该网站的捕捉指标需要重新设计，并且需要一位有学问的人——能够解决复杂统计建模问题的人——来完成这项工作。

"他让我们感到恐慌。" Harry Pavlidis

根据后者在网站赞助的棒球场活动中的写作以及他们的互动，Pavlidis有一种直觉，认为法官"明白了"。不久之后，两人开始喝酒聊天。Pavlidis的直觉得到了证实。法官适合这个职位——更好的是，他愿意接受这个职位。"我见过很多人，" Pavlidis说，"他是唯一一个敢于承担这个任务的人。" [...]

Note: You may have重新调整您的超参数，例如学习率或批大小，以进行此实验 .

问题 (`main_experiment`)：对OWT进行实验（2分）（3 H100小时）

使用与TinyStories相同的模型架构和总训练迭代次数在OpenWebText上训练您的语言模型。这个模型做得怎么样?

交付物：在OpenWebText上您的语言模型的学习曲线。描述从TinyStories到损失的区别——我们应该如何解释这些损失?

Deliverable: Generated text from OpenWebText LM, in the same format as the TinyStories outputs. How is the fluency of this text? Why is the output quality worse even though we have the same model and compute budget as TinyStories?

7.5 Your own modification + leaderboard

Congratulations on getting to this point. You're almost done! You will now try to improve upon the Transformer architecture, and see how your hyperparameters and architecture stack up against other students in the class.

Rules for the leaderboard There are no restrictions other than the following:

`Runtime` Your submission can run for at most 1.5 hours on an H100. You can enforce this by setting `--time=01:30:00` in your slurm submission script.

`Data` You may only use the OpenWebText training dataset that we provide.

Otherwise, you are free to do whatever your heart desires.

If you are looking for some ideas on what to implement, you can checkout some of these resources:

- State-of-the-art open-source LLM families, such as Llama 3 [Grattafiori et al., 2024] or Qwen 2.5 [Yang et al., 2024].
- The NanoGPT speedrun repository (`https://github.com/KellerJordan/modded-nanogpt`), where community members post many interesting modifications for "speedrunning" small-scale language model pretraining. For example, a common modification that dates back to the original Transformer paper is to tie the weights of the input and output embeddings together (see Vaswani et al. [2017] (Section 3.4) and Chowdhery et al. [2022] (Section 2)). If you do try weight tying, you may have to decrease the standard deviation of the embedding/LM head init.

You will want to test these on either a small subset of OpenWebText or on TinyStories before trying the full 1.5-hour run.

As a caveat, we do note that some of the modifications you may find working well in this leaderboard may not generalize to larger-scale pretraining. We will explore this idea further in the scaling laws unit of the course.

Problem (`leaderboard`): Leaderboard (6 points) (10 H100 hrs)

You will train a model under the leaderboard rules above with the goal of minimizing the validation loss of your language model within 1.5 H100-hour.

Deliverable: The final validation loss that was recorded, an associated learning curve that clearly shows a wallclock-time x-axis that is less than 1.5 hours and a description of what you did. We expect a leaderboard submission to beat at least the naive baseline of a 5.0 loss. Submit to the leaderboard here: https://github.com/stanford-cs336/assignment1-basics-leaderboard.

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

7.5 您自己的修改 + 排行榜

恭喜您到达这个阶段。您几乎完成了！您现在将尝试改进Transformer架构，并看看您的超参数和架构与其他学生的表现如何。

排行榜规则 除了以下规则外，没有其他限制：

`Runtime`您的提交最多可以在H100上运行1.5小时。您可以通过在您的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。

否则，你可以随心所欲地做任何你想做的事情。

如果你在寻找一些实现的想法，你可以查看以下这些资源：

- 最先进的开源LLM家族，例如Llama 3 [Grattafiori等人, 2024] 或Qwen 2.5 [Yang et al., 2024].
- NanoGPT速度跑仓库 (`https://github.com/KellerJordan/modded-nanogpt`)，社区成员在这里发布了许多关于"速度跑"小型语言模型预训练的有趣修改。例如，一个可以追溯到原始Transformer论文的常见修改是将输入和输出嵌入的权重绑定在一起（参见Vaswani等人 [2017](第3.4节)和Chowdhery等人 [2022] (第2节)。如果你尝试权重绑定，你可能需要降低嵌入/LM头初始化的标准差。

您在尝试完整的1.5小时运行之前，希望先在OpenWebText的小部分或TinyStories上测试这些内容。

作为一条注意事项，我们注意到，您在这个排行榜上可能发现的一些修改可能不适用于更大规模的预训练。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 (`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，目标是使您的语言模型在1.5 H100小时内验证损失的值最小化。

交付物: 记录的最终验证损失，一个相关的学习曲线，该曲线清楚地显示了实际运行时间x轴小于1.5小时，以及您所做的工作描述。我们期望排行榜提交的内容至少能击败5.0损失的朴素基线。请在此提交排行榜：https://github.com/stanford-cs336/assignment1-basics-leaderboard。

References

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

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

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

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

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

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

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

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

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

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.

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

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.

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

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu

参考文献

罗恩·艾尔丹和袁志立. TinyStories：语言模型可以有多小，仍然能说连贯的英语？，2023. arXiv:2305.07759。

阿隆·戈卡斯拉恩、瓦尼亚·科恩、艾莉·帕夫利克和斯蒂芬妮·泰勒克斯. OpenWebTextcorpus. `http://Skylion007.github.io/OpenWebTextCorpus`，2019。

里科·森尼奇、巴里·哈多和亚历山德拉·伯奇. 使用子词单元进行罕见词的神经机器翻译。在*ACL*，2016。

王长汉、崔炯炫和顾家涛. 基于字节级子词的神经机器翻译，2019. arXiv:1909.03341。

菲利普·盖奇. 数据压缩的新算法。 *C*用户杂志，12(2):23–38，1994年2月。ISSN 0898-9788。

亚历克·拉多夫、杰夫·吴、RewonChild、大卫·刘、达里奥·阿莫迪和伊利亚·苏茨克维尔。语言模型是无监督多任务学习器，2019年。

亚历克·拉多夫、卡蒂克·纳拉西马汉、蒂姆·萨利曼斯和伊利亚·苏茨克维尔。通过生成式预训练改进语言理解，2018年。

阿希什·瓦斯瓦尼、诺亚姆·沙泽尔、尼基·帕马尔、雅各布·乌斯克雷特、利昂·琼斯、艾丹·N·戈麦斯、卢卡斯·凯泽和伊利亚·波洛苏欣。注意力即你所需要。在*NeurIPS*会议论文集，2017年。

Toan Q. Nguyen和Julian Salazar。无需泪水的Transformer：改进自注意力。在<style id='1'>IWSWLT</style>会议论文集，2019年。注意力的规范化。在*IWSWLT*会议论文集，2019年。

熊瑞斌、杨云长、何迪、郑凯、郑舒欣、陈星、张辉怀、蓝艳艳、王立伟，刘铁岩. 关于Transformer架构中的层归一化. 载于*ICML*会议论文集，2020.

Jimmy Lei Ba，Jamie Ryan Kiros，Geoffrey E. Hinton. 层归一化，2016. arXiv:1607.06450. 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，和Guillaume Lample. Llama：开放且高效的基座语言模型，2023. arXiv:2302.13971.

张标，里科·森尼奇. 根均方层归一化. 载于*NeurIPS*会议论文集，2019.

Aaron Grattafiori，Abhimanyu Dubey，Abhinav Jauhri，Abhinav Pandey，Abhishek Kadian，Ahmad Al- Dahle，Aiesha Letman，Akhil Mathur，Alan Schelten，Alex Vaughan，Amy Yang，Angela Fan，Anirudh Goyal，Anthony Hartshorn，Aobo Yang，Archi Mitra，Archie Sravankumar，Artem Korenev，Arthur Hinsvark，Arun Rao，Aston Zhang，AurelienRodriguez，Austen Gregerson，Ava Spataru，Baptiste Roziere，Bethany Biron，Binh Tang，Bobbie Chern，Charlotte Caucheteux，Chaya Nayak，Chloe Bi，Chris Marra，Chris McConnell，Christian Keller，Christophe Touret，Chunyang Wu，Corinne Wong，Cristian Canton Ferrer，Cyrus Nikolaidis，Damien Allonsius，Daniel Song，Danielle Pintz，Danny Livshits，Danny Wyatt，David Esiobu，Dhruv Choudhary，Dhruv Mahajan，Diego Garcia-Olano，Diego Perino，Dieuwke Hupkes，Egor Lakomkin，Ehab AlBadawy，Elina Lobanova，Emily Dinan，Eric Michael Smith，Filip Radenovic，Francisco Guzmán，Frank Zhang，Gabriel Synnaeve，Gabrielle Lee，Georgia Lewis Anderson，Govind Thattai，Graeme Nail，Gregoire Mialon，Guan Pang，Guillem Cucurell，Hailey Nguyen，Hannah Korevaar，Hu Xu，Hugo Touvron，Iliyan Zarov，Imanol Arrieta Ibarra，Isabel Kloumann，Ishan Misra，Ivan Evti- mov，Jack Zhang，Jade Copet，Jaewon Lee，Jan Geffert，Jana Vranes，Jason Park，Jay Mahadeokar，Jeet Shah，Jelmer van der Linde，Jennifer Billock，Jenny Hong，Jenya Lee，Jeremy Fu，Jianfeng Chi，Jianyu

Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Moham-

黄，佳文 刘，杰 王杰超，尤若娜·比特恩，乔·斯皮萨克，金钟秀·帕克，约瑟夫·罗卡，约书亚·约翰斯顿，约书亚·萨克斯，贾俊腾·贾，卡利安·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，李凯，肯尼思·希菲尔德，凯文·斯托恩，哈利德·埃尔阿里尼，克里希蒂卡·伊耶尔，克希特兹·马利克，昆莱尼·丘，库纳尔·巴哈拉，库沙尔·拉克霍蒂亚，劳伦·兰塔拉-耶里，劳伦斯·范德马滕，陈立夫，谭亮，李珍，路易斯·马丁，洛维什·马达安，卢博·马拉，卢卡斯·布莱切尔，卢卡斯·兰扎特，卢克·德·奥利维拉，玛德琳·穆齐，马什·帕苏普莱蒂，曼纳特·辛格，马诺哈拉·帕鲁里，马尔钦·卡达斯，玛丽亚·蒂姆普乌克莉，马修·奥尔德姆，马修·里塔，玛雅·帕夫洛娃，梅拉妮·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·戈伊尔，纳尔杰斯·托拉比，尼古拉·巴什利科夫，尼古拉·博戈耶奇夫，尼拉德里·查特吉，宁张，奥利弗·杜申，奥努尔·切莱比，帕特里克·阿尔拉西，张鹏川，李鹏伟，彼得·瓦西奇，彼得·温，普拉贾瓦尔·巴哈加瓦，普拉蒂克·杜巴尔，普拉文·克里什南，普尼特·辛格·库拉，徐普新，何青，董青晓，拉加万·斯里尼瓦桑，拉贾·甘纳帕蒂，拉蒙·卡尔德勒，里卡多·席尔维拉·卡布拉尔，罗伯特·斯托尼奇，罗伯塔·莱利安，罗汉·马赫什瓦里，罗希特·吉达尔，罗希特·帕特尔，罗曼·索韦斯特雷，罗尼·波利多罗，罗什安·苏姆巴利，罗斯·泰勒，若昂·席尔瓦，侯瑞，王瑞，萨加尔·霍塞尼，萨哈纳·切纳巴萨帕，桑杰·辛格，肖恩·贝尔，赵秀云·索尼娅·金，谢尔盖·叶杜诺夫，聂少良，沙兰·纳兰，沙拉特·拉帕尔蒂，沈胜，万胜业，舒鲁蒂·博萨莱，张顺，西蒙·范登亨德，苏米亚·巴特拉，斯宾塞·惠特曼，斯坦·索特拉，斯蒂芬·科洛特，苏钦·古鲁拉甘，西德尼·博罗丁斯基，塔玛尔·赫曼，塔拉·福勒，塔雷克·沙沙，托马斯·乔治乌，托马斯·西亚洛姆，托比亚斯·斯佩克巴赫，托多·米哈伊洛夫，童晓，乌吉瓦尔·卡尔恩，维达努杰·戈斯瓦米，维布霍尔·古普塔，维尼什·拉马南，维克托·凯尔凯兹，文森特·贡盖，维吉妮·多，维什·沃热蒂，维托尔·阿尔比罗，弗拉丹·佩特罗维奇，朱伟伟，熊文汉，付文银，惠特尼·米尔斯，马丁·马丁内特，王晓东，王晓芳，谭晓青·艾伦·谭，夏德，夏峰，谢学伟，亚埃勒·戈尔茨哈格，亚什什·高尔，亚斯明娜·巴瓦伊，尹文，宋一闻，张宇晨，李悦，毛云宁，扎卡里·德尔皮埃·库德尔特，郑岩，陈正兴，佐伊·帕帕基波斯，阿迪蒂亚·辛格，阿尤什·斯里瓦斯塔瓦，阿布哈·贾因，亚当·凯尔西，亚当·沙因菲尔德，阿迪提亚·甘吉迪，阿道夫·维多利亚，阿胡瓦·戈尔茨坦德，阿贾伊·梅农，阿贾伊·夏尔马，亚历克斯·博森贝格，亚历克谢·贝夫斯克，艾莉·费因斯坦，阿曼达·卡莱特，阿米特·桑加尼，阿莫斯·特奥，阿南姆·尤纳斯，安德烈·卢普，安德烈斯·阿尔瓦拉多，安德鲁·卡普莱斯，安德鲁·顾，安德鲁·胡，安德鲁·波尔顿，安德鲁·瑞恩，安尼克·拉姆查丹尼，安妮·东，安妮·弗朗哥，阿努杰·戈伊尔，阿帕拉吉塔·萨拉夫，阿卡班杜·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，贝奥·詹姆斯，本·莫拉，本杰明·莱昂哈迪，伯尼·黄，贝丝·洛伊德，贝托·德·帕奥拉，巴哈格维·帕兰贾佩，毕晓岚，刘，王长汉，金昌奎，周超，胡切斯特，朱静香，蔡驰，蒂达尔·廷达尔，克里斯托夫·费希滕霍弗，高珊，达蒙·西文，丹娜·比蒂，丹尼尔·克莱默，丹尼尔·李，大卫·阿金斯，大卫·徐，达维德·泰斯特吉尼，黛莉亚·大卫，戴维·帕里希，戴维·帕里

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 Ratanchandani, 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, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy 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, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, 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 Sankaranarayana 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, and Noah Fiedel. PaLM: Scaling language modeling with pathways, 2022. arXiv:2204.02311.

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

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.

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

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

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

Mad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Si nghal, 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 Ratanchandani, 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, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy 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, and Zhiyu Ma. Llama 3 模型群，2024. URL https://arxiv.org/abs/2407.21783.

杨安，杨宝松，张贝辰，惠斌元，郑波，余博文，李成远，刘大恒，黄飞，魏浩然，林欢，杨建，涂建宏，张建伟，杨建新，杨家熙，周景仁，林俊阳，邓康，陆克明，包克勤，杨克新，余磊，李梅，薛明锋，张培，朱琴，门瑞，林润基，李天豪，夏廷宇，任兴章，任轩城，范阳，苏阳，张一长，万宇，刘雨琼，崔泽宇，张振儒，邱子涵。Qwen2.5技术报告。*arXiv*预印本 *arXiv:2412.15115*，2024年。

Chowdhery Aakanksha，Narang Sharan，Devlin Jacob，Bosma Maarten，Mishra Gaurav，Roberts Adam，Barham Paul，Chung Hyung Won，Sutton Charles，Gehrmann Sebastian，Schuh Parker，Shi Kensen，Tsvyashchenko Sasha，Maynez Joshua，Rao Abhishek，Barnes Parker，Tay Yi，Shazeer Noam，Prabhakaran Vinodkumar，Reif Emily，Du Nan，Hutchinson Ben，Pope Reiner，Bradbury James，Austin Jacob，Isard Michael，Gur-Ari Guy，Yin Pengcheng，Duke Toju，Levskaya Anselm，Ghemawat Sanjay，Dev Sunipa，Michalewski Henryk，Garcia Xavier，Misra Vedant，Robinson Kevin，Fedus Liam，Zhou Denny，Ippolito Daphne，刘大卫，Lim Hyeontaek，Zoph Barret，Spiridonov Alexander，Sepassi Ryan，Dohan David，Agra wal Shivani，Omernick Mark，Dai Andrew M.，Pillai Thanumalayan Sankaranarayana，Pellat Marie，Lewkowycz Aitor，Moreira Erica，Child Rewon，Polozov Oleksandr，Lee Katherine，Zhou Zongwei，Wang Xuezhi，Saeta Brennan，Diaz Mark，Firat Orhan，Catasta Michele，Wei Jason，Meier-Hellstern Kathy，Eck Douglas，Dean Jeff，Petrov Slav，Fiedel Noah。PaLM：通过路径扩展语言建模，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性和随机正则化，2016. arXiv:1606.08415。

Stefan Elfwing，Eiji Uchibe 和 Kenji Doya。Sigmoid加权的线性单元用于强化学习中的神经网络函数逼近，2017. URL https://arxiv.org/abs/1702.03118。

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

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

Jianlin Su，Yu Lu，Shengfeng Pan，Bo Wen 和 Yunfeng Liu。Roformer：具有旋转位置嵌入的增强Transformer，2021。

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

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

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proc. of NeurIPS*, 2020.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. arXiv:2001.08361.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. arXiv:2203.15556.

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

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 Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4344–4353, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1443. URL `https://aclanthology.org/D19-1443/`.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=Drrl2gcjzl`.

Diederik P. Kingma 并且 Jimmy Ba. Adam：一种随机优化的方法。在 *ICLR*会议论文集，2015 .

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

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

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

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes 和 Yejin Choi. 神经文本退化的奇怪案例。在*ICLR*会议论文集中，2020 年。

Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency 和 Ruslan Salakhutdinov. Transformer剖析：通过核的视角对Transformer的注意力进行统一理解。在 Kentaro Inui, Jing Jiang, Vincent Ng 和 Xiaojun Wan 编著的 《*2019*年自然语言处理实证方法会议和第*9*届国际自然语言处理联合会议（*EMNLP-IJCNLP*）论文集》 中，第4344–4353页，中国香港，2019年11月。计算语言学协会。doi: 10.18653/v1/D19-1443. URL `https://aclanthology.org/D19-1443/`.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan, Payel Das 和 Siva Reddy. 位置编码对 Transformer 中长度泛化的影响。在 第三十七届神经信息处理系统大会，2023。URL `https://openreview.net/forum?id=Drrl2gcjzl`。